КЛАССИКА COMPUTER SCIENCE

ВНУТРЕННЕЕ УСТРОЙСТВО WINDOWS

СЕДЬМОЕ ИЗДАНИЕ

МАРК РУССИНОВИЧ АЛЕКС ИОНЕСКУ
ДЭВИД СОЛОМОН ПАВЕЛ ЙОСИФОВИЧ

ПИТЕР

СЕРИЯ

КЛАССИКА COMPUTER SCIENCE

ПИТЕР®

Windows Internals
Seventh Edition
Part 1

System architecture, processes, threads, memory management, and more

Pavel Yosifovich, Alex Ionescu,
Mark E. Russinovich, and David A. Solomon

КЛАССИКА COMPUTER SCIENCE

Марк Руссинович, Дэвид Соломон,
Алекс Ионеску, Павел Йосифович

ВНУТРЕННЕЕ УСТРОЙСТВО WINDOWS

СЕДЬМОЕ ИЗДАНИЕ

ИНТЕР®
Санкт-Петербург · Москва · Екатеринбург · Воронеж
Нижний Новгород · Ростов-на-Дону
Самара · Минск
2018

ББК 32.973.2-018.2
УДК 004.451
В60

Руссинович М., Соломон Д., Ионеску А., Йосифович П.

В60 Внутреннее устройство Windows. 7-е изд. — СПб.: Питер, 2018. — 944 с.: ил. — (Серия «Классика computer science»).

ISBN 978-5-4461-0663-9

С момента выхода предыдущего издания этой книги операционная система Windows прошла длинный путь обновлений и концептуальных изменений, результатом которых стала новая стабильная архитектура ядра Windows 10.

Книга «Внутреннее устройство Windows» создана для профессионалов, желающих разобраться во внутренней жизни основных компонентов Windows 10. Опираясь на эту информацию, разработчикам будет проще находить правильные проектные решения, создавая приложения для платформы Windows, и решать сложные проблемы, связанные с их эксплуатацией. Системные администраторы, зная, что находится у операционной системы «под капотом», смогут разобраться с поведением системы и быстрее решать задачи повышения производительности и диагностики сбоев. Специалистам по безопасности пригодится информация о борьбе с уязвимостями операционной системы.

Прочитав эту книгу, вы будете лучше разбираться в работе Windows и в истинных причинах того или иного поведения ОС.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.973.2-018.2
УДК 004.451

ISBN 978-0735684188 англ.

ISBN 978-5-4461-0663-9

Оглавление

Моей семье — жене Идит и нашим детям Даниэль, Амиту и Йоаву; спасибо вам за терпение и поддержку во время этой непростой работы.

Павел Йосифович

Моим родителям, которые направляли и вдохновляли меня на моем пути к мечте, и моей семье, которая была рядом со мной всеми этими бесчисленными вечерами.

Алекс Ионеску

Нашим родителям, которые направляли и вдохновляли нас на пути к мечте.

Марк Руссинович и *Дэвид Соломон*

Введение

Седьмое издание книги *«Внутреннее устройство Windows»* предназначено для профессионалов (разработчиков, специалистов по безопасности и системных администраторов), желающих более глубоко разобраться в работе основных компонентов Microsoft Windows 10 и Windows Server 2016. Разработчики смогут лучше понять обоснование того или иного проектного решения при создании приложений для Windows, и смогут успешнее производить отладку сложных проблем.

Книга пригодится и системным администраторам, так как понимание «скрытых» принципов работы операционной системы позволяет лучше понять ее поведение и облегчить устранение неполадок в системе, если возникают сбои.

Специалисты по безопасности узнают, как злоумышленники могут воспользоваться уязвимостями операционной системы и вызвать нежелательное поведение, а также ознакомятся с защитными мерами и средствами безопасности, реализованными в современных версиях Windows.

Прочитав эту книгу, вы будете лучше разбираться в работе Windows и в причинах того или иного поведения ОС.

История книги

Это седьмое издание книги, которая сначала называлась *«Inside Windows NT»* (Microsoft Press, 1992) и была написана Хелен Кастер (Helen Custer) еще до выхода Microsoft Windows NT 3.1. *«Inside Windows NT»* была первой книгой, написанной о Windows NT и предоставившей ключевую информацию о сути архитектуры и конструкции системы. *«Inside Windows NT, Second Edition»* (Microsoft Press, 1998) была написана Дэвидом Соломоном (David Solomon). Она дополнила исходную книгу описанием Windows NT 4.0, а материал излагался на более глубоком техническом уровне.

Книга *«Inside Windows 2000, Third Edition»* (Microsoft Press, 2000) вышла под авторством Дэвида Соломона (David Solomon) и Марка Руссиновича (Mark Russinovich). В нее было добавлено множество новых тем, например: запуск и завершение работы, внутреннее устройство служб, реестра, драйверов файловой системы и сети. В ней также были рассмотрены изменения, внесенные в ядро Windows 2000, на-

пример: модель драйверов Windows Driver Model (WDM), Plug and Play, диспетчер энергопотребления, Windows Management Instrumentation (WMI), шифрование, объект задания и службы терминалов. В книгу «*Windows Internals, Fourth Edition*» были включены обновления, связанные с выходом Windows XP и Windows Server 2003, и дополнительный контент, предназначенный для IT-профессионалов в применении их знаний внутреннего устройства Windows — например, в использовании основных инструментов из комплекта Windows Sysinternals и в анализе аварийных дампов.

Книга «*Windows Internals, Fifth Edition*» (Microsoft Press, 2009) была обновлена под выход Windows Vista и Windows Server 2008. В это время Марк Руссинович перешел на полную ставку в Microsoft (где он теперь является техническим директором Azure), а у книги появился новый соавтор Алекс Ионеску. В новом материале описан загрузчик образов, средства отладки пользовательского режима, механизм ALPC (Advanced Local Procedure Call) и Hyper-V. Следующее издание, «*Windows Internals, Sixth Edition*»[1] (Microsoft Press, 2012), было полностью обновлено с учетом многочисленных изменений ядра в Windows 7 и Windows Server 2008 R2, с добавлением множества новых экспериментов в соответствии с изменениями в инструментарии.

Изменения, внесенные в седьмое издание

С выхода последнего издания этой книги система Windows прошла несколько обновлений, конечным результатом которых стал выход Windows 10 и Windows Server 2016. Система Windows 10, которая в настоящее время считается основной версией Windows, прошла несколько изданий, от первого выпуска до производственной версии. Каждая версия помечается номером версии из четырех цифр, обозначающим год и месяц выпуска, — например, Windows 10, версия 1703, которая была опубликована в марте 2017 года. Отсюда следует, что система Windows с выхода Windows 7 прошла как минимум шесть версий (на момент написания книги).

Начиная с Windows 8, корпорация Microsoft запустила процесс конвергенции ОС, полезный с точки зрения как разработчика, так и команды разработки Windows. В Windows 8 и Windows Phone 8 все началось с конвергенции ядра, затем в Windows 8.1 и Windows Phone 8.1 процесс продолжился конвергенцией современных приложений. История конвергенции завершилась в системе Windows 10, работающей на настольных/портативных компьютерах, серверах, XBOX One, телефонах (Windows Mobile 10), HoloLens и различных IoT-устройствах (Internet of Things).

С завершением грандиозной унификации пришло время для нового издания серии, которое наконец-то синхронизировалось с почти пятилетними изменениями и по-

[1] Внутреннее устройство Microsoft Windows. 6-е изд. — СПб.: Питер, 2013. — 800 с.: ил. — (Серия «Мастер-класс»).

явлением более стабильной архитектуры ядра. Соответственно, в новом издании книги рассматриваются аспекты Windows с Windows 8 до Windows 10, версия 1703. Кроме того, в новом издании в число соавторов вошел Павел Йосифович.

Практические эксперименты

Даже без доступа к исходному коду Windows вы многое можете почерпнуть о внутреннем устройстве Windows из таких инструментальных средств, как отладчик ядра, утилиты из пакета Sysinternals и инструменты, разработанные специально для этой книги. Когда инструментальное средство может быть использовано для показа или демонстрации некоторых аспектов внутреннего поведения Windows, действия, которые можно попытаться выполнить самостоятельно, описаны во врезках «ЭКСПЕРИМЕНТ». Они встречаются по всей книге, и мы хотим, чтобы вы провели все эти эксперименты по мере чтения книги — наглядное доказательство внутренней работы Windows сильнее отпечатается в вашей памяти, чем простое чтение описания этой работы.

Незатронутые темы

Windows это большая и сложная ОС. В этой книге мы не описываем все, что имеет отношение к внутреннему устройству Windows, но делаем упор на рассмотрение основных системных компонентов. Например, в книге не дается описание COM+, распределенной объектно-ориентированной программной инфраструктуры Windows, или среды Microsoft .NET Framework, которая является основой приложений с управляемым кодом. Поскольку книга посвящена внутреннему устройству, а не использованию, программированию или системному администрированию, в ней не рассматриваются вопросы использования, программирования или настройки Windows.

Предупреждение и предостережение

Поскольку в данной книге рассматривается недокументированное поведение внутренней архитектуры и функционирования операционной системы Windows (например, внутренних структур и функций ядра), в материале книги возможны изменения между выпусками.

Под «возможными изменениями» не обязательно имеется в виду, что подробности, рассмотренные в данной книге, будут меняться от выпуска к выпуску, но не рассчитывайте на то, что они не претерпят вообще никаких изменений. Любое программное обеспечение, использующее эти недокументированные интерфейсы в будущих релизах Windows, может оказаться неработоспособным. Хуже того, программы,

работающие в режиме ядра (например, драйверы устройств) и использующие эти недокументированные интерфейсы, могут при запуске новых выпусков Windows вызвать фатальный сбой с возможной потерей данных.

Одним словом, никогда не используйте внутреннюю функциональность Windows, разделы реестра, поведение, API или любые другие недокументированные подробности, описанные в книге, при разработке любых программ, предназначенных для конечного пользователя или любых других целей, кроме исследования и документирования. Поиск официальной документации по конкретным темам всегда следует начинать с MSDN (Microsoft Software Development Network).

Что мы ожидаем от читателя

Книга предполагает, что читатель уверенно работает с Windows на уровне опытного пользователя, а также понимает основные концепции операционных систем и оборудования: регистры процессора, память, процессы и программные потоки. В некоторых разделах книги также может пригодиться понимание функций, указателей и других конструкций языка программирования C.

Структура книги

Книга разделена на две части (как и в шестом издании); первую часть вы сейчас держите в руках.

- Глава 1 «Концепции и средства» знакомит читателя с концепциями внутреннего строения Windows и представляет основные инструменты, используемые в книге. Чрезвычайно важно начать чтение с этой главы, потому что в ней содержится вся вводная информация, необходимая для понимания остального материала.
- В главе 2 «Архитектура системы» представлена архитектура и основные компоненты Windows; некоторые из них изложены достаточно подробно. Другие концепции более подробно рассматриваются в последующих главах.
- Глава 3 «Процессы и задания» содержит подробное описание реализации процессов в Windows и различных операций с ними. Также здесь описаны задания как механизмы управления наборами процессов и поддержки контейнеров Windows.
- Глава 4 «Потоки» рассказывает об управлении, планировании и других операциях с программными потоками в Windows.
- Глава 5 «Управление памятью» показывает, как диспетчер памяти работает с физической и виртуальной памятью и как процессы и драйверы могут использовать память.

- Глава 6 «Подсистема ввода/вывода» показывает, как работает система ввода/вывода в Windows и как она интегрируется с драйверами устройств для формирования механизмов работы с периферийными устройствами ввода/вывода.
- Глава 7 «Безопасность» посвящена различным механизмам безопасности, встроенным в Windows. В частности, здесь рассматриваются защитные меры, которые сейчас стали частью системы борьбы с эксплойтами.

Благодарности

Прежде всего мы хотим поблагодарить Павла Йосифовича (Pavel Yosifovich), присоединившегося к этому проекту. Его участие сыграло исключительно важную роль для выпуска книги; только благодаря многим ночам, проведенным им за изучением подробностей Windows и сбором информации об изменениях в шести выпусках Windows, эта книга появилась на свет.

Книга не содержала бы столько технических подробностей и не была бы настолько точной, если бы не участие и поддержка ключевых участников группы разработки Microsoft Windows и других экспертов из Microsoft. Мы хотим поблагодарить людей, которые предоставили технические рецензии и/или исходный материал для книги или просто обеспечивали поддержку и помогали авторам: Акила Шриниваса (Akila Srinivasan), Алессандро Пилотти (Alessandro Pilotti), Андреа Аллиеви (Andrea Allievi), Энди Лурс (Andy Luhrs), Арун Кишан (Arun Kishan), Бен Хиллис (Ben Hillis), Билл Мессмер (Bill Messmer), Крис Клейнханс (Chris Kleynhans), Дипу Томас (Deepu Thomas), Юджин Бак (Eugene Bak), Джейсон Ширк (Jason Shirk), Джеремайя Кокс (Jeremiah Cox), Джо Бялек (Joe Bialek), Джон Ламберт (John Lambert), Джон Ленто (John Lento), Джон Берри (Jon Berry), Кай Су (Kai Hsu), Кен Джонсон (Ken Johnson), Лэнди Ванг (Landy Wang), Логан Габриэль (Logan Gabriel), Люк Ким (Luke Kim), Мэтт Миллер (Matt Miller), Мэтью Вулман (Matthew Woolman), Мехмет Иган (Mehmet Iyigun), Мишель Бержерон (Michelle Bergeron), Минсан Ким (Minsang Kim), Мохамед Мансур (Mohamed Mansour), Нэйт Уорфилд (Nate Warfield), Нирадж Сингх (Neeraj Singh), Ник Джадж (Nick Judge), Павел Лебединский (Pavel Lebedynskiy), Рич Тернер (Rich Turner), Сарухан Карадемир (Saruhan Karademir), Саймон Поуп (Simon Pope), Стивен Финниган (Stephen Finnigan) и Стивен Хафнагел (Stephen Hufnagel).

Хочется еще раз поблагодарить Ильфака Гуилфанова (Ilfak Guilfanov) из компании Hex-Rays (*www.hex-rays.com*) за лицензии IDA Pro Advanced и Hex-Rays, которые были предоставлены Алексу Ионеску (Alex Ionescu) более 10 лет назад, благодаря чему он смог ускорить свой анализ ядра Windows, а также за непрестанную поддержку и разработку средств декомпиляции, сделавших возможным написание этой книги без доступа к исходному коду.

И наконец, авторы хотят поблагодарить замечательный коллектив Microsoft Press, воплотивший эту книгу в реальность. Для Девона Масгрейва (Devon Musgrave)

этот проект стал последним на должности рецензента издательства, а Кейт Шауп (Kate Shoup) стала руководителем проекта. Шон Монингстар (Shawn Morningstar), Келли Тэлбот (Kelly Talbot) и Корина Лебеджоара (Corina Lebegioara) также внесли свой вклад в качество книги.

Список опечаток и качество книги

Мы приложили все усилия к тому, чтобы обеспечить точность материала книги и прилагаемого контента. Все ошибки, о которых было сообщено с момента публикации книги, указаны на сайте Microsoft Press по адресу:

https://aka.ms/winint7ed/errata

Если вы обнаружите ошибку, которая еще не указана в списке, вы можете сообщить нам об этом на той же странице.

Если вам понадобится дополнительная поддержка, обратитесь в службу поддержки Microsoft Press по адресу *mspinput@microsoft.com*.

Пожалуйста, учтите, что поддержка продуктов Microsoft по указанным выше адресам не предоставляется.

От издательства

Обращаем ваше внимание, что в данном издании были оставлены оригинальные скриншоты (с англоязычными названиями элементов интерфейса). В тексте книги использованы русские названия согласно официальной терминологии Microsoft (*https://www.microsoft.com/ru-ru/language*), а в скобках приведены англоязычные термины. Такой подход позволит вам легко ориентироваться в книге и использовать ее с любыми настройками ОС.

Оригинальное седьмое издание книги планируется выпустить в виде двух частей (как и шестое издание), поэтому на страницах данной книги вы найдете ссылки на вторую часть, которая (на момент выпуска первого тиража русскоязычного перевода) еще не была написана.

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

Глава 1
Концепции и средства

В этой главе будут представлены ключевые концепции и термины операционной системы (ОС) Microsoft Windows: Windows API, процессы, программные потоки, виртуальная память, режим ядра и пользовательский режим, объекты, дескрипторы, безопасность и реестр. Также мы рассмотрим некоторые средства, которые могут использоваться для исследования внутреннего строения Windows, например отладчик ядра, системный монитор и важнейшие программы из пакета Windows Sysinternals (*http://www.microsoft.com/technet/sysinternals*). Кроме того, мы объясним, как использовать пакеты Windows Driver Kit (WDK) и Windows Software Development Kit (SDK) для получения дальнейшей информации о внутреннем строении Windows.

После прочтения этой главы проверьте себя: все ли вы поняли? Если вы не усвоите материал этой главы, то не поймете материал оставшейся части книги.

Версии операционной системы Windows

В книге рассматриваются самые последние версии ОС Microsoft Windows для клиента и сервера: Windows 10 (32-разрядная для x86 и ARM, 64-разрядная для x64) и Windows Server 2016 R2 (существует только в 64-разрядной версии). Материал книги относится ко всем версиям, если в тексте явно не указано обратное. Для справки в табл. 1.1 перечислены названия продуктов семейства Windows, их внутренние номера версий и даты релиза.

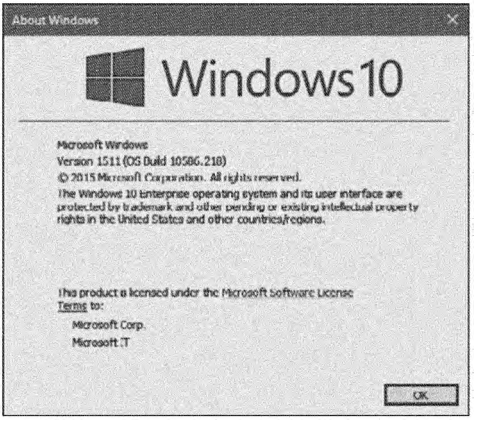
Начиная с Windows 7, нумерация версий перестала быть очевидной. Системе был присвоен номер версии 6.1 вместо 7. Из-за популярности Windows XP, когда в Windows Vista номер версии был повышен до 6.0, в некоторых приложениях проверка версии ОС стала работать некорректно — разработчики проверяли, что основная версия больше или равна 5, а дополнительная версия больше или равна 1; в Windows Vista это условие не выполнялось. Компания Microsoft усвоила урок и решила оставить основную версию 6 с повышением дополнительной версии до 2 (больше 1), чтобы свести к минимуму подобные несовместимости. Впрочем, в Windows 10 номер версии был обновлен до 10.0.

Таблица 1.1. Релизы операционной системы Windows

Название продукта	Внутренний номер версии	Дата релиза
Windows NT 3.1	3.1	июль 1993 г.
Windows NT 3.5	3.5	сентябрь 1994 г.
Windows NT 3.51	3.51	май 1995 г.
Windows NT 4.0	4.0	июль 1996 г.
Windows 2000	5.0	декабрь 1999 г.
Windows XP	5.1	август 2001 г.
Windows Server 2003	5.2	март 2003 г.
Windows Server 2003 R2	5.2	декабрь 2005 г.
Windows Vista	6.0	январь 2007 г.
Windows Server 2008	6.0 (Service Pack 1)	март 2008 г.
Windows 7	6.1	октябрь 2009 г.
Windows Server 2008 R2	6.1	октябрь 2009 г.
Windows 8	6.2	октябрь 2012 г.
Windows Server 2012	6.2	октябрь 2012 г.
Windows 8.1	6.3	октябрь 2013 г.
Windows Server 2012 R2	6.3	октябрь 2013 г.
Windows 10	10.0 (сборка 10240)	июль 2015 г.
Windows 10 версия 1511	10.0 (сборка 10586)	ноябрь 2015 г.
Windows 10 версия 1607 (Anniversary Update)	10.0 (сборка 14393)	июль 2016 г.
Windows Server 2016	10.0 (сборка 14393)	октябрь 2016 г.

ПРИМЕЧАНИЕ Начиная с Windows 8, функция Windows API GetVersionEx по умолчанию возвращает номер версии 6.2 (Windows 8) независимо от фактической версии ОС. (Эта функция также объявлена устаревшей.) Это сделано для того, чтобы свести к минимуму проблемы совместимости; кроме того, такой подход показывает, что проверка версии ОС в большинстве случаев не является оптимальным решением. Дело в том, что некоторые компоненты могут устанавливаться «раньше времени», без согласования с официальным выпуском Windows. Тем не менее, если вам нужно узнать фактическую версию ОС, вы можете получить ее при помощи функции VerifyVersionInfo или более новых вспомогательных API проверки версий: `IsWindows8OrGreater`, `IsWindows8Point1OrGreater`, `IsWindows10OrGreater`, `IsWindowsServer` и т. д. Кроме того, совместимость с разными операционными системами ОС может быть обозначена в манифесте исполняемого файла, что приводит к изменению результатов вызова этой функции. (За подробностями обращайтесь к разделу «Загрузчик образа» главы 3.)

Для просмотра информации о версии Windows можно воспользоваться программой командной строки ver или ее графической версией winver. Вот как выглядит результат выполнения winver в Windows 10 Enterprise версии 1511:



Графическая версия также выводит номер сборки Windows (10586.218 в данном примере); он может быть полезен для участников программы предварительной оценки Windows Insiders (зарегистрировавшихся для получения ранних ознакомительных версий Windows). Информация может пригодиться и для управления обновлениями безопасности, потому что в ней указан уровень установленного исправления.

Windows 10 и будущие версии Windows

С выходом Windows 10 компания Microsoft объявила, что обновление Windows отныне пойдет в более быстром темпе. Официальной версии «Windows 11» не будет; вместо этого Windows Update (или другая модель корпоративного обслуживания) будет обновлять существующую версию Windows 10 до новой версии. На момент написания книги произошло два обновления: в ноябре 2015 года (известное как «версия 1511» по году и месяцу) и в июле 2015 (версия 1607, известная под маркетинговым названием *«Anniversary Update»*).

ПРИМЕЧАНИЕ На внутреннем уровне Microsoft продолжает строить версии Windows «волнами». Например, исходной версии Windows 10 было присвоено кодовое название Threshold 1, тогда как обновление в ноябре 2015 года называлось Threshold 2. Следующие три фазы обновлений назывались Redstone 1 (версия 1607), Redstone 2 и Redstone 3.

Windows 10 и OneCore

За прошедшие годы появилось несколько разновидностей Windows. Кроме «массовых» версий Windows, работающих на PC, существует ответвление Windows 2000 для игровой приставки Xbox 360. Система Windows Phone 7 использует версию на базе Windows CE (ОС реального времени от Microsoft). Конечно, со-

провождение и расширение всех этих кодовых баз создавало немало сложностей, поэтому в Microsoft решили свести воедино разные ядра и базовые двоичные модули платформенной поддержки. Все началось с использования в Windows 8 и Windows Phone 8 общего ядра (а в Windows 8.1 и Windows Phone 8.1 — единого Windows Runtime API). В Windows 10 слияние завершилось; единая платформа, получившая название OneCore, работает на PC, смартфонах, игровой приставке Xbox One, HoloLens и устройствах IoT (Internet of Things), таких как Raspberry Pi 2.

Понятно, что все эти форм-факторы устройств очень сильно отличаются друг от друга. Некоторые функции на ряде устройств просто отсутствуют. Например, поддержка мыши или физической клавиатуры на устройстве HoloLens не имеет смысла, поэтому вряд ли можно ожидать наличия соответствующих компонентов в версии Windows 10 для таких устройств. Однако ядро, драйверы и двоичные файлы базовой платформы фактически остаются неизменными (с настройками на уровне реестра и/или политики по соображениям производительности или другим причинам). Пример такой политики приведен в разделе «Наборы API-функций» главы 3 «Процессы и задания».

В книге будет рассматриваться внутреннее строение ядра OneCore независимо от того, на каком устройстве оно выполняется. Тем не менее описанные в книге эксперименты предполагают использование настольного компьютера с мышью и клавиатурой, и воспроизвести их на других устройствах (скажем, на телефоне или Xbox One) непросто, а иногда и официально невозможно.

Фундаментальные концепции и термины

В этом разделе представлены фундаментальные концепции Windows, необходимые для понимания материала остальных глав книги. Многие концепции, такие как программные потоки и виртуальная память, подробно рассматриваются в последующих главах.

Windows API

Windows API (Application Programming Interface) — программный интерфейс пользовательского режима для ОС семейства Windows. До появления 64-разрядных версий Windows программный интерфейс 32-разрядных версий ОС Windows назывался *Win32 API* в отличие от исходного 16-разрядного Windows API, программного интерфейса для исходных 16-разрядных версий Windows. В этой книге термин *Windows API* относится как к 32-разрядным, так и 64-разрядным программным интерфейсам Windows.

ПРИМЕЧАНИЕ Иногда мы используем термин Win32 API вместо Windows API. В любом случае он все равно относится как к 32-разрядным, так и к 64-разрядным версиям.

ПРИМЕЧАНИЕ Описание Windows API содержится в документации Windows SDK. (См. раздел «Windows SDK» далее в этой главе.) Документация доступна бесплатно по адресу *https://developer.microsoft.com/en-us/windows/desktop/develop*. Также она включается во все уровни подписки MSDN (Microsoft Developer Network), программы поддержки разработчиков от компании Microsoft. Отличное введение в программирование с использованием базового Windows API представлено в книге «Windows via C/C++, Fifth Edition» Джеффри Рихтера (Jeffrey Richter) и Кристофа Насарра (Christophe Nasarre) (Microsoft Press, 2007).

Разновидности Windows API

Изначально Windows API состоял только из функций в стиле C. Сегодня разработчикам доступны тысячи функций. Выбор языка C был естественным на момент появления Windows, потому что он был своего рода «наименьшим общим кратным» (т. е. написанный на нем код также мог использоваться из других языков) и он был достаточно низкоуровневым для предоставления сервиса ОС. Обратной стороной было огромное количество функций в сочетании с недостаточной последовательностью выбора имен и отсутствием логических группировок (вроде пространств имен C++). Эти сложности привели к тому, что в некоторых новых API используется другой механизм — модель COM (Component Object Model, «модель составного объекта»).

Технология COM изначально создавалась для того, чтобы приложения Microsoft Office могли взаимодействовать друг с другом и передавать данные между документами (например, чтобы в документ Word можно было вставить диаграмму Excel или презентацию PowerPoint). Эта функциональность получила название OLE (Object Linking and Embedding, «связывание и внедрение объектов»). Сначала технология OLE была реализована на базе старого механизма передачи сообщений в Windows, который назывался DDE (Dynamic Data Exchange, «динамический обмен данными»). Технология DDE обладала рядом непреодолимых ограничений, поэтому был разработан новый коммуникационный механизм — COM. Более того, в первом варианте, который был представлен около 1993 года, технология COM изначально называлась OLE 2.

COM базируется на двух основополагающих принципах. Во-первых, клиенты взаимодействуют с объектами (которые иногда называются серверными объектами COM) через интерфейсы — четко определенные контракты с набором логически связанных методов, сгруппированных посредством механизма диспетчеризации по виртуальным таблицам (этот же механизм обычно применяется компиляторами C++ для реализации диспетчеризации виртуальных функций). Таким образом обеспечивается двоичная совместимость и снимаются проблемы с декорированием имен компилятором. Соответственно, такие методы могут вызываться из многих других языков (и компиляторов), включая C, C++, Visual Basic, языки .NET, Delphi и т. д. Второй принцип — динамическая загрузка компонентов (вместо статической компоновки с клиентом).

Термин *«сервер COM»* обычно относится к DLL-библиотеке или исполняемому файлу (EXE), в котором реализованы классы COM. COM также содержит ряд

важных функций, связанных с безопасностью, межпроцессным маршалингом, потоковой моделью и т. д. Подробное знакомство с COM выходит за рамки книги; отличное описание можно найти в книге Дона Бокса (Don Box) «Essential COM» (Addison-Wesley, 1998).

ПРИМЕЧАНИЕ Среди примеров API, доступ к которым осуществляется через COM, можно назвать DirectShow, Windows Media Foundation, DirectX, DirectComposition, WIC (Windows Imaging Component) и BITS (Background Intelligent Transfer Service).

Windows Runtime

В Windows 8 появились новый API и исполнительная среда поддержки *Windows Runtime* (иногда используется сокращение WinRT — не путайте с Windows RT, версии ОС Windows на базе ARM, поддержка которой была прекращена). Windows Runtime состоит из платформенных сервисов, предназначенных для разработчиков приложений *Windows Apps* (ранее также использовались термины *Metro Apps*, *Modern Apps*, *Immersive Apps* и *Windows Store Apps*). Приложения Windows Apps подходят для разных форм-факторов устройств, от миниатюрных IoT-устройств до телефонов, планшетов, ноутбуков, настольных систем, и даже таких устройств, как Xbox One и Microsoft HoloLens.

С точки зрения API платформа WinRT строится на базе COM, добавляя в базовую инфраструктуру COM различные расширения. Например, в WinRT доступны полные метаданные типов (хранящиеся в файле WINMD и основанные на формате метаданных .NET), расширяющие аналогичную концепцию библиотек типов в COM. С точки зрения архитектуры API она обладает намного большей целостностью, чем классические функции Windows API: в ней реализованы иерархии пространств имен, последовательная схема назначения имен и паттерны программирования.

Приложения Windows Apps строятся по новым правилам и не похожи на привычные приложения Windows (которые теперь называются настольными приложениями Windows или классическими приложениями Windows). Эти правила описаны в главе 9 «Механизмы управления» части 2[1].

Отношения между различными API и приложениями не столь прямолинейны. Настольные приложения могут использовать подмножество WinRT API. И наоборот, приложения Windows Apps могут использовать подмножество Win32 и COM API. За подробной информацией о том, какие API доступны для каждой платформы приложений, обращайтесь к документации MSDN.

Однако обратите внимание на то, что на базовом двоичном уровне WinRT API все равно строится на основе унаследованных двоичных файлов и API Windows, даже если доступность некоторых API не документирована и официально не под-

[1] Здесь и далее используются ссылки на вторую часть книги, которая на момент выпуска первого тиража русскоязычного перевода еще не опубликована (находится в работе у авторов).

держивается. Это не новый «машинный» API для системы, а ситуация напоминает то, как .NET строится на основе традиционного Windows API.

Приложения, написанные на C++, C# (и других языках .NET) и JavaScript, могут легко использовать WinRT API благодаря языковым проекциям, разработанным для этих платформ. Для C++ компания Microsoft создала нестандартное расширение C++/CX, которое упрощает работу с типами WinRT. Обычная прослойка взаимодействия COM для .NET (с некоторыми расширениями исполнительной среды) позволяет любому языку .NET использовать WinRT API естественно и просто, как если бы это была чистая среда .NET. Чтобы разработчики JavaScript могли работать с WinRT, было разработано расширение *WinJS*, хотя для построения пользовательских интерфейсов разработчикам JavaScript все равно приходится использовать HTML.

ПРИМЕЧАНИЕ Хотя разметка HTML может использоваться в приложениях Windows Apps, они все равно остаются локальными клиентскими приложениями, а не веб-приложениями, загружаемыми с веб-сервера.

.NET Framework

.NET Framework является частью Windows. В табл. 1.2 перечислены версии .NET Framework, устанавливаемые в составе разных версий Windows. Впрочем, новые версии .NET Framework могут устанавливаться и в старых версиях ОС.

Таблица 1.2. Версии .NET Framework, устанавливаемые по умолчанию в Windows

Версия Windows	Версия .NET Framework
Windows 8	4.5
Windows 8.1	4.5.1
Windows 10	4.6
Windows 10 версия 1511	4.6.1
Windows 10 версия 1607	4.6.2

.NET Framework состоит из двух основных компонентов:

- **CLR (Common Language Runtime).** Исполнительная среда .NET; включает JIT-компилятор (Just-In-Time) для преобразования инструкций языка CIL (Common Intermediate Language) в низкоуровневый язык машинных команд процессора, уборщик мусора, систему проверки типов, безопасность обращения к коду и т. д. Среда реализована в виде внутрипроцессного сервера COM (DLL) и использует различные средства, предоставляемые Windows API.
- **.NET Framework Class Library (FCL).** Обширная подборка типов, реализующих функциональность, часто используемую в клиентских и серверных при-

ложениях, – средства пользовательского интерфейса, поддержка сети, работа с базами данных и т. д.

Среда .NET Framework, предоставляющая эти и другие возможности, включая высокоуровневые языки программирования (C#, Visual Basic, F#) и вспомогательные средства, повышает производительность труда разработчика, а также безопасность и надежность приложений. На рис. 1.1 изображены отношения между .NET Framework и ОС Windows.

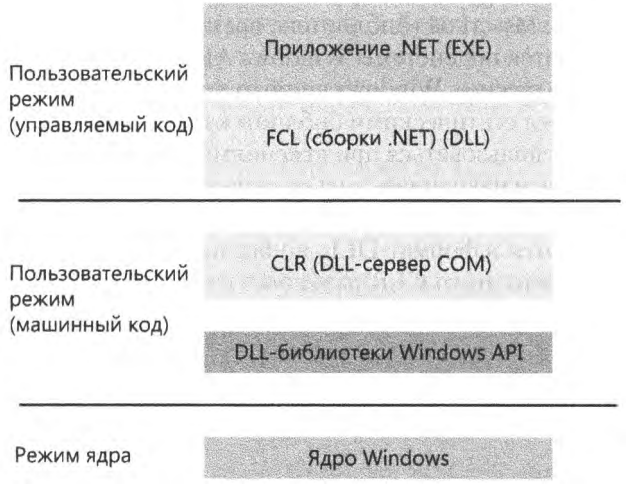


Рис. 1.1. Отношения между .NET и ОС Windows

Службы, функции и процедуры

Некоторые термины в документации пользователя и разработчика Windows имеют разный смысл в разных контекстах. Например, словом «*служба*» (service) может обозначаться код ОС, который может вызываться извне, драйвер устройства или серверный процесс. В следующем списке указано, какой смысл имеют некоторые термины из книги.

- **Функции Windows API.** Документированные, открытые для вызова процедуры Windows API. Примеры – `CreateProcess`, `CreateFile` и `GetMessage`.
- **Системные вызовы (низкоуровневые системные функции).** Недокументированные сервисные функции ОС, которые могут вызываться из пользовательского режима. Например, `NtCreateUserProcess` – внутренняя системная функция, вызываемая функцией Windows `CteateProcess` для создания нового процесса.
- **Вспомогательные функции ядра (процедуры).** Подпрограммы ОС Windows, которые могут вызываться только из режима ядра (см. далее в этой главе). Например, процедура `ExAllocatePoolWithTag` вызывается драйверами устройств для выделения памяти из системного пула Windows (так называемой *кучи*).

- **Службы Windows.** Процессы, запускаемые диспетчером служб Windows. Например, служба планировщика задач выполняется в процессе пользовательского режима, поддерживающего команду schtasks (аналог команд UNIX at и cron). (Заметим, что хотя в системном реестре драйверы устройств Windows определяются как «службы», в книге термин в этом контексте использоваться не будет.)
- **Библиотеки динамической компоновки (DLL).** Подпрограммы, предназначенные для внешнего вызова и объединенные в двоичные файлы, которые могут динамически загружаться приложениями, использующими эти подпрограммы. Примеры: Msvcrt.dll (библиотека времени выполнения C) и Kernel32.dll (одна из библиотек подсистемы Windows API). Приложения и компоненты пользовательского режима Windows широко используют DLL-библиотеки. Их преимущество перед статическими библиотеками заключается в том, что DLL могут совместно использоваться приложениями; система Windows позаботится о том, чтобы в памяти находилась только одна копия кода DLL-библиотеки для всех приложений, работающих с ней. Обратите внимание: библиотечные сборки .NET компилируются в формат DLL, но без неуправляемых экспортируемых подпрограмм. Вместо этого CLR разбирает откомпилированные метаданные для обращения к соответствующим типам и членам.

Процессы

На первый взгляд может показаться, что процессы и программы похожи, но в действительности между ними существуют принципиальные различия. *Программа* — статическая последовательность команд, тогда как *процесс* — контейнер для набора ресурсов, используемых для выполнения программы. На верхнем уровне абстракции процесс Windows включает следующие компоненты:

- **закрытое виртуальное адресное пространство** — множество адресов виртуальной памяти, которая может использоваться процессом;
- **исполняемая программа**, которая определяет первоначальный код и данные и отображается в виртуальное адресное пространство процесса;
- **список открытых дескрипторов** для различных системных ресурсов (семафоров, объектов синхронизации портов, файлов и т. д.), доступных для всех программных потоков в процессе;
- **контекст безопасности** — *маркер доступа* (access token), который идентифицирует пользователя, группы безопасности, привилегии, состояние виртуализации UAC (User Account Control), сеанс и ограниченное состояние учетной записи пользователя, связанное с процессом, а также идентификатор контейнера приложения и связанная с ним информация изоляции;
- **идентификатор процесса** — уникальный идентификатор, который является частью *идентификатора клиента*;

- **по меньшей мере один программный поток** (thread). Пустые процессы теоретически могут существовать, но особой пользы не принесут.

Существуют различные программы для просмотра (и изменения) процессов и информации процессов. Описанные ниже эксперименты показывают, какую информацию о процессах можно получить при помощи некоторых средств такого рода. Многие из этих средств включены в систему Windows, средства отладки для Windows и Windows SDK; другие являются самостоятельными программами из пакета Sysinternals. Многие программы выводят перекрывающиеся подмножества базовых данных процессов и программных потоков, которые иногда обозначаются разными именами.

Вероятно, для просмотра информации о процессах чаще всего используется диспетчер задач (Task Manager). (Выбор названия программы выглядит немного странно, так как в ядре Windows не существует понятия «задачи» (task).) Следующий эксперимент демонстрирует некоторые базовые возможности диспетчера задач.

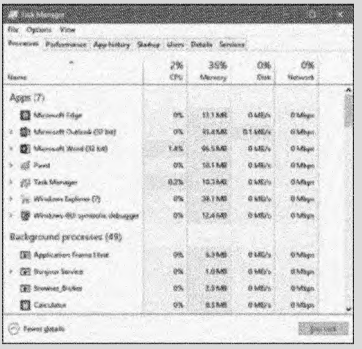
ЭКСПЕРИМЕНТ: ПРОСМОТР ИНФОРМАЦИИ ПРОЦЕССОВ В ДИСПЕТЧЕРЕ ЗАДАЧ

Диспетчер задач, встроенный в Windows, выдает простой список процессов в системе. Диспетчер задач можно запустить четырьмя способами:

- Нажмите Ctrl+Shift+Esc.
- Щелкните правой кнопкой мыши на панели задач и выберите команду диспетчер задач (Start Task Manager).
- Нажмите Ctrl+Alt+Del и щелкните на кнопке Запустить диспетчер задач (Start Task Manager).
- Запустите исполняемый файл Taskmgr.exe.

При первом запуске диспетчер задач работает в «кратком» режиме, в котором выводятся только процессы, имеющие видимое окно верхнего уровня, как на следующем снимке экрана:

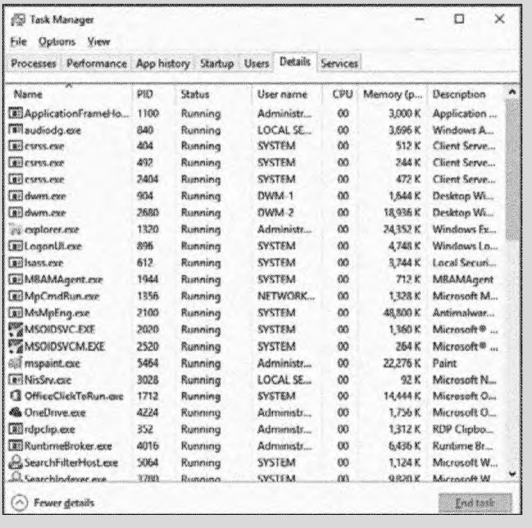
В этом режиме ваши возможности сильно ограничены, поэтому щелкните на кнопке Подробнее (More Details), чтобы открыть полное представление диспетчера задач. По умолчанию выбирается вкладка Процессы (Processes):



На вкладке Процессы (Processes) выводится список процессов, состоящий из четырех столбцов: ЦП (CPU), Память (Memory), Диск (Disk) и Сеть (Network). Чтобы добавить в список другие столбцы, щелкните правой кнопкой мыши на заголовке. Также доступны столбцы Process (Image) name, ИД процесса (Process ID), Тип (Type), Состояние (Status), Издатель (Publisher) и Командная строка (Command Line). Некоторые процессы можно дополнительно развернуть с выводом информации о видимых окнах верхнего уровня, созданных процессом.

Чтобы получить еще больше информации о процессе, щелкните на кнопке Подробнее (Details). Также можно щелкнуть правой кнопкой мыши на процессе и выбрать команду Подробнее (Go to Details), чтобы переключиться на вкладку Подробности (Details) и выбрать этот конкретный процесс.

ПРИМЕЧАНИЕ Вкладка Процессы (Processes) диспетчера задач Windows 7 приблизительно эквивалентна вкладке Подробности (Details) диспетчера задач Windows 8+. На вкладке Приложения (Applications) диспетчера задач Windows 7 выводятся видимые окна верхнего уровня, а не процессы как таковые. В новом диспетчере задач Windows 8+ эта информация теперь содержится на вкладке Процессы (Processes).



На вкладке Подробности (Details) также выводятся процессы, но в более компактном виде. На ней нет информации об окнах, созданных процессами, но больше столбцов с разнообразными данными.

Обратите внимание: процессы идентифицируются по имени образа, экземплярами которого они являются. В отличие от некоторых объектов Windows, процессам не могут присваиваться глобальные имена. Для вывода дополнительной информации щелкните правой кнопкой мыши на заголовке и щелкните на кнопке Выбрать столбцы (Select Columns). Открывается список столбцов, который выглядит так:

Некоторые важнейшие столбцы:

- **Потоки** (Threads) — в этом столбце выводится количество программных потоков в каждом процессе. Это число обычно не меньше 1, так как невозможно напрямую создать процесс, не содержащий ни одного потока (к тому же такой процесс будет бесполезен). Если в списке присутствует процесс с 0 потоков, обычно это означает, что процесс не удается удалить по какой-либо причине — чаще всего из-за ошибки в коде драйвера.
- **Дескрипторы** (Handles) — в этом столбце выводится количество дескрипторов объектов ядра, открытых программными потоками, выполняемыми в процессе. (Дескрипторы рассматриваются далее в этой главе, а также более подробно в главе 8 части 2.)
- **Состояние** (Status) — с этим столбцом дело обстоит сложнее. Для процессов, не имеющих пользовательского интерфейса, в нем обычно выводится значение Выполняется (Running), хотя все потоки могут чего-то ожидать, например сигнала о состоянии объекта ядра или завершения операции ввода/вывода. Другое возможное значение — Приостановлен (Suspended) — встречается в том случае, если все потоки процесса находятся в приостановленном состоянии. Вряд ли это произойдет в результате деятельности самого процесса, хотя может быть следствием вызова для процесса недокументированной функции API `NtSuspendProcess`, чаще всего при помощи служебной программы (например, программы Process Explorer, описанной ниже). Для процессов, создающих пользовательский интерфейс, состояние Выполняется (Running) означает, что пользовательский интерфейс реагирует на действия пользователя. Иначе говоря, поток, создавший окно (или окна), ожидает пользовательского ввода (с технической точки зрения — очереди сообщений, связанной с потоком). Состояние приостановки возможно и без пользовательского интерфейса, но

для приложений Windows Apps (на платформе Windows Runtime) приостановка обычно происходит тогда, когда приложение уходит с первого плана из-за того, что оно было свернуто пользователем. Такие процессы приостанавливаются через 5 секунд, чтобы они не поглощали ресурсы процессора или сети, а новое приложение первого плана могло получить в свое распоряжение все ресурсы машины. Это особенно важно для устройств с питанием от аккумулятора, таких как планшеты или телефоны. Эти и другие сопутствующие механизмы более подробно описаны в главе 9 части 2. Третье возможное значение в столбце Состояние (Status) — Не отвечает (Not Responding). Оно возникает в том случае, если программный поток процесса, создавший пользовательский интерфейс, не проверял свою очередь сообщений на предмет UI-событий по крайней мере 5 секунд. Процесс (а на самом деле поток, которому принадлежит окно) может быть занят работой, интенсивно загружающей процессор, или может ожидать чего-то совершенно иного (например, завершения операции ввода/вывода). В любом случае пользовательский интерфейс «зависает», а Windows сообщает об этом, затеняя такое окно (или окна) и выводя в столбце Состояние (Status) значение Не отвечает (Not Responding).

Каждый процесс также содержит идентификатор своего родительского процесса или процесса-создателя (они могут совпадать, но это не обязательно). Если родитель не существует, то информация не обновляется. Следовательно, процесс может хранить идентификатор несуществующего родителя; это не создает проблем, потому что система не полагается на актуальность этой информации. В программе Process Explorer учитывается время запуска родительского процесса, чтобы избежать связывания дочернего процесса на основании уже переназначенного идентификатора процесса. Следующий эксперимент демонстрирует это поведение.

ПРИМЕЧАНИЕ Почему процесс-родитель может не совпадать с процессом-создателем? В некоторых случаях процессы, которые на первый взгляд были созданы определенным пользовательским приложением, могут использовать участие вспомогательного процесса (процесса-брокера), отвечающего за вызовы API создания процессов. В таких случаях указание процесса-брокера в качестве создателя будет создавать путаницу (и даже ошибки при использовании наследования адресного пространства или дескрипторов), поэтому требуется «смена родителя». Один из примеров такого рода приведен в главе 7 «Безопасность».

ЭКСПЕРИМЕНТ: ПРОСМОТР ДЕРЕВА ПРОЦЕССОВ

Большинство служебных программ не выводит идентификаторы родителей или создателей процессов. Для получения этой информации можно воспользоваться Системным монитором (или запросить идентификатор процесса-создателя из программного кода). Также можно воспользоваться программой Tlist.exe из средств отладки Windows и запросить вывод дерева процессов при помощи ключа `/t`. Пример вывода команды `tlist /t`:

```
System Process (0)
System (4)
  smss.exe (360)
csrss.exe (460)
```

```
wininit.exe (524)
  services.exe (648)
    svchost.exe (736)
      unsecapp.exe (2516)
      WmiPrvSE.exe (2860)
      WmiPrvSE.exe (2512)
      RuntimeBroker.exe (3104)
      SkypeHost.exe (2776)
      ShellExperienceHost.exe (3760) Windows Shell Experience Host
      ApplicationFrameHost.exe (2848) OleMainThreadWndName
      SearchUI.exe (3504) Cortana
      WmiPrvSE.exe (1576)
      TiWorker.exe (6032)
      wuapihost.exe (5088)
    svchost.exe (788)
    svchost.exe (932)
    svchost.exe (960)
    svchost.exe (976)
    svchost.exe (68)
    svchost.exe (380)
    VSSVC.exe (1124)
    svchost.exe (1176)
      sihost.exe (3664)
      taskhostw.exe (3032) Task Host Window
    svchost.exe (1212)
    svchost.exe (1636)
    spoolsv.exe (1644)
    svchost.exe (1936)
    OfficeClickToRun.exe (1324)
    MSOIDSVC.EXE (1256)
      MSOIDSVCM.EXE (2264)
    MBAMAgent.exe (2072)
    MsMpEng.exe (2116)
    SearchIndexer.exe (1000)
      SearchProtocolHost.exe (824)
    svchost.exe (3328)
    svchost.exe (3428)
    svchost.exe (4400)
    svchost.exe (4360)
    svchost.exe (3720)
    TrustedInstaller.exe (6052)
  lsass.exe (664)
csrss.exe (536)
winlogon.exe (600)
  dwm.exe (1100) DWM Notification Window
explorer.exe (3148) Program Manager
  OneDrive.exe (4448)
  cmd.exe (5992) C:\windows\system32\cmd.exe - tlist /t
    conhost.exe (3120) CicMarshalWnd
    tlist.exe (5888)
SystemSettingsAdminFlows.exe (4608)
```

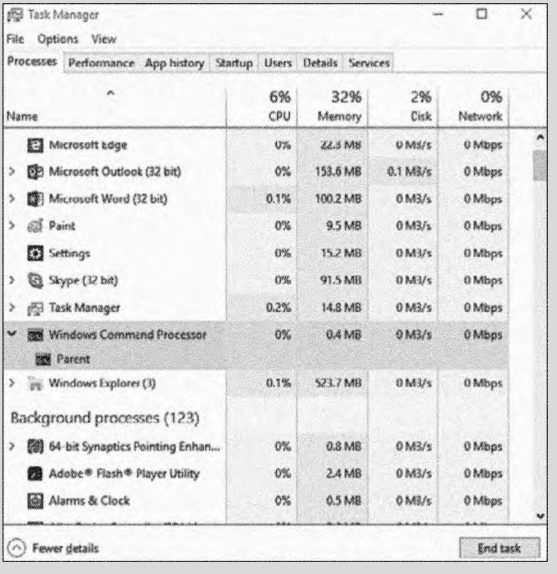
Отношения «родитель—потомок» в этом списке обозначаются отступами. Процессы, не имеющие «живых» родителей, выравниваются по левому краю (как explorer.exe в приведенном примере), потому что, даже если процесс-предок

более высокого уровня существует, отследить его невозможно. Windows хранит только идентификатор процесса-создателя, а не ссылки на всех предков.

Число в круглых скобках — идентификатор процесса, а текст после некоторых процессов — заголовок окна, созданного процессом.

Чтобы убедиться в том, что Windows не хранит ничего, кроме идентификатора родительского процесса, выполните следующие действия:

1. Нажмите клавиши Win+R, введите команду `cmd` и нажмите Enter, чтобы открыть окно командной строки.
2. Введите команду `title parent`, чтобы сменить текст заголовка окна на Parent.
3. Введите команду `start cmd`, чтобы открыть другое окно командной строки.
4. Введите команду `title Child` во втором окне командной строки.
5. Введите команду `mspaint` во втором окне командной строки, чтобы запустить Microsoft Paint.
6. Вернитесь ко второму окну командной строки и введите команду exit. Обратите внимание: окно Paint остается на экране.
7. Нажмите Ctrl+Shift+Esc, чтобы открыть диспетчер задач.
8. Если диспетчер задач находится в режиме сокращенного вывода, щелкните на кнопке Подробнее (More Details).
9. Перейдите на вкладку Процессы (Processes).
10. Найдите приложение Windows Command Processor и раскройте узел. В нем должен появиться заголовок *Parent*, как на следующем снимке экрана:



11. Щелкните правой кнопкой мыши на строке Windows Command Processor и выберите команду Подробнее (Go To Details).
12. Щелкните правой кнопкой мыши на процессе cmd.exe и выберите команду Завершить дерево процессов (End Process Tree).
13. Щелкните на кнопке Завершить дерево процессов (End Process Tree) в диалоговом окне подтверждения.

Первое окно командной строки закрывается, но окно Paint остается, потому что оно было «внуком» закрытого окна командной строки. Так как непосредственный родитель Paint был завершен, между первым процессом и его «внуком» связи не осталось.

Программа Process Explorer из пакета Sysinternals выводит больше информации о процессах и потоках, чем любая другая из существующих программ; по этой причине она будет использоваться во многих экспериментах этой книги. Ниже представлены некоторые уникальные возможности Process Explorer.

- Вывод маркера безопасности процесса (списки групп и привилегий, состояние виртуализации).
- Цветовое выделение для отображения изменений в списках процессов, потоков, DLL-библиотек и дескрипторов.
- Список служб в процессах-хостах служб, с выводимым именем и описанием.
- Список дополнительных атрибутов процессов (например, политики устранения рисков (mitigation) и уровни защиты процессов).
- Процессы, являющиеся частью заданий, и подробная информация о заданиях.
- Процессы-хосты приложений .NET и информация, относящаяся к .NET (например, список доменов приложений, загруженных сборок и счетчиков производительности CLR).
- Процессы, являющиеся хостами Windows Runtime.
- Время запуска процессов и потоков.
- Полный список файлов, отображенных в память (не только DLL).
- Возможность приостановки процесса или потока.
- Возможность уничтожения отдельных потоков.
- Простые средства идентификации процессов, создающих наибольшую нагрузку на процессор за период времени.

ПРИМЕЧАНИЕ Системный монитор может выводить информацию о загрузке процессора для заданного набора процессов, но не обеспечивает автоматического отображения процессов, созданных после начала сеанса. На это способна только ручная трассировка двоичных выходных данных.

Process Explorer также открывает централизованный удобный доступ к разнообразной информации:

- Дерево процессов с возможностью свертки частей дерева.
- Открытые дескрипторы в процессе, включая безымянные.
- Список DLL-библиотек (и файлов, отображенных в память).
- Активность потока в процессе.
- Стеки потоков пользовательского режима и режима ядра с информацией соответствия адресов и имен, полученной при помощи библиотеки Dbghelp.dll из инструментария отладки для Windows.
- Более точные данные загрузки процессора с использованием счетчика циклов потоков, формирующие более точную картину работы процессора (см. главу 4 «Потоки»).
- Уровни целостности.
- Подробная информация о распределении памяти: пиковое выделение памяти, лимиты выгружаемого и невыгружаемого пула памяти ядра (другие программы выводят только текущий размер).

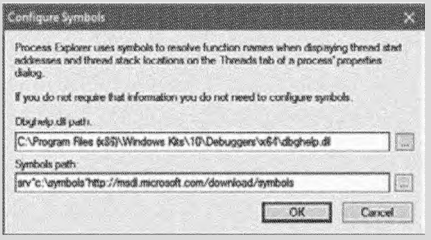
Ниже описан вводный эксперимент с использованием Process Explorer.

ЭКСПЕРИМЕНТ: ПРОСМОТР ИНФОРМАЦИИ О ПРОЦЕССАХ В PROCESS EXPLORER

Загрузите последнюю версию Process Explorer на сайте Sysinternals и запустите ее. Программу можно запустить с привилегиями стандартного пользователя. Также возможен другой вариант: щелкните правой кнопкой мыши на исполняемом файле и выберите команду Запустить от имени администратора (Run As Administrator), чтобы выполнить ее с привилегиями администратора. В этом случае Process Explorer устанавливает драйвер, предоставляющий расширенные возможности. Следующее описание работает одинаково независимо от способа запуска Process Explorer.

При первом запуске Process Explorer следует настроить информацию символических имен. Если этого не сделать, при двойном щелчке на процессе и переходе на вкладку Threads вы получите сообщение о том, что символические имена не настроены. При правильной настройке Process Explorer — получить доступ к символической информации для вывода символического имени стартовой функции потока, а также имен функций из стека вызовов потока. Эта возможность поможет понять, что делает каждый поток в процессе. Чтобы получить доступ к символическим именам, следует установить средства отладки для Windows (см. далее в этой главе). Выберите команду Options ▸ Configure Symbols, введите путь к библиотеке Dbghelp.dll в папке Debugging Tools и действительный путь к символической информации. Например, в 64-разрядной системе следующая

конфигурация будет верна, если средства отладки для Windows установлены в стандартной папке в составе WDK:

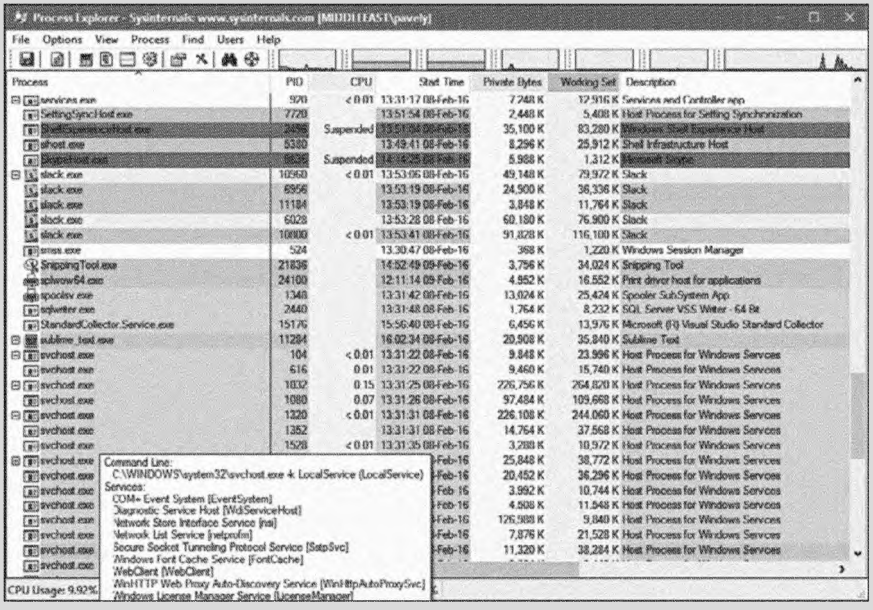


В приведенном примере используется сервер символической информации, а копия символических файлов хранится на локальной машине в папке C:\symbols. (При нехватке места вы можете заменить ее любой другой папкой, скажем, на другом диске.) За дополнительной информацией о настройке символических серверов обращайтесь по адресу *https://msdn.microsoft.com/en-us/library/windows/desktop/ee416588.aspx*.

СОВЕТ Символический сервер Microsoft можно настроить, присвоив переменной окружения `_NT_SYMBOL_PATH` значение, показанное на иллюстрации. Многие программы проверяют значение этой переменной автоматически — Process Explorer, отладчики, являющиеся частью средств отладки для Windows, Visual Studio и т. д. В этом случае вам не придется настраивать каждую программу по отдельности.

При запуске Process Explorer по умолчанию используется режим дерева процессов. Вы можете развернуть нижнюю панель, чтобы вывести списки открытых дескрипторов, DLL и файлов, отображенных в память. (Эта тема рассматривается в главе 5 «Управление памятью» и в главе 8 части 2.) Также выводится экранная подсказка с командной строкой и путем процесса; чтобы увидеть ее, наведите указатель мыши на имя процесса. Для некоторых типов процессов в подсказку также включаются дополнительные сведения:

- службы в процессе-хосте служб (например, Svchost.exe);
- задачи в процессе-хосте задач (например, TaskHostw.exe);
- цель процесса Rundll32.exe, используемая для элементов Панели управления и другой функциональности;
- информация класса COM при размещении в процессе Dllhost.exe (суррогат COM+ по умолчанию);
- информация о поставщике для процессов-хостов WMI (Windows Management Instrumentation), таких как WMIPrvSE.exe (дополнительная информация о WMI приведена в главе 8 части 2);
- информация пакета для процессов Windows Apps (процессы-хосты Windows Runtime; см. раздел «Windows Runtime» этой главы).



Некоторые основные возможности Process Explorer.

1. Обратите внимание: процессы, являющиеся хостами для служб, по умолчанию выделяются розовым цветом. Ваши собственные процессы выделены синим. Чтобы изменить цветовое выделение, откройте меню и выберите команду Options ▸ Configure Colors.

2. Наведите указатель мыши на имя образа процесса. На экране появляется подсказка с полным путем. Как упоминалось ранее, для некоторых типов процессов в подсказку включается дополнительная информация.

3. Выберите команду View ▸ Select Columns. На вкладке Process Image включите вывод пути образа (флажок Image Path).

4. Щелкните на заголовке столбца Process, чтобы отсортировать процессы. Обратите внимание: представление в виде дерева исчезает. (Вы можете либо вывести дерево, либо провести сортировку по любому из показанных столбцов.) Снова щелкните на заголовке столбца Process; вывод переключается на сортировку по убыванию (от Z к A). Третий щелчок возвращает список к режиму дерева.

5. Откройте меню View и снимите пометку команды Show Processes from All Users, чтобы в списке отображались только ваши процессы.

6. Откройте меню Options, выберите команду Difference Highlight Duration и введите новое значение: 3 секунды. Затем запустите новый процесс (любой). Обратите внимание: новый процесс выделяется зеленым цветом на 3 секунды. Завершите новый процесс; прежде чем исчезнуть из списка, он выделяется красным цветом на 3 секунды. Такое выделение может пригодиться для наблюдения за созданием и уничтожением процессов в вашей системе.

7. Дважды щелкните на процессе и исследуйте различные вкладки в списке свойств процесса. (Содержимое этих вкладок будет упоминаться в различных экспериментах в книге при объяснении выводимой на них информации.)

Потоки

Программный поток (или просто *поток*) — последовательность команд внутри процесса, планируемая Windows для исполнения. Без потоков программа процесса не сможет выполняться. Поток содержит следующие важнейшие компоненты:

- Содержимое набора регистров CPU, представляющих состояние процессора.
- Два стека: один для программного потока, который должен использоваться при выполнении в режиме ядра, другой — для выполнения в пользовательском режиме.
- Закрытая область памяти, называемая *локальной памятью потока команд* (TLS, Thread-Local Storage); используется подсистемами, библиотеками времени выполнения и динамическими библиотеками DLL.
- Уникальный идентификатор, называемый *идентификатором потока* (TID, Thread ID); является частью внутренней структуры идентификатора клиента (client ID). Идентификаторы процесса и потока генерируются из одного пространства имен, поэтому они никогда не перекрываются.

Кроме того, потоки иногда обладают собственным контекстом безопасности, который часто используется многопоточными серверными приложениями, олицетворяющими контекст безопасности обслуживаемых клиентов.

Регистры, стеки и закрытая область памяти называются *контекстом* потока. Так как эта информация зависит от архитектуры машины, на которой работает Windows, структура неизбежно привязывается к конкретной архитектуре. Функция Windows `GetThreadContext` предоставляет доступ к этой информации, зависящей от архитектуры (в виде блока `CONTEXT`).

Так как переключение выполнения между потоками требует участия планировщика ядра, эта операция может быть довольно затратной, особенно если два потока часто передают управление между собой. В Windows реализованы два механизма для сокращения этих затрат: *волокна* (fibers) и *планирование пользовательского режима* (UMS, User Mode Scheduling).

ПРИМЕЧАНИЕ Потоки 32-разрядного приложения, выполняемого в 64-разрядной версии Windows, будут содержать как 32-, так и 64-разрядные контексты, которые будут использоваться WoW64 (Windows on Windows 64) для переключения приложения из 32-разрядного режима в 64-разрядный при необходимости. Эти потоки будут иметь два стека пользовательского режима и два блока `CONTEXT`, а обычные функции Windows API будут возвращать 64-разрядный контекст. Впрочем, функция `Wow64GetThreadContext` будет возвращать 32-разрядный контекст. За дополнительной информацией о Wow64 обращайтесь к главе 8 части 2.

Волокна

Волокна позволяют приложению планировать свои потоки выполнения, а не полагаться на механизм приоритетного планирования, встроенный в Windows. Волокна также часто называют «облегченными потоками». В отношении планирования волокна невидимы для ядра, потому что они реализуются в пользовательском режиме в Kernel32.dll. Чтобы использовать волокна, следует сначала вызвать функцию Windows `ConvertThreadToFiber`. Эта функция преобразует поток в работающее волокно. В дальнейшем преобразованное волокно может создавать дополнительные волокна функцией `CreateFiber`. (Каждое волокно может иметь собственный набор волокон.) Однако, в отличие от потоков, волокно не начинает выполняться до того, как оно будет вручную выбрано вызовом функции `SwitchToFiber`. Новое волокно продолжает выполняться, пока не завершится или не вызовет `SwitchToFiber` с выбором другого волокна для выполнения. За дополнительной информацией обращайтесь к описанию функций волокон в документации Windows SDK.

ПРИМЕЧАНИЕ Использовать волокна обычно не рекомендуется, потому что они «невидимы» для ядра. Также волокна создают проблемы с совместным использованием локальной памяти потока (TLS), потому что несколько волокон могут выполняться в одном потоке. И хотя существует локальная память волокон (FLS, Fiber Local Storage), она не решает всех проблем совместного использования, а волокна, ориентированные на ввод/вывод, все равно работают плохо. Наконец, волокна не могут параллельно выполняться на нескольких процессорах и ограничиваются только кооперативной многозадачностью. Как правило, лучше доверить планирование ядру Windows и использовать для решения задач потоки.

Планирование пользовательского режима (UMS)

Потоки UMS (User Mode Scheduling), доступные только в 64-разрядных версиях Windows, предоставляют все основные преимущества волокон при минимуме их недостатков. Потоки UMS обладают собственным состоянием ядра, поэтому они «видимы» для ядра, что позволяет нескольким потокам UMS выдавать блокирующие системные вызовы, совместно использовать ресурсы и конкурировать за них. Или, когда двум и более потокам UMS требуется выполнить работу в пользовательском режиме, они могут периодически переключать контексты выполнения (уступая управление другому потоку) в пользовательском режиме, вместо того чтобы задействовать планировщик. С точки зрения ядра продолжает работать один поток, и ничего не изменилось. Когда поток UMS выполняет операцию, требующую входа в режим ядра (например, вызов системной функции), он переключается на специально выделенный поток режима ядра (так называемое *«направленное переключение контекста»*). И хотя параллельные потоки UMS также не могут выполняться на нескольких процессорах, они работают в условиях модели с вытеснением (pre-emptible), которая не является чисто кооперативной.

Хотя потоки обладают собственным контекстом выполнения, все потоки в процессе совместно используют виртуальное адресное пространство процесса (в дополнение

к остальным ресурсам, принадлежащим процессу). Это означает, что все потоки в процессе обладают полным доступом для чтения/записи к виртуальному адресному пространству процесса. При этом потоки не могут случайно обратиться к адресному пространству другого процесса, если только другой процесс не откроет доступ к части своего закрытого адресного пространства в виде раздела общей памяти (в Windows API используется термин «объект сопоставления файла», file mapping object) или если один процесс не обладает правом открытия другого процесса с использованием межпроцессных функций памяти, таких как `ReadProcessMemory` и `WriteProcessMemory` (процесс, выполняемый с той же учетной записью и не работающий внутри контейнера приложения или другого типа изолированной среды, может получить их по умолчанию, если целевой процесс не установит защиту).

Помимо закрытого адресного пространства и одного или нескольких потоков, у каждого процесса имеется контекст безопасности и список открытых дескрипторов объектов ядра: файлов, разделов общей памяти, объектов синхронизации (мьютексы, события или семафоры) — рис. 1.2.

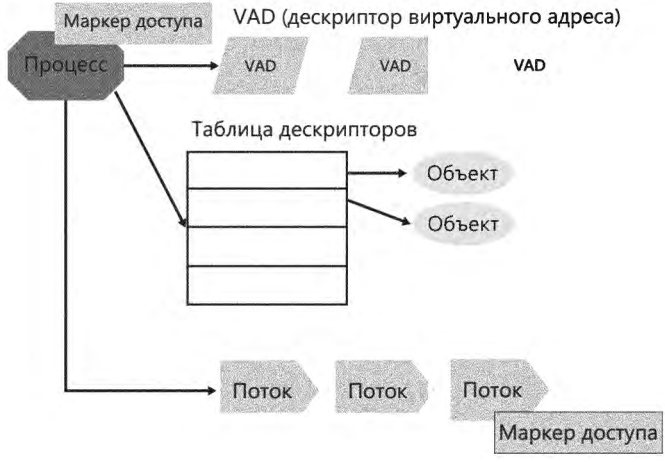


Рис. 1.2. Процесс и его ресурсы

Контент безопасности каждого процесса хранится в объекте, называемом *маркером доступа* (access token). Маркер доступа процесса содержит идентификационные данные безопасности и учетные данные процесса. По умолчанию потоки не имеют собственного маркера доступа, но могут получить его; это позволяет отдельным потокам олицетворять контент безопасности другого процесса (включая процессы в удаленной системе Windows), не затрагивая другие потоки в процессе. (За дополнительной информацией о процессах и безопасности потоков обращайтесь к главе 7.)

Дескрипторы виртуальных адресов (VAD, Virtual Address Descriptors) — структуры данных, используемые диспетчером памяти для отслеживания виртуальных адресов, используемых процессом. Эти структуры данных более подробно описаны в главе 5.

Задания

В Windows реализовано расширение модели процесса — так называемые *задания*. Главная функция объекта задания (job) — обеспечить возможность управления и выполнения операций с группами процессов как с единым целым. Объект задания позволяет управлять некоторыми атрибутами и устанавливает ограничения для процессов, связанных с заданием. Также он хранит основную учетную информацию для всех процессов, связанных с заданием, и для всех заданий, которые были связаны с заданием, но успели завершиться. В каком-то смысле объект задания компенсирует отсутствие структурированного дерева процессов в Windows — тем не менее во многих отношениях он мощнее дерева процессов в стиле UNIX.

ПРИМЕЧАНИЕ Process Explorer может выводить процессы, управляемые заданиями; обычно они обозначаются процессом, но по умолчанию эта возможность не включена (чтобы включить ее, откройте меню Options и выберите команду Configure Colors). Более того, страницы свойств таких процессов содержат дополнительную вкладку Job с информацией о самом объекте задания.

Внутренняя структура процессов и заданий подробнее описана в главе 3, а потоки и алгоритмы планирования потоков — в главе 4.

Виртуальная память

В Windows реализована система виртуальной памяти, основанная на модели плоского (линейного) адресного пространства, которая создает для каждого процесса иллюзию огромного закрытого адресного пространства. Виртуальная память формирует логическое представление памяти, которое может не соответствовать физической структуре. Во время выполнения диспетчер памяти — при поддержке оборудования — транслирует (*отображает*) виртуальные адреса в физические, где фактически хранятся данные. Управляя защитой и отображением адресов, ОС может предотвращать столкновения отдельных процессов или перезапись данных ОС.

Так как в большинстве систем объем физической памяти гораздо меньше суммарного объема виртуальной памяти, используемой работающими процессами, диспетчер памяти переносит (*выгружает*) часть содержимого памяти на диск. Выгрузка данных на диск освобождает физическую память, чтобы она могла использоваться для других процессов или самой ОС. Когда поток обращается по виртуальному адресу, выгруженному на диск, диспетчер виртуальной памяти подгружает информацию обратно в память с диска.

Чтобы пользоваться преимуществами виртуальной памяти, приложения не нужно никак изменять — поддержка оборудования позволяет диспетчеру памяти делать

все необходимое, при этом процессы и потоки ничего не знают о происходящем и никак не содействуют. На рис. 1.3 продемонстрировано использование виртуальной памяти двумя процессами; части памяти отображаются в физическую память (ОЗУ), а другие части выгружаются на диск. Обратите внимание на то, что непрерывные блоки виртуальной памяти могут отображаться в несмежные блоки физической памяти. Эти блоки называются *страницами*, а их размер по умолчанию составляет 4 Кбайт.

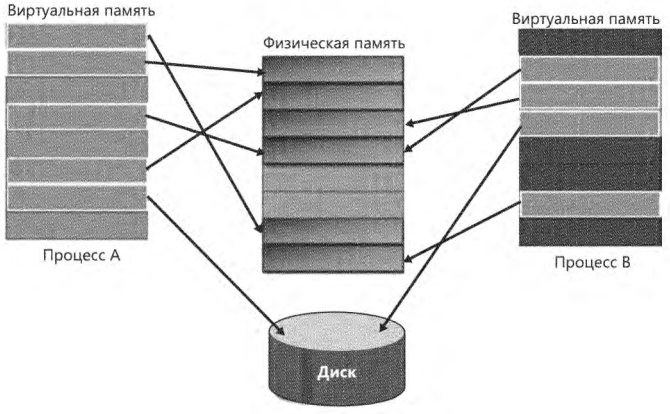


Рис. 1.3. Отображение виртуальной памяти в физическую

Размер виртуального адресного пространства изменяется в зависимости от аппаратной платформы. В 32-разрядных платформах x86 общее виртуальное адресное пространство ограничено теоретическим максимумом 4 Гбайт. По умолчанию Windows выделяет нижнюю половину адресного пространства (адреса от 0x80000000 до 0xFFFFFFFF) процессам для их собственной закрытой памяти, а верхнюю половину (адреса от 0x80000000 до 0xFFFFFFFF) — для собственного защищенного использования памяти ОС. Отображения нижней половины изменяются в соответствии с виртуальным адресным пространством текущего процесса, но отображения верхней половины (в большинстве) всегда состоят из виртуальной памяти ОС. Windows поддерживает параметры времени загрузки (например, квалификатор `increaseuserva` в базе данных Boot Configuration Database (см. главу 5)), которые предоставляют процессам, выполняющим особым образом помеченные программы, возможность использовать до 3 Гбайт закрытого адресного пространства, оставляя 1 Гбайт для ОС. (Под «особой пометкой» мы понимаем флаг обработки расширенного адресного пространства, установленный в заголовке исполняемого образа.) Этот параметр разрешает приложениям, таким как серверы баз данных, хранить большие объемы информации в адресном пространстве процесса, сокращая необходимость в отображении представлений базы данных на диске с повышением общей производительности (хотя в некоторых случаях потеря 1 Гбайт для системы может привести к более ярко выраженной потере произво-

дительности на общесистемном уровне). На рис. 1.4 изображены две типичные структуры виртуального адресного пространства, поддерживаемые 32-разрядными версиями Windows. (Параметр `increaseuserva` позволяет исполняемым образам с установленным флагом обработки расширенного адресного пространства использовать адреса от 2 до 3 Гбайт.)

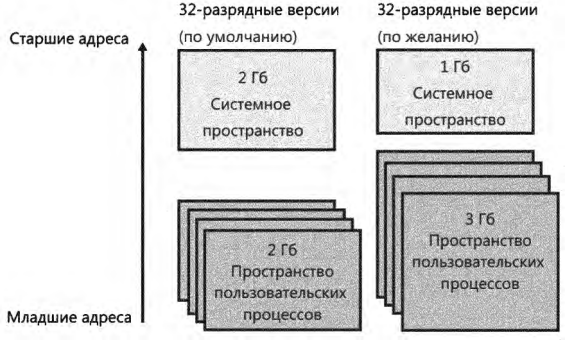


Рис. 1.4. Типичная структура адресного пространства в 32-разрядных версиях Windows

Хотя три гигабайта лучше двух, этого все равно недостаточно для отображения в виртуальное адресное пространство очень больших (многогигабайтных) баз данных. Для решения этой проблемы в 32-разрядных системах Windows предоставляет механизм AWE (Address Windowing Extensions), позволяющий 32-разрядному приложению выделить до 64 Гбайт физической памяти, а затем отображать представления (или окна) в свое 2-гигабайтное виртуальное адресное пространство. И хотя использование AWE перекладывает бремя управления отображением виртуальной памяти в физическую память на разработчика, оно решает проблему с необходимостью прямого обращения к физической памяти большого объема, которая не может быть отображена в 32-разрядное адресное пространство процесса.

64-разрядные версии Windows предоставляют для процессов значительно большее адресное пространство: 128 Тбайт в Windows 8.1, Server 2012 R2 и более поздних системах. На рис. 1.5 изображено упрощенное представление структуры адресного пространства в 64-разрядных системах. (За подробным описанием обращайтесь к главе 5.) Учтите, что эти размеры не определяются архитектурными ограничениями таких платформ. 64 бита адресного пространства — это 2 в 64-й степени, или 16 Эбайт (1 Эбайт = 1024 Пбайт, или 1 048 576 Тбайт), но современное 64-разрядное оборудование ограничивает его меньшими значениями. Неотображаемая область на рис. 1.5 много больше возможной отображаемой области (приблизительно в 1 миллион раз больше в Windows 8), так что масштаб на изображениях не соблюдается (притом очень сильно).

Подробное описание реализации диспетчера памяти, включая преобразование адресов и управление физической памятью в Windows, приведено в главе 5.

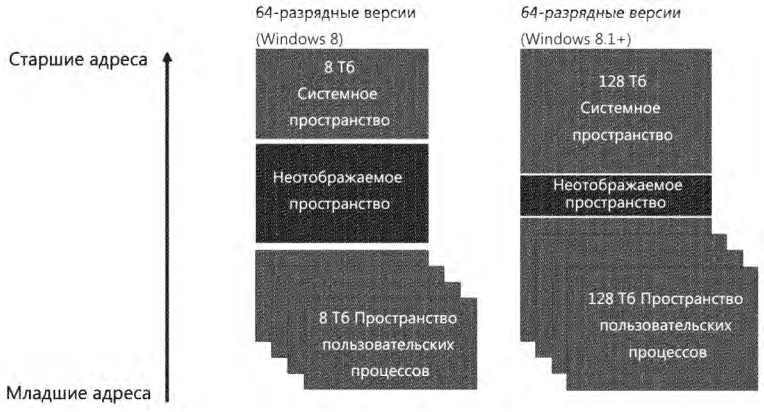


Рис. 1.5. Типичная структура адресного пространства в 64-разрядных версиях Windows

Режим ядра и пользовательский режим

Чтобы пользовательские приложения не могли прочитать критические данные операционной системы и/или изменить их, в Windows предусмотрены два режима доступа к процессору (даже если процессор, на котором работает Windows, поддерживает более двух режимов): *пользовательский режим* (user mode) и *режим ядра* (kernel mode). Код пользовательских приложений выполняется в пользовательском режиме, а код ОС (системные службы, драйверы устройств и т. д.) работает только в режиме ядра. *Режимом ядра* называется режим работы процессора, в котором доступна вся системная память и все команды процессора. У одних процессоров для описания различий между режимами используется термин «уровень привилегий» или «кольцо», другие используют такие термины, как «режим супервизора» и «режим приложения». Независимо от названия низкоуровневое программное обеспечение операционной системы наделяется более высоким уровнем привилегий, чем процессы пользовательского режима, чтобы некорректное поведение приложения не могло нарушить стабильность системы в целом.

ПРИМЕЧАНИЕ Архитектуры процессоров x86 и x64 определяют четыре уровня привилегий (четыре кольца) для защиты системного кода и данных от случайной или злонамеренной перезаписи менее привилегированным кодом. Windows использует уровень привилегий 0 (кольцо 0) для режима ядра и уровень привилегий 3 (кольцо 3) для пользовательского режима. Причина, по которой Windows использует только два уровня, заключается в том, что в некоторых аппаратных архитектурах (ARM в наши дни, MIPS/Alpha в прошлом) реализуется только два уровня привилегий. Выбор минимального уровня, поддерживаемого всеми системами, позволяет реализовать более эффективную и портируемую архитектуру, особенно если учесть, что другие уровни колец x86/x64 не предоставляют таких же гарантий, как комбинация колец 0/3.

Хотя каждый процесс Windows обладает собственным закрытым адресным пространством, код ОС режима ядра и код драйверов устройств совместно используют единое виртуальное адресное пространство. Каждая страница виртуальной памяти снабжается метками, показывающими, в каком режиме доступа должен находиться процессор для выполнения чтения и/или записи в страницу. Страницы системного пространства доступны только из режима ядра, тогда как все страницы пользовательского адресного пространства доступны как из пользовательского режима, так и из режима ядра. Страницы, доступные только для чтения (например, страницы, содержащие статические данные), недоступны для записи из любого режима. Кроме того, на процессорах, поддерживающих защиту памяти с запретом исполнения, Windows помечает страницы с данными как неисполняемые, предотвращая случайное или злонамеренное исполнение кода в областях данных (при включенном механизме защиты DEP (Data Execution Prevension)).

Windows не предоставляет никакой защиты при использовании закрытой системной памяти, доступной для чтения/записи, компонентами, работающими в режиме ядра. Иначе говоря, в режиме ядра код ОС и драйверов устройств обладает полным доступом к памяти системного пространства и может обойти механизмы безопасности Windows при обращении к объектам. Так как основная часть кода ОС Windows работает в режиме ядра, очень важно, чтобы компоненты, работающие в режиме ядра, были тщательно спроектированы и протестированы. Это необходимо для предотвращения возможных нарушений системной безопасности или подрыва стабильности системы.

Отсутствие защиты также показывает, насколько важно сохранять бдительность при загрузке сторонних драйверов устройств — особенно неподписанных, потому что в режиме ядра драйвер обладает полным доступом ко всем данным ОС. Именно по этой причине в Windows 2000 появился механизм цифровой подписи драйверов. Он предупреждает пользователя о попытке добавления неподписанного драйвера Plug-and-Play (или при соответствующей настройке блокирует такие попытки), но не влияет на другие типы драйверов. За дополнительной информацией о цифровой подписи драйверов обращайтесь к главе 6 «Подсистема ввода/вывода». Кроме того, механизм Driver Verifier помогает разработчикам драйверов устройств находить ошибки (такие, как переполнение буфера или утечка памяти), способные вызвать проблемы безопасности или надежности. (Механизм Driver Verifier также рассматривается в главе 6.)

В 64-разрядных и ARM-версиях Windows 8.1 политика подписывания кода режима ядра (KMCS, Kernel-Mode Code-Signing) требует, чтобы все драйверы устройств (а не только драйверы Plug-and-Play) подписывались с криптографическим ключом, выданным одним из ведущих центров сертификации. Пользователь не может выполнить принудительную установку неподписанного драйвера, даже с правами администратора. Тем не менее в качестве разового исключения это ограничение можно отключить вручную. В этом случае драйверы самоподписываются, на обоях рабочего стола выводится надпись «Тестовый режим», а некоторые функции управления цифровыми правами DRM (Digital Rights Management) отключаются.

В Windows 10 компания Microsoft реализовала еще более значительное изменение, которое было введено через год после выхода системы в составе июльского обновления Anniversary Update (версия 1607). В то время все новые драйверы Windows 10 должны были подписываться только двумя возможными центрами сертификации с сертификатом SHA-2 Extended Validation (EV) вместо обычного файлового сертификата SHA-1 и его 20 центрами сертификации. После снабжения подписью EV драйвер оборудования должен был отправляться в Microsoft через портал SysDev (System Device) для аттестации, после которой драйвер получал цифровую подпись Microsoft. Соответственно, ядро принимало только драйверы Windows 10 с подписью Microsoft без каких-либо исключений, кроме упомянутого выше тестового режима. Драйверы, подписанные до выхода Windows 10 (июль 2015 года), на тот момент продолжали загружаться с обычной подписью.

В Windows Server 2016 операционная система действует еще радикальнее. Кроме упоминавшихся требований EV простого аттестационного подписывания недостаточно. Чтобы драйвер Windows 10 загружался в серверной системе, он должен пройти жесткий процесс сертификации WHQL (Windows Hardware Quality Labs) в составе HCK (Hardware Compatibility Kit) и пройти формальную оценку. Только драйверам с подписью WHQL — предоставляющим системным администраторам определенные гарантии совместимости, безопасности, производительности и стабильности — разрешалась загрузка в таких системах. В целом сокращение количества сторонних драйверов, которым разрешалась загрузка в режиме ядра, значительно повысило стабильность и безопасность систем.

В версиях Windows некоторых производителей оборудования, платформ и даже в корпоративных конфигурациях любые из этих политик подписывания могут быть настроены, например, с использованием технологии Device Guard, которая будет кратко описана далее в разделе «Гипервизор» и в главе 7. По этой причине корпоративная версия может требовать подписей WHQL даже в клиентских системах Windows 10 или же запросить ослабление этого требования в системе Windows Server 2016.

Как будет показано в главе 2 «Архитектура системы», пользовательские приложения переключаются из пользовательского режима в режим ядра при вызове системных функций. Например, функция Windows `ReadFile` в конечном итоге должна вызвать внутреннюю функцию Windows, которая выполняет непосредственное чтение данных из файла. Поскольку эта функция обращается к внутренним системным структурам данных, она должна работать в режиме ядра. Специальная команда процессора инициирует переключение из пользовательского режима в режим ядра и заставляет процессор войти в код диспетчеризации системных вызовов в ядре. В свою очередь, этот код вызывает соответствующую внутреннюю функцию: Ntoskrnl.exe или Win32k.sys. Перед возвращением управления пользовательскому потоку процессор снова переключается в пользовательский режим. Таким образом ОС защищает себя и свои данные от анализа и модификации со стороны пользовательских процессов.

ПРИМЕЧАНИЕ Переключение из пользовательского режима в режим ядра (и обратно) не влияет на планирование потоков как таковое. Переключение режима *не* является переключением контекста. Дополнительная информация о диспетчеризации системных вызовов приведена в главе 2.

Таким образом, для пользовательского потока нормально проводить часть времени в пользовательском режиме и часть времени в режиме ядра. Более того, поскольку большая часть графической и оконной системы также выполняется в режиме ядра, процессы приложений, интенсивно работающих с графикой, могут проводить в режиме ядра больше времени, чем в пользовательском режиме. Чтобы убедиться в этом, запустите какое-нибудь графическое приложение (например, Microsoft Paint) и понаблюдайте за распределением времени между пользовательским режимом и режимом ядра при помощи одного из счетчиков производительности из табл. 1.3.

Более современные приложения могут использовать новые технологии (например, Direct2D и DirectComposition), которые выполняют большие объемы вычислений в пользовательском режиме и передают ядру только низкоуровневые данные поверхностей. Таким образом сокращается время, расходуемое на переключение между пользовательским режимом и режимом ядра.

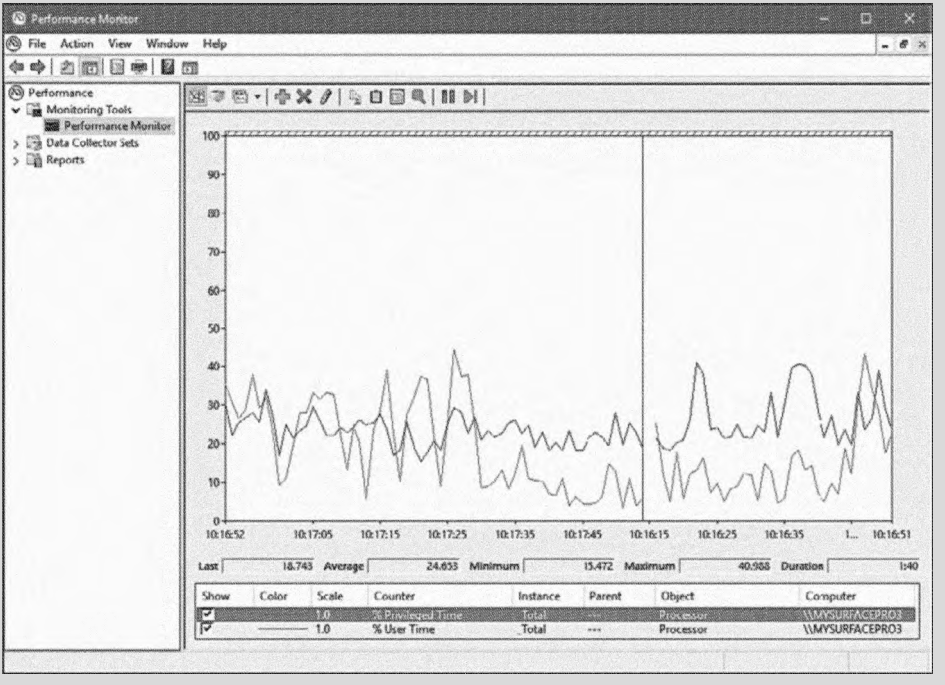
Таблица 1.3. Счетчики производительности, относящиеся к режиму процессора

Объект: счетчик	Функция
Процессор: % работы в привилегированном режиме (Processor: % Privileged Time)	Процент времени, проводимого конкретным процессором (или всеми процессорами) в режиме ядра, за заданный промежуток времени
Процессор: % работы в пользовательском режиме (Processor: % User Time)	Процент времени, проводимого конкретным процессором (или всеми процессорами) в пользовательском режиме, за заданный промежуток времени
Процесс: % работы в привилегированном режиме (Process: % Privileged Time)	Процент времени, проводимого потоками процесса в режиме ядра, за заданный промежуток времени
Процесс: % работы в пользовательском режиме (Process: % User Time)	Процент времени, проводимого потоками процесса в пользовательском режиме, за заданный промежуток времени
Поток: % работы в привилегированном режиме (Thread: % Privileged Time)	Процент времени, проводимого потоком в режиме ядра, за заданный промежуток времени
Поток: % работы в пользовательском режиме (Thread: % User Time)	Процент времени, проводимого потоком в пользовательском режиме, за заданный промежуток времени

ЭКСПЕРИМЕНТ: РЕЖИМ ЯДРА И ПОЛЬЗОВАТЕЛЬСКИЙ РЕЖИМ

Чтобы узнать, сколько времени ваша система проводит за выполнением в режиме ядра и в пользовательском режиме, выполните следующие действия:

1. Откройте меню Пуск (Start) и введите команду Run Performance Monitor (команда должна быть заполнена автоматически еще до того, как вы закончите ее вводить), чтобы запустить Системный монитор.
2. Выберите узел Системный монитор (Performance Monitor) в категории Средства наблюдения (Performance/Monitoring Tools) в дереве на левой панели.
3. Чтобы удалить счетчик по умолчанию, отображающий общее время процессора, щелкните на кнопке Удалить (Delete) на панели инструментов или нажмите клавишу Delete на клавиатуре.
4. Щелкните на кнопке Добавить (Add) (+) на панели инструментов.
5. Разверните категорию Процессор (Processor), щелкните на счетчике % работы в привилегированном режиме (% Privileged Time) и, удерживая клавишу Ctrl, щелкните на счетчике % работы в пользовательском режиме (% User Time).
6. Щелкните на кнопке Добавить (Add), затем щелкните на кнопке OK.
7. Откройте окно командной строки и введите команду `dir \\%computername%\c$/s`, чтобы просканировать структуру каталогов на диске C.



8. По окончании работы закройте программу.

Эту информацию также можно быстро получить в диспетчере задач. Перейдите на вкладку Быстродействие (Performance), щелкните правой кнопкой мыши на графике загрузки процессора и выберите команду Вывод времени ядра (Show Kernel Times). На графике загрузки процессора время, проводимое в режиме ядра, будет отображаться более темным оттенком синего.

Чтобы увидеть, как сам Системный монитор использует данные о времени, проводимом в режиме ядра и пользовательском режиме, добавьте счетчики времени режима ядра и пользовательского режима для каждого процесса в системе.

1. Если Системный монитор был закрыт, снова запустите его. (Если он уже работает, сбросьте отображаемую информацию; для этого щелкните правой кнопкой мыши на области графика и выберите команду Удалить все счетчики (Remove All Counters).)
2. Щелкните на кнопке Добавить (Add) на панели инструментов.
3. В области доступных счетчиков раскройте категорию Процесс (Process).
4. Выберите счетчики % работы в привилегированном режиме (% Privileged Time) и % работы в пользовательском режиме (% User Time).
5. Выберите несколько процессов в поле Экземпляр (Instance) (например, mmc, csrss и Idle).
6. Щелкните на кнопке Добавить (Add), затем щелкните на кнопке OK.
7. Быстро переместите указатель мыши по экрану.
8. Нажмите клавиши Ctrl+H, чтобы включить режим цветового выделения. Текущий выбранный счетчик выделяется черным цветом.
9. Прокрутите содержимое окна, чтобы найти процессы, потоки которых выполнялись при перемещении мыши. Посмотрите, выполнялись ли они в пользовательском режиме или в режиме ядра.

При перемещении мыши вы увидите, что в столбце Экземпляр (Instance) процесса mmc Системного монитора возрастает время как в режиме ядра, так и в пользовательском режиме. Это связано с тем, что процесс выполняет код приложения в пользовательском режиме и вызывает функции Windows, выполняемые в режиме ядра. Также при перемещении мыши можно заметить активность потока режима ядра в процессе с именем csrss. Эта активность обусловлена тем, что к этому процессу присоединен поток низкоуровневого ввода режима ядра подсистемы Windows, обеспечивающий ввод данных с клавиатуры и мыши. (За дополнительной информацией о системных потоках и подсистемах обращайтесь к главе 2.) Наконец, процесс Idle, который проводит почти 100 % времени в режиме ядра, процессом вообще не является — это псевдопроцесс, используемый для учета простоя процессорного времени. Как можно заметить по режиму, в котором выполняются потоки процесса Idle, когда системе Windows нечем заняться, она занимается этим в режиме ядра.

Гипервизор

Последние достижения в сфере приложений и программного обеспечения, такие как появление облачных сервисов и повсеместное распространение устройств IoT, привели к тому, что операционным системам и производителям оборудования приходится искать более эффективные способы виртуализации других «гостей» ОС на оборудовании машины, будь то размещение многочисленных обитателей фермы серверов и работа ста изолированных веб-сайтов на одном сервере или же возможность тестирования разработчиками десятков разновидностей ОС без покупки специализированного оборудования. Потребность в быстрой, эффективной и безопасной виртуализации породила новые модели вычислений и подходов к проектированию программных продуктов. В наши дни некоторые программы работают в контейнерах, которые обеспечивают полную изоляцию виртуальных машин, предназначенных исключительно для выполнения одного стека приложений или инфраструктуры (в качестве примера можно привести технологию Docker, поддерживаемую в Windows 10 и Server 2016); таким образом, границы между управляющей системой и гостями поднимаются на новый уровень.

Для предоставления сервиса виртуализации почти все современные решения пользуются услугами *гипервизора* — специализированного, высокопривилегированного компонента, который обеспечивает виртуализацию и изоляцию всех ресурсов машины, от виртуальной и физической памяти до прерываний устройств и даже устройств с интерфейсами PCI и USB. Например, в основе функциональности клиента Hyper-V, предоставляемой в Windows 8.1 и выше, лежит гипервизор Hyper-V. Все конкурирующие продукты — Xen, KVM, VMware и VirtualBox — реализуют собственные гипервизоры, каждый из которых обладает своими достоинствами и недостатками.

Из-за своей привилегированной природы и из-за уровня доступа даже большего, чем у самого ядра, гипервизор имеет очевидное преимущество перед гостевыми экземплярами других операционных систем: он может обеспечивать защиту и сбор управляющей информации одного экземпляра хоста для предоставления гарантий, превышающих возможности ядра. В Windows 10 компания Microsoft использует гипервизор Hyper-V для предоставления нового набора сервисов виртуализационной безопасности VBS (Virtualization-Based Security):

- **Device Guard** предоставляет сервис HVCI (Hypervisor Code Integrity) для усиления гарантий подписывания кода по сравнению с KMCS и обеспечивает возможность настройки политики подписывания в ОС Windows как для кода пользовательского режима, так и для кода режима ядра.
- **Hyper Guard** защищает ключевые структуры данных и код, относящиеся к режиму ядра и гипервизору.
- **Credential Guard** предотвращает несанкционированный доступ к учетным данным и секретам учетной записи домена в сочетании с безопасной биометрией.
- **Application Guard** — еще более сильная изоляция для браузера Microsoft Edge.

- **Host Guardian** и **Shielded Fabric** используют виртуальный модуль TPM (v-TPM) для защиты виртуальной машины от инфраструктуры, на которой она работает.

Кроме того, гипервизор Hyper-V обеспечивает некоторые ключевые меры защиты ядра от эксплойтов и других видов атак. Главное преимущество всех этих технологий заключается в том, что в отличие от предыдущих средств безопасности на базе ядра они неуязвимы для плохо написанных или вредоносных драйверов независимо от того, подписаны они или нет. Тем самым обеспечивается их высокая устойчивость перед опытными злоумышленниками наших дней. Все это стало возможным благодаря реализации в гипервизоре виртуальных уровней доверия VTL (Virtual Trust Level). Так как обычная операционная система и ее компоненты работают в менее привилегированном режиме (VTL 0), а эти технологии VBS работают в режиме VTL 1 (более высокий уровень привилегий), они неуязвимы даже для кода режима ядра. Код как таковой остается в пространстве привилегий VTL 0. Таким образом, уровни VTL можно рассматривать как ортогональные по отношению к уровням привилегий процессора: режим ядра и пользовательский режим существуют *внутри* каждого уровня VTL, но гипервизор управляет привилегиями между VTL. В главе 2 приведена дополнительная информация об архитектуре, использующей супервизор, а в главе 7 эти механизмы безопасности VBS рассматриваются более подробно.

Микропрограммы

Компоненты Windows все сильнее зависят от безопасности операционной системы и ее ядра, а ядро сейчас зависит от защиты гипервизора. Возникает вопрос: что может гарантировать, что эти компоненты были загружены корректно, и что может проверить их содержимое? Обычно этим занимается *загрузчик* (boot loader), но и ему необходим тот же уровень проверки подлинности, что создает еще более сложную иерархию доверия.

Что же должно стать фундаментом для цепочки доверия, гарантирующим нормальный процесс загрузки? В современной Windows 8 и более поздних системах эта роль отводится системной *микропрограмме* (firmware), которая в сертифицированных системах должна базироваться на спецификации UEFI. Как часть стандарта UEFI, предписываемого Windows (UEFI 2.3.1b; см. *http://www.uefi .org*), должна присутствовать безопасная реализация загрузки с сильными гарантиями и требованиями к качеству сертификации программного обеспечения, относящегося к загрузке.

Процесс проверки подлинности гарантирует безопасную загрузку компонентов Windows с самого начала процесса загрузки. Кроме того, такие технологии, как TPM (Trusted Platform Module), могут получать метрики процесса для предоставления аттестации (как локальной, так и удаленной). На основе партнерских отношений в отрасли Microsoft ведет белый и черный списки компонентов безопас-

ной загрузки UEFI на случай ошибок или взлома загрузочного ПО, а обновления Windows также включают обновления микропрограмм.

Хотя тема микропрограммного ПО не будет рассматриваться до главы 11 «Запуск и завершение работы» в части 2, важно подчеркнуть его значимость в современной архитектуре Windows и значимость тех гарантий, которые оно должно предоставлять.

Службы терминалов и сеансы

Под термином «службы терминалов» (Terminal Services) в Windows понимается поддержка множественных интерактивных сеансов пользователя в одной системе. Благодаря службам терминалов Windows удаленный пользователь может создать сеанс на другой машине, выполнить вход и запустить приложения на сервере. Сервер передает графический интерфейс (GUI) клиенту (а также другие настраиваемые ресурсы, такие как буфер обмена и звук), а клиент передает ввод пользователя обратно на сервер. (По аналогии с X Window System, Windows позволяет запускать приложения в серверной системе с передачей клиенту изображения вместо передачи всего рабочего стола.)

Исходный сеанс считается служебным (или нулевым) сеансом и содержит процессы-хосты системных служб (см. главу 9 части 2). Первый сеанс входа на физической консоли машины считается сеансом 1, а дополнительные сеансы создаются программой подключения к удаленному рабочему столу (Mstsc.exe) или через механизм быстрого переключения пользователей.

Клиентские выпуски Windows допускают подключение к машине одного удаленного пользователя, но если кто-то войдет с консоли, рабочая станция блокируется. Иначе говоря, система может использоваться либо в локальном, либо в удаленном режиме, но не одновременно и в том и в другом. Выпуски Windows, включающие Windows Media Center, допускают один интерактивный сеанс и до четырех сеансов Windows Media Center Extender.

Серверные системы Windows поддерживают два одновременных удаленных подключения. Это сделано для удобства удаленного управления, например, использования управляющих средств, которые требуют выполнения входа на управляемой машине. Они также могут поддерживать более двух сеансов при условии соответствующего лицензирования и настройки в качестве сервера терминалов.

Все клиентские выпуски Windows поддерживают множественные сеансы, которые создаются локально с помощью функции быстрого переключения пользователей; в любой момент времени используется только один из этих сеансов. Когда пользователь решает отключить свой сеанс вместо выполнения выхода из системы (например, если он щелкнет на кнопке Пуск (Start), выберет текущего пользователя и выберет в открывшемся подменю команду Сменить учетную запись (Switch Account) или нажмет Win+L и щелкнет на другом пользователе в левом нижнем

углу экрана), текущий сеанс — т. е. процессы, выполняемые в этом сеансе, и все общесистемные структуры данных, описывающие сеанс, — остается активным в системе, а система возвращается к основному экрану входа (если она еще не находится на этом экране). При входе нового пользователя создается новый сеанс.

Для приложений, которые пожелают получить информацию о выполнении в сеансе сервера терминалов, существует набор Windows API для программного обнаружения и управления различными аспектами служб терминалов. (За подробностями обращайтесь к Windows SDK и документации Remote Desktop Services API.)

В главе 2 кратко описан процесс создания сеансов и эксперименты по просмотру информации сеансов в разных средствах, включая отладчик ядра. В разделе «Диспетчер объектов» главы 8 части 2 описано, как системное пространство имен объектов создается на уровне сеанса и как приложениям, которым нужно знать о других своих экземплярах в той же системе, получить эту информацию. Наконец, в главе 5 рассказано о том, как диспетчер памяти создает данные уровня сеанса и управляет ими.

Объекты и дескрипторы

В ОС Windows *объект ядра* представляет собой отдельный экземпляр статически определенного типа объекта, существующий на стадии выполнения. К *типу объекта* относятся тип данных, определенный в системе, функции, работающие с экземплярами типа данных, и набор атрибутов объекта. Если вы пишете приложения Windows, вам могут встретиться объекты процессов, потоков, файлов и событий — и это лишь несколько примеров. Эти объекты базируются на низкоуровневых объектах, которые создаются системой Windows и находятся под ее управлением. В системе Windows *процесс* является экземпляром типа объекта процесса, *файл* — экземпляром типа объекта файла и т. д.

Атрибутом объекта называется поле данных объекта, определяющее часть состояния объекта. Например, объект типа «процесс» содержит атрибуты, определяющие идентификатор процесса, базовый приоритет планирования и указатель на объект маркера доступа. Методы объектов — средства для выполнения операций с объектами — обычно читают или изменяют атрибуты объектов. Так, метод открытия процесса на входе получает идентификатор процесса, а на выходе возвращает указатель на объект.

ПРИМЕЧАНИЕ При создании объекта с использованием API диспетчера объектов ядра вызывающая сторона передает параметр с именем `ObjectAttributes`. Не путайте этот параметр с более общим смыслом атрибута, используемым в книге.

Самое принципиальное различие между объектом и обычной структурой данных — непрозрачность внутренней структуры объекта. Чтобы прочитать данные или записать их в объект, необходимо вызвать соответствующую сервисную функцию. Вы

не можете напрямую прочитать данные или изменить их в объекте. Это различие отделяет реализацию объекта от кода, который просто использует этот объект — этот принцип позволяет легко менять реализации объекта с течением времени.

Объекты при содействии компонента ядра, называемого *диспетчером объектов*, предоставляют удобные средства для решения четырех важных задач ОС.

- Назначение удобочитаемых имен для системных ресурсов.
- Организация совместного использования ресурсов и данных между процессами.
- Защита ресурсов от несанкционированного доступа.
- Отслеживание ссылок, благодаря чему система понимает, когда объект больше не используется (и может быть автоматически удален из памяти).

Не все структуры данных в ОС Windows являются объектами. В объектах размещаются только те данные, к которым необходимо предоставить совместный доступ, которые нужно защитить, которым нужно назначить имя или которые нужно сделать видимыми для программ пользовательского режима (через системные вызовы). Структуры, используемые только одним компонентом ОС для реализации внутренних функций, объектами не являются. Объекты и дескрипторы (ссылки на объекты) более подробно рассматриваются в главе 8 части 2.

Безопасность

Система Windows изначально разрабатывалась с расчетом на безопасность и выполнение требований различных формальных правительственных и отраслевых спецификаций безопасности, таких как спецификация CCITSE (Common Criteria for Information Technology). Получение рейтинга безопасности, одобренного правительством, дает возможность ОС конкурировать в этой области. Конечно, многие из реализованных возможностей полезны в любой многопользовательской системе.

В состав базовых средств безопасности Windows входят:

- защита на уровне пользователей (дискреционная, т. е. предоставляемая на усмотрение) и обязательная защита для всех совместно используемых системных объектов: файлов, каталогов, процессов, потоков и т. д.;
- система аудита безопасности для учета субъектов (пользователей) и выполняемых ими действий;
- проверка пользователя при входе;
- предотвращение доступа со стороны пользователя к неинициализированным ресурсам (например, свободной памяти или дисковому пространству), освобожденным другим пользователем.

В Windows существует три формы управления доступом к объектам.

- *Управление доступом на уровне пользователей* (дискреционное управление доступом). Именно этот механизм защиты большинство людей представляют себе, когда речь заходит о безопасности ОС. С помощью этого механизма владельцы объектов (например, файлов или принтеров) предоставляют или блокируют доступ другим сторонам. При входе пользователю предоставляется набор учетных данных безопасности, или *контекст безопасности*. При попытке обращения к объекту контекст безопасности пользователя сравнивается со списком управления доступом того объекта, к которому он обращается. Результат сравнения определяет, разрешается ли выполнение запрашиваемой операции. В Windows Server 2012 и Windows 8 эта форма управления доступом усовершенствована реализацией управления доступом на базе атрибутов (DAC, Dynamic Access Control). Однако в списках управления доступом ресурсов не обязательно перечисляются конкретные пользователи и группы. Вместо этого в них указываются необходимые атрибуты, или *утверждения* (claims), для предоставления доступа к ресурсу: например, «Уровень допуска: Совершенно секретно» или «Стаж: 10 лет». Благодаря возможности автоматического заполнения таких атрибутов посредством разбора баз данных и схем SQL через Active Directory, эта существенно более элегантная и гибкая модель безопасности помогает организациям избежать неудобного ручного управления группами и иерархиями групп.

- *Привилегированное управление доступом.* Этот механизм необходим, когда механизма управления доступом на уровне пользователей недостаточно. Он гарантирует, что к защищенным объектам можно будет получить доступ даже в том случае, если их владелец недоступен. Например, если работник уволился из компании, администратор должен как-то получить доступ к файлам, которые были доступны только для этого работника. В таком случае в системе Windows администратор может назначить себя владельцем файла, чтобы управлять правами так, как потребуется.

- *Контроль целостности учетных данных (MIC).* Этот механизм используется, когда необходим дополнительный уровень контроля безопасности для защиты объектов, обращение к которым производится из той же учетной записи пользователя. Он используется для разных целей, от технологии частичной изоляции для приложений Windows Apps (см. далее) и изоляции защищенного режима Internet Explorer (и других браузеров) от конфигурации пользователя, до защиты объектов, созданных учетной записью администратора с повышенными правами, от обращений со стороны учетной записи администратора без повышенных прав. (За дополнительной информацией о механизме контроля учетных записей обращайтесь к главе 7.)

Начиная с Windows 8, для размещения приложений Windows Apps используется изолированная среда («песочница»), которая называется контейнером приложения (AppContainer). Она предоставляет изоляцию от других контейнеров приложений и процессов, не относящихся к приложениям Windows Apps. Код контейнеров приложений может взаимодействовать с *брокерами* (неизолированными процессами,

работающими с учетными данными пользователя), а иногда с другими контейнерами приложений или процессами через четко определенные контракты, предоставляемые Windows Runtime. Классический пример — браузер Microsoft Edge, который выполняется в контейнере приложения и таким образом предоставляет улучшенную защиту от выполняемого внутри него вредоносного кода. Кроме того, сторонние разработчики могут использовать контейнеры приложений для аналогичной изоляции собственных приложений, не являющихся приложениями Windows Apps. Модель контейнера приложения вынуждает к значительному сдвигу традиционных парадигм программирования и переходу от традиционной реализации в форме многопоточного однопроцессного приложения к многопроцессной реализации.

Безопасность задействована во всех аспектах Windows API. Подсистема Windows реализует объектную безопасность по тому же принципу, что и ОС: общие объекты Windows защищаются от несанкционированного доступа за счет назначения им дескрипторов безопасности Windows. Когда приложение в первый раз пытается обратиться к общему объекту, подсистема Windows проверяет, обладает ли оно необходимыми правами. Если проверка безопасности проходит успешно, то подсистема Windows разрешает приложению продолжение операции.

Система безопасности Windows подробно рассматривается в главе 7.

Реестр

Если вы работаете с операционными системами Windows, вероятно, вы уже слышали о системном реестре или заглядывали в него. Трудно говорить о внутреннем устройстве Windows без упоминания реестра — системной базы данных с информацией, необходимой для загрузки и настройки конфигурации системы, общесистемными параметрами, управляющими работой Windows, базой данных безопасности и текущими настройками (например, используемой экранной заставкой). Кроме того, реестр открывает доступ к хранящимся в памяти временным данным, таким как текущее состояние оборудования системы (какие драйверы устройств загружены, какие ресурсы они используют и т. д.), а также счетчикам производительности Windows. Для обращения к счетчикам производительности, которые на самом деле в реестре не хранятся, используются функции реестра (хотя существует новый улучшенный API для обращения к счетчикам безопасности). За дополнительной информацией об обращениях к данным счетчиков производительности обращайтесь к главе 9 части 2.

Многим пользователям и администраторам Windows никогда не приходится обращаться к реестру напрямую (потому что для просмотра и изменения многих параметров конфигурации можно использовать стандартные административные программы). Реестр остается полезным источником внутренней информации Windows, потому что в нем содержатся многие настройки, влияющие на быстродействие и поведение системы. В книге часто упоминаются разные разделы реестра, относящиеся к рассматриваемому компоненту. Большинство разделов реестра, упоминаемых в книге, относится к кусту общесистемной конфигурации HKEY_LOCAL_MACHINE, который в дальнейшем будет обозначаться сокращением *HKLM*.

ВНИМАНИЕ Если вы решите напрямую изменить содержимое реестра, будьте предельно осторожны. Любые изменения могут отрицательно сказаться на производительности системы, или, что еще хуже, помешать успешной загрузке системы.

За дополнительной информацией о реестре и его внутренней структуре обращайтесь к главе 9 части 2.

Юникод

Windows отличается от большинства других операционных систем тем, что многие внутренние текстовые строки хранятся и обрабатываются в виде 16-разрядных символов Юникода (формально UTF-16LE; там, где в книге упоминается Юникод, имеется в виду UTF-16LE, если только в тексте прямо не указано иное). Юникод — стандарт международной кодировки символов, в котором определяются уникальные значения для большинства известных мировых кодировок и поддерживаются 8-, 16- и даже 32-разрядные кодировки для разных символов.

Так как многие приложения работают со строками 8-разрядных (однобайтовых) символов ANSI, многие функции Windows, получающие строковые параметры, имеют две точки входа: для Юникода (широкая, 16-разрядная) и для ANSI (узкая, 8-разрядная). Вызов узкой версии функции Windows сопровождается небольшой потерей производительности, так как входные строковые параметры преобразуются в Юникод перед обработкой системой, а выходные параметры преобразуются из Юникода в ANSI перед возвращением приложению. Таким образом, если у вас имеется старая служба или фрагмент кода, который должен выполняться в Windows, но этот код написан с использованием текстовых строк ANSI, Windows преобразует символы ANSI в Юникод для своего использования. Однако Windows никогда не преобразует данные, хранящиеся в файлах, — приложение само решает, где должны храниться данные храниться, в Юникоде или в ANSI.

Независимо от языка, все версии Windows содержат одни и те же функции. Вместо отдельных версий для разных языков Windows содержит единый двоичный код, чтобы одна установка могла поддерживать много языков (посредством добавления разных языковых пакетов). Приложения также могут пользоваться функциями Windows, что позволяет включать в приложение универсальные двоичные файлы, способные поддерживать разные языки.

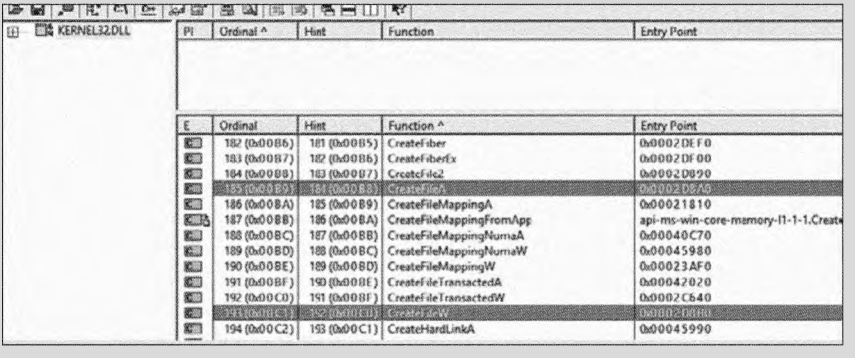
ПРИМЕЧАНИЕ В старых операционных системах 9x не было встроенной поддержки Юникода. Это стало еще одной причиной для создания двух функций (для ASNI и Юникода). Например, функция Windows API `CreateFile` функцией вообще не является; это макрос, который расширяется в одну из двух функций: `CreateFileA` (ANSI) или `CreateFileW` (для Юникода; W означает «wide», т. е. «широкий»). Расширение выполняется на основании константы компиляции с именем UNICODE. Эта константа определяется по умолчанию в проектах Visual Studio C++, потому что работа с функциями Юникода предпочтительна. Впрочем, вместо макроса можно использовать низкоуровневое имя функции.

Следующий эксперимент демонстрирует эти пары функций.

ЭКСПЕРИМЕНТ: ПРОСМОТР ЭКСПОРТИРУЕМЫХ ФУНКЦИЙ

В этом эксперименте мы воспользуемся программой Dependency Walker для просмотра функций, экспортируемых DLL-библиотекой подсистемы Windows.

1. Загрузите Dependency Walker по адресу *http://www.dependencywalker.com*. Если вы работаете в 32-разрядной системе, загрузите 32-разрядную версию Download Dependency. Для 64-разрядной системы загружается 64-разрядная версия. Распакуйте загруженный ZIP-файл в папку по своему выбору.
2. Запустите программу (depends.exe). Откройте меню File, выберите команду Open, перейдите в папку C:\Windows\System32 (предполагается, что система Windows установлена на диске C), найдите файл kernel32.dll и щелкните на кнопке Open.
3. Dependency Walker может выдать диалоговое окно с предупреждением. Не обращайте на него внимания и закройте окно.
4. Вы увидите несколько панелей с вертикальными и горизонтальными линиями разделения. Убедитесь в том, что в дереве вверху слева выбран файл kernel32.dll.
5. Взгляните на вторую панель вверху на правой стороне. На ней выводится список функций, экспортируемых из файла kernel32.dll. Щелкните на заголовке списка Function, чтобы отсортировать его по имени. Найдите функцию `CreateFileA`. Чуть ниже в списке находится и функция `CreateFileW`.



6. Как видите, многие функции, получающие хотя бы один строковый аргумент, в действительности существуют в двух версиях. На иллюстрации видны следующие функции: `CreateFileMappingA/W`, `CreateFileTransactedA/W` и `CreateFileMappingNumaA/W`.
7. Прокрутите список и найдите в нем другие функции. Вы также можете открыть другие системные файлы, такие как user32.dll и advapi32.dll.

Дополнительная информация о Юникоде доступна по адресу *http://www.unicode.org* и в документации по программированию из библиотеки MSDN.

ПРИМЕЧАНИЕ В API на базе COM в системе Windows обычно используются строки Юникода, иногда обозначаемые типом BSTR. По сути это массив символов Юникода, завершенный нуль-символом; за 4 байта до начала массива символов в памяти хранится длина строки в байтах. В Windows Runtime API используются только строки Юникода, обозначаемые типом HSTRING — неизменяемым массивом символов Юникода.

Изучение внутреннего устройства Windows

Хотя большая часть информации в книге основана на чтении исходного кода Windows и беседах с разработчиками, вы вовсе не обязаны принимать все на веру. Многие нюансы, относящиеся к внутреннему устройству Windows, можно обнаружить и продемонстрировать при помощи различных служебных программ: например, включенных в поставку Windows и средства отладки Windows. Эти инструментальные пакеты кратко описаны далее в этом разделе.

Чтобы разжечь вас интерес к исследованию внутреннего устройства Windows, мы включили в книгу врезки «Эксперимент» с описанием конкретных аспектов внутреннего поведения Windows. (Несколько таких врезок уже встречалось вам ранее.) Мы рекомендуем самостоятельно повторить эксперименты, чтобы вы могли понаблюдать в действии многие аспекты, описанные в книге.

В табл. 1.4 перечислены основные средства, представленные в книге, с информацией об их происхождении.

Таблица 1.4. Средства для просмотра информации о внутреннем устройстве Windows

Средство	Имя образа	Происхождение
Startup Programs Viewer	AUTORUNS	Sysinternals
Access Check	ACCESSCHK	Sysinternals
Dependency Walker	DEPENDS	www.dependencywalker.com
Global Flags	GFLAGS	Debugging Tools
Handle Viewer	HANDLE	Sysinternals
Отладчики ядра	WINDBG, KD	WDK, Windows SDK
Object Viewer	WINOBJ	Sysinternals
Системный монитор	PERFMON.MSC	Встроенная программа Windows
Pool Monitor	POOLMON	WDK
Process Explorer	PROCEXP	Sysinternals
Process Monitor	PROCMON	Systinternals
Task (Process) List	TLIST	Средства отладки
Диспетчер задач	TASKMGR	Встроенная программа Windows

Системный монитор и Монитор ресурсов

В этой книге мы часто возвращаемся к Системному монитору (который запускается из папки Администрирование (Administrative Tools) панели управления или командой perfmon в диалоговом окне запуска программы). А если говорить конкретнее, мы сосредоточимся на Системном мониторе и Мониторе ресурсов.

ПРИМЕЧАНИЕ Системный монитор решает три задачи: сбор информации о системе, просмотр журналов счетчиков производительности и установка сигналов (с использованием наборов сборщиков данных, которые также содержат журналы счетчиков производительности, данные трассировки и конфигурации). Для простоты при упоминании Системного монитора мы будем иметь в виду функцию сбора информации о системе.

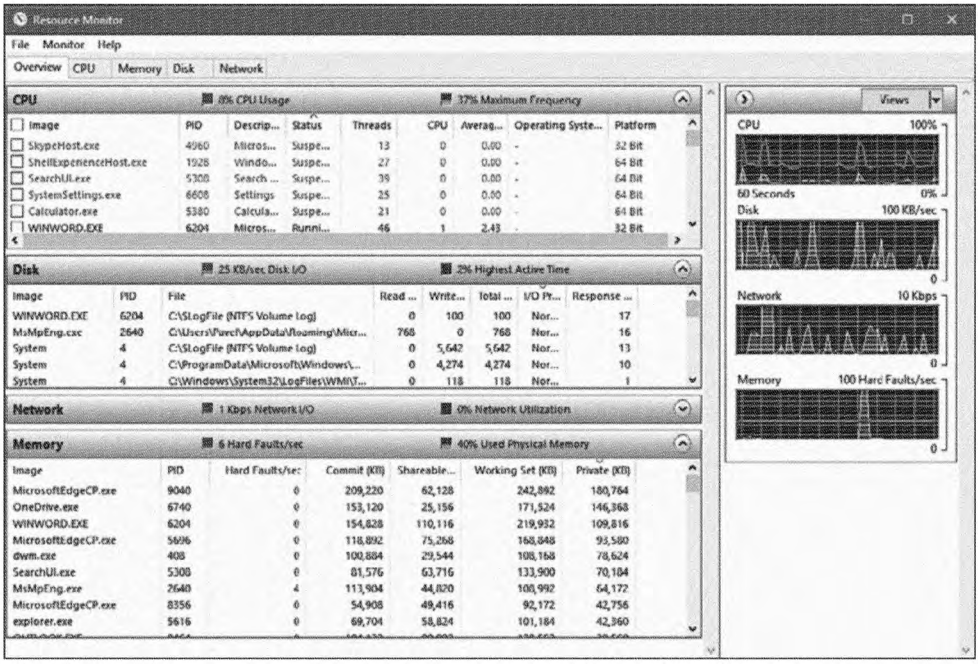
Системный монитор предоставляет больше информации о том, как работает ваша система, чем любая другая программа. Он включает сотни базовых и расширенных счетчиков для разных объектов. Для каждой основной темы, описанной в книге, приводится таблица соответствующих счетчиков производительности. Счетчик производительности содержит краткие описания всех счетчиков. Чтобы просмотреть описания, выберите счетчик в окне Добавить счетчики (Add Counters) и установите флажок Отображать описание (Show Description).

И хотя всю низкоуровневую информацию о системе можно получить в Системном мониторе, Windows также включает программу Монитор ресурсов (запускается из меню Пуск (Start) или с вкладки Быстродействие (Performance) диспетчера задач), отображающую состояние четырех основных системных ресурсов: процессора, диска, сети и памяти. В базовом состоянии эти ресурсы отображаются с тем же уровнем информации, что и в диспетчере задач. Впрочем, они также содержат разделы, которые можно развернуть для получения дополнительной информации. Типичное окно Монитора ресурсов показано на с. 64.

В развернутом виде на вкладке ЦП (CPU) выводится информация об использовании процессора на уровне процессов, как и в диспетчере задач. К ней добавляется столбец для средней загрузки процессора, который может дать лучшее представление о наиболее активных процессах в системе. Вкладка ЦП (CPU) также выводит информацию о службах, об использовании ими процессора и средней загрузки. Каждый процесс, являющийся хостом служб, идентифицируется группой служб, которыми он управляет.

Как и в Process Explorer, при выборе процесса (щелчком на соответствующем флажке) выводится список дескрипторов, открытых процессом, а также список модулей (например, DLL), загруженных в адресное пространство процесса. Поле Найти дескрипторы (Search Handles) также может использоваться для поиска процессов, открывших дескриптор для заданного именованного ресурса.

На вкладке Память (Memory) отображается практически та же информация, которая может быть получена с помощью диспетчера задач, но упорядоченная для всей системы. На графике физической памяти выводится текущее распределение



физической памяти между зарезервированной для оборудования, используемой, измененной, резервной и свободной памятью. Смысл этих терминов объясняется в главе 5.

С другой стороны, на вкладке Диск (Disk) выводится информация ввода/вывода уровня файлов, которая позволяет легко определять, к каким файлам в системе происходит больше всего обращений, чаще всего выполняется запись или чтение. Результаты можно дополнительно отфильтровать по процессам.

На вкладке Сеть (Network) выводятся активные сетевые подключения, процессы, которым они принадлежат, и объем данных, проходящих через них. Эти данные позволяют обнаружить фоновую сетевую активность, которую трудно обнаружить другими способами. Кроме того, на вкладке отображаются подключения TCP, активные в системе, упорядоченные по процессам с такими данными, как удаленный и локальный порт, адрес и время задержки пакетов. Также выводится список прослушиваемых портов для процессов, по которому администратор может определить, какие службы или приложения в настоящее время ожидают подключений через конкретный порт. Для каждого порта указывается протокол и политика брандмауэра.

ПРИМЕЧАНИЕ Ко всем счетчикам производительности Windows можно обращаться на программном уровне. Для получения дополнительной информации проведите поиск по условию «счетчики производительности» в документации MSDN.

Отладка ядра

Под «отладкой ядра» понимается анализ внутренних структур данных ядра и/или пошаговое выполнение функций в ядре. Отладка ядра — полезное средство анализа внутреннего устройства Windows, потому что вы сможете просмотреть внутреннюю системную информацию, которую не удастся получить при помощи других средств, и составить более четкое представление о коде, выполняемом в ядре.

Прежде чем описывать различные способы отладки ядра, рассмотрим некоторые файлы, которые понадобятся вам для выполнения любой разновидности отладки ядра.

Символическая информация для отладки ядра

Символические файлы содержат имена функций и переменных, а также описания структуры и формата структур данных. Они генерируются компоновщиком и используются отладчиками для определения и вывода этих имен во время сеанса отладки. Эта информация обычно не включается в двоичный образ, потому что она не нужна для выполнения кода; двоичные файлы становятся более компактными и быстрее выполняются. С другой стороны, это означает, что при отладке необходимо предоставить отладчику доступ к файлам символических имен для образов, задействованных в сеансе отладки.

Чтобы использовать любые средства отладки ядра для анализа внутренних структур данных ядра Windows — таких, как списки процессов, блоки потоков, списки загруженных драйверов, информация об использовании памяти и т. д. — необходимо иметь нужные символические файлы хотя бы для образа ядра Ntoskrnl.exe. (Этот файл более подробно описан в разделе «Обзор архитектуры» главы 2.) Файлы символических таблиц должны соответствовать версии того образа, на основании которого они были построены. Например, при установке обновления или исправления Windows с модификацией ядра вы должны найти для него соответствующие символические файлы.

И хотя вы можете загрузить и установить символическую информацию для разных версий Windows, обновленные символические файлы для исправлений доступны не всегда. Самый простой способ получить правильную версию символических данных для отладки — воспользоваться сервером символической информации Microsoft с использованием специального синтаксиса пути к символическим данным, который задается в отладчике. Например, следующий путь заставляет средства отладки загрузить необходимую символическую информацию с сервера в интернете и сохранить локальную копию в папке C:\symbols:

```
srv*c:\symbols*http://msdl.microsoft.com/download/symbols
```

Средства отладки для Windows

Пакет средств отладки (Debugging Tools) для Windows содержит расширенные средства отладки, использованные в книге для анализа внутренней структуры Windows. Новейшая версия включается в Windows SDK. (За дополнительной информацией о различных типах установки обращайтесь по адресу *https://msdn.microsoft.com/en-us/library/windows/hardware/ff551063.aspx*.) Эти средства могут использоваться как для отладки процессов пользовательского режима, так и для отладки процессов ядра.

В пакет включены четыре отладчика: cdb, ntsd, kd и WinDbg. Все они основаны на одном отладочном механизме, реализованном в библиотеке DbgEng.dll; он достаточно хорошо документирован в справочном файле. Краткий обзор отладчиков:

- cdb и ntsd — отладчики пользовательского режима с консольным интерфейсом. Они различаются только одним: при запуске из существующего консольного окна ntsd открывает новое консольное окно, а cdb этого не делает.
- kd — отладчик режима ядра с консольным интерфейсом.
- WinDbg может использоваться как отладчик либо пользовательского режима, либо режима ядра, но не одновременно. Отладчик предоставляет графический интерфейс.

Отладчики пользовательского режима (cdb, ntsd и WinDbg, когда используется в этом режиме) практически эквивалентны. Выбор того или иного варианта зависит от личных предпочтений.

Отладчики режима ядра (kd и WinDbg, когда используется в этом режиме) тоже эквивалентны.

Отладка пользовательского режима. Средства отладки также могут присоединяться к процессам пользовательского режима для анализа и/или изменения содержимого памяти процессов. При подключении к процессу возможны два варианта:

- **Агрессивный** (Invasive). Если явно не указано обратное, при подключении к работающему процессу функция Windows `DebugActiveProcess` используется для установления связи между отладчиком и отлаживаемым процессом. Это позволяет вам анализировать и/или изменять память процесса, расставлять точки останова и выполнять другие отладочные операции. Windows позволяет прервать отладку без уничтожения целевого процесса при условии, что отладчик отсоединяется (а не уничтожается).
- **Неагрессивный.** В этом варианте отладчик просто открывает процесс функцией `OpenProcess`. Он не присоединяется к процессу как отладчик. Вы сможете анализировать и/или изменять память целевого процесса, но не назначать точки останова. Это также означает, что неагрессивное подключение возможно, даже если другой отладчик уже присоединился агрессивно.

Также средствами отладки можно открывать файлы дампа процессов пользовательского режима. Файлы дампа рассматриваются в главе 8 части 2, в разделе, посвященном диспетчеризации исключений.

Отладка режима ядра. Как упоминалось ранее, для отладки ядра могут использоваться два отладчика: с интерфейсом командной строки (Kd.exe) и версия с графическим интерфейсом (Windbg.exe). С этими средствами можно выполнять три разновидности отладки режима ядра:

- Открыть файл аварийного дампа, созданный в результате сбоя системы Windows. (За информацией об аварийных дампах ядра обращайтесь к главе 15 «Анализ аварийного дампа», часть 2.)
- Подключение к «живой», работающей системе и анализ состояния системы (или назначение точек останова, если вы занимаетесь отладкой кода драйверов устройств). Для этой операции нужны два компьютера: целевой (отлаживаемая система) и хост (система, в которой работает отладчик). Целевая система также может быть подключена к хосту нуль-модемным кабелем, кабелем IEEE 1394, отладочным кабелем USB 2.0/3.0 или по локальной сети. Целевая система должна загружаться в отладочном режиме. Систему можно настроить для загрузки в отладочном режиме при помощи программы Bcdedit.exe или Msconfig.exe. (Возможно, для этого вам придется отключить безопасную загрузку в настройках BIOS UEFI.) Также возможно подключение через именованный канал — это имеет смысл при отладке Windows 7 или более ранних версий через виртуальную машину, такую как Hyper-V, Virtual Box или VMWare Workstation, предоставлением доступа к последовательному порту гостевой операционной системы как к устройству именованного канала. Для Windows 8 и более поздних гостевых версий вам придется использовать отладку локальной сети, предоставляя сеть, состоящую только из хоста, с использованием виртуальной сетевой платы в гостевой операционной системе. Это приводит к 1000-кратному приросту быстродействия.
- Системы Windows также позволяют подключаться к локальной системе и анализировать ее состояние (это называется *локальной отладкой ядра*). Чтобы инициировать локальную отладку ядра в WinDbg, сначала убедитесь в том, что система переведена в отладочный режим (например, запустите msconfig.exe, перейдите на вкладку Загрузка (Boot), щелкните на кнопке Дополнительные параметры (Advanced Options), выберите вариант Отладка (Debug) и перезапустите Windows). Запустите WinDbg с привилегиями администратора, откройте меню File, выберите команду Kernel Debug, перейдите на вкладку Local и щелкните на кнопке OK (или используйте bcdedit.exe). На рис. 1.6 показан пример вывода для 64-разрядной машины с Windows 10. Некоторые команды отладчика ядра не работают в режиме локальной отладки ядра (например, назначение точек останова или создание дампа памяти командой `.dump`). Впрочем, последнее можно сделать с помощью программы LiveKd, описанной далее в этом разделе.

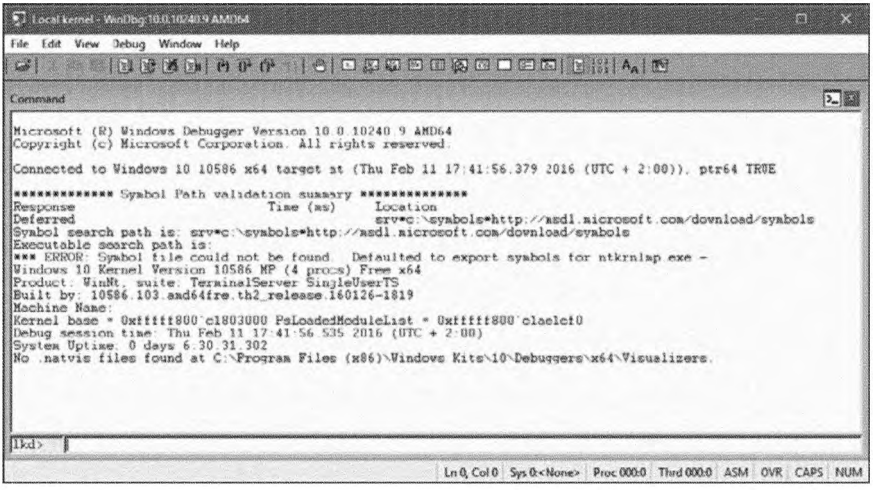


Рис. 1.6. Локальная отладка ядра

После подключения в режиме отладки ядра вы сможете воспользоваться многочисленными командами расширения отладчика (эти команды начинаются с восклицательного знака (!)) для просмотра содержимого внутренних структур данных: потоков, процессов, пакетов запросов ввода/вывода и информации управления памятью. В этой книге при обсуждении каждой темы приводятся связанные с этой темой команды отладчика ядра и примеры вывода. Отличным источником информации служит справочный файл Debugger.chm из папки установки WinDbg: он документирует всю функциональность и расширения отладчика ядра. Кроме того, команда `dt` (Display Type) может вывести в отформатированном виде более 1000 структур ядра, потому что файлы с символической информацией ядра для Windows содержат данные, которые могут использоваться отладчиком для форматирования структур.

ЭКСПЕРИМЕНТ: ВЫВОД ИНФОРМАЦИИ ТИПА ДЛЯ СТРУКТУР ДАННЫХ ЯДРА

Чтобы вывести список структур ядра, данные типа которых содержатся в символической информации ядра, введите команду `dt nt!_*` в отладчике ядра. Ниже приведен пример части вывода (ntkrnlmp — внутреннее имя файла 64-разрядного ядра. За подробностями обращайтесь к главе 2).

```
lkd> dt nt!_*
         ntkrnlmp!_KSYSTEM_TIME
         ntkrnlmp!_NT_PRODUCT_TYPE
         ntkrnlmp!_ALTERNATIVE_ARCHITECTURE_TYPE
         ntkrnlmp!_KUSER_SHARED_DATA
         ntkrnlmp!_ULARGE_INTEGER
         ntkrnlmp!_TP_POOL
         ntkrnlmp!_TP_CLEANUP_GROUP
         ntkrnlmp!_ACTIVATION_CONTEXT
```

```
        ntkrnlmp!_TP_CALLBACK_INSTANCE
        ntkrnlmp!_TP_CALLBACK_PRIORITY
        ntkrnlmp!_TP_CALLBACK_ENVIRON_V3
        ntkrnlmp!_TEB
```

Команда `dt` также может использоваться с универсальными символами для поиска структур. Например, если вы ищете имя структуры для объекта прерывания, введите команду `dt nt!_*interrupt*`:

```
lkd> dt nt!_*interrupt*
        ntkrnlmp!_KINTERRUPT_MODE
        ntkrnlmp!_KINTERRUPT_POLARITY
        ntkrnlmp!_PEP_ACPI_INTERRUPT_RESOURCE
        ntkrnlmp!_KINTERRUPT
        ntkrnlmp!_UNEXPECTED_INTERRUPT
        ntkrnlmp!_INTERRUPT_CONNECTION_DATA
        ntkrnlmp!_INTERRUPT_VECTOR_DATA
        ntkrnlmp!_INTERRUPT_HT_INTR_INFO
        ntkrnlmp!_INTERRUPT_REMAPPING_INFO
```

Далее найденная структура форматируется командой `dt` (отладчик игнорирует регистр символов):

```
lkd> dt nt!_KINTERRUPT
   +0x000 Type             : Int2B
   +0x002 Size             : Int2B
   +0x008 InterruptListEntry : _LIST_ENTRY
   +0x018 ServiceRoutine   : Ptr64     unsigned char
   +0x020 MessageServiceRoutine : Ptr64    unsigned char
   +0x028 MessageIndex     : Uint4B
   +0x030 ServiceContext   : Ptr64 Void
   +0x038 SpinLock         : Uint8B
   +0x040 TickCount        : Uint4B
   +0x048 ActualLock       : Ptr64 Uint8B
   +0x050 DispatchAddress  : Ptr64     void
   +0x058 Vector           : Uint4B
   +0x05c Irql             : UChar
   +0x05d SynchronizeIrql  : UChar
   +0x05e FloatingSave     : UChar
   +0x05f Connected        : UChar
   +0x060 Number           : Uint4B
   +0x064 ShareVector      : UChar
   +0x065 EmulateActiveBoth : UChar
   +0x066 ActiveCount      : Uint2B
   +0x068 InternalState    : Int4B
   +0x06c Mode             : _KINTERRUPT_MODE
   +0x070 Polarity         : _KINTERRUPT_POLARITY
   +0x074 ServiceCount     : Uint4B
   +0x078 DispatchCount    : Uint4B
   +0x080 PassiveEvent     : Ptr64 _KEVENT
   +0x088 TrapFrame        : Ptr64 _KTRAP_FRAME
   +0x090 DisconnectData   : Ptr64 Void
   +0x098 ServiceThread    : Ptr64 _KTHREAD
   +0x0a0 ConnectionData   : Ptr64 _INTERRUPT_CONNECTION_DATA
   +0x0a8 IntTrackEntry    : Ptr64 Void
```

```
   +0x0b0 IsrDpcStats      : _ISRDPCSTATS
   +0x0f0 RedirectObject   : Ptr64 Void
   +0x0f8 Padding          : [8] UChar
```

Обратите внимание: команда `dt` по умолчанию не выводит субструктуры (структуры внутри структур). Для вывода субструктур используется ключ `-r` или `-b`. Например, при использовании одного из этих ключей объект прерывания ядра выводит формат структуры `_LIST_ENTRY`, хранящейся в поле `InterruptListEntry`. (Различия между ключами `-r` и `-b` описаны в документации.)

```
lkd> dt nt!_KINTERRUPT -r
   +0x000 Type           : Int2B
   +0x002 Size           : Int2B
   +0x008 InterruptListEntry : _LIST_ENTRY
      +0x000 Flink       : Ptr64 _LIST_ENTRY
          +0x000 Flink       : Ptr64 _LIST_ENTRY
          +0x008 Blink       : Ptr64 _LIST_ENTRY
      +0x008 Blink       : Ptr64 _LIST_ENTRY
          +0x000 Flink       : Ptr64 _LIST_ENTRY
          +0x008 Blink       : Ptr64 _LIST_ENTRY
   +0x018 ServiceRoutine   : Ptr64     unsigned char
```

Команда `dt` даже позволяет задать уровень рекурсии структур; нужное число указывается за ключом `-r`. В следующем примере выбирается один уровень рекурсии:

```
lkd> dt nt!_KINTERRUPT -r1
```

В справочном файле средств отладки для Windows объясняется, как настроить и использовать отладчики ядра. За дополнительной информацией об использовании отладчиков ядра, предназначенной прежде всего для разработчиков драйверов устройств, обращайтесь к документации WDK.

Программа LiveKd

LiveKd — бесплатная программа из пакета Sysinternals, которая позволяет использовать только что описанные стандартные отладчики ядра компании Microsoft без загрузки системы в отладочном режиме. Такое решение может использоваться в ситуации, когда требуется провести диагностику уровня ядра на машине, не загруженной в отладочном режиме. Некоторые проблемы трудно надежно воспроизвести, поэтому при перезагрузке в отладочном режиме проблема может и не проявиться.

LiveKd запускается так же, как WinDbg или kd. LiveKd передает все заданные вами параметры командной строки выбранному вами отладчику. По умолчанию LiveKd запускает отладчик ядра с интерфейсом командной строки (kd). Чтобы он запускал WinDbg, используйте ключ `-w`. Для просмотра описаний ключей LiveKd используйте ключ `-?`.

LiveKd передает отладчику имитацию файла аварийного дампа, чтобы в LiveKd можно было выполнять операции, поддерживаемые для аварийных дампов. Так как LiveKd моделирует дамп на базе физической памяти, отладчик ядра может столкнуться с ситуацией, при которой структуры данных находятся в процессе изменения системой, а их логическая целостность нарушена. Каждый раз при запуске отладчик начинает со свежего представления состояния системы. Если вы захотите обновить этот «снимок состояния», введите команду q для выхода из отладчика. LiveKd спросит, хотите ли вы запустить его снова.

Если отладчик зацикливается при выводе, нажмите Ctrl+C, чтобы прервать вывод и завершить работу. Если отладчик зависнет, нажмите Ctrl+Break — эта комбинация завершает процесс отладки. В этом случае LiveKd предложит перезапустить отладчик.

Windows Software Development Kit

Пакет Windows SDK (Software Development Kit) доступен в составе подписки MSDN. Его можно бесплатно загрузить по адресу *https://developer.microsoft.com/en-US/windows/downloads/windows-10-sdk*. Visual Studio также предоставляет возможность установить SDK как часть установки VS. Версии, содержащиеся в Windows SDK, всегда соответствуют новейшей версии операционной системы Windows, тогда как в составе Visual Studio может поставляться более старая версия, актуальная на момент выпуска. Кроме средств отладки Windows она содержит заголовочные файлы C и библиотеки, необходимые для компиляции и компоновки приложений Windows. С позиций внутреннего устройства Windows в Windows SDK заслуживают внимания заголовочные файлы Windows API, например C:\Program Files (x86)\Windows Kits\10\Include, и инструментальные средства SDK (поищите папку Bin). Также стоит обратить внимание на документацию: вы можете работать с ней в интернете или загрузить для автономной работы. Некоторые средства также поставляются в виде примеров исходного кода для Windows SDK и MSDN Library.

Windows Driver Kit

Пакет WDK (Windows Driver Kit) также доступен в программе подписки MSDN. Как и Windows SDK, его можно загрузить бесплатно. Документация WDK включена в MSDN Library.

Хотя пакет WDK предназначен для разработчиков драйверов устройств, это богатый источник информации о внутреннем устройстве Windows. Например, хотя в главе 6 описывается архитектура системы ввода/вывода, модель драйверов и базовые структуры данных драйверов устройств, в ней отсутствует подробное описание отдельных функций поддержки ядра. Документация WDK содержит подробное описание всех функций поддержки ядра Windows и механизмов, используемых драйверами устройств, как в форме учебника, так и в форме справочника.

Кроме документации WDK содержит заголовочные файлы (прежде всего ntddk.h, ntifs.h и wdm.h) с определениями ключевых внутренних структур данных и констант, а также интерфейсов ко многим внутренним системным функциям. Эти файлы помогают в исследовании внутренних структур данных Windows с отладчиком ядра; хотя в книге приводится общее строение и содержимое этих структур, подробных описаний уровня полей (размеры и типы данных) мы не приводим. Впрочем, некоторые структуры данных — заголовки диспетчеризации объектов, блоки ожидания, события, мутанты, семафоры и т. д. — полностью описаны в WDK.

Если вы захотите изучить систему ввода/вывода и модель драйверов более подробно, читайте документацию WDK — особенно руководства *Kernel-Mode Driver Architecture Design Guide* и *Kernel-Mode Driver Reference*. Также вам могут пригодиться книги *Programming the Microsoft Windows Driver Model, Second Edition* Уолтера Они (Walter Oney) (Microsoft Press, 2002) и *Developing Drivers with the Windows Driver Foundation* Пенни Орвик (Penny Orwick) и Гая Смита (Guy Smith) (Microsoft Press, 2007).

Средства Sysinternals

Во многих экспериментах в книге используются бесплатные программы, которые можно загрузить на сайте Sysinternals. Большинство из них написал Марк Руссинович, соавтор этой книги. Самые популярные программы пакета — Process Explorer и Process Monitor. Учтите, что многие из этих программ требуют установки и выполнения драйверов режима ядра, а для этого необходимы привилегии администратора, хотя некоторые могут запускаться с ограниченной функциональностью и выводом из стандартной (непривилегированной) учетной записи пользователя.

Так как средства Sysinternals часто обновляются, следите за тем, чтобы у вас была установлена новейшая версия. Чтобы получать оповещения об обновлениях инструментария, следите за блогом на сайте Sysinternals (у которого также имеется новостной канал RSS). Описания всех программ, информация об их использовании и практические примеры решения проблем приведены в книге *Windows Sysinternals Administrator's Reference* Марка Руссиновича (Mark Russinovich) и Аарона Маргосиса (Aaron Margosis) (Microsoft Press, 2011). С вопросами и обсуждениями обращайтесь на форумы Sysinternals.

Заключение

В этой главе были представлены ключевые технические концепции и термины Windows, которые будут использоваться в книге. Также в ней дан обзор многих полезных средств, предназначенных для анализа внутреннего устройства Windows.

Итак, к началу исследования внутреннего устройства системы все готово. Начнем с общего обзора архитектуры системы и ее ключевых компонентов.

Глава 2
Архитектура системы

После знакомства с важнейшими терминами, концепциями и средствами можно переходить к исследованию целей проектирования и внутренней структуры операционной системы (ОС) Microsoft Windows. В этой главе рассматривается общая структура систем: основные компоненты, их взаимодействия и контекст их выполнения. Чтобы заложить основу для понимания внутреннего устройства Windows, начнем с обзора требований и целей, определивших исходную архитектуру и спецификацию системы.

Требования и цели проектирования

Разработка спецификации Windows NT в 1989 году определялась следующими требованиями:

- Реализация полноценной 32-разрядной ОС с вытесняющей многозадачностью, реентерабельностью и поддержкой виртуальной памяти.
- Выполнение на разных аппаратных архитектурах и платформах.
- Выполнение и хорошее масштабирование в системах с симметричной многопроцессорной обработкой.
- Эффективная платформа для распределенных вычислений (как в роли сетевого клиента, так и в роли сервера).
- Выполнение большинства существующих 16-разрядных приложений MS-DOS и Microsoft Windows 3.1.
- Выполнение правительственных требований к соответствию спецификации POSIX 1003.1.
- Выполнение правительственных и отраслевых требований к безопасности ОС.
- Простая адаптация к глобальному рынку за счет поддержки Юникода.

Чтобы определиться с тысячами решений, которые необходимо принимать для создания системы, удовлетворяющей этим требованиям, группа проектирования Windows NT в начале работы над проектом сформулировала следующие цели проектирования.

- **Расширяемость.** Код должен быть написан с учетом возможности комфортного расширения и изменения рыночных требований.
- **Портируемость.** Система должна нормально выполняться на разных аппаратных архитектурах и с относительной простотой переноситься на новые системы в соответствии с требованиями рынка.
- **Расширяемость и надежность.** Система должна защищаться как от внутренних сбоев, так и от попыток внешнего воздействия. Необходимо предотвратить любые попытки приложений повредить ОС или другим приложениям.
- **Совместимость.** Хотя система Windows NT должна расширять существующую технологию, ее пользовательский интерфейс и API должны быть совместимыми с предыдущими версиями Windows и MS-DOS. Система также должна взаимодействовать с другими системами, такими как UNIX, OS/2 и NetWare.
- **Быстродействие.** На всех аппаратных платформах система должна работать с максимально возможным быстродействием и скоростью отклика (в рамках других целей проектирования).

Изучая в подробностях внутреннюю структуру и принципы работы Windows, вы увидите, как эти цели проектирования и рыночные требования были успешно интегрированы в построение системы. Но прежде чем переходить к исследованию, мы рассмотрим общую модель архитектуры Windows и сравним ее с другими современными операционными системами.

Модель операционной системы

В большинстве многопользовательских операционных систем приложения отделяются от самой ОС. Код ядра ОС выполняется в привилегированном режиме процессора (называемом в книге *режимом ядра*), для него доступны системные данные и оборудование. Код приложений выполняется в непривилегированном режиме процессора (так называемом пользовательском режиме), ему предоставляется ограниченный набор интерфейсов и ограниченный доступ к системным данным, а прямой доступ к оборудованию заблокирован. Когда программа пользовательского режима вызывает системную функцию, процессор выполняет специальную команду, которая переключает вызывающий поток в режим ядра. По завершении системной функции ОС переключает контекст потока обратно в пользовательский режим и дает возможность вызывающей стороне продолжить работу.

Как и многие системы UNIX, Windows является монолитной ОС в том смысле, что большая часть кода ОС и кода драйверов устройств использует одно защищенное пространство памяти защищенного режима. Это означает, что любой компонент ОС или драйвер устройства теоретически может повредить данные, используемые другими системными компонентами ОС. Однако, как было показано в главе 1 «Концепции и средства», Windows решает эту проблему за счет повышения каче-

ства и контроля происхождения сторонних драйверов через такие программы, как WHQL и KMCS, с одновременным внедрением дополнительных технологий защиты ядра, таких как безопасность на базе виртуализации, функции Device Guard и Hyper Guard. В этом разделе вы увидите, как эти компоненты взаимодействуют между собой, а более подробная информация будет приведена в главе 7 «Безопасность» и в главе 8 «Системные механизмы».

Конечно, все эти компоненты ОС полностью защищены от некорректно работающих приложений, потому что приложения не могут напрямую обращаться к коду или данным привилегированной части ОС (хотя они могут быстро вызывать другие сервисные функции ядра). Защита — одна из причин, по которым Windows пользуется репутацией надежной и стабильной системы как сервер приложений и платформа рабочих станций, но при этом быстрой и подвижной с точки зрения базовых сервисов ОС, таких как управление виртуальной памятью, файловый ввод/вывод, сетевые коммуникации и совместное использование принтера.

В компонентах режима ядра Windows также воплощаются базовые принципы объектно-ориентированного проектирования. Например, обычно они не обращаются к структурам данных друг друга для получения доступа к информации, поддерживаемой отдельными компонентами. Вместо этого они используют формальные интерфейсы для передачи параметров, а также чтения и/или записи структур данных.

Несмотря на повсеместное использование объектов для представления общих системных ресурсов, Windows не является объектно-ориентированной системой в прямом смысле слова. Большая часть кода ОС написана на языке C для обеспечения портируемости. В языке программирования C нет прямой поддержки объектно-ориентированных конструкций, таких как полиморфные функции или наследование классов. Таким образом, реализация объектов на базе C в системе Windows заимствует некоторые возможности объектно-ориентированных языков, но не зависит от них.

Обзор архитектуры

После краткого обзора целей проектирования Windows мы рассмотрим ключевые системные компоненты, формирующие архитектуру системы. Упрощенная версия архитектуры изображена на рис. 2.1. Учтите, что диаграмма имеет достаточно общий характер и показано на ней не все. Например, сетевые компоненты и различные уровни драйверов устройств на ней не показаны.

На рис. 2.1 прежде всего обратите внимание на линию, разделяющую части пользовательского режима и режима ядра ОС Windows. Блоки выше линии представляют процессы пользовательского режима, а компоненты ниже линии — сервисные средства ОС режима ядра. Как упоминалось в главе 1, потоки пользовательского режима выполняются в закрытом адресном пространстве процессов (хотя во время выполнения в режиме ядра они получают доступ к системному пространству).

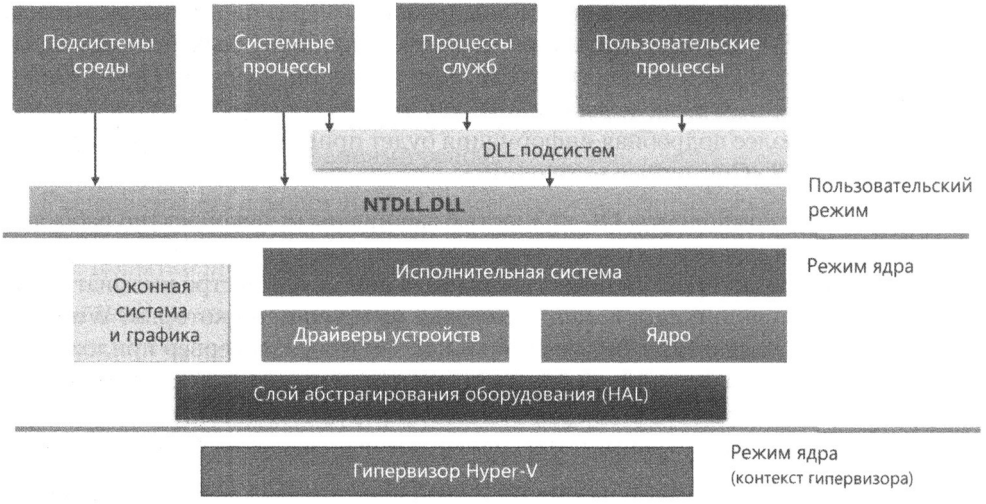


Рис. 2.1. Упрощенная архитектура Windows

Таким образом, системные процессы, процессы служб, пользовательские процессы и подсистемы среды обладают собственными закрытыми адресными пространствами. Также хорошо видна вторая разделяющая линия между компонентами режима ядра Windows и гипервизором. Строго говоря, гипервизор также работает на том же уровне привилегий процессора (0), что и ядро, но благодаря использованию специализированных команд процессора (VT-x у процессоров Intel, SVM у AMD) он может изолироваться от ядра одновременно с сохранением контроля над ним (и приложениями). По этим причинам также часто приходится слышать термин *«кольцо -1»* (формально неточный).

Ниже перечислены четыре базовых типа процессов пользовательского режима.

- **Пользовательские процессы.** Эти процессы относятся к одному из следующих типов: 32-разрядные или 64-разрядные приложения Windows (приложения Windows Apps, работающие на базе среды Windows Runtime в Windows 8 и выше, включаются в эту категорию), 16-разрядные приложения Windows 3.1, 16-разрядные приложения MS-DOS, 32-разрядные и 64-разрядные приложения POSIX. Следует заметить, что 16-разрядные приложения могут выполняться только в 32-разрядных версиях Windows, а приложения POSIX в Windows 8 уже не поддерживаются.
- **Процессы служб.** К этой категории относятся процессы, являющиеся хостами для служб Windows, например службы планировщика задач и диспетчера печати. Обычно к службам предъявляется требование независимости выполнения от входа пользователя. Многие серверные приложения Windows (такие, как Microsoft SQL Server и Microsoft Exchange Server) также включают компоненты, выполняемые как службы. Службы более подробно описаны в главе 9 «Механизмы управления», часть 2.

- **Системные процессы.** Фиксированные процессы (такие, как процесс входа или диспетчер сеансов) не являются службами Windows. Иначе говоря, они не запускаются диспетчером служб.
- **Серверные процессы подсистем среды.** Эти процессы реализуют часть поддержки среды ОС, предоставляемой пользователю и программисту. Windows NT изначально поставлялась с тремя подсистемами среды: Windows, POSIX и OS/2. Но подсистема OS/2 включалась только до Windows 2000, а подсистема POSIX в последний раз была включена в поставку Windows XP. Ultimate- и Enterprise-выпуски клиента Windows 7, а также все серверные версии Windows 2008 R2, включают поддержку расширенной подсистемы POSIX, которая называется SUA (Subsystem for UNIX-based Applications). Сейчас подсистема SUA не поддерживается и уже не включается как необязательная часть в версии Windows (клиентские или серверные).

ПРИМЕЧАНИЕ Windows 10 версии 1607 включает подсистему Windows для Linux (WSL, Windows Subsystem for Linux) в бета-версии только для разработчиков. Однако она не является полноценной подсистемой по критериям, описанным в этом разделе. В этой главе WSL и поставщики Pico рассматриваются более подробно. За информацией о процессах Pico обращайтесь к главе 3 «Процессы и задания».

На рис. 2.1 обратите внимание на блок DLL подсистем под блоками Процессы служб и Пользовательские процессы. В системе Windows пользовательские приложения не вызывают низкоуровневые сервисные функции ОС Windows напрямую. Вместо этого они проходят через одну или несколько *динамических библиотек (DLL) подсистем*. Роль DLL подсистем заключается в преобразовании документированных функций в соответствующие внутренние (и обычно недокументированные) вызовы системных функций, реализованные в основном в Ntdll.dll. Это преобразование может включать (а может не включать) отправку сообщения процессу, обслуживающему пользовательский процесс.

К категории режима ядра Windows относятся следующие компоненты:

- **Исполнительная система.** Исполнительная система содержит базовые сервисные функции ОС: управление памятью, управление процессами и потоками, безопасность, ввод/вывод, сетевая поддержка и межпроцессные коммуникации.
- **Ядро Windows.** Низкоуровневые функции ОС: планирование потоков, диспетчеризация прерываний и исключений и многопроцессорная синхронизация. Оно также предоставляет набор функций и базовых объектов, которые используются исполнительной системой для реализации высокоуровневых конструкций.
- **Драйверы устройств.** В это семейство входят как драйверы физических устройств, преобразующие вызовы пользовательских функций ввода/вывода в конкретные запросы ввода/вывода к устройству, так и драйверы устройств, не относящихся к физическому оборудованию, например драйверы файловой системы или сетевые драйверы.

- **Слой абстрагирования оборудования (HAL).** Прослойка кода, изолирующая ядро, драйверы устройств и прочий исполняемый код Windows от платформенно-зависимых различий в работе оборудования (например, различий между системными платами).
- **Оконная и графическая система.** Реализация функций графического интерфейса (GUI), также известных как функции GDI: работа с окнами, элементы пользовательского интерфейса и графический вывод.
- **Уровень гипервизора.** Состоит всего из одного компонента: самого гипервизора. В этой среде нет ни драйверов, ни других модулей. При этом сам гипервизор состоит из нескольких внутренних уровней и служб: собственный диспетчер памяти, планировщик виртуальных процессов, управление прерываниями и таймером, функции синхронизации, разделы (экземпляры виртуальных машин) и внутрипроцессные коммуникации (IPC, Inter-Process Communication) и многие другие.

В табл. 2.1 указаны имена файлов некоторых базовых компонентов ОС Windows. (Вы должны знать имена этих файлов, потому что мы будем ссылаться на некоторые системные файлы в книге по именам.) Каждый компонент более подробно рассматривается далее в этой главе и в последующих главах.

Таблица 2.1. Основные системные файлы Windows

Имя файла	Компоненты
Ntoskrnl.exe	Исполнительная система и ядро
Hal.dll	HAL
Win32k.sys	Часть подсистемы Windows режима ядра (GUI)
Hvix64.exe (Intel), Hvax64.exe (AMD)	Гипервизор
.sys les in \SystemRoot\System32\Drivers	Основные файлы драйверов: Direct X, Volume Manager, TCP/IP, TPM и поддержка ACPI
Ntdll.dll	Внутренние вспомогательные функции и заглушки диспетчеризации системных сервисных функций
Kernel32.dll, Advapi32.dll, User32.dll, Gdi32.dll	DLL основных подсистем Windows

Но прежде чем рассмотреть подробно эти системные компоненты, следует познакомиться с архитектурой ядра Windows. Начнем с того, как в Windows обеспечивается портируемость между разными аппаратными архитектурами.

Портируемость

Система Windows проектировалась с учетом возможности работы на разных аппаратных архитектурах. Исходная версия Windows NT должна была поддерживать

архитектуры x86 и MIPS. Вскоре была добавлена поддержка Alphs AXP от DEC (Digital Equipment Corporation была куплена компанией Compaq, которая позднее объединилась с Hewlett-Packard). (Хотя Alpha AXP является 64-разрядным процессором, Windows NT работала в 32-разрядном режиме. Во время разработки Windows 2000 64-разрядная версия работала на Alpha AXP, но так и не была выпущена.)

Поддержка четвертой архитектуры процессоров — Motorola PowerPC — была добавлена в Windows NT 3.51. Тем не менее из-за изменившихся требований рынка поддержка архитектур MIPS и PowerPC была прекращена до начала работы над Windows 2000. Позднее компания Compaq прекратила поддержку архитектуры Alpha AXP, и в результате Windows 2000 поддерживалась только на архитектуре x86. В Windows XP и Windows Server 2003 была добавлена поддержка двух семейств 64-разрядных процессоров: семейства Intel Itanium IA-64 и семейства AMD64 с ее эквивалентом Intel 64-bit Extension Technology (EM64T). Две последние реализации назывались *64-разрядными расширениями* и в этой книге называются x64. (О том, как Windows выполняет 32-разрядные приложения в 64-разрядной системе, рассказано в главе 8 части 2.) Кроме того, на момент выхода Server 2008 R2 системы IA-64 в Windows более не поддерживаются.

Новые версии Windows поддерживают архитектуру процессоров ARM. Например, Windows RT была версией Windows 8, работавшей на архитектуре ARM, хотя поддержка этой версии с тех пор была прекращена.

Windows 10 Mobile — наследник операционных систем Windows Phone 8.x — работает на процессорах на базе ARM, таких как модели Qualcomm Snapdragon. Windows 10 IoT работает как на x86, так и на устройствах ARM, таких как Raspberry Pi 2 (использует процессор ARM Cortex-A7) и Raspberry Pi 3 (использует ARM Cortex-A53). Возможно, с переходом оборудования ARM на 64-разрядную модель в какой-то момент появится поддержка нового семейства процессоров AArch64, или ARM64, на котором работает все большее количество устройств.

В Windows портируемость между аппаратными архитектурами и платформами достигается двумя основными способами:

Многоуровневая архитектура. Windows имеет многоуровневую архитектуру: на нижнем уровне находятся части системы, зависящие от конкретной архитектуры процессора или от платформы и оформленные в виде отдельных модулей, чтобы верхние уровни системы были изолированы от различий между архитектурами и аппаратными платформами. Два ключевых компонента, обеспечивающих портируемость ОС, — ядро (находится в файле Ntoskrnl.exe) и HAL (находится в файле Hal.dll). Оба компонента более подробно описаны далее в этой главе. Функции, зависящие от архитектуры (например, переключение контекста потоков и диспетчеризация ловушек (traps)) реализованы в ядре. Функции, которые могут различаться между системами с одной архитектурой (например, разные системные платы), реализованы в HAL. Кроме них единственным компонентом со значительным объемом кода, привязанного к конкретной архитектуре, является диспетчер памяти, но даже он

относительно мал по сравнению с системой в целом. Гипервизор использует похожую архитектуру: большинство компонентов используется совместно реализациями для AMD (SVM) и Intel (VT-x), с набором специализированных компонентов для каждого процессора — отсюда два имени файла в табл. 2.1.

Использование языка C. Подавляющее большинство кода Windows написано на языке C, лишь некоторые части написаны на C++. Язык ассемблера используется только для частей O, которые должны напрямую взаимодействовать с системным оборудованием (например, обработчики прерываний) или которые в высшей степени чувствительны к быстродействию (например, переключение контекста). Код на языке ассемблера встречается не только в ядре и HAL, но и в других местах базовой ОС (например, функциях, реализующих заблокированные команды, а также в одном модуле поддержки локального вызова процедур), в части режима ядра подсистемы Windows и даже в некоторых библиотеках пользовательского режима (например, в коде запуска процессов в Ntdll.dll (системная библиотека, рассматривается далее в этой главе).

Симметричная многопроцессорная архитектура

Многозадачность (multitasking) — метод, применяемый ОС для совместного использования одного процессора несколькими программными потоками. Но если компьютер оснащен несколькими процессорами, он может выполнять несколько потоков одновременно. Таким образом, если многозадачная ОС только создает видимость одновременного выполнения нескольких потоков, ОС с многопроцессорной обработкой реализует одновременное выполнение — по одному потоку для каждого процессора.

Как упоминалось в начале главы, одна из ключевых целей при проектировании Windows заключалась в том, что ОС должна хорошо работать на многопроцессорных компьютерных системах. Windows является ОС с симметричной многопроцессорной обработкой (SMP, Symmetric Multiprocessing). «Главного» процессора не существует — и ОС, и пользовательские потоки могут планироваться для выполнения на любом процессоре. Кроме того, все процессоры используют всего одно пространство памяти. Эта модель отличается от асимметричной многопроцессорной обработки (ASMP, Asymmetric Multiprocessing), в которой ОС обычно выбирает один процессор для выполнения кода ядра ОС, а на остальных процессорах выполняется только пользовательский код. Различия между двумя многопроцессорными моделями продемонстрированы на рис. 2.2.

Windows также поддерживает четыре основные характеристики современных многопроцессорных систем: *многоядерность*, *одновременную многопоточность* (SMT, Simultaneous Multi-threading), *разнородность* и *асимметричный доступ к памяти* (NUMA, Non-Uniform Memory Access). Эти характеристики кратко описаны далее. (За полным, подробным описанием поддержки планирования в таких системах обращайтесь к соответствующему разделу главы 4 «Потоки».)

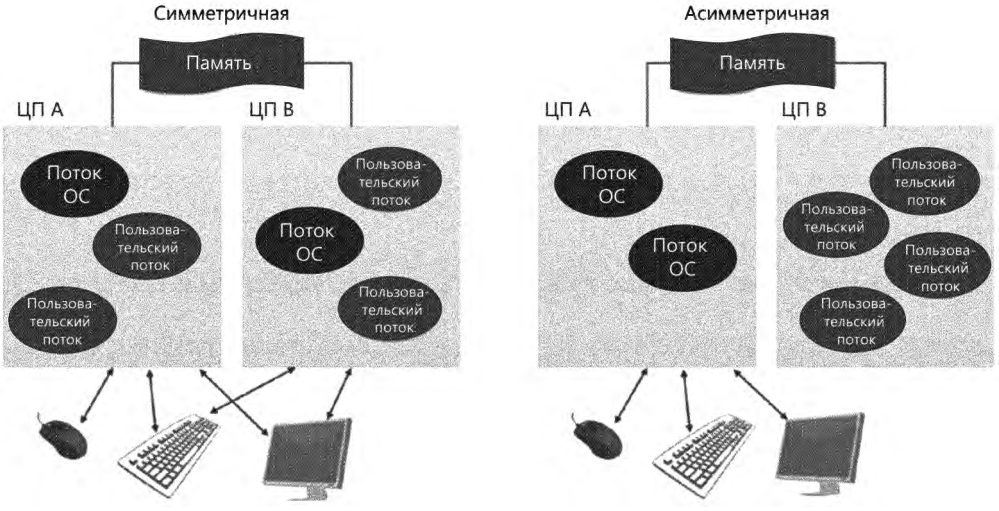


Рис. 2.2. Симметричная и асимметричная многопроцессорная обработка

Одновременная многопоточность впервые появилась в системах Windows при включении поддержки технологии гиперпоточности Intel (Hyper-Threading), позволяющей смоделировать два логических процессора для каждого физического ядра. В более новых процессорах с микроархитектурой Zen реализована похожая технология SMT, также удваивающая количество логических процессоров.

Каждый логический процессор обладает собственным состоянием, но исполнительное ядро и встроенный кэш используются совместно. Это позволяет одному логическому процессору действовать, пока другие логические процессоры простаивают (например, после промаха кэша или неверного прогнозирования ветвления). По неизвестной причине в маркетинговой литературе обеих компаний дополнительные ядра называются «*потоками*», поэтому часто приходится слышать формулировки типа «четыре ядра, восемь потоков». Это означает, что планироваться могут до восьми потоков; соответственно, существуют восемь логических процессоров. Алгоритмы планирования усовершенствованы для оптимального использования машин с SMT, например, посредством планирования потоков на свободном физическом процессоре вместо выбора свободного логического процессора на физическом процессоре, остальные логические процессоры которого заняты. За дополнительной информацией о планировании потоков обращайтесь к главе 4.

В системах NUMA процессоры объединяются в группы, называемые *узлами* (nodes). Каждый узел использует собственные процессоры и память и подключается к большей системе через соединительную шину с когерентностью кэша. Windows на системах NUMA также работает как система SMP, в которой все процессоры обладают доступом ко всей памяти; просто обращение к локальной памяти узлов осуществляется быстрее, чем обращение к памяти других узлов. Система стремится повысить быстродействие за счет планирования потоков на процессоры

того же узла, к которому относится используемая память. Она пытается выполнять запросы на выделение памяти в узле, но при необходимости выделяет память из других узлов.

Естественно, Windows также обладает встроенной поддержкой многоядерных систем. Так как в этих системах существуют реальные физические ядра, исходный код SMP в Windows обращается с ними как с разными процессорами, если не считать некоторых учетных операций и задач идентификации (таких, как лицензирование, см. ниже), различающихся между ядрами одного процессора и ядрами разных процессоров. Это особенно важно при использовании топологии кэша для оптимизации совместного доступа к данным.

Наконец, ARM-версии Windows также поддерживают технологию *разнородной многопроцессорной обработки*, реализация которой на таких процессорах называется big.LITTLE. Эта разновидность архитектур на базе SMP отличается от традиционных тем, что не все ядра процессоров идентичны по своим возможностям, но, в отличие от «чистой» разнородной многопроцессорной обработки, могут выполнять те же команды. Различия проявляются в тактовой частоте и потребляемой мощности в состоянии полной нагрузки/бездействия, что позволяет объединять группы более медленных ядер с более быстрыми.

Представьте операцию отправки электронной почты на старой двухъядерной 1-гигагерцовой системе, подключенной к Интернету по современному каналу связи. Вряд ли она будет работать медленнее, чем восьмиядерная машина с частотой 3,6 ГГц, потому что «узкими местами» обычно становятся скорость ввода с клавиатуры и пропускная способность канала, а не простая вычислительная мощность. Тем не менее даже в самом глубоком режиме энергосбережения такая современная система с большой вероятностью будет использовать существенно большую мощность, чем старая система. Даже если бы машина могла понизить свою тактовую частоту до 1 ГГц, старая система могла бы, например, понизить свою тактовую частоту до 200 МГц.

Возможность объединения таких старых мобильных процессоров с ультрасовременными платформами на базе ARM в сочетании с совместимым планировщиком ядра ОС позволяет поднять вычислительную мощность до максимума, когда требуется включение всех ядер, необходимо выдержать баланс (поддержание части мощных ядер в активном состоянии и части менее мощных для выполнения других задач) или же работать в режиме чрезвычайно низкого энергопотребления (в активном состоянии остается только одно маломощное ядро — этого достаточно для получения SMS и отправки электронной почты). За счет поддержания так называемых разнородных политик планирования Windows 10 позволяет потокам выбирать политику, отвечающую их потребностям, и взаимодействовать с планировщиком и диспетчером питания для ее оптимальной поддержки. Эти политики более подробно рассматриваются в главе 4.

Система Windows изначально не проектировалась с расчетом на ограничение количества процессоров, если не считать политик лицензирования, различающихся

в разных выпусках Windows. Однако для удобства и эффективности Windows отслеживает информацию о процессорах (общее количество, занятость и т. д.) в битовой маске (иногда называемой *маской сходства*), количество бит в которой совпадает с разрядностью встроенного типа данных машины (32-разрядная или 64-разрядная). Это позволяет процессору напрямую выполнять операции с битами в регистрах. По этой причине количество процессоров в системах Windows изначально ограничивалось разрядностью машинного слова, потому что разрядность маски сходства невозможно было увеличивать произвольно. Для обеспечения совместимости, а также для поддержки систем с большим количеством процессоров в Windows реализована конструкция более высокого порядка: *группа процессоров* (processor group). Группа процессоров представляет собой набор процессоров, которые могут определяться одной маской сходства, а ядро и приложения могут выбирать нужную группу при обновлениях маски сходства. Совместимые приложения могут запрашивать информацию о количестве поддерживаемых групп (в настоящее время ограничивается 20; максимальное количество логических процессоров в настоящее время ограничивается 640), а затем перебрать битовые маски всех групп. В настоящее время старые приложения продолжают работать, видя только текущую группу. За дополнительной информацией о том, как Windows распределяет процессоры по группам (которая также относится к NUMA), обращайтесь к главе 4.

Как упоминалось ранее, фактическое количество поддерживаемых лицензированных процессоров зависит от выпуска Windows (см. табл. 2.2). Это число хранится в файле системной политики лицензирования (который фактически содержит набор пар «имя/значение») %SystemRoot%\ServiceProfiles\LocalService\AppData\Local\Microsoft\WSLicense\tokens.dat в переменной с именем `kernel-RegisteredProcessors`.

Масштабируемость

Одним из ключевых аспектов многопроцессорных систем является *масштабируемость*. Чтобы код ОС правильно работал в системах SMP, он должен придерживаться строгих рекомендаций и правил. В многопроцессорных системах конкуренция за обладание ресурсами и другие вопросы производительности решаются сложнее, чем в однопроцессорных системах, и это обстоятельство должно быть учтено при проектировании системы. Windows включает ряд возможностей, сыгравших критическую роль в ее успехе как многопроцессорной ОС.

- Код ОС может выполняться на любом из существующих процессоров и на нескольких процессорах одновременно.
- Возможность существования в одном процессе нескольких программных потоков, каждый из которых может выполняться одновременно на разных процессорах.

- Высокоточные механизмы синхронизации в ядре (спин-блокировки, спин-блокировки с очередями и пуш-блокировки, описанные в главе 8 части 2), а также в драйверах устройств и серверных процессах, позволяющие большему количеству компонентов работать параллельно на нескольких процессорах.
- Механизмы программирования, облегчающие эффективную реализацию многопоточных серверных процессов, хорошо масштабируемых в многопроцесорных системах, такие как порты завершения ввода/вывода (см. главу 6 «Подсистема ввода/вывода»).

Масштабируемость ядра Windows улучшалась со временем. Например, в Windows 2003 появились очереди планирования на уровне отдельных процессоров с высокоточными блокировками, реализующими возможность параллельного принятия решений планировки потоков на нескольких процессорах. В Windows 7 и Windows Server 2008 R2 была исключена глобальная блокировка планировщика во время операций диспетчеризации с ожиданием. Это постепенное повышение точности блокировок также происходило и в других областях: в диспетчере памяти, диспетчере кэша и диспетчере объектов.

Различия между клиентскими и серверными версиями

Windows поставляется как в клиентских, так и в серверных вариантах. Существуют шесть настольных клиентских версий Windows 10: Windows 10 Home, Windows 10 Pro, Windows 10 Education, Windows 10 Pro Education, Windows 10 Enterprise и Windows 10 Enterprise Long Term Servicing Branch (LTSB). К числу других, ненастольных, изданий относятся Windows 10 Mobile, Windows 10 Mobile Enterprise и Windows 10 IoT Core, IoT Core Enterprise и IoT Mobile Enterprise. Существует еще больше разновидностей, предназначенных для регионов с особыми потребностями, например серии N.

Windows Server 2016 существует в шести разных версиях: Windows Server 2016 Datacenter, Windows Server 2016 Standard, Windows Server 2016 Essentials, Windows Server 2016 MultiPoint Premium Server, Windows Storage Server 2016 и Microsoft Hyper-V Server 2016.

Ниже перечислены различия между этими версиями:

- Модель ценообразования по количеству ядер (а не по количеству процессоров) для Server 2016 Datacenter и Standard.
- Общее количество поддерживаемых логических процессоров.
- Для серверных систем — разрешенное для выполнения количество контейнеров Hyper-V (клиентские системы поддерживают только контейнеры Windows на базе пространств имен).

- Объем поддерживаемой физической памяти (фактически наивысший физический адрес ОЗУ, который может использоваться системой; более подробно об ограничениях физической памяти см. в главе 5 «Управление памятью»).
- Количество поддерживаемых параллельных сетевых подключений (например, для служб файлов и печати в клиентских версиях разрешено до 10 параллельных подключений).
- Поддержка мультисенсорного ввода и композиции рабочего стола.
- Поддержка таких функций, как BitLocker, загрузка VHD, AppLocker, Hyper-V и более 100 других настраиваемых параметров политики лицензирования.
- Многоуровневые службы, поставляемые в выпуски Windows Server, которые не входят в клиентские выпуски (службы каталогов, Host Guardian, Storage Spaces Direct, защищенные виртуальные машины и кластеризация).

В табл. 2.2 перечислены различия в поддержке памяти и процессоров в некоторых выпусках Windows 10, Windows Server 2012 R2 и Windows Server 2016. Подробная сравнительная схема разных выпусков Windows Server 2012 R2 доступна по адресу *https://www.microsoft.com/en-us/download/details.aspx?id=41703*. Информация об ограничениях работы с памятью в выпусках Windows 10 и Server 2016 и более ранних доступна по адресу *https://msdn.microsoft.com/en-us/library/windows/desktop/aa366778.aspx*.

Таблица 2.2. Ограничения по количеству процессоров и объему памяти в некоторых выпусках Windows

	Количество поддерживаемых физических процессоров (32-разрядная версия)	**Поддерживаемая физическая память (32-разрядная версия)**	**Количество логических/физических процессоров (64-разрядная версия)**	**Поддерживаемая физическая память (версии x64)**
Windows 10 Home	1	4 Гбайт	1 физический процессор	128 Гбайт
Windows 10 Pro	2	4 Гбайт	2 физических процессора	2 Тбайт
Windows 10 Enterprise	2	4 Гбайт	2 физических процессора	2 Тбайт
Windows Server 2012 R2 Essentials	–	–	2 физических процессора	64 Гбайт
Windows Server 2016 Standard	–	–	512 логических процессоров	24 Тбайт
Windows Server 2016 Datacenter	–	–	512 логических процессоров	24 Тбайт

И хотя ОС Windows существует в нескольких клиентских или серверных вариантах, они совместно используют общий набор базовых системных файлов, включая образ ядра Ntoskrnl.exe (и PAE-версию Ntkrnlpa.exe), библиотеки HAL, драйверы устройств, базовые системные средства и DLL.

При таком количестве разных выпусков Windows, имеющих одинаковый образ ядра, как система узнает, какой выпуск загружается? По значениям параметров реестра `ProductType` и `ProductSuite` из раздела HKLM\SYSTEM\CurrentControlSet\Control\ProductOptions. Параметр `ProductType` указывает, является ли система клиентской или серверной (любого типа). Эти значения загружаются в реестр на основании файла политики лицензирования (см. выше). Допустимые значения перечислены в табл. 2.3. Их можно запросить из пользовательского режима функцией `VerifyVersionInfo` или из драйвера устройства функциями режима ядра `RtlGetVersion` и `RtlVerifyVersionInfo` (обе функции документированы в WDK).

Таблица 2.3. Значения параметра реестра ProductType

Выпуск Windows	Значение ProductType
Клиент	`WinNT`
Сервер (контроллер домена)	`LanmanNT`
Сервер (только сервер)	`ServerNT`

Другой параметр реестра, `ProductPolicy`, содержит кэшированную копию данных из файла tokens.dat, по которым различаются выпуски Windows и предоставляемая ими функциональность.

Итак, если основные файлы практически одинаковые для клиентских и серверных версий, как системы различаются в работе? Если коротко, серверные системы оптимизируются по умолчанию для максимального быстродействия в роли высокопроизводительных серверов приложений, тогда как клиентская версия (хотя и обладает серверными возможностями) оптимизируется для скорости реакции в роли интерактивного настольного компьютера. Например, в зависимости от типа продукта во время загрузки некоторые решения из области распределения ресурсов принимаются по-разному (например, размер и количество куч (пулов) ОС, количество внутренних системных рабочих потоков, размер системного кэша данных). Кроме того, некоторые решения политик времени выполнения (например, способ балансировки диспетчером памяти запросов системной памяти и памяти процессов) также различаются между серверными и клиентскими выпусками. Даже некоторые нюансы планирования потоков по умолчанию обладают разным поведением в зависимости от семейства (длина интервала времени по умолчанию, или *квант* потока; за подробностями обращайтесь к главе 4). В тех случаях, когда между двумя продуктами существуют значительные практические различия, мы будем особо выделять их в соответствующих главах в остальной части книги. Если в тексте явно не указано обратное, весь материал книги относится как к клиентским, так и к серверным версиям.

ЭКСПЕРИМЕНТ: ОПРЕДЕЛЕНИЕ ВОЗМОЖНОСТЕЙ, АКТИВИЗИРУЕМЫХ ПОЛИТИКОЙ ЛИЦЕНЗИРОВАНИЯ

Как упоминалось ранее, Windows поддерживает более 100 разных функциональных возможностей, которые могут активизироваться механизмом лицензирования программного продукта. Эти параметры политики определяют различия не только между клиентской и серверной установкой, но и между выпусками ОС: как, например, поддержка BitLocker (доступная и в серверных версиях Windows, и в выпусках Pro и Enterprise клиентской версии Windows). Для вывода многих параметров политики можно воспользоваться программой SlPolicy из загружаемых материалов, прилагаемых к книге.

Параметры политики упорядочиваются по *подсистемам*, представляющим модуль, к которому применяется политика. Для вывода списка всех подсистем, известных программе, запустите SlPolicy.exe с ключом `-f`:

```
C:\>SlPolicy.exe -f
Software License Policy Viewer Version 1.0 (C)2016 by Pavel Yosifovich
Desktop Windows Manager
Explorer
Fax
Kernel
IIS
…
```

Укажите имя любой подсистемы после ключа, чтобы вывести данные политики для этой подсистемы. Например, для получения информации об ограничении количества процессоров и доступной памяти используется подсистема Kernel. Примерный вывод программы на машине с Windows 10 Pro выглядит так:

```
C:\>SlPolicy.exe -f Kernel
Software License Policy Viewer Version 1.0 (C)2016 by Pavel Yosifovich
Kernel
------
Maximum allowed processor sockets: 2
Maximum memory allowed in MB (x86): 4096
Maximum memory allowed in MB (x64): 2097152
Maximum memory allowed in MB (ARM64): 2097152
Maximum physical page in bytes: 4096
Device Family ID: 3
Native VHD boot: Yes
Dynamic Partitioning supported: No
Virtual Dynamic Partitioning supported: No
Memory Mirroring supported: No
Persist defective memory list: No
```

Или другой пример: вывод для подсистемы kernel в выпуске Windows Server 2012 R2 Datacenter будет выглядеть примерно так:

```
Kernel
------
Maximum allowed processor sockets: 64
Maximum memory allowed in MB (x86): 4096
```

```
Maximum memory allowed in MB (x64): 4194304
Add physical memory allowed: Yes
Add VM physical memory allowed: Yes
Maximum physical page in bytes: 0
Native VHD boot: Yes
Dynamic Partitioning supported: Yes
Virtual Dynamic Partitioning supported: Yes
Memory Mirroring supported: Yes
Persist defective memory list: Yes
```

Отладочная сборка

Существует специальная внутренняя отладочная версия Windows, называемая *отладочной сборкой* (внешний доступ к ней открыт только для Windows 8.1 и более ранних версий в составе подписки MSDN Operating Systems). Она представляет собой результат компиляции исходного кода Windows с установленным флагом времени компиляции `DBG`, с которым включается код условной отладки и трассировки. Кроме того, для простоты понимания машинного кода двоичные файлы Windows не подвергаются последующей обработке для оптимизации структуры кода с целью ускорения выполнения. (См. раздел «Debugging performance-optimized code» в справочном файле средств отладки для Windows.)

Отладочная сборка изначально предоставлялась для содействия разработчикам драйверов устройств, потому что она выполняет более жесткую проверку ошибок при вызове функций режима ядра драйверами устройств или другим системным кодом. Например, если драйвер (или другой фрагмент кода режима ядра) обратится с некорректным вызовом к системной функции, проверяющей параметры (например, при получении спин-блокировки на неверном уровне запроса прерывания), система остановит выполнение при обнаружении проблемы и не допустит порчу структуры данных и последующий возможный фатальный сбой системы. Так как полная отладочная сборка часто была нестабильной и не работала во многих средах, компания Microsoft предоставляет отладочную версию ядра и HAL только для Windows 10 и более поздних версий. Это позволяет разработчикам получить ту же пользу при взаимодействии с кодом ядра и HAL, не сталкиваясь с проблемами полной отладочной сборки. Отладочные версии ядра и HAL свободно распространяются в составе WDK (каталог `\Debug` корневого пути установки). За подробными инструкциями по поводу того, как с ними работать, обращайтесь к разделу «Installing Just the Checked Operating System and HAL» в документации WDK.

ЭКСПЕРИМЕНТ: ПРОВЕРКА ВЫПОЛНЕНИЯ ОТЛАДОЧНОЙ СБОРКИ

Не существует встроенных средств, которые выводили бы информацию о том, выполняете вы отладочную или нормальную версию ядра. Однако эту информацию можно получить из свойства Debug класса WMI `Win32_OperatingSystem`.

Следующий сценарий PowerShell выводит это свойство. (Чтобы воспроизвести этот пример, откройте сервер сценариев PowerShell.)

```
PS C:\Users\pavely> Get-WmiObject win32_operatingsystem | select debug
debug
-----
False
```

В этой системе отладочная сборка не выполняется, потому что свойство Debug содержит значение `False`.

Большая часть дополнительного кода в двоичных файлах отладочных сборок является результатом использования макросов `ASSERT` и/или `NT_ASSERT`, определяемых в заголовочном файле WDK `Wdm.h` и описываемых в документации WDK. Эти макросы проверяют условие — например, действительность структуры данных или параметра. Если результат выражения равен `FALSE`, макрос либо вызывает функцию режима ядра `RtlAssert`, которая вызывает `DbgPrintEx` для отправки текста отладочного сообщения в буфер отладочных сообщений, либо выдает прерывание проверки условия (прерывание 0x2B в системах x64 и x86). При наличии присоединенного отладчика ядра и загруженной символической информации сообщение выводится автоматически, а за ним пользователю предлагается выбрать дальнейшее действие (активизировать точку останова, игнорировать, завершить процесс или завершить поток). Если система не была загружена с отладчиком ядра (при помощи параметра `debug` в базе данных конфигурации загрузки) и отладчик ядра не присоединен, нарушенная проверка условия приведет к фатальному сбою системы. Небольшой список проверок, выполняемых некоторыми вспомогательными функциями ядра, приведен в разделе «Checked Build ASSERTs» документации WDK (учтите, что этот список не поддерживается и содержит устаревшую информацию).

Отладочная сборка также полезна для системного администратора из-за дополнительной подробной трассировки, которую можно включить для некоторых компонентов. (За подробными инструкциями обращайтесь к статье Microsoft Knowledge Base Article 314743 «HOWTO: Enable Verbose Debug Tracing in Various Drivers and Subsystems».) Выводимая информация передается во внутренний отладочный буфер сообщений с использованием функции `DbgPrintEx`, упоминавшейся ранее. Чтобы увидеть отладочные сообщения, присоедините отладчик ядра к целевой системе (для чего необходимо загрузить целевую систему в отладочном режиме), используйте команду `!dbgprint` во время выполнения локальной отладки ядра или воспользуйтесь программой `Dbgview.exe` из пакета Sysinternals. Тем не менее многие последние версии Windows отказались от такого типа отладочного вывода и используют трассировку препроцессора Windows (WPP) или технологию TraceLogging; оба способа строятся на базе технологии Event Tracing for Windows (ETW). Преимущество новых механизмов вывода информации заключается в том, что они не ограничиваются отладочными версиями компонентов (что особенно полезно сейчас, когда полная отладочная сборка стала недоступной), а для просмотра информации можно пользоваться такими средствами, как программа

WPA (Windows Performance Analyzer), ранее известная под названием XPerf или Windows Perfomance Toolkit, TraceView (из WDK), или команда расширения `!wmiprint` отладчика ядра.

Наконец, отладочная сборка может пригодиться для тестирования кода пользовательского режима просто из-за других временных характеристик системы (из-за дополнительных проверок, происходящих в ядре, и того факта, что компоненты компилируются без оптимизаций). Часто многопоточные ошибки синхронизации связаны с конкретными состояниями хронометража. При запуске тестов в системе, в которой работает отладочная сборка (или, по крайней мере, отладочное ядро и HAL), из-за изменившихся временных характеристик всей системы на поверхность могут всплыть ошибки, которые не встречаются в нормальной системе.

Обзор архитектуры безопасности на основе виртуализации

Как было показано в главе 1 и в этой главе, разделение между пользовательским режимом и режимом ядра обеспечивает защиту ОС от кода пользовательского режима (злонамеренного или нет). Но если нежелательный фрагмент кода режима ядра проникает в систему (из-за неисправленного ядра или уязвимости драйвера или потому, что пользователя обманом убедили установить вредоносный или уязвимый драйвер), безопасность системы фактически нарушена, потому что весь код режима ядра обладает полным доступом ко всей системе. Технологии, описанные в главе 1, использующие гипервизор для предоставления дополнительных гарантий против атак, образуют набор средств безопасности на основе виртуализации (VBS, Virtualization-Based Security), совершенствующих естественную изоляцию на основе привилегий через введение виртуальных уровней безопасности (VTL, Virtual Trust Levels). Кроме простого введения нового ортогонального механизма изоляции доступа к памяти, оборудованию и ресурсам процессора, технология VTL также требует нового кода и компонентов для управления более высокими уровнями доверия. Обычному ядру и драйверам, работающим на уровне VTL 0, не разрешается управление и определение ресурсов VTL 1; это будет противоречить самой цели.

На рис. 2.3 показана архитектура Windows 10 Enterprise и Server 2016 при активной безопасности VBS. (Также иногда используется термин *Virtual Secure Mode*, или *VSM*.) В релизах Windows 10 версии 1607 и Server 2016 он всегда активен по умолчанию, если поддерживается оборудованием. В более старых версиях Windows 10 его можно активизировать при помощи политики или в диалоговом окне Add Windows Features (выберите режим Isolated User Mode).

Как показано на рис. 2.3, код пользовательского режима/режима ядра, рассмотренный ранее, работает на базе гипервизора Hyper-V, как и на рис. 2.1. Различие состоит в том, что с включенным механизмом VBS появляется уровень VTL 1, который содержит свое собственное безопасное ядро, работающее в привилегиро-

ванном режиме процессора (кольцо 0 в x86/x64). Аналогичным образом появился пользовательский режим времени выполнения, называемый IUM (Isolated User Mode), который работает в непривилегированном режиме (кольцо 3).

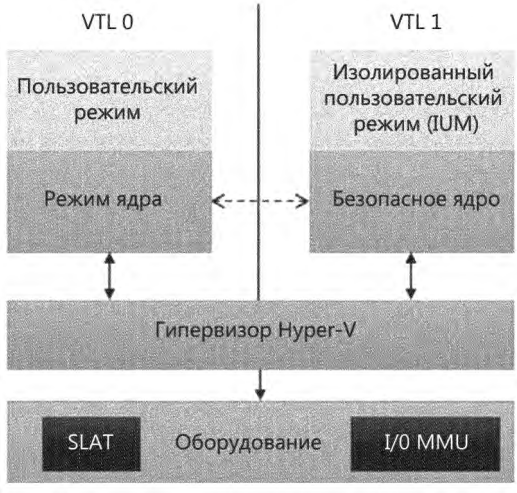


Рис. 2.3. Архитектура VBS в Windows 10 и Server 2016

В этой архитектуре безопасное ядро представлено отдельным двоичным файлом, который хранится на диске под именем securekernel.exe. Что касается IUM, это и среда, которая ограничивает системные вызовы, разрешенные для DLL пользовательского режима (определяя тем самым, какие из этих DLL могут быть загружены), и фреймворк, который добавляет специальные вызовы системных функций, выполняемых только в VTL 1. Доступ к этим дополнительным системным вызовам осуществляется по аналогии с обычными системными вызовами: через внутреннюю системную библиотеку с именем lumdll.dll (версия Ntdll.dll для VTL 1) и библиотеку, обращенную к подсистеме Windows, с именем lumbase.dll (версия Kernelbase.dll для VTL 1). Реализация IUM, в основном использующая те же стандартные библиотеки Win32 API, обеспечивает сокращение затрат памяти приложений пользовательского режима VTL 1, потому что в них, по сути, задействован тот же код пользовательского режима, что и в их аналогах VTL 0. Важное замечание: механизмы копирования при записи, о которых вы подробнее узнаете в главе 5, не позволяют приложениям VTL 0 модифицировать двоичный код, используемый VTL 1.

В VBS действуют стандартные правила взаимодействия обычного пользователя с ядром, но теперь они дополняются факторами VTL. Другими словами, код режима ядра, выполняемый на уровне VTL 0, не может контактировать с кодом пользовательского режима, выполняемого в VTL 1, потому что уровень VTL 1 более привилегирован. Тем не менее код пользовательского режима, выполняемый на уровне VTL 1, тоже не может контактировать с режимом ядра, выполняемым

на уровне VTL 0, потому что пользователь (кольцо 3) не может контактировать с ядром (кольцо 0). Аналогичным образом приложения пользовательского режима VTL 1 все равно должны проходить через обычные системные вызовы Windows и соответствующие проверки доступа, если они пожелают обратиться к ресурсам.

Происходящее можно рассматривать так: уровни привилегий (пользовательский режим или режим ядра) обеспечивают полномочия. VTL, с другой стороны, обеспечивают изоляцию. Хотя приложение пользовательского режима VTL 1 не обладает бóльшими полномочиями, чем приложение VTL 0 или драйвер, оно от них изолируется. По сути, приложения VTL 1 не просто обладают бóльшими полномочиями; во многих случаях их полномочия намного меньше. Так как безопасное ядро не реализует весь диапазон системных возможностей, оно специально отбирает, какие системные вызовы будут перенаправляться ядру VTL 0. Любые разновидности ввода/вывода, включая работу с файлами, сетью и реестром, полностью запрещены. Графика также полностью исключается. Взаимодействия с любыми драйверами запрещены.

С другой стороны, безопасное ядро, работающее на уровне VTL 1 и в режиме ядра, обладает полным доступом к ресурсам и памяти VTL 0. Оно может использовать гипервизор для ограничения доступа ОС VTL 0 к некоторым участкам памяти за счет использования аппаратной поддержки процессора, называемой SLAT (Second Level Address Translation). SLAT лежит в основе технологии Credential Guard, которая позволяет хранить секретную информацию в этих ячейках.

Аналогичным образом безопасное ядро может использовать технологию SLAT для управления выполнением определенных участков памяти — ключевой аспект Device Guard.

Чтобы обычные драйверы устройства не могли использовать оборудование для прямых обращений к памяти, система задействует другой аппаратный компонент — блок управления памятью (I/O MMU, Input/Output Memory Management Unit), который фактически виртуализирует доступ к памяти для устройств. Он предотвращает прямые обращения к участкам физической памяти гипервизора или безопасного ядра средствами DMA (Direct Memory Access). Такие обращения работали бы в обход SLAT, потому что виртуальная память в них не задействована.

Поскольку гипервизор является первым системным компонентом, который запускается начальным загрузчиком, он может программировать SLAT и I/O MMU так, как считает нужным, определяя исполнительные среды VTL 0 и VTL 1. Затем, находясь в VTL 1, загрузчик выполняется снова и загружает безопасное ядро, которое может продолжить настройку системы по своим потребностям. Только после этого VTL снижается, и начинает работать нормальное ядро; теперь оно существует в своей «тюрьме» VTL 0, не имея возможности выйти за ее пределы.

Так как процессы пользовательского режима в VTL 1 изолированы, потенциально вредоносный код, хотя и бессилен оказать влияние на систему, может работать

скрытно, пытаться вызывать защищенные системные функции (что позволит ему подписывать собственные секреты) и теоретически создавать нежелательные взаимодействия с другими процессами VTL 1 или ядром. По этой причине в VTL 1 может выполняться только особый класс специально подписанных двоичных *трастлетов* (trustlets). Каждый трастлет имеет уникальный идентификатор и сигнатуру, а в безопасном ядре жестко закодирована информация о трастлетах, созданных до настоящего момента. Соответственно, невозможно создавать новые трастлеты без обращения к безопасному ядру (которое доступно только для Microsoft), а существующие трастлеты модифицироваться не могут (тогда специальная электронная подпись Microsoft стала бы недействительной). Подробнее о трастлетах см. в главе 3.

Введение безопасного ядра и VBS — интересный шаг в архитектуре современных ОС. Дополнительные аппаратные модификации шин (таких, как PCI и USB) делают возможной поддержку целого класса безопасных устройств, которые в сочетании с минималистичным безопасным HAL, безопасным диспетчером Plug-and-Play и безопасной инфраструктурой User-Mode Device Framework предоставят некоторым приложениям VTL 1 прямой и изолированный доступ к специальным устройствам (например, устройствам ввода биометрических данных или смарт-карт). Скорее всего, эти возможности будут использоваться в новых версиях Windows 10.

Ключевые компоненты системы

После знакомства с высокоуровневой архитектурой Windows можно переходить к более глубокому изучению внутреннего строения и роли каждого из ключевых компонентов ОС. На рис. 2.4 представлена более подробная и полная диаграмма базовой архитектуры системы Windows и компонентов, изображенных на рис. 2.1. Как и в предыдущем случае, на диаграмме представлены не все компоненты (особенно сетевая поддержка, которая подробно рассматривается в главе 10 «Сеть», часть 2).

В следующих разделах подробнее рассматривается каждый из основных элементов этой диаграммы. В главе 8 части 2 рассматриваются основные управляющие механизмы, используемые системой (такие, как диспетчер объектов, прерывания и т. д.).

В главе 11 «Запуск и завершение работы» описывается процесс запуска и завершения работы Windows, а в главе 9 освещаются такие управляющие механизмы, как реестр, процессы служб и WMI. В других главах еще подробнее исследуется внутренняя структура и принципы работы ключевых областей: процессов и потоков, управления памятью, безопасности, диспетчера ввода/вывода, управления дисковым пространством, диспетчера кэша, файловой системы Windows (NTFS) и сети.

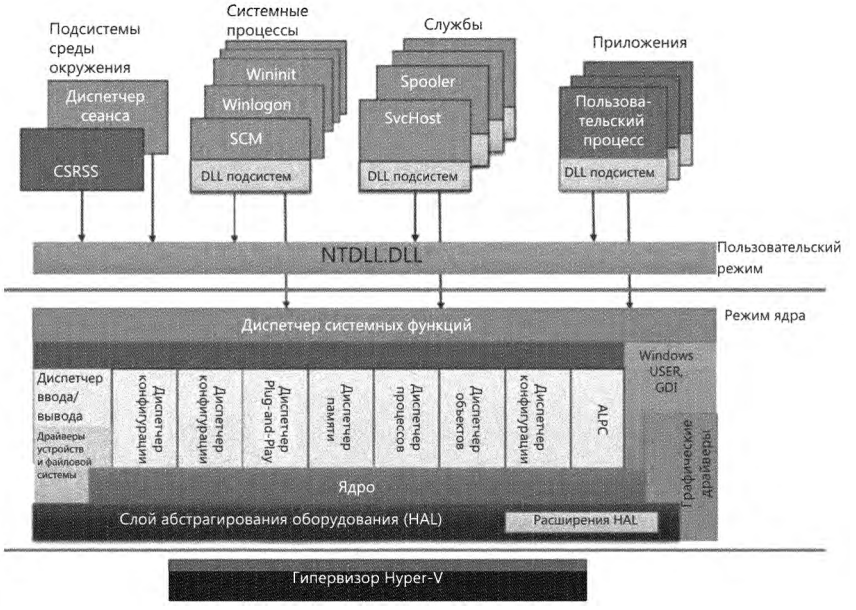


Рис. 2.4. Архитектура Windows

Подсистемы среды и DLL среды

Подсистема среды нужна, чтобы предоставлять прикладным программам некоторое подмножество сервисных функций базовой исполнительной системы Windows. Разные подсистемы открывают доступ к разным подмножествам встроенных сервисных функций Windows. Это означает, что в приложении, построенном на базе одной подсистемы, можно делать то, что не может быть сделано в приложении на базе другой подсистемы. Например, приложения Windows не могут использовать функцию SUA `fork`.

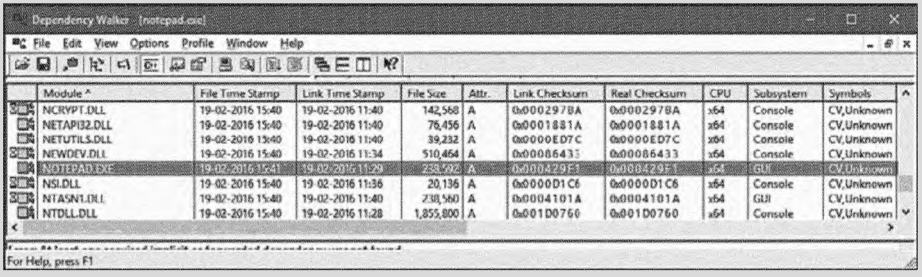
Каждый исполняемый образ (.exe) связывается с одной и только одной подсистемой. При выполнении образа код создания процесса анализирует код типа подсистемы в заголовке образа, чтобы оповестить правильную подсистему о новом процессе. Код типа задается параметром компоновщика Microsoft Visual Studio `/SUBSYSTEM` (или при помощи записи `SubSystem` в странице свойств Linker/System свойств проекта).

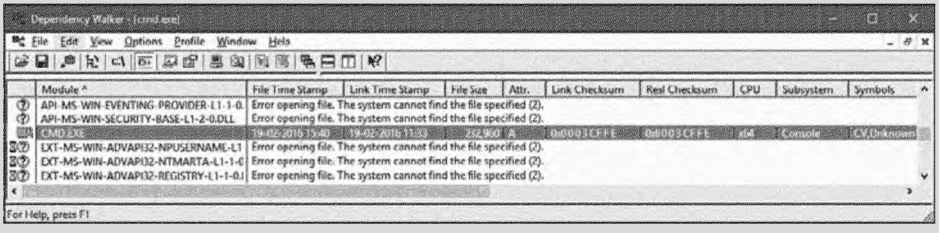
Как упоминалось ранее, пользовательские приложения не вызывают системные функции Windows напрямую. Вместо этого они проходят через одну или несколько DLL подсистем. Эти библиотеки экспортируют документированный интерфейс, который может вызываться программами, скомпонованными с этой подсистемой. Например, DLL-библиотеки подсистемы Windows (такие, как Kernel32.dll, Advapi32.dll, User32.dll и Gdi32.dll) реализуют функции Windows API. DLL-библиотека подсистемы SUA (Psxdll.dll) используется для реализации функций SUA API (в версиях Windows с поддержкой POSIX).

!

ЭКСПЕРИМЕНТ: ПРОСМОТР ТИПА ПОДСИСТЕМЫ ОБРАЗА

Для просмотра типа подсистемы образа можно воспользоваться программой Dependency Walker (Depends.exe). Например, обратите внимание на типы двух разных образов Windows, Notepad.exe (простой текстовый редактор) и Cmd.exe (командная строка Windows):





Как видите, Блокнот (Notepad) является программой с графическим интерфейсом, а Cmd — консольной (текстовой) программой. Может показаться, что для графических и текстовых программ используются две разные подсистемы, но в действительности подсистема всего одна, а графические программы могут создавать консоли (вызовом функции `AllocConsole`) подобно тому, как консольные программы могут отображать графический интерфейс.

Когда приложение вызывает функцию из DLL подсистемы, возможны три варианта:

- Функция полностью реализована в пользовательском режиме в DLL подсистемы. Другими словами, процессу подсистемы среды сообщение не посылается, и сервисные функции исполнительной системы Windows не вызываются. Функция выполняется в пользовательском режиме, а результаты возвращаются вызывающей стороне. К числу таких функций относятся `GetCurrentProcess` (всегда возвращает `-1` — значение, которым обозначается текущий процесс во всех функциях, связанных с процессами) и `GetCurrentProcessId`. (Идентификатор работающего процесса не изменяется, поэтому значение читается из кэша, чтобы обойтись без вызова ядра.)
- Функция требует одного или нескольких вызовов к исполнительной среде Windows. Например, функции Windows `ReadFile` и `WriteFile` требуют вызова

внутренних (и недокументированных для использования из пользовательского режима) системных функций ввода/вывода Windows `NtReadFile` и `NtWriteFile` соответственно.

- Функция требует выполнения некоторой работы в процессе подсистемы среды. (Процессы подсистемы среды, работающие в пользовательском режиме, отвечают за поддержание состояния клиентских приложений, работающих под их управлением.) В этом случае запрос к подсистеме среды производится при помощи сообщения ALPC (см. главу 2 части 2), которое отправляется подсистеме для выполнения некоторой операции. DLL подсистемы ожидает ответа перед тем, как возвращать управление на сторону вызова.

Некоторые функции представляют собой комбинации второго и третьего пункта — как, например, функции Windows `CreateProcess` и `ExitWindowsEx`.

Запуск подсистемы

Подсистемы запускаются процессом диспетчера сеансов (Smss.exe). Информация для запуска подсистемы хранится в разделе реестра HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\SubSystems. На рис. 2.5 показаны значения из этого раздела (для Windows 10 Pro).

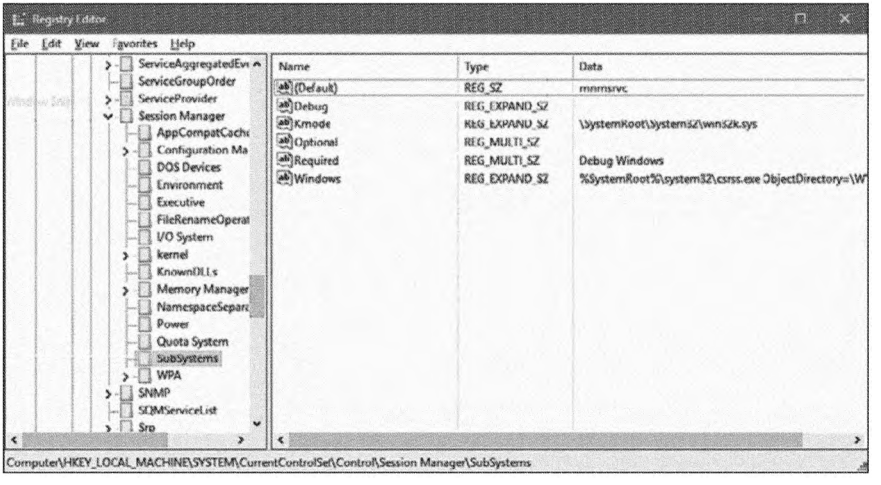


Рис. 2.5. Информация подсистемы в редакторе реестра Windows

В параметре Required перечисляются подсистемы, загружаемые при загрузке системы. Значение состоит из двух строк: `Windows` и `Debug`. Параметр Windows определяет файл подсистемы Windows Csrss.exe (*Client/Server Runtime Subsystem*). Параметр `Debug` пуст (это значение не используется после Windows XP, но значение остается в реестре для обеспечения совместимости), он ничего не делает.

Параметр Optional описывает необязательные подсистемы; в данном случае он также пуст, потому что подсистема SUA в Windows 10 не поддерживается. Если бы она была доступна, то значение данных Posix указывало бы на другой параметр, ссылающийся на файл Psxss.exe (процесс подсистемы POSIX). Параметр Optional означает «загружаемый по требованию», т. е. в первый раз, когда будет обнаружен образ POSIX. Параметр реестра Kmode содержит имя файла, содержащего часть подсистемы Windows режима ядра — Win32k.sys (см. далее в этом разделе).

А теперь поближе познакомимся с подсистемами среды Windows.

Подсистема Windows

Хотя система Windows проектировалась с расчетом на поддержку нескольких независимых подсистем среды, с практической точки зрения реализация всего кода для работы с окнами и экранного ввода/вывода в каждой подсистеме привела бы к появлению большого количества дублирующихся системных функций, что в конечном итоге отрицательно сказалось бы как на размере, так и на быстродействии системы. Так как Windows является основной подсистемой, проектировщики Windows решили, что базовые функции будут размещаться в ней, а другие подсистемы будут обращаться с вызовами к подсистеме Windows для выполнения экранного ввода/вывода. Таким образом, подсистема SUA вызывает сервисные функции из подсистемы Windows для выполнения операций экранного ввода/вывода.

В результате этого проектировочного решения подсистема Windows является необходимым компонентом для любой системы Windows — даже в серверных системах, не предусматривающих входа интерактивного пользователя. Из-за этого процесс помечается как критичный (а это означает, что если он завершится по какой-либо причине, в системе происходит фатальный сбой).

Подсистема Windows состоит из следующих основных компонентов:

- Для каждого сеанса экземпляр процесса подсистемы среды (Csrss.exe) загружает четыре DLL-библиотеки (Basesrv.dll, Winsrv.dll, Sxssrv.dll и Csrsrv.dll), которые обеспечивают поддержку следующей функциональности:
  - Различные служебные задачи, связанные с созданием и удалением процессов и потоков.
  - Завершение работы приложений Windows (через API `ExitWindowsEx`).
  - Файл .ini с аналогами различных параметров реестра для обеспечения обратной совместимости.
  - Отправка уведомлений ядра (например, от диспетчера Plug-and-Play) приложениям Windows в виде сообщений Windows (`WM_DEVICECHANGE`).
  - Компоненты поддержки процессов 16-разрядной виртуальной машины DOS (VDM) (только 32-разрядная версия Windows).

- Поддержка SxS (Side-by-Side)/Fusion и кэша манифеста.
- Некоторые функции поддержки естественных языков для организации кэширования.

ПРИМЕЧАНИЕ Пожалуй, самый важный момент заключается в том, что код режима ядра, обеспечивающий работу потока низкоуровневого ввода и потока рабочего стола (отвечающий за указатель мыши, ввод с клавиатуры и работу с окном рабочего стола), находится в потоках, работающих внутри Winsrv.dll. Кроме того, экземпляры Csrss.exe, связанные с интерактивными пользовательскими сеансами, содержат пятую DLL-библиотеку, которая называется «каноническим драйвером экрана», или CDD (Canonical Display Driver) (Cdd.dll). CDD отвечает за взаимодействие с поддержкой DirectX в ядре (см. далее) при каждом обновлении вертикальной развертки (VSync) для перерисовки видимого состояния рабочего стола без традиционной поддержки GDI с аппаратным ускорением.

- Драйвер устройства режима ядра (Win32k.sys), который содержит следующие компоненты:
  - диспетчер окон, который управляет отображением окон; управляет экранным выводом; получает данные от клавиатуры, мыши и других устройств и передает пользовательские сообщения приложениям;
  - интерфейс графических устройств (GDI, Graphic Device Interface) – библиотека функций для устройств графического вывода; включает функции вывода текста, рисования линий и геометрических фигур, а также манипуляций с графическими объектами;
  - обертки для поддержки DirectX, реализованной в другом драйвере ядра (Dxgkrnl.sys).
- Процесс консольного хоста (Conhost.exe), предоставляющий поддержку консольных приложений.
- Диспетчер окон рабочего стола (Desktop Window Manager) (Dwm.exe), позволяющий формировать видимое изображение окна на одной поверхности через CDD и DirectX.
- DLL подсистемы (такие, как Kernel32.dll, Advapi32.dll, User32.dll и Gdi32.dll), преобразующие документированные функции Windows API в соответствующие им вызовы недокументированных (для пользовательского режима) системных функций Ntoskrnl.exe и Win32k.sys.
- Драйверы графических устройств для аппаратно-зависимых драйверов экрана, драйверов принтеров и драйверы для мини-портов видео.

ПРИМЕЧАНИЕ В результате инициативы по рефакторингу архитектуры Windows, которая называлась MinWin, DLL подсистемы обычно строятся из конкретных библиотек, реализующих наборы API-функций, которые затем компонуются в DLL подсистемы и обрабатываются с использованием специальной схемы перенаправления. Подробнее об этой инициативе см. в разделе «Загрузчик образов» главы 3.

Windows 10 и Win32k.sys

Базовые требования к управлению окнами на устройствах с Windows 10 сильно зависят от самого устройства. Например, полноценной настольной системе с Windows необходимы все функции диспетчера окон, такие как окна с изменением размеров, родительские окна, дочерние окна и т. д. Версиям Windows Mobile 10 на телефонах и малых планшетах многие из этих функций не нужны, потому что на переднем плане находится окно, которое нельзя свернуть. Нельзя также изменить его размер и т. д. То же относится к устройствам IoT, у которых экрана может вообще не быть.

По этим причинам функциональность Win32K.sys была распределена между несколькими модулями ядра, так что не все модули необходимы в конкретной системе. Это приводит к значительному сокращению «поверхности атаки» диспетчера окон за счет сокращения сложности кода и исключения многих унаследованных фрагментов. Несколько примеров:

- На телефонах (Windows Mobile 10) Win32k.sys загружает Win32kMin.sys и Win32kBase.sys.
- На полноценных настольных системах Win32k.sys загружает Win32kBase.sys и Win32kFull.sys.
- На некоторых системах IoT Win32k.sys может быть достаточно Win32kBase.sys.

Приложения вызывают стандартные функции USER для создания на экране элементов пользовательского интерфейса, таких как окна и кнопки. Диспетчер окон передает эти запросы GDI, что приводит к их передаче драйверам графических устройств, где они форматируются для устройства графического вывода. Драйвер экрана на пару с драйвером мини-порта видео завершает поддержку вывода на экран.

GDI предоставляет набор стандартных двумерных функций, которые позволяют приложениям взаимодействовать с графическими устройствами, ничего не зная о самих устройствах. Функции GDI играют роль посредника между приложениями и графическими устройствами, такими как драйверы экрана и драйверы принтеров. GDI интерпретирует запросы приложения на графический вывод и отправляет запросы драйверам графических устройств. Кроме того, он предоставляет стандартный интерфейс для приложений, которые используют различные устройства графического вывода. Этот интерфейс позволяет коду приложения действовать независимо от аппаратных устройств и их драйверов. GDI адаптирует свои приложения для возможностей устройства, часто разделяя запрос на несколько более удобных частей. Например, некоторые устройства понимают прямые команды на рисование эллипса; другие требуют, чтобы уровень GDI интерпретировал команду как серию пикселов, расположенных по некоторым координатам. Подробнее об архитектуре графики и видеодрайвера см. в разделе «Design Guide» главы «Display (Adapters and Monitors)» документации WDK.

Так как значительная часть подсистемы — и особенно функциональность экранного ввода/вывода — работает в режиме ядра, лишь немногие функции Windows в ко-

нечном итоге отправляют сообщения процессу подсистемы Windows: это функции создания и завершения процессов и потоков, а также функции назначения букв устройств DOS (как при использовании subst.exe).

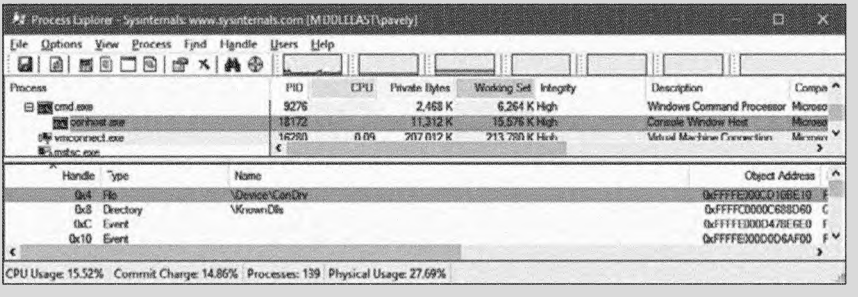
Как правило, работающее приложение Windows не создает многих (а возможно, даже немногих) переключений контекста в процесс подсистемы Windows, кроме необходимости перерисовки нового положения указателя мыши, обработки ввода с клавиатуры и прорисовки экрана через CDD.

ХОСТ КОНСОЛЬНОГО ОКНА

В исходной архитектуре подсистемы Windows процесс подсистемы (Csrss.exe) отвечал за управление консольными окнами, и каждое консольное приложение (такое, как командная строка Cmd.exe) взаимодействовало с Csrss.exe. В Windows 7 для каждого консольного окна в системе использовался отдельный процесс: хост консольного окна (Conhost.exe). (Одно консольное окно может совместно использоваться несколькими консольными приложениями, как, например, при запуске окна командной строки из командной строки. По умолчанию вторая командная строка совместно использует консольное окно первой.) Консольный хост Windows 7 подробно рассмотрен в главе 2 шестого издания этой книги.

В Windows 8 и последующих версиях консольная архитектура снова изменилась. Процесс Conhost.exe остался, но он теперь порождается от консольного процесса (вместо Csrss.exe, как в Windows 7) драйвером консоли (\Windows\System32\Drivers\ConDrv.sys). Этот процесс взаимодействует с Conhost.exe через драйвер консоли (ConDrv.sys), отправляя запросы чтения, записи, управления вводом/выводом и другие типы запросов ввода/вывода. Conhost.exe играет роль сервера, а процесс, использующий консоль, — роль клиента. Это изменение избавляет Csrss.exe от необходимости получать ввод с клавиатуры (как часть потока низкоуровневого ввода), отправлять его через Win32k.sys процессу Conhost.exe, а затем использовать ALPC для отправки его Cmd.exe. Вместо этого приложение командной строки может напрямую получать ввод от драйвера консоли через операции чтения/записи, избегая лишних переключений контекста.

На следующем экране Process Explorer показан дескриптор, который Conhost.exe держит открытым для объекта устройства, предоставляемого ConDrv.sys, с именем \Device\ConDrv. (Подробнее об именах устройств и вводе/выводе см. в главе 6.)



Обратите внимание: Conhost.exe является дочерним процессом для консольного процесса (в данном случае Cmd.exe). Создание Conhost инициируется загрузчиком образа для образов подсистемы Console или по требованию, если образ подсистемы GUI вызывает функцию Windows API `AllocConsole`. (Конечно, GUI и Console в каком-то смысле похожи: и то и другое — разновидности подсистем Windows.) Настоящей «рабочей силой» Conhost.exe является загружаемая DLL-библиотека (\Windows\System32\ConhostV2.dll), которая включает основной код, взаимодействующий с драйвером консоли.

Другие подсистемы

Как упоминалось ранее, система Windows изначально поддерживала подсистемы POSIX и OS/2. Так как эти подсистемы теперь в Windows не включаются, в книге они не рассматриваются. Общая концепция подсистем при этом остается в силе, а система может быть расширена новыми подсистемами, если такая необходимость вдруг возникнет в будущем.

Поставщики Pico и подсистема Windows для Linux

Традиционная модель подсистем расширяема; она, очевидно, обладает достаточной мощностью, если могла поддерживать POSIX и OS/2 в течение десятилетия, но у нее есть два технических недостатка, которые затрудняют широкое применение двоичных файлов, не относящихся к платформе Windows, за исключением нескольких специализированных ситуаций:

- Как упоминалось ранее, так как информация подсистем извлекается из заголовка PE (Portable Executable), для ее перестроения в формате исполняемого файла Windows PE (.exe) требуется исходный код исходного двоичного файла. При этом все системные вызовы и зависимости в стиле POSIX преобразуются в импортированные конструкции библиотеки Psxdll.dll.
- Модель ограничивается функциональностью, предоставляемой подсистемой Win32 (поверх которой она иногда работает) или ядром NT. Следовательно, подсистема инкапсулирует (вместо эмуляции) поведение, необходимое для приложений POSIX. Иногда это приводит к нетривиальным проблемам совместимости.

Наконец, следует заметить, что, как следует из названия, проектировщики подсистемы POSIX/SUA ориентировались на приложения POSIX/UNIX, которые доминировали на рынке серверов десятилетия назад, а не на приложения Linux, распространенные сегодня.

Решение этих проблем требовало нового подхода к построению подсистемы — подхода, который не требовал бы традиционной для пользовательского режима инкапсуляции других системных вызовов и исполнения традиционных образов PE. К счастью, проект Drawbridge от Microsoft Research предоставил идеальную

основу для обновленного подхода к созданию подсистем. Результатом стала реализация модели Pico.

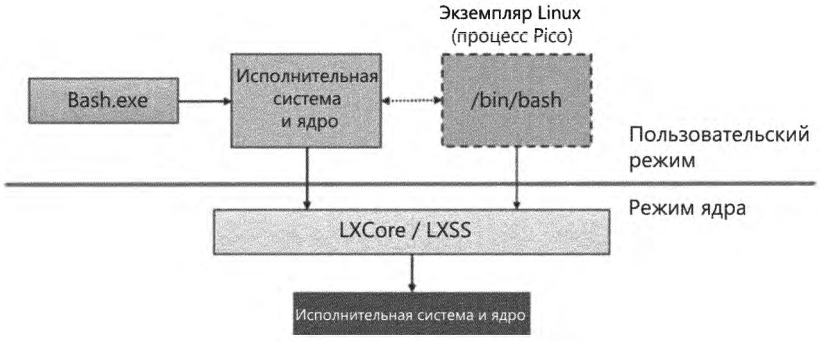
В этой концепции была определена концепция *поставщика Pico* — специального драйвера режима ядра, который получает доступ к специализированным интерфейсам ядра через API `PsRegisterPicoProvider`. Специализированные интерфейсы обладают двумя основными преимуществами:

- Они позволяют поставщику создавать процессы Pico и потоки с настройкой их контекстов исполнения и сегментов и сохранять данные в соответствующих структурах `EPROCESS` и `ETHREAD` (подробнее об этих структурах см. в главах 3 и 4).
- Они позволяют поставщику получать разнообразные уведомления, когда такие процессы или потоки участвуют в некоторых системных операциях: вызовах системных функций, исключениях, APC, ошибках страниц, завершении, смене контекста, приостановке/возобновлении и т. д.

В Windows 10 версии 1607 присутствует один поставщик Pico: Lxss.sys со своим «напарником» Lxcore.sys. Как подсказывает имя, это компонент подсистемы Windows для Linux (WSL), а эти драйверы образуют интерфейс поставщика Pico для него.

Так как поставщик Pico получает почти все возможные переходы в пользовательский режим и режим ядра и из них (будь то вызовы системных функций или исключения, например), при условии, что он распознает адресное пространство процесса Pico (или процессов), на базе которого он работает, а его код может исполняться в исходном формате, «настоящее» ядро не так уж важно — при условии, что эти переходы обрабатываются абсолютно прозрачно. Поэтому процессы Pico, работающие под управлением поставщика Pico WSL, сильно отличаются от обычных процессов Windows (см. главу 3) — например, у них нет библиотеки Ntdll.dll, которая всегда загружается в обычных процессах. Вместо этого их память содержит такие структуры, как vDSO — специальный образ, встречающийся только в системах Linux/BSD.

Более того, если процессы Linux должны работать прозрачно, они должны выполняться без перекомпиляции в исполняемый формат Windows PE. Так как ядро Windows не знает, как отображать другие типы образов, эти образы не могут запускаться через API `CreateProcess` процессами Windows и никогда не вызывают такие API сами (потому что понятия не имеют о том, что выполняются в Windows). Такая поддержка совместимости предоставляется совместно поставщиком Pico и диспетчером LXSS, который представляет собой службу пользовательского режима. Первый реализует закрытый интерфейс, который используется им для взаимодействия с диспетчером LXSS. Второй реализует интерфейс на базе COM, который используется для взаимодействия со специальным процессом запуска, в настоящее время известным как Bash.exe, и с управляющим процессом Lxrun.exe. На следующей диаграмме представлен обзор компонентов, образующих WSL.



Реализация поддержки широкого спектра приложений Linux — серьезное дело. Linux использует сотни системных вызовов — почти столько же, сколько само ядро Windows. Хотя поставщик Pico может использовать существующие функциональные возможности Windows (значительная часть которых строилась для поддержки исходной подсистемы POSIX, как, например, поддержка `fork()`), в некоторых случаях функциональность приходится реализовывать заново. Например, хотя для хранения файлов используется реальная файловая система NTFS (а не EXTFS), поставщик Pico содержит полную реализацию файловой системы Linux VFS (Virtual File System), которая включает поддержку индексных узлов, inotify(), /sys, /dev и других аналогичных пространств имен на базе файловой системы Linux с соответствующим поведением. Кроме того, хотя поставщик Pico может использовать WSK (Windows Sockets for Kernel) для работы с сетью, в него включаются сложные обертки для реального поведения сокетов, чтобы он мог поддерживать сокеты доменов UNIX, сокеты Linux NetLink и стандартные интернет-сокеты.

В других случаях существующие средства Windows просто не обладали достаточной совместимостью, причем различия могли быть весьма неочевидными. Например, в Windows существует драйвер именованного канала (Npfs.sys), который поддерживает традиционный механизм каналов IPC. Тем не менее ряд нетривиальных отличий от каналов Linux приводил к нарушению работы приложений. Таким образом, требовалось заново реализовывать каналы для приложений Linux без использования драйвера ядра Npfs.sys.

На момент написания книги эта функциональность официально находилась в стадии бета-тестирования с возможностью значительных изменений, поэтому в книге внутреннее строение этой подсистемы не рассматривается. Впрочем, мы еще вернемся к процессам Pico в главе 3. Когда подсистема вырастет за пределы бета-версии, вероятно, вы увидите в MSDN официальную документацию и стабильные API для взаимодействия с процессами Linux из Windows.

Ntdll.dll

Ntdll.dll — специальная системная библиотека, предназначенная прежде всего для использования DLL подсистем и родных приложений. (В данном контексте

родными (native) называются образы, не привязанные ни к какой конкретной подсистеме.) Она содержит функции двух типов:

- Заглушки диспетчеризации для системных функций исполнительной системы Windows.
- Внутренние вспомогательные функции, используемые подсистемами, DLL подсистем и т. д.

Первая группа функций предоставляет интерфейс для системных функций Windows, которые могут вызываться из пользовательского режима. Существует более 450 таких функций: `NtCreateFile`, `NtSetEvent` и т. д. Как упоминалось ранее, многие возможности этих функций доступны через Windows API. (Впрочем, некоторые функции предназначены только для использования конкретными внутренними компонентами ОС.)

Для каждой из этих функций Ntdll.dll содержит одноименную точку входа. Код функции содержит команду для конкретной архитектуры, которая инициирует переход в режим ядра для вызова диспетчера системных сервисных функций. (Эта тема более подробно рассматривается в главе 8 части 2.) После проверки параметров этот диспетчер системных функций вызывает системную сервисную функцию режима ядра, которая содержит реальный код в Ntoskrnl.exe. Следующий эксперимент показывает, как выглядят такие функции.

ЭКСПЕРИМЕНТ: ПРОСМОТР КОДА ДИСПЕТЧЕРА СИСТЕМНЫХ СЕРВИСНЫХ ФУНКЦИЙ

Откройте версию WinDbg, которая соответствует архитектуре вашей системы (например, версию x64 в 64-разрядной Windows). Откройте меню File и выберите команду Open Executable. Перейдите в папку %SystemRoot%\System32 и выберите файл Notepad.exe.

Запускается программа Блокнот, а отладчик прерывается на исходной точке останова. Это происходит на очень ранней стадии жизни процесса, которую можно просмотреть командой k (call stacK). Вы увидите несколько функций, начинающихся с `Ldr` — признак загрузчика (LoaDeR) образов. Главная функция Notepad еще не выполнена, это означает, что окна Блокнота на экране вы не увидите.

Установите точку останова в `NtCreateFile` в Ntdll.dll (отладчик игнорирует регистр символов):

```
bp ntdll!ntcreatefile
```

Введите команду g (Go) или нажмите F5, чтобы разрешить продолжение выполнения. Отладчик останавливается практически сразу, а вывод на экране выглядит примерно так (x64):

```
Breakpoint 0 hit
ntdll!NtCreateFile:
00007ffa'9f4e5b10 4c8bd1        mov     r10,rcx
```

Возможно, имя функции будет отображаться в виде `ZwCreateFile`. `ZwCreateFile` и `NtCreateFile` относятся к одному символическому имени в пользовательском режиме. Теперь введите команду u (Unassembled), чтобы просмотреть несколько ближайших команд:

```
00007ffa'9f4e5b10 4c8bd1          mov     r10,rcx
00007ffa'9f4e5b13 b855000000      mov     eax,55h
00007ffa'9f4e5b18 f604250803fe7f01 test     byte ptr [SharedUserData+0x308
(00000000'7ffe0308)],1
00007ffa'9f4e5b20 7503            jne ntdll!NtCreateFile+0x15
(00007ffa'9f4e5b25)
00007ffa'9f4e5b22 0f05            syscall
00007ffa'9f4e5b24 c3              ret
00007ffa'9f4e5b25 cd2e            int 2Eh
00007ffa'9f4e5b27 c3              ret
```

В регистр EAX записывается номер системной сервисной функции (55 в шестнадцатеричной системе в данном случае) для этой ОС (Windows 10 Pro x64). Обратите внимание на команду `syscall`: это она переводит процессор в режим ядра с передачей управления диспетчеру системных сервисных функций, где значение EAX используется для выбора сервисной функции `NtCreateFile`. Также обратите внимание на проверку флага (1) со смещением 0x308 в SharedUserData (подробнее об этой структуре см. в главе 4). Если этот флаг установлен, то выполнение идет по другому пути с использованием команды `int` 2Eh. Если включить функцию Credential Guard VBS, описанную в главе 7, этот флаг будет установлен на вашей машине, потому что гипервизор может реагировать на команду `int` более эффективно, чем команда syscall, и это поведение более желательно для Credential Guard.

Как упоминалось ранее, в главе 8 части 2 этот механизм описан более подробно (как и поведение syscall и `int`). А пока попробуйте найти другие системные функции, такие как `NtReadFile`, `NtWriteFile` и `NtClose`.

В разделе «Обзор архитектуры безопасности на основе виртуализации» было показано, что приложения IUM могут использовать другой двоичный модуль, похожий на Ntdll.dll, который называется IumDll.dll. Эта библиотека также содержит системные функции, но с другими индексами. Если в вашей системе включен механизм Credential Guard, вы можете повторить описанный эксперимент: откройте меню File в WinDbg, выберите команду Open Crash Dump и выберите файл IumDll.dll. В следующем примере вывода обратите внимание на то, что у индекса системной функции старший бит установлен, а проверка `SharedUserData` не производится; для таких системных вызовов (которые называются защищенными системными вызовами) всегда используется команда `syscall`:

```
0:000> u iumdll!IumCrypto
iumdll!IumCrypto:
00000001'80001130 4c8bd1          mov     r10,rcx
00000001'80001133 b802000008      mov     eax,8000002h
00000001'80001138 0f05            syscall
00000001'8000113a c3              ret
```

Ntdll.dll также содержит множество вспомогательных функций, включая загрузчик образов (функции с префиксом `Ldr`), диспетчер кучи и коммуникационные функции процессов подсистемы Windows (функции с префиксом `Csr`). Ntdll.dll также включает общие библиотечные функции времени выполнения (функции с префиксом `Rtl`), поддержку отладки в режиме пользователя (функции с префиксом `DbgUi`), трассировку событий для Windows (функции с префиксом `Etw`), диспетчер асинхронных вызовов процедур (APC, Asynchronous Procedure Call) и диспетчер исключений. (APC, как и исключения, кратко рассматривается в главе 6, а более подробно в главе 8 части 2.)

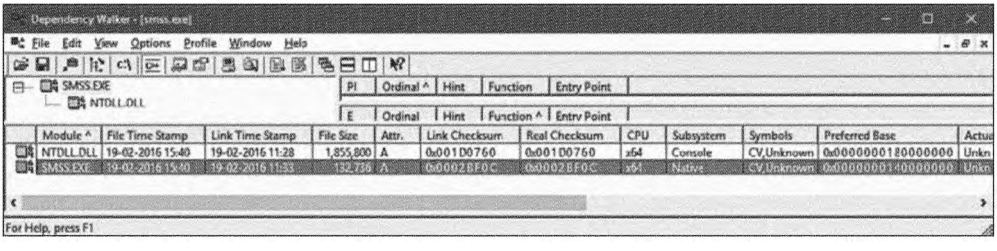
Наконец, в Ntdll.dll находится небольшое подмножество функций времени выполнения C (CRT, C Run-Time), ограниченное функциями, входящими в строковые и стандартные библиотеки (`memcpy`, `strcpy`, `sprintf` и т. д.); эти функции полезны для собственных приложений, описанных в следующем разделе.

Собственные образы

Некоторые образы (исполняемые модули) не принадлежат никакой подсистеме. Другими словами, они не компонуются с DLL подсистем, такими как Kernel32.dll для подсистемы Windows. Вместо этого они компонуются только с Ntdll.dll — «наименьшим общим кратным» для всех подсистем. Так как собственный (native) API, предоставляемый Ntdll.dll, в основном недокументирован, такие образы обычно строятся только компанией Microsoft. Примером служит процесс диспетчера сеансов (Smss.exe, см. далее в этой главе). Smss.exe — первый создаваемый процесс пользовательского режима (он создается непосредственно ядром), поэтому он не может зависеть от подсистемы Windows, так как Csrss.exe (процесс подсистемы Windows) еще не запустился.

Собственно, Smss.exe отвечает за запуск Csrss.exe. Другой пример — утилита Autochk, которая иногда выполняется при запуске системы для проверки дисков. Так как она выполняется на относительно ранней стадии загрузки (Autochk запускается Smss.exe), она не может зависеть от какой-либо подсистемы.

Ниже приведена информация Smss.exe в Dependency Walker, из которой видно, что программа зависит только от Ntdll.dll. Обратите внимание: в качестве типа подсистемы указано значение `Native`.



Исполнительная система

Исполнительная система образует верхний уровень Ntoskrnl.exe. (Ядро — нижний уровень.) Он включает следующие типы функций.

- **Функции, экспортируемые и вызываемые из пользовательского режима.** Эти функции, называемые системными сервисными функциями, экспортируются через Ntdll.dll (как функция `NtCreateFile` из предыдущего эксперимента). Большая часть сервисных функций доступна через Windows API или API другой подсистемы среды. Тем не менее некоторые сервисные функции недоступны ни через какие документированные функции подсистем. Примеры — функции ALPC и различные функции получения информации (такие, как `NtQueryInformationProcess`), специализированные функции (такие, как `NtCreatePagingFile`) и т. д.
- **Функции драйверов устройств, вызываемые через функцию DeviceIoControl.** Эта функция предоставляет общий интерфейс из пользовательского режима в режим ядра для вызова функций драйверов устройств, не связанных с чтением или записью. Хорошими примерами служат драйверы, используемые программами Process Explorer и Process Monitor из пакета Sysinternals, а также упоминавшийся ранее драйвер консоли (ConDrv.sys).
- **Функции, которые вызываются только из режима ядра — экспортируемые и документируемые в WDK.** К их числу относятся различные вспомогательные функции, включая функции диспетчера ввода/вывода (с префиксом `Io`), общие функции исполнительной системы (`Ex`) и другие функции, необходимые для разработчиков драйверов устройств.
- **Функции, которые вызываются только из режима ядра, но не документированы в WDK.** К их числу относятся функции, вызываемые видеодрайвером загрузки (функции с префиксом `Inbv`).
- **Функции, которые определяются как глобальные символические имена, но не экспортируются.** К их числу относятся внутренние вспомогательные функции, вызываемые из Ntoskrnl.exe, в числе которых функции с префиксами `Iop` (внутренние вспомогательные функции диспетчера ввода/вывода) или `Mi` (внутренние вспомогательные функции управления памятью).
- **Функции, внутренние по отношению к модулю, не определенные как глобальные символические имена.** Эти функции используются исключительно исполнительной системой и ядром.

Исполнительная система состоит из следующих основных компонентов, которые подробно рассматриваются в дальнейших главах этой книги.

- **Диспетчер конфигурации.** Диспетчер конфигурации, рассматриваемый в главе 9 части 2, отвечает за реализацию и управление системным реестром.

- **Диспетчер процессов.** Диспетчер процессов, рассматриваемый в главах 3 и 4, создает и завершает процессы и потоки. Поддержка процессов и потоков реализована в ядре Windows; исполнительная система добавляет дополнительную семантику и функции к этим низкоуровневым объектам.
- **SRM (Security Reference Monitor).** Монитор SRM, описанный в главе 7, обеспечивает соблюдение политик безопасности на локальном компьютере. Он реализует защиту ресурсов ОС, обеспечивая безопасность и аудит объектов.
- **Диспетчер ввода/вывода.** Диспетчер ввода/вывода, рассмотренный в главе 6, реализует аппаратно-независимый ввод/вывод и отвечает за диспетчеризацию запросов к соответствующим драйверам устройств для дальнейшей обработки.
- **Диспетчер PnP (Plug and Play).** Диспетчер PnP, описанный в главе 6, определяет, какие драйверы необходимы для поддержки конкретного устройства, и загружает эти драйверы. Он получает требования к аппаратным ресурсам для каждого устройства и затем на основании требований каждого устройства диспетчер PnP выделяет аппаратные ресурсы: порты ввода/вывода, IRQ, каналы DMA и блоки памяти. Он также отвечает за отправку уведомлений при изменении набора устройств (добавление или удаление устройства) в системе.
- **Диспетчер питания.** Диспетчер питания (см. главу 6), управление энергопотреблением процессора (PPM, Processor Power Management) и инфраструктура управления питанием (PoFx, Power management Framework) координируют события электропитания и генерируют уведомления управления питанием ввода/вывода к драйверам устройств. Когда система свободна, PPM можно настроить для сокращения энергопотребления посредством перевода ЦП в спящий режим. Изменения в энергопотреблении отдельных устройств обрабатываются драйверами устройств, но координируются диспетчером питания и PoFx. Для некоторых классов устройств терминальный диспетчер тайм-аута также управляет тайм-аутом физического вывода в зависимости от использования устройств.
- **Функции WMI (Windows Management Instrumentation), WDM (Windows Driver Model).** Эти функции, рассматриваемые в главе 9 части 2, разрешают драйверам устройств публиковать информацию быстродействия и конфигурации и получать команды от службы WMI пользовательского режима. Потребители информации WMI могут находиться на локальном компьютере или на удаленном компьютере в сети.
- **Диспетчер памяти.** Диспетчер памяти, рассмотренный в главе 5, реализует виртуальную память — схему управления памятью, которая предоставляет каждому процессу большое закрытое адресное пространство, размер которого превышает доступную физическую память. Диспетчер памяти также обеспечивает поддержку диспетчера кэша. В этом ему помогают система предварительной выборки (prefetcher) и диспетчер хранилища, которые также рассматриваются в главе 5.
- **Диспетчер кэша.** Диспетчер кэша, рассматриваемый в главе 14 «Диспетчер кэша» части 2, повышает быстродействие файлового ввода/вывода за счет хра-

нения часто используемых дисковых данных в основной памяти для быстрого доступа. Также операции записи на диск откладываются, а обновления некоторое время хранятся в памяти перед отправкой на диск. Как вы вскоре увидите, для этого используется поддержка отображаемых файлов диспетчером памяти.

Кроме того, исполнительная система содержит четыре основные группы вспомогательных функций, которые используются только что перечисленными компонентами. Около трети этих вспомогательных функций документировано в WDK, потому что они также используются драйверами устройств. Вспомогательные функции делятся на четыре категории.

- **Диспетчер объектов.** Диспетчер объектов создает, управляет и удаляет исполнительные объекты Windows и абстрактные типы данных, которые используются для представления ресурсов ОС, таких как процессы, потоки и различные объекты синхронизации. Диспетчер объектов рассматривается в главе 8 части 2.
- **Механизм ALPC (Asynchronous LPC).** Механизм ALPC, рассматриваемый в главе 8 части 2, передает сообщения между клиентским и серверным процессом на одном компьютере. Среди прочего, ALPC используется как локальный транспортный механизм удаленного вызова процедур (RPC) – Windows-реализации стандартного механизма коммуникаций между клиентскими и серверными процессами по сети.
- **Функции библиотек времени выполнения.** К их числу относятся функции обработки строк, арифметических операций, преобразований типов данных и операций со структурами безопасности.
- **Вспомогательные функции исполнительной системы.** Функции выделения системной памяти (из выгружаемого и невыгружаемого пула), доступа к памяти с блокировкой, а также специальным типам механизмов синхронизации – таким, как исполнительные ресурсы, быстрые мьютексы и пуш-блокировки.

Исполнительная система также содержит много других инфраструктурных функций, часть из которых лишь кратко упоминается в книге.

- **Библиотека отладчика ядра.** Обеспечивает возможность отладки ядра из отладчика, поддерживающего KD — портируемый протокол, поддерживаемый на разных видах транспорта, таких как USB, Ethernet и IEEE 1394, и реализуемый отладчиками WinDbg и Kd.exe.
- **Отладочная инфраструктура пользовательского режима.** Отвечает за отправку событий отладочному API пользовательского режима, позволяет назначать точки останова и выполнять код в пошаговом режиме, а также изменять контексты выполняемых потоков.
- **Библиотека гипервизора и библиотека VBS.** Предоставляют поддержку ядра для среды защищенной виртуальной машины и оптимизации некоторых частей кода, когда система знает, что она выполняется в клиентском разделе (виртуальной среде).

- **Диспетчер ошибок.** Диспетчер ошибок предоставляет обходные решения для нестандартных физических устройств.
- **Диспетчер проверки драйверов.** Диспетчер проверки драйверов реализует необязательные проверки целостности драйверов и кода режима ядра (см. главу 6).
- **Трассировка событий Windows (ETW, Event Tracing for Windows).** ETW предоставляет вспомогательные функции для общесистемной трассировки событий для режима ядра и пользовательского режима.
- **WDI (Windows Diagnostic Infrastructure).** WDI обеспечивает интеллектуальную трассировку системной активности на основании диагностических сценариев.
- **Вспомогательные функции WHEA (Windows Hardware Error Architecture).** Функции предоставляют общую инфраструктуру для сообщений об ошибках.
- **Библиотека файловой системы времени выполнения (FSRTL, File-System Runtime Library).** Предоставляет общие вспомогательные функции для драйверов файловой системы.
- **KSE (Kernel Shim Engine).** KSE предоставляет обертки совместимости драйверов и дополнительную поддержку ошибок устройств. Для этого используется база данных и инфраструктура оберток совместимости, описанная в главе 8 части 2.

Ядро

Ядро состоит из набора функций Ntoskrnl.exe, предоставляющих фундаментальные механизмы системы. К их числу относятся сервисные функции планирования потоков и синхронизации, используемые исполнительными компонентами, и низкоуровневая поддержка, зависящая от аппаратной архитектуры, — диспетчеризация прерываний и исключений, зависящая от архитектуры процессора. Код ядра пишется в основном на C, а вставки на языке ассемблера резервируются для задач, требующих доступа к специализированным командам процессора и регистрам, недоступным напрямую из языка C.

Некоторые функции ядра (как и различные исполнительные вспомогательные функции, упомянутые в предыдущем разделе) документированы в WDK (чтобы найти их, поищите функции, имена которых начинаются с префикса `Ke`), потому что они необходимы для реализации драйверов устройств.

Объекты ядра

Ядро предоставляет низкоуровневую базу четко определенных примитивов ОС с предсказуемым поведением и механизмов, при помощи которых компоненты исполнительной системы более высокого уровня решают свои задачи. Ядро изолируется от остальной части исполнительной подсистемы, реализуя механизмы ОС и избегая принятия решений. Исполнительной системе предоставляются прак-

тически все решения, за исключением планирования потоков и диспетчеризации, которые реализуются ядром.

За пределами ядра исполнительная система представляет потоки и другие ресурсы совместного использования в виде объектов. С этими объектами связываются некоторые дополнительные затраты ресурсов, обусловленные политикой, — дескрипторы для выполнения операций, проверки безопасности для защиты и квоты ресурсов, уменьшаемые при их создании. Эти дополнительные затраты устраняются в ядре, которое реализует набор более простых объектов, называемых *объектами ядра*; эти объекты помогают ядру управлять централизованной обработкой и поддерживать создание исполнительных объектов. Большинство объектов исполнительного уровня инкапсулирует один или несколько объектов ядра, включая их атрибуты, определяемые ядром.

Один набор объектов ядра — так называемые *управляющие объекты* (control objects) — устанавливает семантику управления различными функциями ОС. В этот набор входит объект асинхронного вызова процедур (APC), объект отложенного вызова процедур (DPC) и несколько объектов, используемых диспетчером ввода/вывода, например объект прерывания.

Другой набор объектов ядра — так называемые объекты диспетчеризации — включает средства синхронизации, влияющие на планирование потоков или изменяющие его. К числу объектов диспетчеризации относятся объекты потока ядра, мьютекса (называемого *мутантом* (mutant) в терминологии ядра), события, пары событий ядра, семафора, таймера и таймера с ожиданием.

Исполнительная система использует функции ядра для создания экземпляров объектов ядра, для выполнения операций с ними и для построения более сложных объектов, которые она предоставляет в пользовательском режиме. Объекты более подробно рассматриваются в главе 8 части 2, а процессы и потоки — в главах 3 и 4 соответственно.

KPCR и KPRCB

Для хранения данных, относящихся к конкретному процессору, ядро использует структуру данных, которая называется *KPCR* (Kernel Processor Control Region). В KPCR хранится основная информация: таблица диспетчеризации прерываний процессора (IDT, Interrupt Dispatch Table), сегмент состояния задачи (TSS, Task State Segment) и глобальная таблица дескрипторов (GDT, Global Descriptor Table). Также она включает состояние контроллера прерываний, которое используется совместно с другими модулями, такими как драйвер ACPI и HAL. Для простоты обращений к KPCR ядро хранит указатель на структуру в регистре `fs` в 32-разрядных версиях Windows и в регистре `gs` в системах Windows x64.

KPCR также содержит вложенную структуру данных, называемую блоком управления процессором (KPRCB, Kernel Processor Control Block). В отличие от структуры KPCR, документированной для сторонних драйверов и других внутренних

компонентов ядра Windows, KPRCB — закрытая структура, используемая только кодом ядра в Ntoskrnl.exe. Она содержит следующие данные.

- Информация планирования (например, текущий, следующий поток и поток простоя, запланированные для выполнения на процессоре).
- База данных диспетчеризации для процессора, включающая очереди готовности для всех уровней приоритета.
- Очередь DPC.
- Производитель процессора и идентификационные данные: модель, выпуск, скорость и биты функциональных возможностей.
- Топология процессора и NUMA: информация об узлах, число ядер в пакете, число логических процессоров в ядре и т. д.
- Размеры кэшей.
- Информация учета времени (например, время DPC и прерываний).

KPRCB также содержит всю статистику процессора, в том числе:

- статистику ввода/вывода;
- статистику диспетчера кэша (см. главу 14 части 2);
- статистику DPC;
- статистику диспетчера памяти (см. главу 5).

Наконец, KPRCB иногда используется для хранения структур выравнивания границ кэша для каждого процессора, используемых для оптимизации доступа к памяти, особенно в NUMA-системах. Например, резервные списки невыгружаемой и выгружаемой памяти хранятся в KPRCB.

ЭКСПЕРИМЕНТ: ПРОСМОТР KPCR И KPRCB

Для просмотра содержимого KPCR и KPRCB можно воспользоваться командами отладчика ядра `!pcr` и `!prcb`. Если во втором случае не включить флаги, отладчик по умолчанию выводит информацию для процессора 0. Чтобы задать конкретный процессор, добавьте его номер после команды — например, `!prcb 2`. С другой стороны, первая команда всегда выводит информацию о текущем процессоре, который можно изменить в сеансе удаленной отладки. При локальной отладке для получения адреса KPCR можно воспользоваться расширением `!prc` с номером процессора и последующей заменой `@$pcr` этим адресом. Не используйте другие данные, отображаемые командой `!pcr`. Это расширение считается устаревшим и выводит некорректные данные. Следующий пример показывает, как выглядит вывод команд `dt nt!_KPCR @$pcr` и `!prcb` (Windows 10 x64):

```
lkd> dt nt!_KPCR @$pcr
   +0x000 NtTib            : _NT_TIB
   +0x000 GdtBase          : 0xfffff802'a5f4bfb0 _KGDTENTRY64
```

```
   +0x008 TssBase          : 0xfffff802'a5f4a000 _KTSS64
   +0x010 UserRsp          : 0x0000009b'1a47b2b8
   +0x018 Self             : 0xfffff802'a280a000 _KPCR
   +0x020 CurrentPrcb      : 0xfffff802'a280a180 _KPRCB
   +0x028 LockArray        : 0xfffff802'a280a7f0 _KSPIN_LOCK_QUEUE
   +0x030 Used_Self        : 0x0000009b'1a200000 Void
   +0x038 IdtBase          : 0xfffff802'a5f49000 _KIDTENTRY64
   +0x040 Unused           : [2] 0
   +0x050 Irql             : 0 ''
   +0x051 SecondLevelCacheAssociativity : 0x10 ''
   +0x052 ObsoleteNumber   : 0 ''
   +0x053 Fill0            : 0 ''
   +0x054 Unused0          : [3] 0
   +0x060 MajorVersion     : 1
   +0x062 MinorVersion     : 1
   +0x064 StallScaleFactor : 0x8a0
   +0x068 Unused1          : [3] (null)
   +0x080 KernelReserved   : [15] 0
   +0x0bc SecondLevelCacheSize : 0x400000
   +0x0c0 HalReserved      : [16] 0x839b6800
   +0x100 Unused2          : 0
   +0x108 KdVersionBlock   : (null)
   +0x110 Unused3          : (null)
   +0x118 PcrAlign1        : [24] 0
   +0x180 Prcb             : _KPRCB
lkd> !prcb
PRCB for Processor 0 at fffff803c3b23180:
Current IRQL -- 0
Threads-- Current ffffe0020535a800 Next 0000000000000000 Idle fffff803c3b99740
Processor Index 0 Number (0, 0) GroupSetMember 1
Interrupt Count -- 0010d637
Times -- Dpc 000000f4 Interrupt 00000119
         Kernel 0000d952 User 0000425d
```

Вы также можете воспользоваться командой dt для получения непосредственного дампа структур данных _KPRCB, потому что команда отладчика предоставляет адрес структуры (в предшествующем выводе он выделен жирным шрифтом). Например, если вы хотите узнать скорость процессора, определенную на момент загрузки, обратитесь к полю MHz следующей командой:

```
lkd> dt nt!_KPRCB fffff803c3b23180 MHz
   +0x5f4 MHz : 0x893
lkd> ? 0x893
Evaluate expression: 2195 = 00000000'00000893
```

На этой машине на момент загрузки процессор работал приблизительно на частоте 2,2 ГГц.

Поддержка оборудования

Другая важная задача ядра — абстрагирование, или изоляция, исполнительной среды и драйверов устройств от различий между аппаратными архитектурами,

поддерживаемыми Windows. Эта задача подразумевает обработку различий в функциональности, таких как обработка прерываний, диспетчеризация исключений и синхронизация многопроцессорной обработки. Даже для этих функций, связанных с оборудованием, архитектура ядра стремится довести до максимума объем общего кода. Ядро поддерживает набор интерфейсов, портируемых и семантически идентичных между архитектурами. Большая часть кода, реализующего эти портируемые интерфейсы, также идентична между архитектурами.

Некоторые из этих интерфейсов по-разному реализуются в различных архитектурах или частично реализуются кодом, привязанным к конкретной архитектуре. Архитектурно независимые интерфейсы могут вызываться на любой машине, а семантика интерфейса одинакова, независимо от изменений кода, связанных с конкретной архитектурой. Некоторые интерфейсы ядра, такие как функции спинблокировки, описанные в главе 8 части 2, в действительности реализованы в HAL (см. следующий раздел), потому что их реализация может различаться в системах, принадлежащих к одному архитектурному семейству.

Ядро также содержит небольшой объем кода с интерфейсами для x86, необходимыми для поддержки старых 16-разрядных программ MS-DOS (в 32-разрядных системах). Эти интерфейсы x86 не являются портируемыми в том смысле, что они не могут вызываться на машине, основанной на любой другой архитектуре. Например, этот код, относящийся к x86, поддерживает вызовы для использования виртуального режима 8086, необходимые для эмуляции кода реального режима на старых видеокартах.

Другие примеры кода, привязанного к архитектуре в ядре, — интерфейсы, обеспечивающие поддержку буфера преобразования и кэша процессора. Эта поддержка требует разного кода для разных архитектур из-за различий в способах реализации кэша.

Еще один пример — переключение контекста. Хотя на высоком уровне абстракции для выбора потока и переключения контекста применяется один и тот же алгоритм (сохранить контекст предыдущего потока, загрузить контекст нового потока, запустить новый поток), в реализациях для разных процессоров существуют архитектурные различия. Так как контекст описывается состоянием процессора (регистры и т. д.), сохраняемые и загружаемые данные зависят от архитектуры.

Слой абстрагирования оборудования (HAL)

Как упоминалось в начале этой главы, одной из важнейших целей при проектировании Windows была возможность портирования между разнообразными аппаратными платформами. С появлением OneCore и бесчисленными форм-факторами существующих устройств она становится еще более важной. Слой абстрагирования оборудования, или HAL (Hardware Abstraction Layer), играет ключевую роль в обеспечении этой портируемости. HAL представляет собой загружаемый модуль режима ядра (Hal.dll), предоставляющий низкоуровневый интерфейс к аппаратной

платформе, на которой работает Windows. HAL скрывает подробности, зависящие от оборудования: интерфейсы ввода/вывода, контроллеры прерываний и механизмы многопроцессорных коммуникаций — словом, все аппаратно-зависимые функции, привязанные к конкретной архитектуре.

Итак, внутренние компоненты Windows и драйверы устройств, написанные пользователями, не работают напрямую с оборудованием, а для обеспечения портируемости вызывают функции HAL, когда им требуется информация, зависящая от платформы. По этой причине многие функции HAL документированы в WDK. За дополнительной информацией о HAL и использовании драйверов устройств обращайтесь к WDK.

И хотя в стандартный вариант установки Windows для настольной системы включена пара HAL для x86 (табл. 2.4), Windows может во время загрузки выбрать нужную версию HAL. Таким образом, устраняется проблема, существовавшая в предыдущих версиях Windows при попытке загрузить установленный экземпляр Windows в системе другого типа.

Таблица 2.4. Список HAL для x86

Имя файла HAL	Поддерживаемые системы
Halacpi.dll	Компьютеры с поддержкой ACPI (Advanced Configuration and Power Interface, «усовершенствованный интерфейс управления конфигурацией и питанием»). Используется для однопроцессорных машин без поддержки APIC. (Если хотя бы одно условие не выполняется, используется другая версия HAL в следующей строке.)
Halmacpi.dll	Компьютеры с поддержкой APIC (Advanced Programmable Interrupt Controller, «усовершенствованный программируемый контроллер прерываний») и ACPI. Существование APIC подразумевает поддержку SMP

На машинах на базе x64 и ARM существует только один образ HAL, называемый Hal.dll. Это объясняется тем, что все машины x64 имеют одинаковые конфигурации системной платы, потому что процессорам необходима поддержка ACPI и APIC. Следовательно, в поддержке машин без ACPI или стандартным контроллером PIC нет необходимости. Также все системы ARM поддерживают ACPI и используют контроллеры прерываний, как и в случае со стандартным APIC; в этом случае тоже хватает одного образа HAL.

С другой стороны, хотя контроллеры прерываний похожи, они не идентичны. Кроме того, фактически используемые таймеры и контроллеры памяти/DMA в некоторых системах ARM отличаются от других.

Наконец, в мире IoT некоторые стандартные устройства PC (например, контроллер DMA в архитектуре Intel) могут отсутствовать, и даже в системах на базе PC может потребоваться поддержка другого контроллера. Для решения этой проблемы в старых версиях Windows каждый производитель должен был поставлять специализированную версию HAL для каждой возможной комбинации платформ.

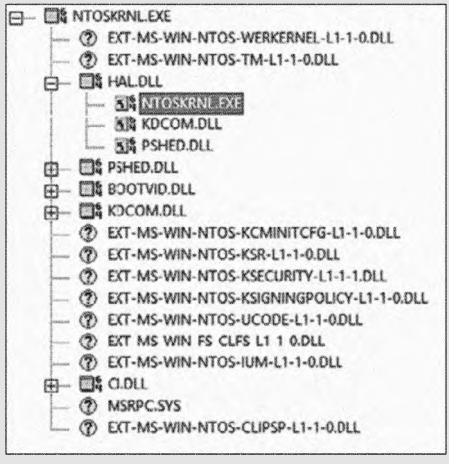
В наши дни такое решение, приводящее к значительным объемам дублирующегося кода, стало нереальным. Вместо этого Windows поддерживает модули, называемые *расширениями HAL*, — дополнительные DLL-библиотеки на диске, которые могут загружаться загрузчиком при наличии конкретного оборудования, запрашивающего эти библиотеки (обычно через ACPI и конфигурацию на основе реестра). Скорее всего, ваша настольная система с Windows 10 включает HalExtPL080.dll и HalExtIntcLpioDMA.dll; например, вторая библиотека используется на некоторых платформах Intel с пониженным энергопотреблением.

Создание расширений HAL требует сотрудничества с компанией Microsoft, и такие файлы должны снабжаться цифровой подписью с использованием специального сертификата расширения HAL, доступного только производителям оборудования. Кроме того, они сильно ограничиваются в возможностях использования API и взаимодействия через ограниченный механизм импорта/экспорта, не использующий традиционный механизм образов PE. Например, при попытке проведения с расширением HAL в следующем эксперименте никакие функции не отображаются.

ЭКСПЕРИМЕНТ: ПРОСМОТР NTOSKRNL.EXE И ЗАВИСИМОСТЕЙ ОБРАЗА HAL

Чтобы лучше понять отношения между ядром и образами HAL, воспользуйтесь программой Dependency Walker (Depends.exe) для изучения таблиц экспорта и импорта. Чтобы проанализировать образ в Dependency Walker, откройте меню File, выберите команду Open и выберите нужный файл образа.

Пример результата, который может быть получен при просмотре зависимостей Ntoskrnl.exe в этой программе (пока не обращайте внимания на ошибки, выводимые Dependency Walker из-за невозможности разобрать наборы API-функций):



Обратите внимание: Ntoskrnl.exe компонуется с библиотекой HAL, которая в свою очередь компонуется с Ntoskrnl.exe. (Они используют функции друг друга.) Ntoskrnl.exe также компонуется со следующими библиотеками:

- **Pshed.dll.** Платформенно-зависимый драйвер ошибок оборудования (PSHED, Platform-Specific Hardware Error Driver) предоставляет абстракцию средств сообщения об ошибках оборудования для используемой платформы. Для этого он скрывает подробности механизмов обработки ошибок платформы от ОС и предоставляет логически целостный интерфейс к ОС Windows.
- **Bootvid.dll.** Загрузочный видеодрайвер (Boot Video Driver) в системах x86 предоставляет поддержку команд VGA, необходимых для вывода загрузочных сообщений и логотипа во время запуска системы.
- **Kdcom.dll.** Коммуникационная библиотека протокола отладчика ядра (KD, Kernel Debugger Protocol).
- **Ci.dll.** Библиотека целостности (подробнее см. в главе 8 части 2).
- **Msrpc.sys.** Драйвер клиента Microsoft RPC (RPC) для режима ядра позволяет ядру (и другим драйверам) взаимодействовать с сервисными функциями пользовательского режима через RPC или же выполнять маршалинг ресурсов в кодировке MES. Например, этот драйвер используется ядром для обеспечения маршалинга данных к сервисным функциям PnP пользовательского режима и обратно.

Подробное описание информации, выводимой этой программой, приведено в справочном файле Dependency Walker (Depends.hlp).

Мы предложили не обращать внимания на ошибки, выдаваемые Dependency Walker при обработке наборов API-функций, потому что авторы не обновили программу для правильной обработки этого механизма. Хотя реализация наборов API-функций будет описана в главе 3 в разделе «Загрузчик образов», вы все равно можете использовать вывод Dependency Walker для анализа других потенциальных зависимостей ядра, так как эти наборы API-функций могут указывать на реальные модули. Следует заметить, что при описании наборов API-функций используется понятие контрактов, а не DLL или библиотек. Важно заметить, что любое число таких контрактов (и даже все они) на вашей машине может отсутствовать. Их присутствие зависит от комбинации факторов: выпуска ОС, платформы и производителя оборудования.

- **Контракт Werkernel.** Предоставляет поддержку сообщений об ошибках Windows (WER, Windows Error Reporting) в ядре, например, при оперативном создании дампа ядра.
- **Контракт Tm.** Диспетчер транзакций ядра (KTM), описанный в главе 8 части 2.
- **Контракт Kcminitcfg.** Отвечает за нестандартную исходную конфигурацию, которая может потребоваться на некоторых платформах.
- **Контракт Ksr.** Обеспечивает горячую перезагрузку ядра (KSR, Kernel Soft Reboot) и сохранение некоторых областей памяти для ее поддержки, особенно на некоторых мобильных и IoT-платформах.
- **Контракт Ksecurity.** Содержит дополнительные политики для процессов контейнеров приложений (т. е. приложений Windows Apps), работающих в пользовательском режиме на некоторых устройствах и выпусках Windows.

- **Контракт Ksigningpolicy.** Содержит дополнительные политики для целостности кода пользовательского режима (UMCI, User-Mode Code Integrity) либо для поддержки процессов, не являющихся процессами контейнеров приложений в некоторых выпусках, либо для дополнительной настройки Device Guard и/или средств безопасности App Locker на некоторых платформах/выпусках Windows.
- **Контроллер Ucode.** Библиотека обновления микрокода для платформ, которые могут поддерживать обновления микрокода процессоров (таких, как Intel и AMD).
- **Контроллер Clfs.** Драйвер CLFS (Common Log File System), используемый (среди прочего) транзакционным реестром (TxR, Transactional Registry). Подробнее о TxR читайте в главе 8 части 2.
- **Контракт Ium.** Дополнительные политики для трастлетов IUM, работающих в системе, которые могут понадобиться в некоторых выпусках Windows: например, для реализации защищенных виртуальных машин в Datacenter Server. Трастлеты рассматриваются в главе 3.

Драйверы устройств

Драйверы устройств подробно рассматриваются в главе 6, а в этом разделе дается краткий обзор типов драйверов. Также мы расскажем, как получить список драйверов, установленных и загруженных в вашей системе.

Windows поддерживает драйверы режима ядра и драйверы пользовательского режима, но в этом разделе рассматриваются только драйверы ядра. Термин «драйвер устройства» подразумевает физические устройства, но существуют и другие типы драйверов устройств, не связанные с оборудованием напрямую (их список приводится ниже). В этом разделе мы сосредоточимся на драйверах устройств, предназначенных для управления физическими устройствами.

Драйверы устройств представляют собой загружаемые модули режима ядра (файлы, которые обычно имеют суффикс .sys), которые обеспечивают интерфейс между диспетчером ввода/вывода и соответствующим оборудованием. Они выполняются в режиме ядра в одном из трех контекстов.

- В контексте пользовательского потока, инициирующего функцию ввода/вывода (такие, как операция чтения).
- В контексте системного потока режима ядра (такие, как запросы от диспетчера PnP).
- В результате прерывания, а следовательно, не в контексте какого-то конкретного потока, а в контексте того потока, который был текущим в момент возникновения прерывания.

Как упоминалось в предыдущем разделе, драйверы устройств в Windows не работают с оборудованием напрямую, а вызывают для взаимодействия с ним функции

HAL. Драйверы обычно пишутся на C и/или C++. Таким образом, при правильном использовании функций HAL драйверы могут быть портируемыми между архитектурами процессоров, поддерживаемыми Windows, на уровне исходного кода и портируемыми на двоичном уровне в пределах семейства архитектур.

Существует несколько типов драйверов устройств.

- **Драйверы физических устройств.** Драйверы этой категории используют HAL для работы с оборудованием, чтобы записывать вывод или получать ввод от физических устройств или из сети. Есть много типов драйверов физических устройств: драйверы шин, драйверы устройств, обеспечивающих интерфейс пользователя, драйверы запоминающих устройств большой емкости и т. д.
- **Драйверы файловой системы.** Драйверы Windows, получающие файлово-ориентированные запросы ввода/вывода и преобразующие их в запросы ввода/вывода для конкретного устройства.
- **Драйверы-фильтры файловой системы.** Драйверы, которые обеспечивают зеркалирование или шифрование дисков, выполняют сканирование для поиска вирусов, перехватывают запросы ввода/вывода и выполняют дополнительную обработку перед передачей ввода/вывода на следующий уровень (а в некоторых случаях блокируют операцию).
- **Сетевые перенаправители и серверы.** Драйверы файловой системы, передающие запросы ввода/вывода файловой системы на машину в сети и получающие такие запросы соответственно.
- **Драйверы протоколов.** Драйверы реализуют сетевые протоколы, такие как TCP/IP, NetBEUI и IPS/SPX.
- **Потоковые драйверы-фильтры ядра.** Эти драйверы объединяются в цепочки для цифровой обработки потоков данных (например, записи или воспроизведения аудио и видео).
- **Программные драйверы.** Модули ядра, выполняющие операции, которые могут быть выполнены только в режиме ядра, по поручению некоторого процесса пользовательского режима. Многие программы из пакета Sysinternals — такие, как Process Explorer и Process Monitor, — используют драйверы для получения информации или выполнения операций, которые невозможно выполнить из API пользовательского режима.

Модель драйвера Windows

Исходная модель драйвера, созданная в первой версии NT (3.1), не поддерживала концепцию PnP (Plug and Play), потому что тогда эта технология еще не была доступна. Ситуация изменилась только после выхода Windows 2000 (и Windows 95/98 на потребительской стороне Windows).

В Windows 2000 добавилась поддержка PnP, Power Options, а также расширение модели драйвера Windows NT, называемое WDM (Windows Driver Model). Унас-

ледованные драйверы Windows NT 4 могут работать в Windows 2000 и выше, но из-за отсутствия поддержки PnP и Power Options системы с такими драйверами будут обладать усеченной функциональностью в этих двух областях.

Изначально модель WDM предоставляла единую модель драйвера, которая была (почти) совместима на уровне исходного кода между Windows 2000/XP и Windows 98/ME. Это было сделано для того, чтобы упростить написание драйверов для физических устройств, поскольку вместо двух кодовых баз хватало одной. Модель WDM имитировалась в Windows 98/ME.

После того как эти операционные системы вышли из использования, модель WDM оставалась основной моделью для написания драйверов физических устройств для Windows 2000 и более поздних версий.

С точки зрения WDM драйверы делятся на три типа.

- **Драйверы шин.** Драйвер шины обслуживает контроллер шины, адаптер, мост или любое другое устройство, у которого имеются дочерние устройства. Драйверы шины являются обязательными, обычно их предоставляет компания Microsoft. У каждой разновидности типа шины в системе (например, PCI, PCMCIA и USB) имеется один драйвер шины. Третьи стороны могут писать драйверы шин, чтобы обеспечить поддержку новых шин, таких как VMEbus, Multibus и Futurebus.

- **Функциональные драйверы.** Функциональные драйверы — основная категория драйверов устройств, предоставляющих рабочий интерфейс к своему устройству. Этот драйвер необходим, если только устройство не используется в низкоуровневой реализации, при которой ввод/вывод осуществляется драйвером шины и драйверами-фильтрами шин (например, SCSI PassThru). Функциональный драйвер по определению располагает полной информацией о конкретном устройстве; обычно это единственный драйвер, который работает с регистрами устройства.

- **Драйверы-фильтры.** Драйвер-фильтр используется или для расширения функциональности устройства существующего драйвера, или для модификации запросов или ответов ввода/вывода от других драйверов. Драйверы-фильтры часто используются для исправления ответов оборудования, возвращающего некорректную информацию о требованиях к аппаратным ресурсам. Драйверы-фильтры являются необязательными; они могут существовать в любом количестве выше или ниже функционального драйвера и выше драйвера шины. Обычно драйверы-фильтры поставляются изготовителями комплектного оборудования (OEM, Original Equipment Manufacturer) или независимыми производителями оборудования (IHV, Independent Hardware Vendor).

В среде драйверов WDM ни один драйвер не управляет всеми аспектами работы устройства. Драйвер шины обеспечивает передачу информации устройств диспетчеру PnP, тогда как функциональный драйвер непосредственно работает с устройством.

В большинстве случаев низкоуровневые драйверы-фильтры изменяют поведение физического оборудования. Например, если устройство сообщает своему драйверу шины, что ему необходимо 4 порта ввода/вывода, тогда как в действительности ему нужно 16 портов, драйвер-фильтр более низкого уровня для этого устройства перехватывает список аппаратных ресурсов, переданный драйвером шины диспетчеру PnP, и обновляет количество портов ввода/вывода.

Высокоуровневые драйверы-фильтры обычно расширяют функциональность устройства. Например, высокоуровневый драйвер-фильтр диска может реализовать дополнительные проверки безопасности.

Обработка прерываний рассматривается в главе 8 части 2, а в узком контексте драйверов устройств — в главе 6. Подробнее о диспетчере ввода/вывода, WDM, Plug and Play и управлении питанием также см. в главе 6.

Windows Driver Foundation

WDF (Windows Driver Foundation) упрощает разработку драйверов Windows, предоставляя в распоряжение разработчика два фреймворка: KDMF (Kernel-Mode Driver Framework) и UMDF (User-Mode Driver Framework). Разработчики используют KMDF для написания драйверов для Windows 2000 SP4 и выше, тогда как UMDF поддерживает Windows XP и более поздние версии.

KMDF предоставляет простой интерфейс к WDM, скрывая сложность модели от создателя драйвера без модификации нижележащей модели драйвера шины/функционального драйвера/фильтра. Драйверы KMDF реагируют на события, которые они могут регистрировать, и обращаются с вызовами к библиотеке KMDF для выполнения работы, не привязанной к оборудованию, которым они управляют, такой как обобщенное управление питанием или синхронизация. (Ранее каждому драйверу приходилось реализовывать эти функции самостоятельно.) В отдельных случаях до 200 строк кода WDM можно заменить одним вызовом функции KMDF.

UMDF дает возможность реализовать некоторые классы драйверов: прежде всего шины на базе USB и других протоколов с высокой задержкой, используемых в видеокамерах, MP3-проигрывателях, сотовых телефонах и принтерах, — в виде драйверов пользовательского режима. С UMDF каждый драйвер пользовательского режима выполняется фактически в виде службы пользовательского режима; ALPC используется для взаимодействия с драйвером-оберткой пользовательского режима, который предоставляет непосредственный доступ к оборудованию. Если в драйвере UMDF происходит сбой, процесс завершается и обычно перезапускается. Таким образом, система не становится нестабильной; устройство просто остается недоступным на время перезапуска службы-хоста.

UMDF существует в двух основных версиях: версия 1.x доступна для всех версий ОС с поддержкой UMDF; последней является версия 1.11, доступная в Windows 10. Эта версия использует C++ и COM для написания драйверов, что удобно для программистов пользовательского режима, но из-за этого модель UMDF отличается

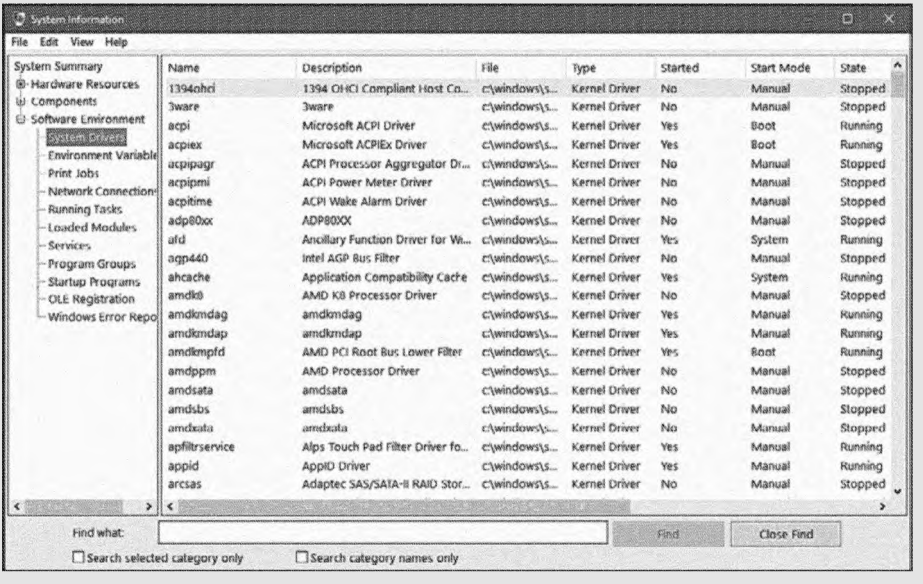
от KMDF. Версия 2.0 UMDF, появившаяся в Windows 8.1, базируется на той же объектной модели, что и KMDF, поэтому модели программирования двух фреймворков очень похожи. Наконец, инструментарий WDF был переведен Microsoft на модель с открытым кодом; на момент написания книги он был доступен на GitHub по адресу *https://github.com/Microsoft/Windows-Driver-Frameworks*.

Универсальные драйверы Windows

Начиная с Windows 10, термином «универсальные драйверы Windows» обозначается возможность написания драйверов устройств, использующих API и интерфейсы драйверов устройств (DDI, Device Driver Interface), предоставляемые общим ядром Windows 10. Эти драйверы обладают двоичной совместимостью для конкретной архитектуры процессора (x86, x64, ARM) и могут использоваться в неизменном виде для разных форм-факторов, от устройств IoT до телефонов, от HoloLens и Xbox One до портативных и настольных компьютеров. Универсальные драйверы могут использовать в качестве своей модели драйвера KMDF, UNDF 2.x или WDM.

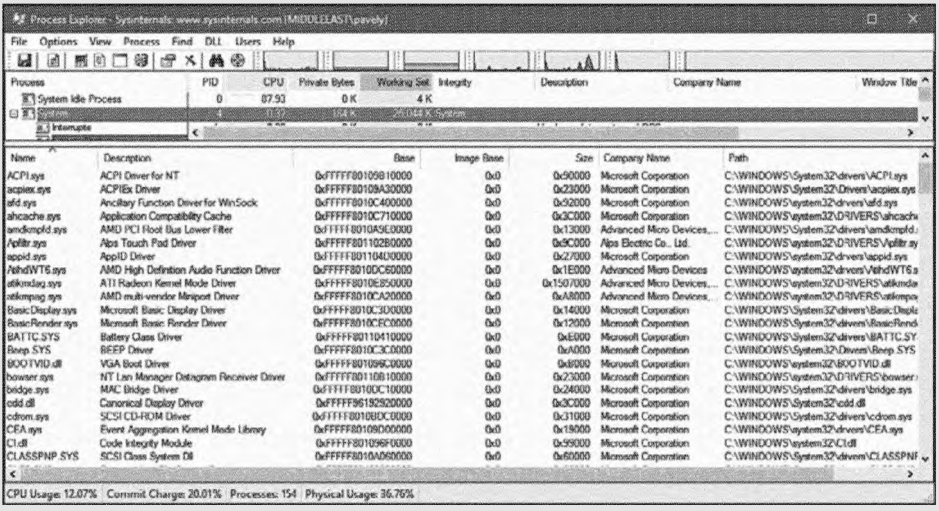
ЭКСПЕРИМЕНТ: ПРОСМОТР УСТАНОВЛЕННЫХ ДРАЙВЕРОВ УСТРОЙСТВ

Чтобы получить список установленных драйверов, запустите программу Сведения о системе (System Information) (Msinfo32.exe). Чтобы запустить программу, щелкните на кнопке Пуск (Start) и введите имя программы Msinfo32. В списке Сведения о системе раскройте узел Программная среда (Software Enviroment) и откройте раздел Системные драйверы (System Drivers). Пример вывода списка установленных драйверов:



В окне выводится список драйверов устройств, определенных в реестре, с указанием типа и состояния (работает/остановлен). Драйверы устройств и процессы служб Windows определяются в одном месте: HKLM\SYSTEM\CurrentControlSet\Services. При этом они различаются по коду типа. Например, тип 1 обозначает драйвер устройства режима ядра. За полным списком информации, хранимой в реестре для драйверов устройств, обращайтесь к главе 9 части 2.

Также для просмотра текущего списка загруженных драйверов можно выбрать процесс System в программе Process Explorer и открыть режим представления DLL. Ниже приведен пример вывода. (Чтобы открыть дополнительные столбцы, щелкните правой кнопкой мыши на заголовке столбца и выберите команду Select Columns, чтобы просмотреть все возможные столбцы для модулей на вкладке DLL.)



НЕДОКУМЕНТИРОВАННЫЕ ИНТЕРФЕЙСЫ

Просмотр имен экспортируемых или глобальных символических имен в ключевых системных образах (таких, как Ntoskrnl.exe, Hal.dll или Ntdll.dll) может быть весьма поучительным: вы получите представление о том, на что способна система Windows, по сравнению с тем, что официально документировано и поддерживается сегодня. Конечно, то, что вы знаете имена этих функций, еще не значит, что вы можете вызывать их (или что вам стоит это делать) — интерфейсы не документированы, а значит, подвержены изменениям. Мы рекомендуем взглянуть на эти функции просто для того, чтобы получить более полное представление о внутренней работе Windows, а не для того, чтобы работать в обход поддерживаемых интерфейсов.

Например, при просмотре списка функций в Ntdll.dll вы получите список всех системных сервисных функций, предоставляемых системой Windows DLL-библиотекам подсистем пользовательского режима, и сможете сравнить их

с подмножеством, предоставляемым каждой подсистемой. Хотя многие из этих функций однозначно соответствуют документированным и поддерживаемым функциям Windows, некоторые из них недоступны через Windows API.

И наоборот, будет интересно изучить импортируемые функции DLL подсистем Windows (такие, как Kernel32.dll или Advapi32.dll) и функции, вызываемые ими в Ntdll.dll.

Также заслуживает внимания образ Ntoskrnl.exe. Хотя многие из экспортируемых функций, используемых драйверами устройств режима ядра, документированы в WDK, есть достаточно много недокументированных. Возможно, вам будет интересно просмотреть таблицу импортирования Ntoskrnl.exe и HAL; в этой таблице перечислены функции HAL, используемые Ntoskrnl.exe, и наоборот.

В табл. 2.5 приведены наиболее распространенные префиксы имен функций для исполнительных компонентов. Каждый из основных исполнительных компонентов также использует модификацию префикса для обозначения внутренних функций — либо за первой буквой префикса следует буква `i` (от «internal», т. е. «внутренний»), либо за всем префиксом следует буква `p` (от «private», т. е. «закрытый»). Например, префикс `Ki` представляет внутренние функции ядра, а префикс `Psp` — внутренние вспомогательные функции процессов.

Таблица 2.5. Самые распространенные префиксы

Префикс	Компонент
`Alpc`	Расширенные локальные вызовы процедур
`Cc`	Общий кэш
`Cm`	Диспетчер конфигурации
`Dbg`	Поддержка отладки ядра
`Dbgk`	Отладочная инфраструктура для пользовательского режима
`Em`	Диспетчер ошибок
`Etw`	Трассировка событий для Windows
`Ex`	Исполнительные вспомогательные функции
`FsRtl`	Библиотека файловой системы времени выполнения
`Hv`	Библиотека Hive
`Hvl`	Библиотека гипервизора
`Io`	Диспетчер ввода/вывода
`Kd`	Отладчик ядра
`Ke`	Ядро
`Kse`	Оболочка совместимости ядра
`Lsa`	Локальная система безопасности
`Mm`	Диспетчер памяти

Префикс	Компонент
Nt	Системные сервисные функции NT (доступны из пользовательского режима через системные вызовы)
Ob	Диспетчер объектов
Pf	Система предварительной выборки
Po	Диспетчер питания
PoFx	Инфраструктура управления питанием
Pp	Диспетчер PnP
Ppm	Диспетчер управления питанием процессора
Ps	Поддержка процессов
Rtl	Библиотека времени выполнения
Se	SRM
Sm	Диспетчер хранилища
Tm	Диспетчер транзакций
Ttm	Диспетчер тайм-аута терминала
Vf	Проверка драйверов
Vsl	Библиотека виртуального безопасного режима
Wdi	Диагностическая инфраструктура Windows
Wfp	Windows Fingerprint
Whea	Windows Hardware Error Architecture
Wmi	Windows Management Instrumentation
Zw	Зеркальная точка входа для системных функций (начинающихся с Nt), которая назначает предыдущим режимом доступа режим ядра; тем самым предотвращается проверка параметров, потому что системные функции Nt проверяют параметры только в том случае, если предыдущим был пользовательский режим

Имена этих экспортируемых функций проще расшифровываются, если вы понимаете схему формирования имен системных функций Windows. Общий формат имен выглядит так:

<Префикс><Операция><Объект>

В этом формате *Префикс* — внутренний компонент, экспортирующий функцию, *Операция* сообщает, что происходит с объектом или ресурсом, а *Объект* идентифицирует то, с чем выполняется операция.

Например, `ExAllocatePoolWithTag` — исполнительная вспомогательная функция для выделения памяти из выгружаемого или невыгружаемого пула. А `KeInitializeThread` — функция для создания и инициализации объекта ядра для потока.

Системные процессы

Системные процессы, описанные ниже, присутствуют в каждой системе Windows 10. Один из них (Idle) процессом вообще не является; три других — System, Secure System и Memory Compression — не являются полноценными процессами, поскольку в них не работает исполняемый файл пользовательского режима. Такие процессы, называемые минимальными, описаны в главе 3.

- **Процесс Idle.** Содержит один поток для каждого процессора для учета времени бездействия процессора.
- **Процесс System.** Содержит большинство системных потоков и дескрипторов режима ядра.
- **Процесс Secure System.** Содержит адресное пространство безопасного ядра в VTL 1 (если работает).
- **Процесс Memory Compression.** Содержит сжатый рабочий набор процессов пользовательского режима (см. главу 5).
- **Диспетчер сеансов** (Smss.exe).
- **Подсистема Windows** (Csrss.exe).
- **Инициализация сеанса 0** (Wininit.exe).
- **Процесс входа** (Winlogon.exe).
- **Диспетчер служб** (Services.exe) и создаваемые им дочерние процессы (такие, как системный обобщенный процесс, выполняющий функции хоста служб (Svchost.exe).
- **Локальная служба аутентификации** (Lsass.exe), и если активен механизм Credential Guard — изолированный локальный сервер аутентификации (Lsaiso.exe).

Чтобы понять, как связаны эти процессы, полезно рассмотреть дерево процессов, т. е. иерархию отношений «родитель—потомок» между процессами. Если вы знаете, каким процессом был создан любой другой процесс, вам будет проще понять, откуда эти процессы взялись. На рис. 2.6 показано дерево процессов, построенное на основе загрузочной трассировки. Чтобы выполнить загрузочную трассировку, откройте меню Options в Process Monitor и выберите команду Enable Boot Logging. Затем перезапустите систему, снова откройте Process Monitor, откройте меню Tools и выберите команду Process Tree (или нажмите Ctrl+T). Благодаря Process Monitor вы видите процессы, которые уже завершились (они обозначены затемненным значком).

В следующих разделах описаны ключевые системные процессы, показанные на рис. 2.6. Хотя в этих разделах кратко указывается порядок запуска процесса, в главе 11 части 2 содержится подробное описание шагов, связанных с загрузкой и запуском Windows.

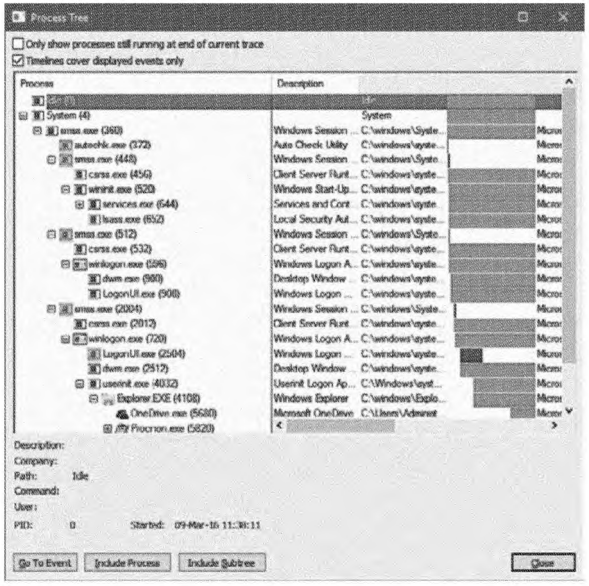


Рис. 2.6. Исходное дерево системных процессов

Процесс простоя

На первом месте в списке на рис. 2.6 стоит процесс Idle. Как будет показано в главе 3, процессы идентифицируются по именам своих образов. Однако этот процесс — как и процессы System, Secure System и Memory Compression processes — не выполняет реальный образ пользовательского режима. Иначе говоря, в каталоге \Windows нет файла с именем System Idle Process.exe. Кроме того, из-за подробностей реализации имя, показанное для этого процесса, отличается от программы к программе. Процесс Idle учитывает время бездействия в системе. По этой причине количество «потоков» в этом «процессе» равно количеству логических процессоров в системе. В табл. 2.6 перечислены некоторые имена, присваиваемые процессу Idle (идентификатор процесса 0). Процесс Idle подробно рассматривается в главе 3.

Таблица 2.6. Имена процесса с идентификатором 0 в разных программах

Программа	Имя процесса с идентификатором 0
Диспетчер задач	System Idle process
Process Status (Pstat.exe)	Idle process
Process Explorer (Procexp.exe)	System Idle process
Task List (Tasklist.exe)	System Idle process
Tlist (Tlist.exe)	System process

А теперь рассмотрим разные системные потоки и предназначение всех системных процессов, в которых выполняются реальные образы.

Процесс System и системные потоки

Процесс System (идентификатор процесса 4) объединяет потоки особого типа, выполняемые только в режиме ядра: *системные потоки режима ядра*. Системные потоки обладают всеми атрибутами и контекстами обычных потоков пользовательского режима (такими, как аппаратный контекст, приоритет и т. д.), но отличаются тем, что они выполняются только в режиме ядра с выполнением кода, загруженного в системном пространстве, будь то в Ntoskrnl.exe или в любом другом загруженном драйвере устройства. Кроме того, системные потоки не имеют адресного пространства пользовательского режима, а следовательно, должны выделять всю динамическую память из куч памяти ОС, таких как выгружаемый или невыгружаемый пул.

ПРИМЕЧАНИЕ В Windows 10 версии 1511 диспетчер задач называет этот процесс System «System and Compressed Memory». Это объясняется появлением в Windows 10 новой функции сжатия памяти для хранения большего объема информации процессов в памяти (вместо выгрузки на диск). Этот механизм более подробно описан в главе 5. Запомните, что термин «процесс System» относится к этому процессу независимо от того, какое имя отображается в той или иной программе. В Windows 10 версии 1607 и Server 2016 процессу было возвращено прежнее имя System. Дело в том, что для сжатия памяти в них используется новый процесс Memory Compression. В главе 5 этот процесс рассматривается более подробно.

Системные потоки создаются функциями `PsCreateSystemThread` или `IoCreateSystemThread` (обе функции документированы в WDK). Эти потоки могут вызываться только из режима ядра. Windows, как и различные драйверы устройств, создает системные потоки во время инициализации системы для выполнения операций, требующих контекста потока, например, выдачи запросов и ожидания ввода/вывода или других объектов или опроса состояния устройства. Например, диспетчер памяти использует системные потоки для реализации таких функций, как запись измененных страниц в файл подкачки или отображаемые файлы, выгрузка и загрузка процессов в память и т. д. Ядро создает системный поток, называемый *диспетчером набора балансировки* (balance set manager); он активизируется один раз в секунду для инициирования различных событий, связанных с планированием и управлением памятью. Диспетчер кэша также использует системные потоки для реализации ввода/вывода с опережающим чтением и отложенной записью. Драйвер устройства файлового сервера (Srv2.sys) использует системные потоки для реакции на сетевые запросы ввода/вывода к файловым данным на дисковых разделах, находящихся в совместном доступе к сети. Даже для драйвера флоппи-дисковода создается системный поток, обеспечивающий опрос устройства. (Опрос в данном случае более эффективен, потому что драйвер флоппи-дисковода на базе прерываний потребляет большое количество системных ресурсов.) Дополнительная информация по конкретным системным потокам приводится в главах с описанием соответствующего компонента.

По умолчанию системные потоки принадлежат процессу System, но драйвер устройства может создать системный поток в любом процессе. Например, драйвер устройства подсистемы Windows (Win32k.sys) создает системный поток в каноническом драйвере экрана (Cdd.dll) — части процесса подсистемы Windows (Csrss.exe), чтобы он мог легко обращаться к данным адресного пространства пользовательского режима этого процесса.

Когда вы занимаетесь диагностикой или анализом системы, полезно иметь возможность установить соответствие отдельных системных потоков с драйвером или даже подпрограммой, содержащей код. Например, на файловом сервере под высокой нагрузкой процесс System с большой вероятностью будет занимать значительную долю процессорного времени. Но если вы знаете, что во время выполнения процесса System работает «какой-то системный поток», этого недостаточно для того, чтобы определить, какой именно драйвер устройства или компонент ОС выполняется.

Итак, если в процессе System выполняются потоки, сначала определите, какие именно (например, при помощи Системного монитора или программы Process Explorer). После того как вы определите работающий поток (или потоки), выясните, в каком драйвере началось выполнение системного потока. По крайней мере, вы будете знать, какой драйвер с большой вероятностью создал поток. Например, в Process Explorer щелкните правой кнопкой мыши на процессе System и выберите команду Properties. Затем на вкладке Threads щелкните на заголовке столбца CPU, чтобы самый активный поток оказался наверху. Выберите этот поток и щелкните на кнопке Module, чтобы увидеть файл, из которого был запущен код на вершине стека. Так как процесс System был защищенным в более ранних версиях Windows, Process Exporer не может показать стек вызовов.

Процесс Secure System

Процесс Secure System (переменный идентификатор процесса) формально является «домом» для защищенного адресного пространства ядра VTL 1, дескрипторов и системных потоков. Несмотря на это, поскольку планирование, управление объектами и управление памятью принадлежат ядру VTL 0, никакие из этих сущностей не будут связаны с этим процессом. Его единственное назначение — предоставить пользователям (например, в таких программах, как диспетчер задач и Process Explorer) наглядный признак того, что механизм VBS в настоящее время активен (показывая тем самым, что хотя бы что-то его использует).

Процесс Memory Compression

Процесс Memory Compression использует свое адресное пространство пользовательского режима для хранения сжатых страниц памяти, которые соответствуют резервной памяти, вытесненной из рабочих наборов некоторых процессов (см. главу 5). В отличие от процесса Secure System, процесс Memory Compression действительно является хостом для нескольких системных потоков, обычно отображаемых

с именами SmKmStoreHelperWorker и SmStReadThread. Оба потока принадлежат диспетчеру хранилища, который управляет сжатием памяти.

Кроме того, в отличие от других процессов System в списке, этот процесс действительно хранит свою память в адресном пространстве пользовательского режима. Это означает, что к нему применяется усечение рабочего набора и в средствах мониторинга для него может отображаться значительное использование памяти. Собственно, если вы обратитесь к вкладке Быстродействие (Performance) диспетчера задач, на которой теперь выводится как используемая, так и сжатая память, вы увидите, что размер рабочего набора процесса Memory Compression равен объему сжатой памяти.

Диспетчер сеансов

Диспетчер сеансов (%SystemRoot%\System32\Smss.exe) — первый процесс пользовательского режима, создаваемый в системе. Этот процесс создается системным потоком пользовательского режима, который выполняет последнюю фазу инициализации исполнительной системы и ядра. Он создается в виде процесса PPL (Protected Process Light), о котором рассказано в главе 3.

При запуске Smss.exe он проверяет, является ли он первым (главным) экземпляром Smss.exe или экземпляром самого себя, который запускается главным экземпляром Smss.exe для создания сеанса. Если при запуске присутствуют аргументы командной строки, значит, это второй случай. Создавая несколько экземпляров самих себя в ходе загрузки и создании сеансов служб терминалов, Smss.exe может создать несколько сеансов одновременно — до четырех параллельных сеансов и еще по одному за каждый дополнительный процессор, помимо первого. Эта возможность повышает быстродействие входа в системах на базе сервера терминалов при одновременном подключении нескольких пользователей. После того как сеанс завершит инициализацию, копия Smss.exe завершается. В результате остается активным только исходный процесс Smss.exe (за информацией о службах терминалов обращайтесь к разделу «Службы терминалов и сеансы» главы 1).

Главный экземпляр Smss.exe выполняет следующие одноразовые операции инициализации.

1. Процесс и его исходный поток помечаются как критические. Если процесс или поток, помеченный как критический, по какой-то причине завершится, в Windows происходит фатальный сбой. Подробнее см. в главе 3.
2. Это заставляет процесс рассматривать некоторые ошибки (например, некорректное использование дескрипторов или нарушение структуры кучи) как критические, а также открывает возможность блокировки исполнения динамического кода (механизм устранения рисков).
3. Базовый приоритет процесса повышается до 11.
4. Если система поддерживает горячее добавление процессоров, она активизирует автоматическое обновление соответствия (affinity) процессоров. Это

означает, что при добавлении новых процессоров новые сеансы будут использовать новые процессоры. Подробнее о динамическом добавлении процессоров см. в главе 4.

5. Пул потоков инициализируется для обработки команд ALPC и других рабочих элементов.
6. Создается порт ALPC с именем \SmApiPort для получения команд.
7. Инициализируется локальная копия NUMA-топологии системы.
8. Создается мьютекс с именем `PendingRenameMutex` для синхронизации операций переименования файлов.
9. Создается исходный блок окружения процесса и обновляется переменная `Safe Mode` (при необходимости).
10. На основании значения параметра `ProtectionMode` в разделе HKLM\SYSTEM\CurrentControlSet\Control\Session Manager создаются дескрипторы безопасности, которые будут использоваться для различных системных ресурсов.
11. На основании значения параметра `ObjectDirectories` в разделе HKLM\SYSTEM\CurrentControlSet\Control\Session Manager создаются описанные им каталоги диспетчера объектов, такие как \RPC Control и \Windows. Также сохраняются программы, перечисленные в параметрах `BootExecute`, `BootExecuteNoPnpSync` и `SetupExecute`.
12. Путь к программе сохраняется в параметре `S0InitialCommand` из раздела HKLM\SYSTEM\CurrentControlSet\Control\Session Manager.
13. Читается значение `NumberOfInitialSessions` из раздела HKLM\SYSTEM\CurrentControlSet\Control\Session Manager; оно игнорируется, если система находится в режиме изготовления (manufacturing mode).
14. Читаются операции переименования файлов, указанные в параметрах `PendingFileRenameOperations` и `PendingFileRenameOperations2` из раздела HKLM\SYSTEM\CurrentControlSet\Control\Session Manager.
15. Читаются значения параметров `AllowProtectedRenames`, `ClearTempFiles`, `TempFileDirectory` и `DisableWpbtExecution` из раздела HKLM\SYSTEM\CurrentControlSet\Control\Session Manager.
16. Читается список DLL из параметра `ExcludeFromKnownDllList`, находящегося в разделе HKLM\SYSTEM\CurrentControlSet\Control\Session Manager.
17. Читается информация о файле подкачки, хранящаяся в разделе HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management — в частности, списки из параметров `PagingFiles` и `ExistingPageFiles`, а также параметры конфигурации `PagefileOnOsVolume` и `WaitForPagingFiles`.
18. Читаются и сохраняются значения, хранящиеся в разделе HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\ DOS Devices.

!

19. Читается и сохраняется список из параметра KnownDlls, хранящегося в разделе HKLM\SYSTEM\CurrentControlSet\Control\Session Manager.

20. Создаются общесистемные переменные среды, определенные в параметре HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Environment.

21. Создается каталог \KnownDlls, а также каталог \KnownDlls32 в 64-разрядных системах с использованием WoW64.

22. В каталоге \Global?? в пространстве имен диспетчера объектов создаются символические ссылки на устройства, определенные в разделе HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\DOS Devices.

23. Создается корневой каталог \Sessions в пространстве имен диспетчера объектов.

24. Создается защищенный слот сообщений и префиксы именованного канала для защиты приложений от фальсифицирующих атак, которые могут произойти, если вредоносное приложение пользовательского режима будет выполнено до службы.

25. Выполняются программные части списков BootExecute и BootExecuteNoPnpSync, разобранных ранее. (По умолчанию используется программа Autochk.exe, выполняющая проверку диска.)

26. Инициализируется оставшаяся часть реестра (кусты HKLM\Software, SAM и Security).

27. Выполняется двоичный модуль WPBT (Windows Platform Binary Table), зарегистрированный в соответствующей таблице ACPI (если его выполнение не было отключено в реестре). Часто используется производителями средств для борьбы с кражами для принудительного выполнения на очень ранней стадии двоичного модуля Windows, который может связаться с владельцем или организовать выполнение других сервисных функций даже в свежеустановленной системе. Эти процессы должны компоноваться только с Ntdll.dll (т. е. принадлежать «родной» подсистеме).

28. Обрабатываются незавершенные переименования файлов, заданные в ключах реестра, упомянутых ранее (если не используется загрузка среды восстановления Windows).

29. Инициализируется информация файла(-ов) подкачки и выделенного файла дампа на основании разделов HKLM\System\CurrentControlSet\Control\Session Manager\Memory Management и HKLM\System\CurrentControlSet\Control\CrashControl.

30. Проверяется совместимость системы с технологией охлаждения памяти, используемой в системах NUMA.

31. Сохраняется старый файл подкачки, создается файл аварийного дампа и новые файлы подкачки на основании информации о предшествующем сбое.

32. Создаются дополнительные динамические переменные среды, такие как PROCESSOR_ARCHITECTURE, PROCESSOR_LEVEL, PROCESSOR_IDENTIFIER и PROCESSOR_REVISION, основанные на данных реестра и системной информации, запрашиваемой у ядра.

33. Запускаются программы из списка HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\SetupExecute. Для них используются те же правила, что и для BootExecute на шаге 11.

34. Создается безымянный объект раздела, совместно используемый дочерними процессами (например, Csrss.exe) для обмена информацией с Smss.exe. Дескриптор этого раздела передается дочерним процессам через механизм наследования дескрипторов. Подробнее о наследовании дескрипторов см. в главе 8 части 2.

35. Открываются известные DLL, которые отображаются на постоянные разделы (отображаемые файлы), — кроме тех, которые перечислены как исключения в предшествующих проверках реестра (по умолчанию не указаны).

36. Создается поток для ответов на запросы создания сеансов.

37. Создается экземпляр Smss.exe для инициализации сеанса 0 (неинтерактивный сеанс).

38. Создается экземпляр Smss.exe для инициализации сеанса 1 (интерактивный сеанс) и при соответствующей настройке реестра — дополнительные экземпляры Smss.exe для дополнительных интерактивных сеансов для будущих возможных входов пользователей. Когда экземпляр Smss.exe создает эти экземпляры, он каждый раз запрашивает явное создание нового идентификатора сеанса с передачей флага PROCESS_CREATE_NEW_SESSION функции NtCreateUserProcess. Это эквивалентно вызову внутренней функции диспетчера памяти MiSessionCreate, которая создает необходимые сеансовые структуры данных режима ядра (такие, как объект Session) и задает диапазон виртуальных адресов сеанса, используемый частью режима ядра подсистемы Windows (Win32k.sys) и другими драйверами устройств сеансового пространства. Подробнее см. в главе 5.

После выполнения всех этих действий Smss.exe неограниченно долго ожидает по дескриптору сеанса 0 Csrss.exe. Так как процесс Csrss.exe помечен как критический (а также является защищенным процессом; см. главу 3), при завершении Csrss.exe ожидание никогда не завершится, потому что в системе произойдет фатальный сбой.

Сеансовый стартовый экземпляр Smss.exe выполняет следующие действия:

- создание процесса(-ов) подсистем для сеанса (по умолчанию подсистема Windows Csrss.exe);
- создание экземпляра Winlogon (интерактивные сеансы) или же исходной команды сеанса 0, которой по умолчанию является Wininit (для сеанса 0) при отсутствии изменений из-за параметров реестра, упомянутых выше. Дополнительная информация об этих двух процессах приведена ниже.

Наконец, промежуточный процесс Smss.exe завершается, а процессы подсистемы и Winlogon (или Wininit) остаются без родителя.

Процесс инициализации Windows

Процесс Wininit.exe выполняет следующие функции инициализации системы.

1. Он сам и главный поток помечаются как критические, чтобы в случае преждевременного выхода в системе, загруженной в отладочном режиме, происходила передача управления отладчику. (В противном случае произойдет фатальный сбой.)
2. Это заставляет процесс рассматривать некоторые ошибки (например, некорректное использование дескрипторов или нарушение структуры кучи) как критические.
3. Инициализируется поддержка разделения состояний, если она поддерживается выпуском Windows.
4. Создается событие с именем `Global\FirstLogonCheck` (вы можете наблюдать его в Process Explorer или WinObj в каталоге \BaseNamedObjects); оно используется процессами Winlogon для определения того, какой из процессов Winlogon должен запуститься первым.
5. Создается событие `WinlogonLogoff` в каталоге диспетчера объектов BaseNamedObjects, предназначенное для использования экземплярами Winlogon. Это событие инициируется в начале операции выхода.
6. Базовый приоритет самого процесса повышается до высокого (13), а приоритет его главного потока — до 15.
7. При отсутствии специальной настройки в параметре `NoDebugThread` из раздела HKLM\Software\Microsoft\Windows NT\CurrentVersion\Winlogon создается очередь с периодическим таймером, который передает управление процессу пользовательского режима по распоряжению отладчика ядра. Это позволяет удаленному отладчику ядра выполнить присоединение Winlogon с выходом в любые другие приложения пользовательского режима.
8. Имя компьютера сохраняется в переменной среды `COMPUTERNAME`, после чего происходит обновление и настройка информации, относящейся к TCP/IP, — например, имени домена и имени хоста.
9. Задаются значения переменных среды `USERPROFILE`, `ALLUSERSPROFILE`, `PUBLIC` и `ProgramData` для профиля по умолчанию.
10. Создается временный каталог %SystemRoot%\Temp (например, C:\Windows\Temp).
11. Организуется загрузка шрифтов и DWM (диспетчера окон рабочего стола), если сеанс 0 является интерактивным, что зависит от выпуска Windows.
12. Создается исходный терминал, который состоит из оконной станции (которой всегда присваивается имя Winsta0) и двух рабочих столов (Winlogon и Default) для процессов, которые должны работать в сеансе 0.

13. Инициализируется ключ шифрования LSA в зависимости от того, хранится ли он локально или должен вводиться в интерактивном режиме. За дополнительной информацией о хранении локальных ключей аутентификации обращайтесь к главе 7.
14. Создается диспетчер служб (SCM или Services.exe). Краткое описание приводится ниже, а более подробная информация — в главе 9 части 2.
15. Запускается служба подсистемы локальной аутентификации (Lsass.exe), и если активен механизм Credential Guard — изолированный трастлет LSA (Lsaiso.exe). Для этого также требуется запрос ключа VBS от UEFI. Подробнее о Lsass.exe и Lsaiso.exe см. в главе 7.
16. Если в настоящее время инициализация не завершена (т. е. это первая загрузка на только что установленной копии системы, обновление с переходом на новую основную версию ОС или ознакомительную версию), запускается программа инициализации.
17. Начинается бесконечное ожидание запроса на завершение работы или завершение одного из упоминавшихся ранее системных процессов (если только не задан параметр `DontWatchSysProcs` в разделе `Winlogon`, упоминавшийся на шаге 7). В любом случае работа системы завершается.

Диспетчер служб

В этом разделе рассматриваются службы, являющиеся процессами пользовательского режима. Службы похожи на процессы демонов Linux в том отношении, что они также могут настраиваться для автоматического запуска во время загрузки системы без необходимости интерактивного входа. Они также могут запускаться вручную (например, из административной программы Службы, с использованием средства sc.exe или вызовом функции Windows `StartService`). Как правило, службам не обязательно взаимодействовать с пользователем, выполнившим вход, хотя в некоторых особых условиях это возможно. Кроме того, хотя большинство служб работает от имени специальных учетных записей (`SYSTEM` или `LOCAL SERVICE`), другие могут выполняться в одном контексте безопасности с учетными записями пользователей, выполнивших вход. (Подробнее см. в главе 9 части 2.)

Диспетчер служб (SCM, Service Control Manager) — специальный системный процесс, выполняющий образ %SystemRoot%\System32\Services.exe, ответственный за запуск, остановку и взаимодействие с процессами служб. Он также является защищенным процессом, что усложняет возможную фальсификацию. Программы служб в действительности представляют собой обычные образы Windows, которые вызывают специальные функции Windows для взаимодействия с SCM. Эти функции выполняют такие действия, как регистрация успешного запуска службы, реакция на запросы статуса, приостановка или завершение. Службы определяются в разделе реестра HKLM\SYSTEM\CurrentControlSet\Services.

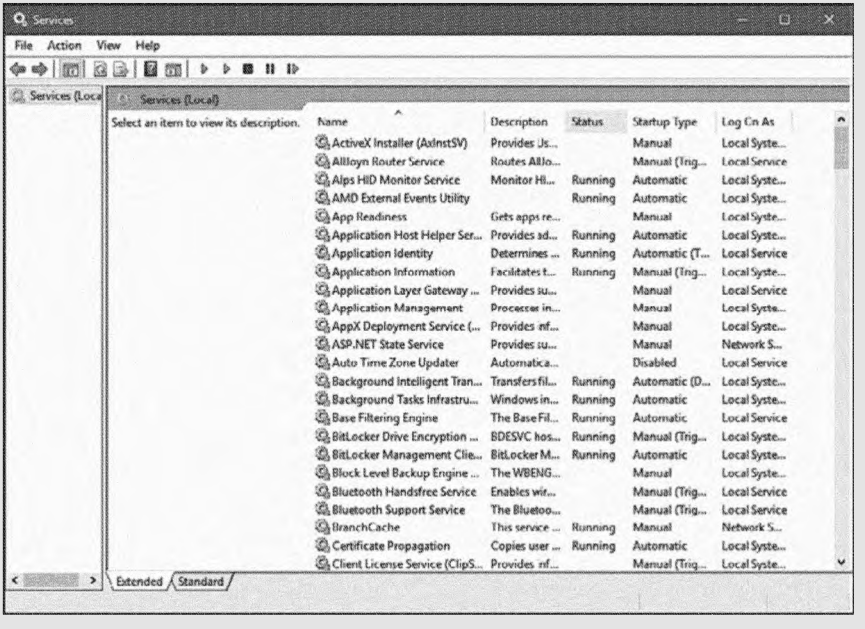
Помните, что у служб три имени: имя процесса, которое вы видите выполняющимся в системе, внутреннее имя в реестре и отображаемое имя для административного приложения **Службы** (Services). (Не у всех служб есть отображаемое имя — если его нет, вместо него выводится внутреннее имя.) Службы также могут иметь поле описания с подробностями о назначении службы.

Чтобы связать процесс службы со службами, содержащимися в этом процессе, используйте команду `tlist/s` (из средств отладки Windows) или `tasklist/svc` (встроенная программа Windows). Учтите, что между процессами служб и работающими службами не всегда существует однозначное соответствие, потому что некоторые службы используют процесс совместно с другими службами. В реестре параметр `Type` в разделе службы показывает, выполняется ли служба в отдельном процессе или использует процесс вместе с другими службами в образе.

Ряд других компонентов Windows реализуется в формате служб: диспетчер печати, журнал событий, планировщик задач и разные сетевые компоненты. За дополнительной информацией о службах обращайтесь к главе 9 части 2.

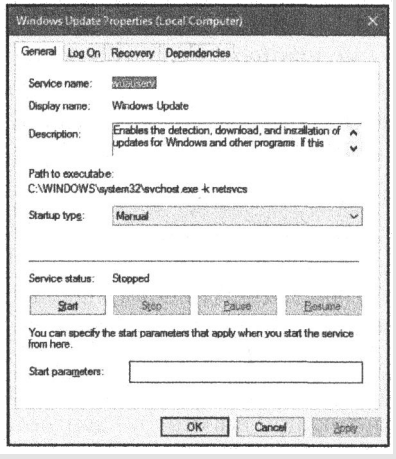
ЭКСПЕРИМЕНТ: ВЫВОД СПИСКА УСТАНОВЛЕННЫХ СЛУЖБ

Чтобы вывести список установленных служб, откройте панель управления, выберите категорию Администрирование (Administrative tools) и выберите приложение Службы (Services.) Также вместо этого можно щелкнуть на кнопке Пуск (Start) и ввести команду `services.msc`. Результат выглядит так:



!

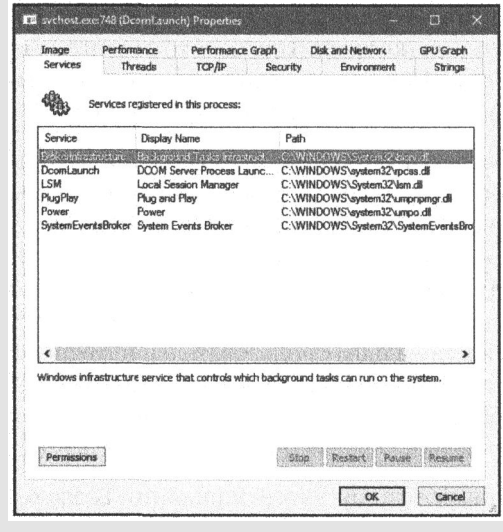
Чтобы просмотреть подробный список свойств службы, щелкните правой кнопкой мыши на службе и выберите команду Свойства (Properties). Например, свойства службы Windows Update выглядят так:



Обратите внимание: в поле Исполняемый файл (Path to Executable) указана программа, содержащая службу, и ее командная строка. Помните, что некоторые службы используют процесс совместно с другими службами; однозначное соответствие существует не всегда.

ЭКСПЕРИМЕНТ: ПРОСМОТР ИНФОРМАЦИИ О ПРОЦЕССАХ СЛУЖБ

Process Explorer выделяет процессы, которые являются хостами для одной или нескольких служб. (По умолчанию процессы выделяются розовым цветом, но вы можете сменить цвет. Для этого откройте меню Options и выберите команду Configure Colors.) Если сделать двойной щелчок на процессе, являющемся хостом службы, открывается вкладка Services со списком служб внутри процесса, именем раздела реестра с определением службы, отображаемое имя для администратора, текст описания службы (если он есть) и для служб Svchost.exe — путь к DLL-библиотеке, реализующей службу. Например, список служб в одном из процессов Svchost.exe, выполняемых от имени учетной записи System, выглядит так:



Winlogon, LogonUI и Userinit

Процесс входа Windows (%SystemRoot%\System32\Winlogon.exe) обеспечивает вход и выход интерактивных пользователей. Winlogon.exe получает уведомления о запросах на вход, когда пользователь вводит комбинацию клавиш SAS (Secure Attention Sequence). По умолчанию в Windows используется комбинация SAS Ctrl+Alt+Delete. SAS существует для защиты пользователей от программ перехвата паролей, имитирующих процесс входа, потому что комбинация клавиш не может быть перехвачена приложением пользовательского режима.

Аспекты идентификации и аутентификации процесса входа реализуются через DLL-библиотеки, называемые *поставщиками учетных данных* (credential providers). Стандартные поставщики учетных данных Windows реализуют стандартные интерфейсы аутентификации Windows: пароли и смарт-карты. Windows 10 предоставляет биометрического поставщика учетных данных: механизм распознавания лиц, известный как *Windows Hello*. Однако разработчики могут предоставить собственных поставщиков учетных данных для реализации других механизмов идентификации и аутентификации вместо стандартного метода с вводом имени/пароля, например, основанных на анализе голоса или биометрических устройствах (таких, как сканер отпечатка пальца). Так как Winlogon.exe является критическим системным процессом, от которого зависит работа системы, поставщики учетных данных и пользовательский интерфейс для вывода диалогового окна входа выполняются в дочернем процессе Winlogon.exe с именем LogonUI.exe. Когда процесс Winlogon.exe обнаруживает SAS, он запускает процесс, инициализирующий поставщиков учетных данных. Когда пользователь вводит свои учетные данные (в форме, требуемой поставщиком) или закрывает интерфейс входа, процесс LogonUI.exe завершается. Winlogon.exe также может загрузить дополнительные DLL сетевых поставщиков, необходимые для выполнения вторичной аутентификации. Эта возможность позволяет нескольким сетевым поставщикам собрать данные идентификации и аутентификации во время нормального входа.

После того как имя пользователя и пароль (или другой информационный пакет по требованию поставщика учетных данных) будут введены, они передаются процессу локальной службы аутентификации (Lsass.exe, см. главу 7) для проверки. Lsass.exe вызывает соответствующий пакет аутентификации, реализованный в форме DLL, для выполнения фактической проверки, например проверки совпадения пароля с данными, хранящимися в Active Directory или SAM (часть реестра с определением локальных пользователей и групп). Если механизм Credential Guard активен и происходит вход в домен, Lsass.exe связывается с изолированным трастлетом LSA (Lsaiso.exe, см. главу 7) для получения машинного ключа, необходимого для проверки действительности запроса аутентификации.

При успешной аутентификации Lsass.exe вызывает функцию SRM (например, `NtCreateToken`) для генерирования объекта маркера доступа, содержащего профиль безопасности пользователя. Если в системе используется механизм контроля учетных записей UAC (User Account Control) и входящий пользователь является

членом административной группы или обладает привилегиями администратора, Lsass.exe создает вторую, ограниченную версию маркера. Затем маркер доступа используется Winlogon для создания исходного процесса(-ов) в сеансе пользователя. Исходный процесс(-ы) хранится в параметре реестра `Userinit` раздела реестра HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon. По умолчанию используется Userinit.exe, но образов в списке может быть несколько.

Процесс Userinit.exe выполняет частичную инициализацию окружения пользователя, например запуск сценария входа и восстановление сетевых подключений. Затем он обращается в реестре к параметру `Shell` (в разделе Winlogon, упоминавшемся выше) и создает процесс для запуска оболочки, определяемой в системе (по умолчанию Explorer.exe). Затем Userinit завершает работу. Вот почему Explorer (Проводник) отображается в списке без родителя — его родитель завершился. Как было объяснено в главе 1, tlist.exe и Process Explorer выравнивают по левому краю процессы, родители которых не выполняются. Также можно сказать, что Explorer является «внуком» Winlogon.exe.

Процесс Winlogon.exe активизируется не только во время входа и выхода, но и каждый раз при перехвате SAS с клавиатуры. Например, при нажатии Ctrl+Alt+Delete при выполненном входе появляется экран безопасности Windows с командами завершения работы, запуска диспетчера задач, блокировки рабочей станции, завершения работы системы и т. д. Winlogon.exe и LogonUI.exe — процессы, обеспечивающие это взаимодействие.

Полное описание этапов процесса входа приведено в главе 11 части 2. Более подробно об аутентификации см. в главе 7. Описания вызываемых функций, взаимодействующих с Lsass.exe (функций с префиксами `Lsa`), приведены в документации Windows SDK.

Заключение

В этой главе представлен общий обзор системной архитектуры Windows. В ней рассматриваются ключевые компоненты Windows и взаимодействия между ними. В следующей главе более подробно описаны процессы, входящие в число фундаментальных сущностей Windows.

Глава 3
Процессы и задания

В этой главе рассматриваются структуры данных и алгоритмы, относящиеся к процессам и заданиям Windows. Сначала мы опишем создание процесса в целом. Затем рассмотрим внутренние структуры, составляющие процесс. Далее вы узнаете о защищенных процессах и о том, чем они отличаются от процессов незащищенных. Далее мы опишем действия по созданию процесса (и его исходного потока). Глава завершается описанием заданий.

Процессы связаны со многими компонентами Windows, поэтому многие термины и структуры данных (рабочие наборы, потоки, объекты и дескрипторы, кучи системной памяти и т. д.) упоминаются в этой главе, но подробно рассматриваются в других главах. Чтобы в полной мере понять эту главу, вы должны знать термины и концепции, представленные в главе 1 «Концепции и средства» и главе 2 «Архитектура системы» — в частности, понимать различия между процессами и потоками, знать структуру виртуального адресного пространства Windows и представлять суть различий между пользовательским режимом и режимом ядра.

Создание процесса

Windows API предоставляет несколько функций для создания процессов. Простейшая из них — функция `CreateProcess` — пытается создать процесс с таким же маркером доступа, как у процесса-создателя. Если необходимо использовать другой маркер, можно воспользоваться функцией `CreateProcessAsUser`, которая получает дополнительный аргумент — дескриптор объекта маркера, который уже был получен каким-то образом (например, при вызове `LogonUser`).

К числу других функций создания процессов относятся функции `CreateProcessWithTokenW` и `CreateProcessWithLogonW` (обе принадлежат advapi32.dll). Функция `CreateProcessWithTokenW` похожа на `CreateProcessAsUser`, но они различаются уровнем необходимых привилегий для вызывающей стороны. (За подробностями обращайтесь к документации Windows SDK.) `CreateProcessWithLogonW` представляет собой удобный сокращенный способ входа с учетными данными указанного пользователя и создания процесса с полученным маркером за один вызов. Обе функции обращаются к службе вторичного входа (seclogon.dll с хостом SvcHost.Exe), выдавая вызов RPC для фактического создания процесса.

SecLogon выполняет вызов в своей внутренней функции SlrCreateProcessWithLogon, и если все прошло нормально, в итоге вызывает CreateProcessAsUser. Служба SecLogon по умолчанию настраивается для ручного запуска, поэтому при первом вызове CreateProcessWithTokenW или CreateProcessWithLogonW происходит запуск службы. Если попытка запуска завершается неудачей (например, администратор может отключить службу), вызовы функций не проходят. Также эти функции используются программой командной строки runas — вероятно, вам знакомой.

На рис. 3.1 изображена диаграмма вызовов, описанных выше.

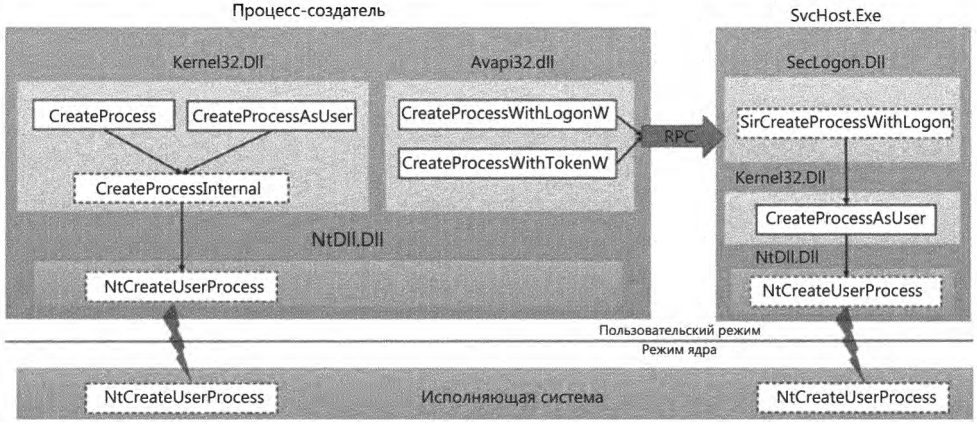


Рис. 3.1. Функции создания процесса. Внутренние функции помечены пунктирными рамками

Все функции, документированные выше, рассчитаны на файл PE (Portable Executable) с правильной структурой (хотя расширение EXE не обязательно), пакетный файл или 16-разрядное COM-приложение. В остальном они не знают, как связать файл с определенным расширением (например, .txt) с исполняемым файлом (например, приложением Блокнот). Эта функциональность предоставляется Оболочкой Windows в таких функциях, как ShellExecute и ShellExecuteEx. Эти функции могут получить произвольный файл (не только исполняемые файлы) и попытаться найти исполняемый файл на основании расширения и настроек реестра в HKEY_CLASSES_ROOT. (См. главу 9 «Механизмы управления» части 2.)

Затем ShellExecute(Ex) вызывает CreateProcess с подходящим исполняемым файлом и добавляет в командную строку аргументы для реализации намерения пользователя (например, имени файла TXT для его редактирования в Блокноте).

В конечном итоге все ветви выполнения ведут к общей внутренней функции CreateProcessInternal, которая начинает реальную работу по созданию процесса Windows пользовательского режима. Если все проходит нормально, CreateProcessInternal вызывает NtCreateUserProcess из Ntdll.dll, чтобы передать управление в режим ядра и продолжить процесс создания режима ядра в одноименной функции (NtCreateUserProcess), части исполняющей системы.

Аргументы функций CreateProcess*

Стоит обсудить аргументы семейства функций `CreateProcess*`, часть которых будет рассматриваться при описании логики выполнения `CreateProcess`. Процесс, созданный из пользовательского режима, всегда создается с одним потоком. Этот поток в конечном итоге будет выполнять главную функцию исполняемого файла. Важные аргументы функций `CreateProcess*`:

- Для `CreateProcessAsUser` и `CreateProcessWithTokenW` — дескриптор маркера, под которым будет выполняться новый процесс. Аналогичным образом для `CreateProcessWithLogonW` требуется имя пользователя, домен и пароль.
- Путь к исполняемому файлу и аргументы командной строки.
- Дополнительные атрибуты безопасности, применяемые к новому процессу и создаваемому объекту потока.
- Флаг, указывающий, должны ли все дескрипторы текущего (создающего) процесса, помеченные как наследуемые, наследоваться (копироваться) в новый процесс. (Подробнее о дескрипторах и наследовании дескрипторов см. в главе 8 «Системные механизмы».)
- Различные флаги, влияющие на создание процессов. Несколько примеров (за полным списком обращайтесь к документации Windows SDK):
  - `CREATE_SUSPENDED` — исходный поток нового процесса создается в приостановленном состоянии. Последующий вызов `ResumeThread` заставляет поток начать выполнение;
  - `DEBUG_PROCESS` — создающий процесс объявляет, что он является отладчиком, создающим новый процесс под своим контролем;
  - `EXTENDED_STARTUPINFO_PRESENT` — вместо `STARTUPINFO` (см. ниже) передается расширенная структура `STARTUPINFOEX`.
- Необязательный блок среды нового процесса (с указанием переменных среды). Если среда не задана, она наследуется от создающего процесса.
- Необязательный текущий каталог нового процесса. (Если он не задан, используется текущий каталог создающего процесса.) Созданный процесс позднее может установить другой текущий каталог вызовом `SetCurrentDirectory`. Текущий каталог процесса используется в различных вариантах поиска с неполным путем (например, при загрузке DLL с указанием только имени файла).
- Структура `STARTUPINFO` или `STARTUPINFOEX` с дополнительными данными конфигурации для создания процесса. `STARTUPINFOEX` содержит дополнительное непрозрачное поле, представляющее список атрибутов процесса и потока (фактически массив пар «ключ/значение»). Эти атрибуты заполняются вызовами `UpdateProcThreadAttributes` для каждого необходимого атрибута. Некоторые атрибуты не документированы и используются во внутренней реализации, как при создании приложений Магазина (см. следующий раздел).

- Структура `PROCESS_INFORMATION`, содержащая выходные данные успешного создания процесса. Структура содержит новый уникальный идентификатор процесса, новый уникальный идентификатор потока, дескриптор нового процесса и дескриптор нового потока. Значения дескрипторов могут пригодиться создающему процессу, если он хочет выполнить какие-то операции с новым процессом или потоком после создания.

Создание современных процессов Windows

В главе 1 были описаны новые типы приложений, появившиеся в Windows 8 и Windows Server 2012. Термины для обозначения таких приложений меняются со временем, но мы будем называть их *современными приложениями* (а также приложениями UWP или иммерсивными процессами), чтобы отличить их от классических (также называемых настольными) приложений.

Создание процесса современного приложения не сводится к простому вызову `CreateProcess` с правильным путем к исполняемому файлу; также появляются некоторые обязательные аргументы командной строки. Другое требование заключается в добавлении недокументированного атрибута процесса (с использованием `UpdateProcThreadAttribute`) с ключом `PROC_THREAD_ATTRIBUTE_PACKAGE_FULL_NAME` и значением, которому присваивается полное имя пакета приложения Магазина. Хотя атрибут не документирован, существуют и другие способы (с точки зрения API) выполнения приложений Магазина. Например, Windows API включает интерфейс COM с именем `IApplicationActivationManager`, который реализуется классом COM с идентификатором CLSID `CLSID_ApplicationActivationManager`. Один из методов этого интерфейса, `ActivateApplication`, может использоваться для запуска приложения Магазина после получения идентификатора `AppUserModelId` из полного имени пакета вызовом `GetPackageApplicationIds`. (За дополнительной информацией об этих API обращайтесь к Windows SDK.)

Имена пакетов и типичный механизм создания процесса приложения Магазина с момента нажатия пользователя на плитке современного приложения до итогового вызова `CreateProcess` описаны в главе 9 части 2.

Создание других разновидностей процессов

Хотя приложения Windows запускают либо классические, либо современные приложения, исполняющая система включает поддержку других видов процессов, которые должны запускаться в обход Windows API, таких как собственные процессы, минимальные процессы или процессы Pico. Например, в главе 2 упоминалось о существовании диспетчера сеансов Smss, который является примером машинного образа. Так как он создается ядром, очевидно, процесс создания не использует API `CreateProcess`; вместо этого `NtCreateUserProcess` вызывается напрямую. Аналогичным образом, когда Smss создает Autochk (программа проверки

диска) или Csrss (процесс подсистемы Windows), функции Windows API также недоступны, и вместо них необходимо использовать `NtCreateUserProcess`. Кроме того, собственные процессы не могут создаваться на базе приложений Windows, так как функция `CreateProcessInternal` отвергает образы с типом обычной подсистемы. Для устранения этих сложностей библиотека Ntdll.dll включает экспортированную вспомогательную функцию `RtlCreateUserProcess`, которая предоставляет более простую и удобную обертку для `NtCreateUserProcess`.

Как подсказывает имя, функция `NtCreateUserProcess` используется для создания процессов пользовательского режима. Однако, как мы узнали из главы 2, Windows также включает ряд процессов пользовательского режима, например процессы System и Memory Compression (которые являются минимальными процессами), а также возможность управления процессами Pico со стороны поставщиков, например подсистемой Windows для Linux. Вместо этого созданием таких процессов занимается системная функция `NtCreateProcessEx`, часть возможностей которой зарезервирована исключительно для вызовов из режима ядра (например, созданием минимальных процессов).

Наконец, поставщики Pico вызывают вспомогательную функцию, которая обеспечивает как создание минимальных процессов, так и инициализацию контекста поставщика Pico — `PspCreatePicoProcess`. Эта функция не экспортируется и доступна только поставщикам Pico через специальный интерфейс.

Как будет показано далее в этой главе, хотя `NtCreateProcessEx` и `NtCreateUserProcess` формально являются разными системными функциями, для выполнения работы используются одни и те же внутренние функции: `PspAllocateProcess` и `PspInsertProcess`. Все возможные способы создания процессов, упоминавшиеся до настоящего момента, и любые возможные способы, которые можно себе представить, от командлетов WMI PowerShell до драйвера режима ядра, в конечном итоге приходят в эту точку.

Внутреннее устройство процессов

В этом разделе описаны ключевые структуры данных процессов Windows, поддерживаемые различными компонентами системы, а также различные механизмы и средства для анализа этих данных.

Каждый процесс Windows представляется структурой `EPROCESS` (Executive Process). Кроме многих атрибутов, относящихся к процессу, `EPROCESS` содержит ряд других взаимосвязанных структур данных и указателей на них. Например, каждый процесс содержит один или несколько потоков, каждый из которых представлен структурой `ETHREAD` (Executive Thread). (Структуры данных потоков рассматриваются в главе 4 «Потоки».)

Структура `EPROCESS` и большинство связанных с ней структур данных существует в системном адресном пространстве. Исключение составляет блок PEB (Process

Environment Block), который существует в адресном пространстве процесса (пользовательском), потому что он содержит информацию, доступную для кода пользовательского режима. Кроме того, некоторые структуры данных процесса, используемые при управлении памятью (например, список рабочих наборов), действительны только в контексте текущего процесса, поскольку хранятся в системном пространстве, принадлежащем процессу. (Подробнее об адресном пространстве процессов см. в главе 5 «Управление памятью».)

Для каждого процесса, выполняющего программу Windows, процесс подсистемы Windows (Csrss) поддерживает параллельную структуру с именем `CSR_PROCESS`. Кроме того, часть подсистемы Windows, относящаяся к режиму ядра (Win32k.sys), поддерживает структуру данных уровня процесса `W32PROCESS`, которая создается при первом вызове из потока функции Windows USER или GDI, реализованной в режиме ядра. Это происходит сразу же после загрузки библиотеки User32.dll. Типичные функции, инициирующие загрузку этой библиотеки, — `CreateWindow(Ex)` и `GetMessage`.

Так как часть подсистемы Windows режима ядра интенсивно использует графику с аппаратным ускорением на базе DirectX, инфраструктура компонента GDI (Graphics Device Interface) заставляет графическое ядро DirectX (Dxgkrnl.sys) инициализировать собственную структуру `DXGPROCESS`. Эта структура содержит информацию для объектов DirectX (поверхности, шейдеры и т. д.), а также счетчики и параметры политики, относящиеся к GPGPU, для планирования как вычислений, так и управления памятью.

За исключением процесса Idle, каждая структура `EPROCESS` инкапсулируется в виде объекта процесса диспетчером объектов (см. главу 8 части 2). Поскольку процессы не являются именованными объектами, они не отображаются в программе WinObj (из пакета Sysinternals). Впрочем, в каталоге \ObjectTypes виден объект `Type` с именем `Process` (в WinObj). Дескриптор процесса предоставляет (через связанные с процессом API) доступ к некоторым данным структуры `EPROCESS` и некоторым из связанных с ней структур.

Многие другие драйверы и компоненты системы, регистрируя уведомления о создании процессов, могут создавать собственные структуры данных для хранения информации на уровне процессов. (Функции `PsSetCreateProcessNotifyRoutine(Ex, Ex2)`, документированные в WDK, предоставляют такую возможность.) При анализе затрат памяти процесса часто приходится учитывать размер таких структур, хотя получить точное числовое значение практически невозможно. Кроме того, некоторые из этих функций позволяют таким компонентам запрещать (блокировать) создание процессов. Таким образом, производителям средств защиты от вредоносных программ предоставляется архитектурный механизм включения усовершенствований безопасности в операционную систему — с применением черных списков на базе хешей либо других средств.

Для начала сосредоточимся на объекте `Process`. На рис. 3.2 изображены ключевые поля структуры `EPROCESS`.

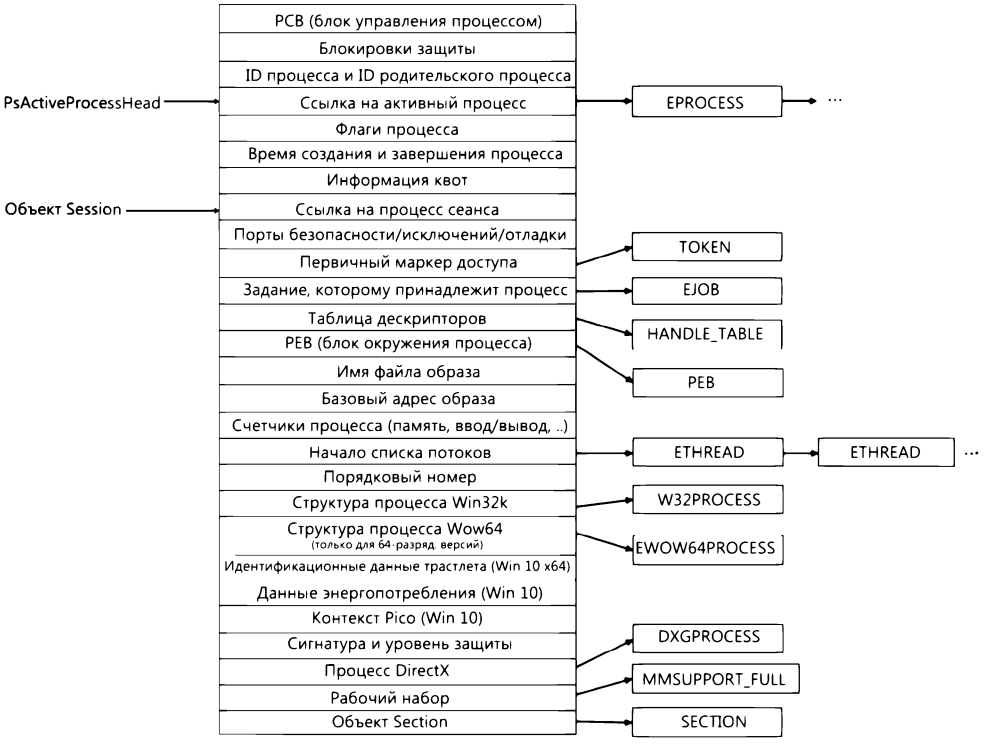


Рис. 3.2. Важнейшие поля структуры EPROCESS

Рис. 3.3. Важнейшие поля структуры KPROCESS

Мы уже показывали, как API и компоненты ядра разбиваются на изолированные многоуровневые модули с собственными соглашениями имен. Структуры данных процесса строятся по похожему принципу. Как видно из рис. 3.2, первое поле структуры процесса называется PCB (Process Control Block). Это структура типа `KPROCESS` (Kernel Process). Хотя функции исполняющей системы хранят информацию в `EPROCESS`, диспетчер, планировщик и код учета прерываний/времени — части ядра операционной системы — используют `KPROCESS`. Это позволяет ввести абстрактную прослойку между высокоуровневой функциональностью исполняющей системы и низкоуровневой реализацией некоторых функций, а также предотвратить нежелательные зависимости между уровнями. На рис. 3.3 показаны ключевые поля структуры `KPROCESS`.

ЭКСПЕРИМЕНТ: ВЫВОД ФОРМАТА СТРУКТУРЫ EPROCESS

Чтобы получить список полей, образующих структуру EPROCESS, и их смещений в шестнадцатеричной форме, введите команду `dt nt!_eprocess` в отладчике ядра. (Подробнее об отладчике ядра и выполнении отладки режима ядра в локальной системе см. в главе 1.) Вывод (сокращенный для экономии места) в 64-разрядной системе Windows 10 выглядит так:

```
lkd> dt nt!_eprocess
    +0x000 Pcb              : _KPROCESS
    +0x2d8 ProcessLock      : _EX_PUSH_LOCK
    +0x2e0 RundownProtect   : _EX_RUNDOWN_REF
    +0x2e8 UniqueProcessId  : Ptr64 Void
    +0x2f0 ActiveProcessLinks : _LIST_ENTRY
...
    +0x3a8 Win32Process     : Ptr64 Void
    +0x3b0 Job              : Ptr64 _EJOB
...
    +0x418 ObjectTable      : Ptr64 _HANDLE_TABLE
    +0x420 DebugPort        : Ptr64 Void
    +0x428 WoW64Process     : Ptr64 _EWOW64PROCESS
...
    +0x758 SharedCommitCharge : Uint8B
    +0x760 SharedCommitLock : _EX_PUSH_LOCK
    +0x768 SharedCommitLinks : _LIST_ENTRY
    +0x778 AllowedCpuSets : Uint8B
    +0x780 DefaultCpuSets : Uint8B
    +0x778 AllowedCpuSetsIndirect : Ptr64 Uint8B
    +0x780 DefaultCpuSetsIndirect : Ptr64 Uint8B
```

Первое поле структуры (`Pcb`) содержит вложенную структуру типа KPROCESS. В этой структуре хранятся данные планирования и учета времени. Формат структуры KPROCESS выводится так же, как и формат структуры EPROCESS:

```
lkd> dt nt!_kprocess
    +0x000 Header           : _DISPATCHER_HEADER
    +0x018 ProfileListHead  : _LIST_ENTRY
    +0x028 DirectoryTableBase : Uint8B
    +0x030 ThreadListHead   : _LIST_ENTRY
```

```
    +0x040 ProcessLock      : Uint4B
    ...
    +0x26c KernelTime       : Uint4B
    +0x270 UserTime         : Uint4B
    +0x274 LdtFreeSelectorHint : Uint2B
    +0x276 LdtTableLength   : Uint2B
    +0x278 LdtSystemDescriptor : _KGDTENTRY64
    +0x288 LdtBaseAddress : Ptr64 Void
    +0x290 LdtProcessLock : _FAST_MUTEX
    +0x2c8 InstrumentationCallback : Ptr64 Void
    +0x2d0 SecurePid      : Uint8B
```

Команда `dt` также позволяет просмотреть содержимое одного или нескольких полей; для этого введите их имена после имени структуры. Например, команда `dt nt!_eprocess UniqueProcessId` выводит поле с идентификатором процесса. В случае если поле представляет структуру (как поле Pcb структуры EPROCESS, которое содержит вложенную структуру KPROCESS), добавьте точку после имени поля, чтобы отладчик вывел вложенную структуру. Например, для просмотра KPROCESS также можно ввести команду `dt nt!_eprocess Pcb`. Рекурсию можно продолжить; добавьте новые имена полей (из KPROCESS) и т. д. Наконец, ключ `-r` команды `dt` позволяет выполнить рекурсивный обход по всем вложенным структурам. Число, добавленное после ключа, управляет глубиной рекурсии при работе команды.

Команда `dt` в приведенном ранее виде выводит формат выбранной структуры, но не содержимое конкретного экземпляра этого типа структуры. Чтобы вывести экземпляр конкретного процесса, укажите адрес структуры EPROCESS в аргументе команды `dt`. Для получения адресов почти всех структур EPROCESS в системе используйте команду `!process 0 0` (исключением является процесс простоя системы). Так как структура KPROCESS находится в самом начале EPROCESS, адрес EPROCESS также подойдет в качестве адреса KPROCESS в команде `dt _kprocess`.

ЭКСПЕРИМЕНТ: ИСПОЛЬЗОВАНИЕ КОМАНДЫ !PROCESS ОТЛАДЧИКА ЯДРА

Команда `!process` отладчика ядра выводит подмножество информации, хранящейся в объекте процесса и в связанных с ним структурах. Вывод для каждого процесса делится на две части. Сначала выводится информация о процессе, как показано ниже. Если идентификатор или адрес процесса не указан, то команда !process выводит информацию о процессе — владельце потока, выполняемого на процессоре 0. В однопроцессорной системе им будет сам отладчик WinDbg (или livekd, если он используется вместо WinDbg).

```
lkd> !process
PROCESS ffffe0011c3243c0
    SessionId: 2 Cid: 0e38 Peb: 5f2f1de000 ParentCid: 0f08
    DirBase: 38b3e000 ObjectTable: ffffc000a2b22200 HandleCount:
    <Data Not Accessible>
    Image: windbg.exe
```

```
VadRoot ffffe0011badae60 Vads 117 Clone 0 Private 3563. Modified 228.
Locked 1.
DeviceMap ffffc000984e4330
Token                             ffffc000a13f39a0
ElapsedTime                       00:00:20.772
UserTime                          00:00:00.000
KernelTime                        00:00:00.015
QuotaPoolUsage[PagedPool]         299512
QuotaPoolUsage[NonPagedPool]      16240
Working Set Sizes (now,min,max)  (9719, 50, 345) (38876KB, 200KB, 1380KB)
PeakWorkingSetSize                9947
VirtualSize                       2097319 Mb
PeakVirtualSize                   2097321 Mb
PageFaultCount                    13603
MemoryPriority                    FOREGROUND
BasePriority                      8
CommitCharge                      3994
Job                               ffffe0011b853690
```

После вывода основной информации о процессе следует список потоков в процессе. Этот вывод объясняется в разделе «Эксперимент: использование команды `!thread` отладчика ядра» главы 4.

Также к числу команд, отображающих информацию о процессе, относится команда `!handle`, которая выводит таблицу дескрипторов процесса (см. раздел «Дескрипторы объектов и таблица дескрипторов процесса» главы 8 части 2). Структуры процесса и безопасности потоков описаны в главе 7 «Безопасность».

Обратите внимание: в выходных данных приводится адрес PEB. Его можно использовать с командой `!peb`, описанной в следующем эксперименте, для просмотра удобного представления блока PEB произвольного процесса; также вы можете использовать обычную команду `dt` со структурой `_PEB`. Но поскольку блок PEB находится в адресном пространстве пользовательского режима, адрес действителен только в контексте его процесса. Чтобы обратиться к блоку PEB другого процесса, необходимо сначала переключить WinDbg на этот процесс. Для этого можно воспользоваться командой `.process /P`, за которой следует указатель на EPROCESS.

Если вы используете последнюю версию Windows 10 SDK, в обновленной версии WinDbg под адресом PEB размещается гиперссылка; щелчок на этой ссылке автоматически выполняет команду `.process` и команду `!peb`.

Блок PEB размещается в адресном пространстве пользовательского режима описываемого им процесса. Он содержит информацию, необходимую для загрузчика образов, диспетчера кучи и других компонентов Windows, которые должны обращаться к нему из пользовательского режима; предоставлять доступ ко всей этой информации через системные вызовы было бы слишком затратно. Структуры `EPROCESS` и `KPROCESS` доступны только из режима ядра. Важнейшие поля PEB изображены на рис. 3.4 и подробно рассматриваются позднее в этой главе.

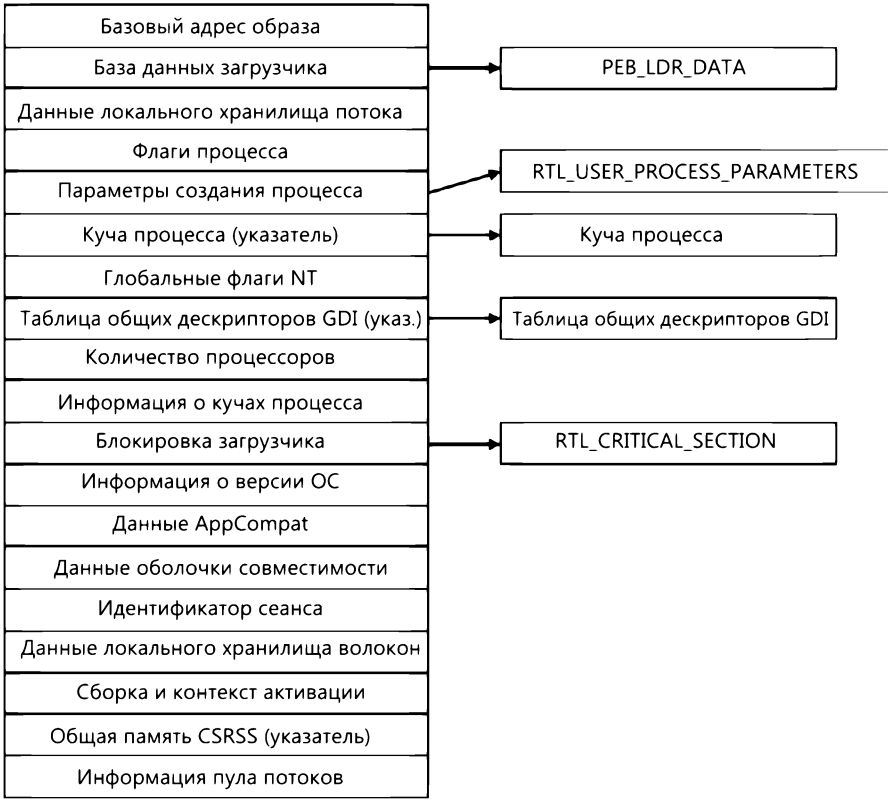


Рис. 3.4. Важнейшие поля структуры PEB

ЭКСПЕРИМЕНТ: ПРОСМОТР PEB

Структуру PEB можно вывести командой `!peb` отладчика ядра. Эта команда выводит PEB процесса, который является владельцем потока, выполняемого в настоящее время на процессоре 0. Информация из предыдущего эксперимента также позволяет использовать указатель на PEB в аргументе команды.

```
lkd> .process /P ffffe0011c3243c0 ; !peb 5f2f1de000
PEB at 0000003561545000
    InheritedAddressSpace:    No
    ReadImageFileExecOptions: No
    BeingDebugged:            No
    ImageBaseAddress:         00007ff64fa70000
    Ldr                       00007ffdf52f5200
    Ldr.Initialized:          Yes
    Ldr.InInitializationOrderModuleList: 000001d3d22b3630 . 000001d3d6cddb60
    Ldr.InLoadOrderModuleList:           000001d3d22b3790 . 000001d3d6cddb40
    Ldr.InMemoryOrderModuleList:         000001d3d22b37a0 . 000001d3d6cddb50
                    Base TimeStamp                     Module
            7ff64fa70000 56ccafdd Feb 23 21:15:41 2016 C:\dbg\x64\windbg.exe
```

```
            7ffdf51b0000 56cbf9dd Feb 23 08:19:09 2016 C:\WINDOWS\SYSTEM32\
                                  ntdll.dll
            7ffdf2c10000 5632d5aa Oct 30 04:27:54 2015 C:\WINDOWS\system32\
                                  KERNEL32.DLL
...
```

Структура `CSR_PROCESS` содержит информацию о процессах, относящихся к подсистеме Windows (Csrss). Соответственно, структура `CSR_PROCESS` связывается только с приложениями Windows (например, у Smss ее нет). Кроме того, поскольку каждый сеанс содержит собственный экземпляр подсистемы Windows, структуры `CSR_PROCESS` поддерживаются процессом Csrss в каждом отдельном сеансе. Базовая структура `CSR_PROCESS` изображена на рис. 3.5, а более подробное описание приведено далее в этой главе.

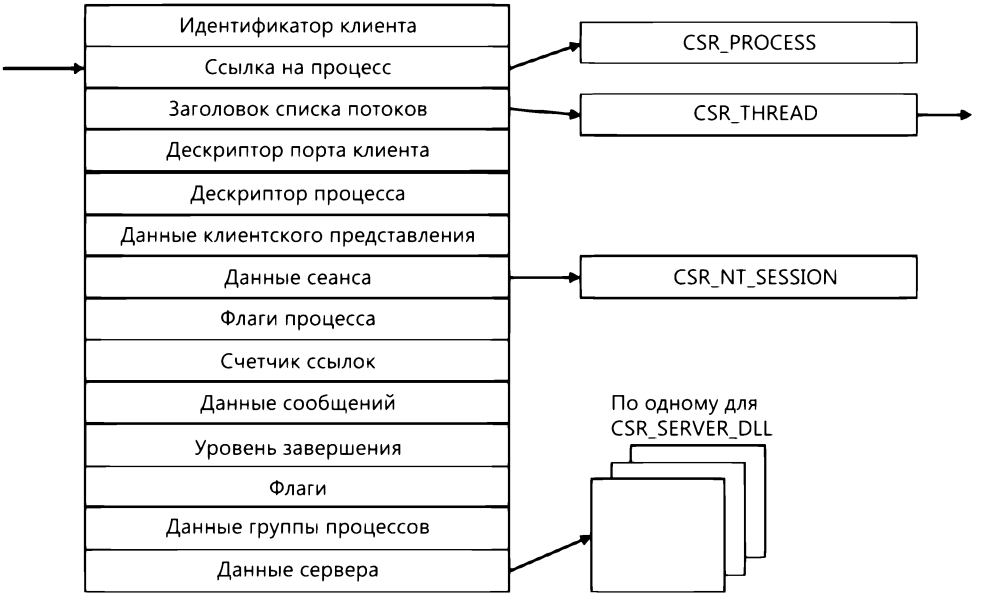


Рис. 3.5. Поля структуры CSR_PROCESS

ЭКСПЕРИМЕНТ: АНАЛИЗ CSR_PROCESS

Процессы Csrss являются защищенными (о защищенных процессах рассказано далее в этой главе), поэтому присоединить отладчик пользовательского режима к процессу Csrss невозможно (даже с повышенными привилегиями или в неагрессивном режиме). Вместо этого используется отладчик ядра.

Начнем с вывода списка существующих процессов Csrss:

```
lkd> !process 0 0 csrss.exe
PROCESS ffffe00077ddf080
    SessionId: 0  Cid: 02c0    Peb: c4e3fc0000  ParentCid: 026c
    DirBase:   ObjectTable: ffffc0004d15d040  HandleCount: 543.
    Image: csrss.exe

PROCESS ffffe00078796080
    SessionId: 1  Cid: 0338    Peb: d4b4db4000  ParentCid: 0330
    DirBase:   ObjectTable: ffffc0004ddff040  HandleCount: 514.
    Image: csrss.exe
```

Возьмите любой процесс и измените контекст отладчика для конкретного процесса, чтобы были видны его модули пользовательского режима:

```
lkd> .process /r /P ffffe00078796080
Implicit process is now ffffe000'78796080
Loading User Symbols
.............
```

Ключ `/p` меняет контекст процесса отладчика на переданный объект процесса (EPROCESS), а ключ `/r` запрашивает загрузку символических имен пользовательского режима. Теперь вы можете просмотреть список модулей командой `lm` или просмотреть структуру `CSR_PROCESS`:

```
lkd> dt csrss!_csr_process
   +0x000 ClientId         : _CLIENT_ID
   +0x010 ListLink         : _LIST_ENTRY
   +0x020 ThreadList       : _LIST_ENTRY
   +0x030 NtSession        : Ptr64 _CSR_NT_SESSION
   +0x038 ClientPort       : Ptr64 Void
   +0x040 ClientViewBase   : Ptr64 Char
   +0x048 ClientViewBounds : Ptr64 Char
   +0x050 ProcessHandle    : Ptr64 Void
   +0x058 SequenceNumber   : Uint4B
   +0x05c Flags            : Uint4B
   +0x060 DebugFlags       : Uint4B
   +0x064 ReferenceCount   : Int4B
   +0x068 ProcessGroupId   : Uint4B
   +0x06c ProcessGroupSequence : Uint4B
   +0x070 LastMessageSequence : Uint4B
   +0x074 NumOutstandingMessages : Uint4B
   +0x078 ShutdownLevel    : Uint4B
   +0x07c ShutdownFlags    : Uint4B
   +0x080 Luid             : _LUID
   +0x088 ServerDllPerProcessData : [1] Ptr64 Void
```

`W32PROCESS` — последняя из системных структур данных процессов, которые будут рассмотрены в этой главе. Она содержит всю информацию, необходимую коду управления графикой и окнами в ядре (Win32k) для поддержания информации состояния процессов GUI (которые определялись ранее как процессы, выполнившие хотя бы один системный вызов USER/GDI). Базовая структура `W32PROCESS` показана на рис. 3.6. К сожалению, так как информация типа для структур Win32k

недоступна в виде общедоступных символических имен, мы не сможем легко продемонстрировать эксперимент с выводом этой информации. Как бы то ни было, обсуждение структур данных и концепций, относящихся к графике, выходит за рамки темы книги.

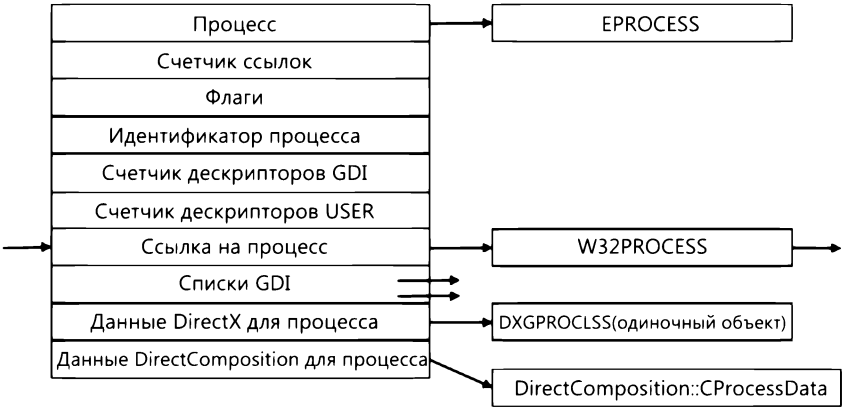


Рис. 3.6. Поля структуры процесса Win32k

Защищенные процессы

В модели безопасности Windows любой процесс, запущенный с маркером доступа, содержащим привилегию отладки (такую, как у учетной записи администратора), может запросить любые требуемые ему права на доступ к любому другому процессу, запущенному на машине. Например, он может выполнять произвольное чтение и запись в произвольных участках памяти процесса, вставлять код, приостанавливать и возобновлять выполнение потоков и запрашивать информацию о других процессах. Такие инструментальные средства, как Process Explorer и диспетчер задач, нуждаются в этих правах и запрашивают их для предоставления пользователям своей функциональности.

Это поведение (гарантирующее, что полный доступ к запущенному коду будет только у администраторов) выглядит логично, но оно противоречит поведению системы, соответствующему требованиям по управлению цифровыми правами, установленным медиаиндустрией для операционных систем компьютеров. Эти права поддерживают воспроизведение современного высококачественного цифрового контента, например, записанного на носителях Blu-ray. Для поддержки надежного и защищенного проигрывания такого контента в Windows Vista и Windows Server 2008 появились *защищенные процессы*. Такие процессы сосуществуют с обычными процессами Windows, но накладывают существенные ограничения на права доступа, которые могут запрашиваться другими процессами системы (даже запущенными с привилегиями администратора).

Защищенные процессы могут быть созданы любым приложением, но операционная система разрешит процессу быть защищенным, только если файл образа был снабжен цифровой подписью со специальным сертификатом Windows Media Certificate. В Windows защищенный путь к носителю — Protected Media Path (PMP) — использует защищенные процессы для защиты дорогостоящих цифровых материалов, и разработчики приложений (например, DVD-проигрывателей) могут воспользоваться защищенными процессами с помощью API-функций Media Foundation.

Процесс Audio Device Graph (Audiodg.exe) является защищенным, потому что через него может быть декодирован защищенный музыкальный контент. С ним непосредственно связан защищенный конвейер Media Foundation (Mfpmp.exe), который тоже является защищенным процессом по аналогичным причинам (по умолчанию он не запускается). Принадлежащий системе отчета об ошибках Windows Error Reporting (WER; см. главу 8 части 2) клиентский процесс (Werfaultsecure.exe) также может быть запущен в защищенном режиме, поскольку ему нужно иметь доступ к защищенным процессам на случай аварии одного из них. И наконец, сам процесс System защищен, потому что часть расшифрованной информации генерируется драйвером Ksecdd.sys и сохраняется в его памяти пользовательского режима. Процесс System также защищен для обеспечения целостности всех дескрипторов ядра (потому что таблица дескрипторов процесса System содержит все имеющиеся в системе дескрипторы ядра). Поскольку другие драйверы также могут отображать память из адресного пространства пользовательского режима процесса System (например, сертификат целостности кода и данные каталога), это еще одна причина для защиты процесса.

Поддержка защищенных процессов на уровне ядра ведется по двум направлениям: во-первых, во избежание атак внедрения кода основная часть создания процесса происходит в режиме ядра. (Последовательность действий при создании защищенных и стандартных процессов описана в следующем разделе.) Во-вторых, у защищенных процессов (и их усовершенствованных «родственников» PPL, описанных в следующем разделе) в структуре `EPROCESS` устанавливаются специальные биты, которые изменяют поведение процедур, связанных с мерами безопасности в диспетчере процессов для отказа в некоторых правах доступа, которые обычно предоставляются администраторам. Фактически, в отношении защищенных процессов предоставляются только следующие права: на запрос процесса и установку ограниченной информации — `PROCESS_QUERY/SET_LIMITED_INFORMATION`, на завершение процесса — `PROCESS_TERMINATE` и на приостановку и возобновление процесса — `PROCESS_SUSPEND_RESUME`. Определенные права доступа отключаются также и в отношении потоков, запущенных внутри защищенных процессов, эти права доступа будут рассмотрены позже в разделе «Внутреннее устройство потоков» главы 4.

Поскольку в Process Explorer для запроса информации о внутренних данных процесса используются стандартные API-функции пользовательского режима, некоторые операции не могут выполняться с защищенными процессами. С другой стороны, такое инструментальное средство, как WinDbg, в режиме отладки ядра использующее для получения этой информации инфраструктуру режима ядра,

будет в состоянии вывести полную информацию. Поведение Process Explorer при работе с такими защищенными процессами, как Audiodg.exe, описано в эксперименте раздела «Внутреннее устройство потоков» главы 4.

ПРИМЕЧАНИЕ Как уже упоминалось в главе 1, для выполнения локальной отладки ядра нужно загрузить систему в режиме отладки (который включается командой `bcdedit /debug on` или путем использования расширенных настроек загрузки в средстве Msconfig). Это оградит от атак на защищенные процессы и на PMP (Protected Media Path), основанные на использовании режима отладки. При загрузке в режиме отладки проигрывание содержимого высокого разрешения работать не будет.

Надежное ограничение прав доступа позволяет ядру оградить защищенный процесс от доступа к нему из пользовательского режима. С другой стороны, поскольку признаком защищенного процесса служит флаг в структуре `EPROCESS`, администратор все же может загрузить драйвер режима ядра, снимающий бит этого флага. Но это будет нарушением PMP-модели и будет считаться потенциально опасным действием, и такой драйвер, скорее всего, будет в конечном итоге заблокирован от загрузки на 64-разрядной системе, поскольку действующая в режиме ядра политика цифровой подписи кода запретит снабжать подписью вредоносный код. Кроме того, защита режима ядра PatchGuard (описанная в главе 7), а также драйвер защищенной среды и аутентификации (Peauth.sys) распознают такие попытки и сообщают о них. Даже на 32-разрядных системах драйвер должен быть распознан PMP-политикой, иначе воспроизведение будет остановлено. Эта политика реализована компанией Microsoft и обнаруживается не на всяком ядре. Блокировка потребует вмешательства Microsoft по идентификации подписи, определения вредоносности кода и обновления ядра.

Облегченные защищенные процессы (PPL)

Как вы только что видели, исходная модель защищенных процессов ориентировалась на цифровую защиту контента. Начиная с Windows 8.1 и Windows Server 2012 R2, появилось расширение модели защищенных процессов — так называемые *облегченные защищенные процессы* (PPL, Protected Process Light).

Процессы PPL защищены так же, как и классические защищенные процессы: код пользовательского режима (даже выполняемый с повышенными привилегиями) не может проникнуть в эти процессы посредством внедрения потоков или получения подробной информации о загруженных DLL. Тем не менее модель PPL добавляет в качество защиты новое измерение: значения атрибутов. Разные подписывающие стороны имеют различные уровни доверия, а это приводит к тому, что некоторые PPL имеют более высокую (или низкую) защиту, чем другие.

Поскольку поддержка управления цифровыми правами (DRM) прошла путь от простой защиты мультимедийного контента до лицензирования Windows и приложений Магазина Windows, стандартные защищенные процессы также теперь

различаются в зависимости от подписывающей стороны. Наконец, различные признанные подписывающие стороны также определяют, какие права доступа будут запрещены для менее защищенных процессов. Например, обычно разрешены только маски доступа `PROCESS_QUERY/SET_LIMITED_INFORMATION` и `PROCESS_SUSPEND_RESUME`. Право завершения `PROCESS_TERMINATE` не разрешено для некоторых подписывающих сторон PPL.

В табл. 3.1 перечислены допустимые значения флага защиты, хранимого в структуре `EPROCESS`.

Таблица 3.1. Допустимые значения защиты для процессов

Внутреннее символическое имя уровня защиты процесса	Тип защиты	Подписывающая сторона
`PS_PROTECTED_SYSTEM (0x72)`	Защищенный	WinSystem
`PS_PROTECTED_WINTCB (0x62)`	Защищенный	WinTcb
`PS_PROTECTED_WINTCB_LIGHT (0x61)`	Защищенный облегченный	WinTcb
`PS_PROTECTED_WINDOWS (0x52)`	Защищенный	Windows
`PS_PROTECTED_WINDOWS_LIGHT (0x51)`	Защищенный облегченный	Windows
`PS_PROTECTED_LSA_LIGHT (0x41)`	Защищенный облегченный	Lsa
`PS_PROTECTED_ANTIMALWARE_LIGHT (0x31)`	Защищенный облегченный	Средства защиты от вредоносных программ
`PS_PROTECTED_AUTHENTICODE (0x21)`	Защищенный	Authenticode
`PS_PROTECTED_AUTHENTICODE_LIGHT (0x11)`	Защищенный облегченный	Authenticode
`PS_PROTECTED_NONE (0x00)`	Нет	Нет

Как видно из табл. 3.1, определено несколько подписывающих сторон с убыванием приоритета. WinSystem обладает наивысшим приоритетом и используется для процесса System и минимальных процессов (таких, как процесс Memory Compression). Для процессов пользовательского режима наивысшим приоритетом обладает WinTCB (Windows Trusted Computer Base); он используется для защиты критических процессов, которые хорошо известны ядру, — для таких процессов ядро может немного опустить свою «планку безопасности». Анализируя приоритеты процессов, помните, что защищенные процессы всегда превосходят PPL, а процессы подписывающих сторон с более высоким приоритетом всегда имеют доступ к низкоприоритетным, но не наоборот. В табл. 3.2 перечислены уровни подписывающих сторон (высокие значения обозначают более высокий приоритет) и приведены примеры их использования. Для вывода их в отладчике используется тип `_PS_PROTECTED_SIGNER`.

Таблица 3.2. Подписывающие стороны и уровни защиты

Имя (PS_PROTECTED_SIGNER)	Уровень	Применение
PsProtectedSignerWinSystem	7	Системные и минимальные процессы (включая процессы Pico)
PsProtectedSignerWinTcb	6	Критические компоненты Windows (с запретом PROCESS_TERMINATE)
PsProtectedSignerWindows	5	Важные компоненты Windows для работы с конфиденциальными данными
PsProtectedSignerLsa	4	Lsass.exe (при настройке для защищенного выполнения)
PsProtectedSignerAntimalware	3	Службы и процессы, предназначенные для защиты от вредоносных программ, включая сторонние (с запретом PROCESS_TERMINATE)
PsProtectedSignerCodeGen	2	NGEN (.NET Native Code Generation)
PsProtectedSignerAuthenticode	1	Работа с контентом DRM или загрузка шрифтов пользовательского режима
PsProtectedSignerNone	0	Защита отсутствует

Казалось бы, что мешает вредоносному процессу заявить, что он является защищенным процессом, и оградиться от средств борьбы с вредоносными программами? Поскольку для запуска в качестве защищенного процесса сертификат Windows Media DRM уже не требуется, компания Microsoft расширила свой модуль целостности кода и включила в него поддержку двух специальных вариантов расширенного использования ключа (EKU, Enhanced Key Usage), которые могут кодироваться в сертификате цифровой подписи кода: 1.3.6.1.4.1.311.10.3.22 и 1.3.6.4.1.311.10.3.20. При наличии одного из этих EKU жестко закодированные строки подписывающей стороны и издателя в сертификате, в сочетании с дополнительными возможными EKU, затем связываются с различными значениями защищенной подписывающей стороны. Например, издатель сертификата Microsoft Windows может предоставить значение защищенной подписывающей стороны `PsProtectedSignerWindows`, но только в том случае, если вместе с ним также присутствует EKU проверки системного компонента Windows (1.3.6.1.4.1.311.10.3.6). Так, на рис. 3.7 показан сертификат процесса Smss.exe, которому разрешен запуск в режиме `WinTcb-Light`.

Наконец, учтите, что уровень защиты процесса также влияет на то, какие DLL ему будет разрешено загружать, — в противном случае из-за ошибки логики или простой замены файла совершенно законный защищенный процесс можно было бы заставить загрузить стороннюю или вредоносную библиотеку, которая теперь будет выполняться на том же уровне защиты, что и процесс. Для реализации этой проверки каждому процессу предоставляется «уровень подписи», который хранится в поле `SignatureLevel` структуры `EPROCESS` и используется для поиска по внутренней таблице соответствующего «уровня подписи DLL», хранящегося в поле `SectionSignatureLevel` в `EPROCESS`. Все DLL-библиотеки, загружаемые в процессе, будут проверяться компонентом целостности кода по тем же правилам, по которым проверяется основной исполняемый файл. Например, процесс с подписы-

вающей стороной «WinTcb» будет загружать только DLL-библиотеки с подписью «Windows» или уровнем выше.

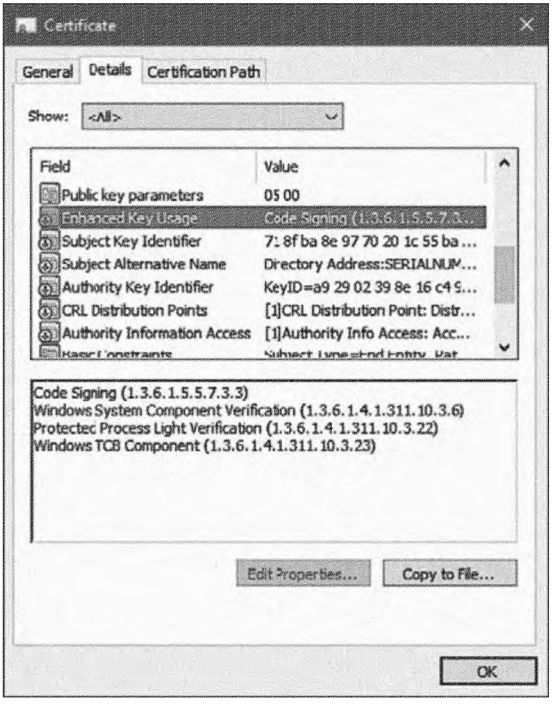


Рис. 3.7. Сертификат Smss

В Windows 10 и Windows Server 2016 следующие процессы снабжаются подписью PPL уровня WinTcb-Lite: smss.exe, csrss.exe, services.exe и wininit.exe. Lsass.exe выполняется как PPL в Windows на базе ARM (таких, как Windows Mobile 10) и может выполняться как PPL в x86/x64, если соответствующая настройка будет включена параметром реестра или политики (подробнее см. в главе 7). Кроме того, некоторые службы могут настраиваться для выполнения в форме Windows PPL или защищенного процесса — как, например, sppsvc.exe (Software Protection Platform). Также на этом уровне защиты работают некоторые процессы-хосты служб (Svchost.exe), поскольку многие службы — например, служба развертывания AppX и подсистема Windows для Linux — также выполняются как защищенные. Подробнее о таких защищенных службах см. в главе 9 части 2.

Тот факт, что эти базовые системные двоичные модули работают с защитой TCB, критичен для безопасности системы. Например, Csrss.exe имеет доступ к некоторым закрытым API, реализуемым диспетчером окон (Win32k.sys), что может предоставить атакующему административный доступ к чувствительным частям ядра. Аналогичным образом Smss.exe и Wininit.exe реализуют логику запуска и управления системой, критичную для выполнения без возможного вмешатель-

ства со стороны администратора. Windows гарантирует, что эти двоичные модули всегда выполняются с правами WinTCB-Lite, поэтому, например, никто не сможет запустить их без указания правильного уровня защиты процесса в атрибутах процесса при вызове `CreateProcess`. Эта гарантия, известная как *минимальный список TCB*, заставляет любые процессы из системного пути, имена которых присутствуют в табл. 3.3, обладать минимальным уровнем защиты и/или уровнем подписи независимо от данных, переданных вызывающей стороной.

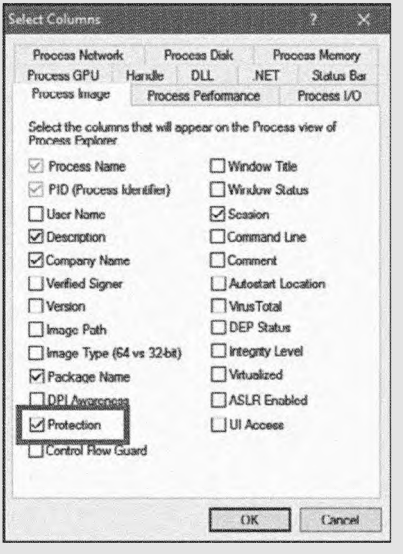
Таблица 3.3. Минимальный уровень TCB

Имя процесса	Минимальный уровень подписи	Минимальный уровень защиты
Smss.exe	Определяется по уровню защиты	WinTcb-Lite
Csrss.exe	Определяется по уровню защиты	WinTcb-Lite
Wininit.exe	Определяется по уровню защиты	WinTcb-Lite
Services.exe	Определяется по уровню защиты	WinTcb-Lite
Werfaultsecure.exe	Определяется по уровню защиты	WinTcb-Full
Sppsvc.exe	Определяется по уровню защиты	Windows-Full
Genvalobj.exe	Определяется по уровню защиты	Windows-Full
Lsass.exe	`SE_SIGNING_LEVEL_WINDOWS`	0
Userinit.exe	`SE_SIGNING_LEVEL_WINDOWS`	0
Winlogon.exe	`SE_SIGNING_LEVEL_WINDOWS`	0
Autochk.exe	`SE_SIGNING_LEVEL_WINDOWS*`	0

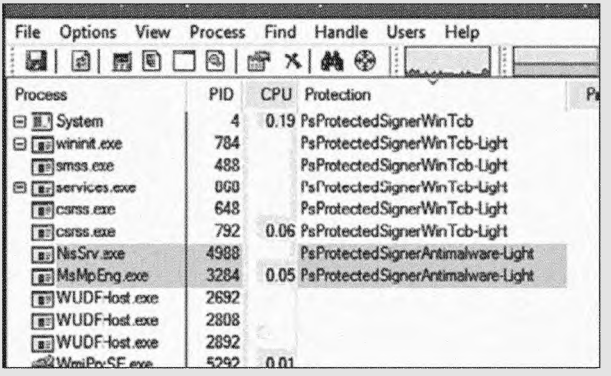
* Только в системах с микропрограммами UEFI

ЭКСПЕРИМЕНТ: ПРОСМОТР ЗАЩИЩЕННЫХ ПРОЦЕССОВ В PROCESS EXPLORER

В этом эксперименте вы увидите, как в Process Explorer отображаются защищенные процессы (любого типа). Запустите Process Explorer и установите флажок Protection на вкладке Process Image для включения в список столбца Protection:



Теперь отсортируйте столбец Protection по убыванию и прокрутите список в начало. Вы увидите в нем все защищенные процессы с указанием типа защиты. Следующий снимок экрана сделан на машине с Windows 10 для x64:



Если выбрать защищенный процесс и посмотреть на нижнюю часть в режиме настройки просмотра DLL, вы ничего не увидите. Дело в том, что Process Explorer для получения информации о загруженных модулях использует API пользовательского режима, а для этого необходим уровень доступа, не предоставляемый для обращения к защищенным процессам. Наиболее заметное исключение — процесс System, который является защищенным, однако Process Explorer показывает для него список загруженных модулей ядра (в основном драйверов), потому что в системных процессах DLL-библиотек нет. Для получения информации используется `EnumDeviceDrivers` — системная API-функция, которой не нужен дескриптор процесса.

Переключившись на представление Handle, вы получите полную информацию о дескрипторах. Причина та же: Process Explorer использует недокументированную API-функцию, которая возвращает все дескрипторы в системе, а для ее работы конкретный дескриптор процесса не нужен. Process Explorer может идентифицировать процесс просто потому, что в эту информацию включается значение PID, связанное с каждым дескриптором.

Сторонняя поддержка PPL

Механизм PPL выводит средства защиты процессов за рамки исполняемых файлов, создаваемых исключительно Microsoft. Один из примеров — программные продукты для борьбы с вредоносными программами (AM, Anti-Malware). Типичный AM-продукт состоит из трех основных компонентов.

- Драйвер ядра, который перехватывает запросы ввода/вывода к файловой системе и/или сети и реализует средства блокировки с использованием объектов, процессов и обратных вызовов потоков.
- Служба пользовательского режима (обычно запущенная от имени привилегированной учетной записи), которая настраивает политики драйвера, получает

оповещения от драйвера об «интересных» событиях (например, обнаружении зараженного файла) и может общаться с локальным сервером или интернет-сервером.

- GUI-процесс пользовательского режима, который передает информацию пользователю и (возможно) позволяет пользователю принимать решения там, где это уместно.

Один из возможных путей атаки вредоносных программ на систему основан на внедрении кода в процесс, работающий с повышенными привилегиями. А еще лучше — прямо в службу, предназначенную для борьбы с вредоносными программами. Таким образом они пытаются заблокировать ее работу. Но если AM-служба будет выполняться как PPL, внедрение кода станет невозможным, а завершение процесса будет запрещено; таким образом обеспечивается улучшенная защита AM-программ от вредоносных программ, не использующих эксплойты уровня ядра.

Чтобы такой сценарий стал возможен, драйвер ядра AM-программы, описанный выше, должен иметь соответствующий драйвер ELAM (Early-Launch Anti Malware). Подробное описание ELAM приведено в главе 7, но ключевое различие заключается в том, что таким драйверам необходим специальный сертификат, предоставленный компанией Microsoft (после проверки издателя программного продукта). После того как такой драйвер будет установлен, его основной исполняемый (PE) файл может содержать дополнительный ресурсный раздел с именем `ELAMCERTIFICATEINFO`. Этот раздел может описывать до трех дополнительных подписывающих сторон (определяемых их открытыми ключами), каждая из которых имеет до трех дополнительных EKU (идентифицируемых значениями OID).

Если система целостности кода распознает любой файл, подписанный одной из этих трех сторон и содержащий один из трех EKU, она разрешает процессу запросить PPL с уровнем `PS_PROTECTED_ANTIMALWARE_LIGHT` (0x31). Каноническим примером служит собственный AM-продукт компании Microsoft — Защитник Windows. Его служба в Windows 10 (MsMpEng.exe) подписана сертификатом для улучшения защиты от вредоносного кода, атакующего как сам AM-продукт, так и его сетевой сервер (NisSvc.exe).

Минимальные процессы и процессы Pico

Могло создаться впечатление, будто типы процессов, рассматривавшиеся до настоящего момента, а также их структуры данных предназначены для исполнения кода пользовательского режима, и для этого они хранят в памяти множество взаимосвязанных структур данных. Однако не все процессы используются для этой цели. Например, как вы уже видели, процесс System представляет собой обычный контейнер для большинства системных потоков, чтобы их время выполнения не влияло на процессы пользовательского режима, а также контейнер для дескрипторов драйверов (называемых дескрипторами ядра), чтобы их владельцами не становились произвольные процессы.

Минимальные процессы

Если функции `NtCreateProcessEx` передается особый флаг, а вызывающая сторона находится в режиме ядра, работа функции слегка изменяется: в ней вызывается API-функция `PsCreateMinimalProcess`.

Это приводит к тому, что процесс создается без многих структур, представленных ранее, а именно:

- не создается адресное пространство имен пользовательского режима, поэтому блок PEB и связанные с ним структуры не существуют;
- в процесс не отображается ни NTDLL, ни информация загрузчика/наборов API-функций;
- никакой объект раздела не связывается с процессом; это означает, что файл с исполняемым образом не будет связан с выполнением или с именем процесса (которое может быть пустым или содержать произвольную строку);
- во флагах `EPROCESS` устанавливается флаг `Minimal`, с которым все потоки становятся минимальными потоками, и для них не создаются структуры пользовательского режима (такие, как TEB или стек пользовательского режима). Подробнее о TEB см. в главе 4.

Как было показано в главе 2, Windows 10 содержит не менее двух минимальных процессов: процесс System и процесс Memory Compression; также он может содержать третий процесс — Secure System, если включены средства безопасности на основе виртуализации (см. главу 2 и главу 7).

Наконец, есть еще один способ создания минимальных процессов в системе Windows 10: включение необязательной подсистемы Windows для Linux (WSL), которая также была описана в главе 2. Это приводит к установке поставщика Pico, состоящего из драйверов Lxss.sys и LxCore.sys.

Процессы Pico

Минимальные процессы имеют ограниченную область применения касательно доступа к виртуальному адресному пространству из компонентов ядра и его защиты. Процессы Pico играют более важную роль: для этого они разрешают специальному компоненту, называемому *поставщиком Pico*, управлять большинством аспектов своего выполнения с позиций операционной системы. Такая степень контроля позволяет поставщику эмулировать поведение совершенно другого ядра операционной системы таким образом, что двоичный образ понятия не имеет о том, что он выполняется в операционной системе на базе Windows. По сути это реализация проекта Drawbridge от Microsoft Research, которая также используется для аналогичной поддержки SQL Server (хотя и с использованием Library OS на базе Windows поверх ядра Linux).

Чтобы система поддерживала процессы Pico, в ней должен присутствовать поставщик. Такой поставщик регистрируется API-функцией `PsRegisterPicoProvider`, но по очень специфическому правилу: поставщик Pico должен быть загружен до того, как будут загружены сторонние драйверы (включая драйверы загрузки).

Вызов этой API-функции разрешен только ограниченному набору из десятка или около того основных драйверов, причем эти драйверы должны быть снабжены сертификатом Microsoft и EKU компонента Windows. В системах Windows с включенным необязательным компонентом WSL основной драйвер называется Lxss.sys; он служит заглушкой до того момента, как будет загружен другой драйвер LxCore.sys, который берет на себя обязанности поставщика Pico посредством перевода различных таблиц диспетчеризации на себя. Кроме того, учтите, что на момент написания книги только один основной драйвер мог зарегистрироваться в качестве поставщика Pico.

Когда поставщик Pico вызывает API-функцию регистрации, он получает набор указателей на функции, которые использует для создания процессов Pico и управления ими:

- функция для создания процесса Pico и функция для создания потока Pico;
- функция для получения контекста (произвольного указателя, который может использоваться поставщиком для хранения произвольных данных) процесса Pico, функция для назначения контекста и еще пара аналогичных функций для потоков Pico. Эти данные используются для заполнения поля `PicoContext` в `ETHREAD` и/или `EPROCESS`;
- функция для чтения структуры контекста процессора (`CONTEXT`) потока Pico и функция для записи этой структуры;
- функция для изменения сегментов FS и/или GS потока Pico; обычно эти функции используются кодом пользовательского режима для хранения указателей на локальную структуру потока (например, TEB в Windows);
- функция для завершения потока Pico и аналогичная функция для процесса Pico;
- функция для приостановки потока Pico и функция для возобновления его выполнения.

Как видите, при помощи этих функций поставщик Pico может создавать полностью настраиваемые процессы и потоки, для которых он управляет исходным состоянием, сегментными регистрами и сопутствующими данными. Тем не менее само по себе это еще не позволяет эмулировать другую операционную систему. Также передается второй набор указателей на функции — на этот раз от поставщика к ядру. Эти функции служат функциями обратного вызова при выполнении некоторых операций, представляющих интерес, потоком или процессом Pico:

- для вызова системной функции потоком Pico с использованием команды `SYSCALL`;

- для инициирования исключений потоками Pico;
- для ошибок страниц при выполнении операций пробирования и блокировки со списками дескрипторов памяти (MDL, Memory Descriptor List) в потоках Pico;
- для запроса имени процесса Pico вызывающей стороной;
- для запроса трассировки стека пользовательского режима процесса Pico из WETW (Event Tracing for Windows);
- для попытки приложения открыть дескриптор для процесса Pico или потока Pico;
- для запроса на завершение процесса Pico;
- для неожиданного завершения потока Pico или процесса Pico.

Кроме того, поставщик Pico также использует защиту от модификации ядра KPP (Kernel Patch Protection), описанную в главе 7, для защиты как своих обратных вызовов, так и системных вызовов с целью предотвращения регистрации вредоносных поставщиков Pico поверх нормальных.

Становится ясно, что, при таком беспрецедентном доступе к любому возможному переходу из пользовательского режима в режим ядра или видимым взаимодействиям режима ядра с пользовательским режимом между процессом/потоком Pico и внешним миром, вся функциональность может быть полностью инкапсулирована поставщиком Pico (и соответствующими библиотеками пользовательского режима) для формирования реализации совершенно другого ядра, отличного от ядра Windows (конечно, с некоторыми исключениями, потому что правила планирования потоков и правила управления памятью все еще продолжают действовать). Правильно написанные приложения не должны быть чувствительными к таким внутренним алгоритмам, потому что эти алгоритмы могут изменяться даже в пределах операционной системы, в которой они обычно выполняются.

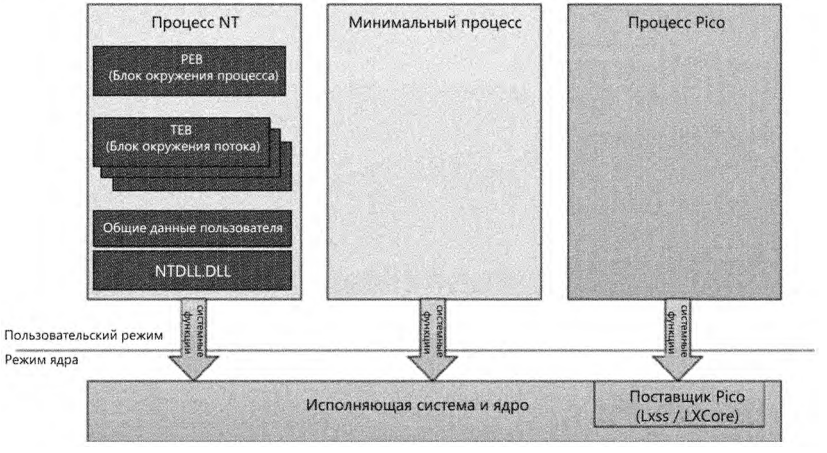


Рис. 3.8. Типы процессов

Итак, поставщики Pico по сути представляют собой специально написанные модули ядра, реализующие необходимые обратные вызовы для реакции на список возможных событий (см. выше), которые могут инициироваться процессами Pico. Так у WSL появляется возможность выполнения немодифицированных двоичных файлов Linux ELF в пользовательском режиме, которая ограничивается только полнотой эмуляции системных функций и сопутствующей функциональности.

Чтобы картина сравнения обычных процессов NT с минимальными процессами и процессами Pico была более полной, на рис. 3.8 представлены структуры каждой разновидности процессов.

Трастлеты (безопасные процессы)

Как упоминалось в главе 2, в Windows появились новые средства обеспечения безопасности на основе виртуализации (VBS, Virtualization-Based Security) — Device Guard и Credential Guard. Они повышают безопасность данных операционной системы и пользователя за счет применения гипервизора. Вы видели, как одна из таких функций, Credential Guard (более подробно рассматривается в главе 7), работает в новой среде изолированного пользовательского режима (IUM). Последняя хотя и является непривилегированной (кольцо 3), имеет виртуальный уровень доверия 1 (VTL 1), что дает ей защиту от обычного мира VTL 0, в котором существуют как ядро NT (кольцо 0), так и приложения (кольцо 3). Посмотрим, как ядро организует выполнение таких процессов и какие структуры данных используются такими процессами.

Структура трастлета

Прежде всего, хотя трастлеты представляют собой файлы в традиционном для Windows формате PE (Portable Executable), они обладают рядом дополнительных свойств, специфических для IUM.

- Они могут импортировать только из ограниченного набора системных DLL Windows (C/C++ Runtime, KernelBase, Advapi, RPC Runtime, CNG Base Crypto и NTDLL) из-за ограниченного числа системных функций, доступных для трастлетов. Математические DLL, работающие только со структурами данных (такие, как NTLM, ASN.1 и т. д.), тоже доступны, поскольку они не вызывают никаких системных функций.
- Возможно импортирование из доступной для них системной DLL-библиотеки, специфической для IUM. Эта библиотека (`Iumbase`) предоставляет интерфейс Base IUM System API с поддержкой слотов сообщений, блоков хранения данных, криптографии и т. д. Библиотека в конечном итоге обращается с вызовом к Iumdll.dll — версии Ntdll.dll для VTL 1, которая содержит безопасные системные функции (системные функции, реализованные безопасным ядром и не передаваемые обычному ядру VTL 0).

- Они содержат раздел PE с именем `.tPolicy`, в котором находится экспортированная глобальная переменная с именем `s_IumPolicyMetadata`. Она содержит метаданные, на основе которых безопасное ядро реализует параметры политики, связанные с разрешением доступа к трастлету из VTL 0 (возможность отладки, поддержка аварийного дампа и т. д.).
- Они подписываются сертификатом, содержащим EKU изолированного пользовательского режима (1.3.6.1.4.1.311.10.3.37). На рис. 3.9 представлены данные сертификата для файла LsaIso.exe, в которых отображается его IUM EKU.

Кроме того, для запуска трастлета при использовании `CreateProcess` должен быть указан конкретный атрибут процесса — как для запроса выполнения в IUM, так и для задания конкретных свойств запуска. Ниже приводится описание метаданных политики и атрибутов процесса.

Рис. 3.9. EKU сертификат трастлета

Метаданные политики трастлетов

Метаданные политики включают различные параметры для настройки того, насколько «доступен» будет трастлет из VTL 0. Они описываются структурой, присутствующей в упоминавшемся ранее экспорте `s_IumPolicyMetadata`, которая содержит номер версии (в настоящее время равный 1) и идентификатор трастлета — уникальное число, которое однозначно определяет этот конкретный трастлет среди существующих (например, BioIso.exe присвоен идентификатор трастлета 4).

Наконец, метаданные содержат массив параметров политики. В настоящее время поддерживаются параметры, перечисленные в табл. 3.4. Поскольку эти политики являются частью подписанных данных, любая попытка изменить их приведет к нарушению подписи IUM, и выполнение станет невозможным.

Таблица 3.4. Параметры политики трастлетов

Политика	Описание	Дополнительная информация
ETW	Включает или отключает ETW	
Отладка	Настраивает отладку	Отладка может быть включена постоянно, только при отключении SecureBoot или при использовании механизма «запрос/ответ» по требованию
Аварийный дамп ядра	Включает или запрещает аварийный дамп	
Ключ аварийного дампа	Задает открытый ключ для шифрования аварийного дампа	Дампы могут отправляться в группу продукта из компании Microsoft. У этой группы имеется закрытый ключ для расшифровки
GUID аварийного дампа	Задает идентификатор для ключа аварийного дампа	Открывает возможность использования/идентификации группой продукта нескольких ключей
Родительский дескриптор безопасности	Формат SDDL	Используется для проверки ожидаемого владельца/родительского процесса
Версия родительского дескриптора безопасности	Идентификатор версии формата SDDL	Используется для проверки ожидаемого владельца/родительского процесса
SVN	Версия безопасности	Уникальное число, которое может использоваться трастлетом (наряду с его идентификатором) при шифровании сообщений AES256/GCM
Идентификатор устройства	Идентификатор PCI безопасного устройства	Трастлет может взаимодействовать только с безопасным устройством (Secure Device), имеющим совпадающий идентификатор PCI
Возможность	Активизирует возможности VTL 1	Открывает доступ к API создания безопасных разделов, DMA, MMIO-доступ пользовательского режима к безопасным устройствам и API безопасного хранения
Идентификатор сценария	Идентификатор сценария для этого двоичного файла	Этот идентификатор, закодированный в формате GUID, должен задаваться трастлетами при создании разделов безопасного образа, чтобы гарантировать их соответствие заранее известному сценарию

Атрибуты трастлета

Запуск трастлета требует правильного использования атрибута `PS_CP_SECURE_PROCESS`, который сначала используется для проверки того, что вызывающая сторона действительно хочет создать трастлет, а также что выполняется именно тот трастлет, который *хочет* выполнить вызывающая сторона. Для этого в атрибут встраивается идентификатор трастлета, который должен совпадать с идентификатором трастлета, указанным в метаданных политики. Кроме того, может быть задан один или несколько атрибутов, представленных в табл. 3.5.

Таблица 3.5. Атрибуты трастлетов

Атрибут	Описание	Дополнительная информация
Ключ почтового ящика	Используется для получения данных почтового ящика	Почтовые ящики (mailbox) используются трастлетами для обмена данными со средой VTL 0 (для этого должен быть известен ключ трастлета)
Идентификатор взаимодействия	Задает идентификатор взаимодействия, который должен использоваться при работе с IUM API безопасного хранилища	Безопасное хранилище позволяет трастлетам обмениваться данными друг с другом — при условии, что они имеют одинаковый идентификатор взаимодействия (Collaboration ID). Если идентификатор отсутствует, вместо него используется идентификатор экземпляра трастлета
Идентификатор сеанса TK	Определяет идентификатор сеанса, используемый с Crypto	

Встроенные системные трастлеты

На момент написания книги Windows 10 содержит пять разных трастлетов, которые определяются идентификаторами, перечисленными в табл. 3.6. Обратите внимание: идентификатор трастлета 0 представляет само безопасное ядро.

Таблица 3.6. Встроенные трастлеты

Имя двоичного файла (идентификатор трастлета)	Описание	Параметры политики
Lsalso.exe (1)	Трастлет защиты учетных данных и ключей	Разрешить ETW, Запретить отладку, Разрешить шифрование аварийного дампа
Vmsp.exe (2)	Рабочий процесс безопасной виртуальной машины (трастлет vTPM)	Разрешить ETW, Запретить отладку, Запретить аварийный дамп, Разрешить возможность безопасного хранилища, Проверить родительский дескриптор безопасности S-1-5-83-0 (NT VIRTUAL MACHINE\Virtual Machines)

Имя двоичного файла (идентификатор трастлета)	Описание	Параметры политики
Unknown (3)	Трастлет регистрации ключей vTPM	Неизвестно
BioIso.exe (4)	Трастлет безопасной биометрии	Разрешить ETW, Запретить отладку, Разрешить шифрование аварийного дампа
FsIso.exe (5)	Трастлет сервера безопасных фреймов	Запретить ETW, Разрешить отладку, Разрешить возможность создания безопасных разделов, Использовать идентификатор сценария {AE53FC6E-8D89-4488-9D2E-4D008731C5FD}

Идентификация трастлета

У трастлетов есть несколько механизмов идентификации, которые могут использоваться системой.

- **Идентификатор трастлета.** Целое число, жестко запрограммированное в метаданных политики трастлета, которое также должно использоваться в атрибутах создания процесса трастлета. Оно гарантирует, что системе будет известен лишь небольшой набор трастлетов, а при вызове будет запущен нужный трастлет.
- **Экземпляр трастлета.** Криптографически безопасное 16-байтовое случайное число, сгенерированное безопасным ядром. Без использования идентификатора взаимодействия экземпляр трастлета гарантирует, что API безопасного хранилища разрешит только одному экземпляру трастлета читать/записывать данные в двоичный объект хранения данных.
- **Идентификатор взаимодействия.** Используется, когда трастлет хочет разрешить другим трастлетам с тем же идентификатором или другими экземплярами того же трастлета предоставить доступ к тому же двоичному объекту хранения данных. Если идентификатор присутствует, идентификатор экземпляра трастлета будет игнорироваться при вызове API-функций чтения и записи.
- **Версия безопасности (SVN, Security Version).** Используется для трастлетов, требующих сильного криптографического доказательства происхождения подписанных или зашифрованных данных. Используется при шифровании данных AES256/GCM механизмами Credential и Key Guard, а также используется службой Cryptograph Report.
- **Идентификатор сценария.** Используется для трастлетов, создающих именованные (основанные на идентификационных данных) безопасные объекты ядра, такие как безопасные разделы. Это значение GUID подтверждает, что трастлет создает такие объекты, как часть заранее определенного сценария (для этого

объекты помечаются в пространстве имен этим кодом GUID). Соответственно, другие трастлеты, желающие открыть объекты с одинаковыми именами, должны иметь одинаковый идентификатор сценария. Также обратите внимание на то, что присутствовать могут сразу несколько идентификаторов сценариев, но трастлеты не могут использовать более одного из них.

Изолированные службы пользовательского режима

К преимуществам запуска в качестве трастлета относится не только защита от атак из нормальной среды (VTL 0), но и доступ к привилегированным и защищенным системным функциям, которые предоставляются только безопасным ядром трастлетам. К их числу относятся следующие механизмы.

- **Безопасные устройства** (`IumCreateSecureDevice`, `IumDmaMapMemory`, `IumGetDmaEnabler`, `IumMapSecureIo`, `IumProtectSecureIo`, `IumQuerySecureDevice Information`, `IopUnmapSecureIo`, `IumUpdateSecureDeviceState`). Предоставляют доступ к безопасным устройствам ACPI и/или PCI, недоступным из VTL 0 и монопольно принадлежащим безопасному ядру (и его вспомогательным механизмам безопасного слоя HAL и безопасного PCI). Трастлеты с соответствующими возможностями (см. «Метаданные политики трастлетов» ранее в этой главе) могут отображать регистры таких устройств в VTL 1 IUM, а также выполнять пересылку данных средствами DMA (Direct Memory Access). Кроме того, трастлеты могут служить драйверами устройств пользовательского режима для такого оборудования, используя инфраструктуру SDF (Secure Device Framework) в SDFHost.dll. Эта функциональность используется средствами безопасной биометрии для Windows Hello, такими как безопасные смарт-карты USB (через PCI) или веб-камера/сканера отпечатков пальцев (через ACPI).

- **Безопасные разделы** (`IumCreateSecureSection`, `IumFlushSecureSectionBuffers`, `IumGetExposedSecureSection`, `IumOpenSecureSection`). Предоставляют возможность совместного использования физических страниц с применением драйвера VTL 0 (который использует `VslCreateSecureSection`) через безопасные разделы, а также позволяют организовать совместный доступ к данным исключительно с VTL 1 в форме именованных безопасных разделов (с использованием механизма, описанного ранее в разделе «Идентификация трастлета») с другими трастлетами или экземплярами того же трастлета. Для использования этой функциональности трастлетам необходима возможность безопасных разделов (см. раздел «Метаданные политики трастлетов»).

- **Почтовые ящики** (`IumPostMailbox`). Позволяют трастлету совместно использовать до 8 слотов, содержащих около 4 Кбайт данных, с компонентом нормального (VTL 0) ядра, которое может вызывать `VslRetrieveMailbox` с передачей идентификатора слота и секретного ключа почтового ящика. Например, Vid.sys в VTL 9 использует его для получения различных секретов, используемых механизмом vTPM из трастлета Vmsp.exe.

- **Идентификационные ключи** (`IumGetIdk`). Позволяют трастлету получить либо уникальный ключ дешифрования, либо ключ подписи. Эти ключи уникальны для машины и могут быть получены только от трастлета. Они являются важнейшей частью механизма Credential Guard, в котором они используются для однозначной проверки машины и того, что учетные данные поступили от IUM.
- **Криптографические функции** (`IumCrypto`). Позволяют трастлету шифровать и дешифровать данные с локальным и/или действующим на уровне загрузки сеансовым ключом, сгенерированным безопасным ядром и доступным только для IUM, для получения дескрипторов привязки, для получения режима FIPS безопасного ядра и для получения затравки генератора случайных чисел (ГСЧ), генерируемой только безопасным ядром для IUM. Также позволяет трастлету генерировать отчеты, снабженные подписью IDK, хешированием SHA-2 и временной пометкой, которые содержат идентификационные данные и SVN трастлета, дамп его метаданных политики, признак его подключения к отладчику, а также любые другие запрашиваемые данные, находящиеся под управлением трастлета. Такие данные могут использоваться как своего рода метрика трастлета, аналогичная TPM и доказывающая, что трастлет не подвергался злонамеренному вмешательству.
- **Безопасное хранилище** (`IumSecureStorageGet`, `IumSecureStoragePut`). Позволяет трастлетам использовать возможность безопасного хранилища (описанную ранее в разделе «Метаданные политики трастлетов») для сохранения больших двоичных объектов (BLOB, Binary Large Object) произвольного размера и их последующего чтения либо по уникальному экземпляру трастлета, либо по тому же идентификатору взаимодействия, как у другого трастлета.

Системные функции, доступные для трастлетов

В своем стремлении к минимизации поверхности атаки безопасное ядро пытается предоставить небольшое подмножество (менее 50) сотен системных функций, которые могут вызываться обычными приложениями (VTL 0). Эти системные функции представляют минимум, необходимый для совместимости с системными DLL-библиотеками, которые могут использоваться трастлетами (см. раздел «Структура трастлета») и сервисами, необходимыми для поддержки исполнительной среды RPC (Rpcrt4.dll) и трассировки ETW.

- **API рабочих фабрик и потоков.** Поддержка API пула потоков (используемого RPC) и слотов TLS, используемых загрузчиков.
- **API информации процесса.** Поддержка слотов TLS и выделения памяти в стеке потоками.
- **API событий, семафоров, ожидания и завершения.** Поддержка пула потоков и синхронизации.
- **API расширенного локального вызова процедур (ALPC).** Поддержка локальных вызовов RPC на основе транспорта ncalrpc.

- **API информации о системе.** Поддержка чтения информации безопасной загрузки, основной системной информации и информации NUMA для Kernel32.dll и масштабирования пула потоков, быстродействия и подмножеств информации времени.
- **API маркеров.** Минимальная поддержка олицетворения RPC.
- **API выделения виртуальной памяти.** Поддержка выделения памяти диспетчером кучи пользовательского режима.
- **API разделов.** Поддержка функциональности загрузчика (для образов DLL) и функциональности безопасных разделов (созданных/открытых безопасными системными функциями, описанными выше).
- **API управления трассировкой.** Поддержка ETW.
- **API исключений и продолжения работы.** Поддержка структурированной обработки исключений (Structured Exception Handling, SEH).

Из списка очевидно, что поддержка таких операций, как операции ввода/вывода с устройствами (с файлами или физическими устройствами), невозможна (прежде всего из-за отсутствия API `CreateFile`), как поддержка операций ввода/вывода с реестром. Также отсутствует поддержка создания других процессов и использование графических API (в VTL 1 нет драйвера Win32k.sys). Как следствие, трастлеты должны быть изолированными служебными рабочими процессами (в VTL 1) для своих сложных «напарников», взаимодействующих с пользователем (в VTL 0); вся передача данных осуществляется только через ALPC или безопасные разделы (дескриптор которого будет передан через ALPC). В главе 7 «Безопасность» мы подробнее обсудим реализацию конкретного трастлета — Lsalso.exe, предоставляющего механизмы Credential и Key Guard.

ЭКСПЕРИМЕНТ: ИДЕНТИФИКАЦИЯ БЕЗОПАСНЫХ ПРОЦЕССОВ

Кроме идентификации по имени безопасные процессы могут быть идентифицированы в отладчике ядра. Во-первых, каждый безопасный процесс имеет безопасный идентификатор PID, представляющий дескриптор в таблице дескрипторов безопасного ядра. Он используется нормальным (VTL 0) ядром при создании потоков в процессе и запросах на его завершение. Во-вторых, с самими потоками связывается потоковый объект cookie, который представляет их индекс в таблице потоков безопасного ядра.

Попробуйте ввести следующие команды в отладчике ядра:

```
lkd> !for_each_process .if @@(((nt!_EPROCESS*)${@#Process})->Pcb.SecurePid) {
.printf "Trustlet: %ma (%p)\n", @@(((nt!_EPROCESS*)${@#Process})-
>ImageFileName),
@#Process }
Trustlet: Secure System (ffff9b09d8c79080)
Trustlet: LsaIso.exe (ffff9b09e2ba9640)
Trustlet: BioIso.exe (ffff9b09e61c4640)
lkd> dt nt!_EPROCESS ffff9b09d8c79080 Pcb.SecurePid
```

```
   +0x000 Pcb           :
      +0x2d0 SecurePid     : 0x00000001'40000004
lkd> dt nt!_EPROCESS ffff9b09e2ba9640 Pcb.SecurePid
   +0x000 Pcb           :
      +0x2d0 SecurePid     : 0x00000001'40000030
lkd> dt nt!_EPROCESS ffff9b09e61c4640 Pcb.SecurePid
   +0x000 Pcb           :
      +0x2d0 SecurePid     : 0x00000001'40000080
lkd> !process ffff9b09e2ba9640 4
PROCESS ffff9b09e2ba9640
    SessionId: 0  Cid: 0388    Peb: 6cdc62b000  ParentCid: 0328
    DirBase: 2f254000  ObjectTable: ffffc607b59b1040  HandleCount:  44.
    Image: LsaIso.exe
        THREAD ffff9b09e2ba2080  Cid 0388.038c  Teb: 0000006cdc62c000
Win32Thread:
0000000000000000 WAIT
lkd> dt nt!_ETHREAD ffff9b09e2ba2080 Tcb.SecureThreadCookie
   +0x000 Tcb                    :
      +0x31c SecureThreadCookie     : 9
```

Порядок работы функции CreateProcess

В этой главе уже были показаны различные структуры данных, используемые при работе с состоянием процесса и при управлении процессом, а также возможность исследования этой информации с помощью различных инструментальных средств и команд отладчика. В данном разделе вы увидите, как и когда эти структуры данных создаются и заполняются, а также получите общее представление о том, как создается и завершается процесс. Как вы уже видели, все документированные функции создания процессов в конечном итоге вызывают `CreateProcessInternalW`, поэтому мы начнем именно с этой функции.

Создание процесса Windows состоит из нескольких этапов, выполняемых тремя частями операционной системы: Windows-библиотекой Kernel32.dll, работающей на стороне клиента (реальная работа начинается с вызова `CreateProcessInternalW`), исполняющей системой Windows и процессом подсистемы Windows (Csrss). Поскольку архитектура подсистем Windows рассчитана на разные варианты окружения, создание объекта процесса исполняющей системы (который может использоваться другими подсистемами) отделено от работы, выполняемой при создании процесса подсистемы Windows. Поэтому, хотя следующее описание порядка выполнения Windows-функции `CreateProcess` выглядит довольно сложным, нужно иметь в виду, что часть работы характерна для семантики, добавленной подсистемой Windows, в отличие от основной работы, необходимой для создания объекта процесса исполняющей системы.

В следующем списке собраны основные этапы создания процесса с помощью Windows-функций `CreateProcess*`. Операции, выполняемые на каждом этапе, подробно рассматриваются в следующих разделах.

ПРИМЕЧАНИЕ Многие шаги `CreateProcess` связаны с конфигурацией виртуального адресного пространства процесса, а следовательно, относятся ко многим понятиям управления памятью и структурами, описанными в главе 5.

1. Проверка приемлемости параметров; преобразование флагов и настроек подсистемы Windows в их внутренние аналоги; разбор, проверка и преобразование списка атрибутов во внутренний аналог.
2. Открытие файла образа (`.exe`), предназначенного для выполнения внутри процесса.
3. Создание объекта процесса исполняющей системы.
4. Создание исходного потока (стека, контекста и объекта потока исполняющей системы Windows).
5. Инициализация процесса, специфическая для подсистемы Windows, выполняемая после его создания.
6. Начало выполнения исходного потока (если только не был установлен флаг создания приостановленного потока `CREATE_SUSPENDED`).
7. Завершение в контексте нового процесса и потока инициализации адресного пространства (например, загрузка требуемых DLL-библиотек) и передача управления в точку входа программы.

На рис. 3.10 изображена диаграмма действий Windows при создании процесса.

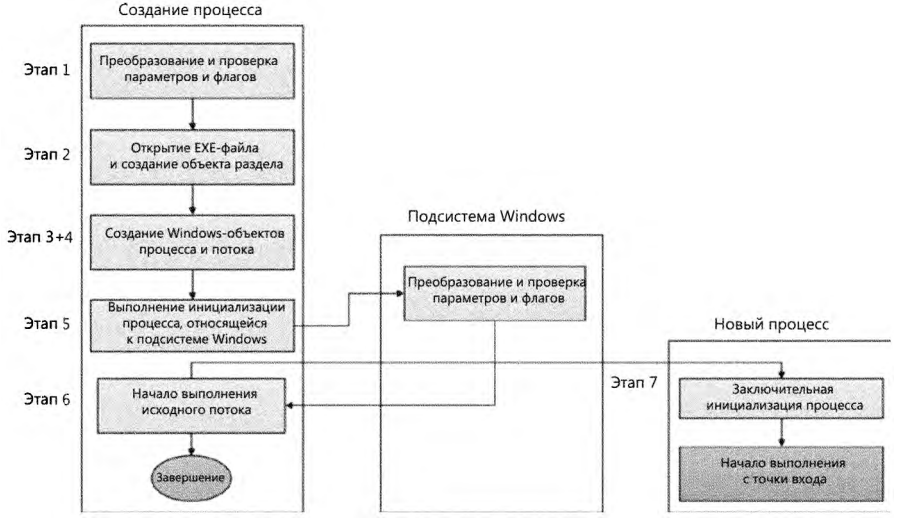


Рис. 3.10. Основные этапы создания процесса

Этап 1. Преобразование и проверка параметров и флагов

Прежде чем открыть на запуск исполняемый образ, функция `CreateProcessInternalW` выполняет действия, описанные ниже.

1. Класс приоритета нового процесса задается независимыми битами в параметре `CreationFlags` функции `CreateProcess*`. Поэтому для одного вызова `CreateProcess*` можно указать более одного класса приоритетов. Чтобы назначить процессу класс приоритета, Windows выбирает для него самый низкий класс приоритета.

   Всего определены шесть классов приоритета процессов, каждый из которых связывается с определенным числом:

   - простой (Idle) или низкий (Low) (4);
   - ниже обычного (Below Normal) (6);
   - обычный (Normal) (8);
   - выше обычного (Above Normal) (10);
   - высокий (High) (13);
   - приоритет реального времени (Real-time) (24).

   Класс приоритета используется как базовый приоритет для потоков, созданных в этом процессе. Это значение не влияет напрямую на сам процесс — только на принадлежащие ему потоки. Описание классов приоритетов процесса и его влияния на планирование потоков приводится в главе 4.

2. Если класс приоритета для нового процесса не указан, по умолчанию для него устанавливается класс приоритета Normal. Если для нового процесса указан класс приоритета Real-time и код, вызвавший процесс, не имеет привилегии на повышение приоритета при планировании (`SE_INC_BASE_PRIORITY_NAME`), то вместо него используется класс приоритета High. Иначе говоря, попытка создания процесса не завершается неудачей только потому, что у вызывающего кода были недостаточные привилегии для создания процесса с классом приоритета Real-time; просто у нового процесса не будет такого высокого приоритета, как Real-time.

3. Если флаги создания определяют, что процесс будет подвергаться отладке, Kernel32 приступает к подключению к исходному коду отладки в Ntdll.dll путем вызова функции `DbgUiConnectToDbg` и получает дескриптор объекта отладки из блока переменных окружения (TEB) текущего потока.

4. Библиотека Kernel32.dll устанавливает по умолчанию жесткий режим ошибки, если таковой указан во флагах создания.

5. Указанный пользователем список атрибутов преобразуется из формата подсистемы Windows в собственный формат, и к нему добавляются внутренние

атрибуты. Возможные атрибуты, которые могут быть добавлены к списку атрибутов, перечислены в табл. 3.7; туда включены и их документированные аналоги из Windows API, если таковые имеются.

ПРИМЕЧАНИЕ Список атрибутов, переданный при вызове CreateProcess*, позволяет вернуть на сторону вызова информацию, выходящую за рамки простого кода статуса, например адрес TEB исходного потока или сведения о разделе образа. Это необходимо при использовании защищенных процессов, поскольку родитель не сможет запросить эту информацию после создания дочернего процесса.

Таблица 3.7. Атрибуты процесса

Исходный атрибут	Эквивалентный атрибут Windows	Тип	Описание
PS_CP_PARENT_PROCESS	PROC_THREAD_ATTRIBUTE_PARENT_PROCESS. Также используется при повышении привилегий	Входной	Дескриптор родительского процесса
PS_CP_DEBUG_OBJECT	Отсутствует — применяется при использовании флага DEBUG_PROCESS	Входной	Объект отладки, если начинается отладка процесса
PS_CP_PRIMARY_TOKEN	Отсутствует — применяется при использовании CreateProcessAsUser/WithTokenW	Входной	Маркер доступа процесса, если была использована процедура CreateProcessAsUser
PS_CP_CLIENT_ID	Отсутствует — возвращается в виде параметра Win32 API (PROCESS_INFORMATION)	Выходной	Возвращение TID и PID исходного потока и процесса
PS_CP_TEB_ADDRESS	Отсутствует — предназначается для внутреннего использования	Выходной	Возвращение адреса TEB исходного потока
PS_CP_FILENAME	Отсутствует — используется в качестве параметра API-функций CreateProcess	Входной	Имя создаваемого процесса
PS_CP_IMAGE_INFO	Отсутствует — предназначается для внутреннего использования	Выходной	Возвращение структуры SECTION_IMAGE_INFORMATION с информацией о версии, флагах и подсистеме исполняющей системы, а также размере стека и точке входа
PS_CP_MEM_RESERVE	Отсутствует — предназначается для внутреннего использования SMSS и CSRSS	Входной	Массив данных резервирования виртуальной памяти, которые должны быть выполнены при создании начального адресного пространства процесса с гарантированной доступностью, поскольку другие операции выделения памяти пока не выполнялись

Исходный атрибут	Эквивалентный атрибут Windows	Тип	Описание
PS_CP_PRIORITY_CLASS	Отсутствует — передается в параметре API-функции CreateProcess	Входной	Класс приоритета, который будет назначен процессу
PS_CP_ERROR_MODE	Отсутствует — передается через флаг CREATE_DEFAULT_ERROR_MODE	Входной	Жесткий режим обработки ошибок для процесса
PS_CP_STD_HANDLE_INFO	Отсутствует — предназначается для внутреннего использования	Входной	Выбор между дублированием стандартных дескрипторов или созданием новых дескрипторов
PS_CP_HANDLE_LIST	PROC_THREAD_ATTRIBUTE_HANDLE_LIST	Входной	Список дескрипторов, принадлежащих родительскому процессу, который должен быть унаследован новым процессом
PS_CP_GROUP_AFFINITY	PROC_THREAD_ATTRIBUTE_GROUP_AFFINITY	Входной	Группа (группы) процессоров, на которой разрешается выполнение потока
PS_CP_PREFERRED_NODE	PROC_THREAD_ATTRIBUTES_PREFERRED_NODE	Входной	Предпочитаемый (идеальный) узел, который должен быть связан с процессом. Касается узла, на котором будет создана куча исходного процесса и стек потока (см. главу 5)
PS_CP_IDEAL_PROCESSOR	PROC_THREAD_ATTTRIBUTE_IDEAL_PROCESSOR	Входной	Предпочитаемый (идеальный) процессор, на котором должно планироваться выполнение потока
PS_CP_UMS_THREAD	PROC_THREAD_ATTRIBUTE_UMS_THREAD	Входной	Содержит UMS-атрибуты, список завершения и контекст
PS_CP_MITIGATION_OPTIONS	PROC_THREAD_MITIGATION_POLICY	Входной	Содержит информацию о том, какие средства устранения рисков (SEHOP, ATL Emulation, NX) должны быть включены/отключены для процесса
PS_CP_PROTECTION_LEVEL	PROC_THREAD_ATTRIBUTE_PROTECTION_LEVEL	Входной	Одно из допустимых значений защиты процессов, представленных в табл. 3.1, или значение PROTECT_LEVEL_SAME, обозначающее такой же уровень защиты, как у родителя
PS_CP_SECURE_PROCESS	Отсутствует — предназначается для внутреннего использования	Входной	Означает, что процесс должен выполняться как трастлет изолированного пользовательского режима (IUM.) Подробнее см. в главе 2 части 2
PS_CP_JOB_LIST	Отсутствует — предназначается для внутреннего использования	Входной	Включает процесс в список заданий

Таблица 3.7 (*окончание*)

Исходный атрибут	Эквивалентный атрибут Windows	Тип	Описание
PS_CP_CHILD_PROCESS_POLICY	PROC_THREAD_ATTRIBUTE_CHILD_PROCESS_POLICY	Входной	Указывает, разрешено ли новому процессу создавать дочерние процессы прямо или косвенно (например, с использованием WMI)
PS_CP_ALL_APPLICATION_PACKAGES_POLICY	PROC_THREAD_ATTRIBUTE_ALL_APPLICATION_PACKAGES_POLICY	Входной	Указывает, должен ли маркер контейнера приложения быть исключен из проверок ACL, включающих группу ALL APPLICATION PACKAGES. Вместо нее будет использоваться группа ALL RESTRICTED APPLICATION PACKAGES
PS_CP_WIN32K_FILTER	PROC_THREAD_ATTRIBUTE_WIN32K_FILTER	Входной	Указывает, должны ли вызовы процессом многих системных функций GDI/USER к Win32k.sys отфильтровываться (блокироваться), или же они должны быть разрешены, но подвергнуты аудиту. Используется браузером Microsoft Edge для сокращения поверхности атаки
PS_CP_SAFE_OPEN_PROMPT_ORIGIN_CLAIM	Отсутствует — предназначается для внутреннего использования	Входной	Используется функциональностью Mark of the Web для обозначения того, что файл поступил от непроверенного источника
PS_CP_BNO_ISOLATION	PROC_THREAD_ATTRIBUTE_BNO_ISOLATION	Входной	Связывает первичный маркер процесса с изолированным каталогом BaseNamedObjects. (Подробнее об именованных объектах см. в главе 8 части 2.)
PS_CP_DESKTOP_APP_POLICY	PROC_THREAD_ATTRIBUTE_DESKTOP_APP_POLICY	Входной	Указывает, будет ли разрешено современному приложению запускать традиционные настольные приложения, и если разрешено — то как именно
Отсутствует — предназначается для внутреннего использования	PROC_THREAD_ATTRIBUTE_SECURITY_CAPABILITIES	Входной	Указатель на структуру SECURITY_CAPABILITIES,которая используется для создания маркера контейнера приложения для процесса перед вызовом NtCreateUserProcess

6. Если процесс является частью объекта задания, но флаги создания запрашивают отдельную виртуальную машину DOS (VDM), то флаг игнорируется.

7. Атрибуты безопасности процесса и исходного потока, переданные функции `CreateProcess`, преобразуются в их внутреннее представление (структуры `OBJECT_ATTRIBUTES`, документированные в WDK).

8. `CreateProcessInternalW` проверяет, должен ли процесс создаваться как современное приложение. Это происходит в том случае, если этот режим задается атрибутом (`PROC_THREAD_ATTRIBUTE_PACKAGE_FULL_NAME`) с полным именем пакета или сам создатель является современным (и родительский процесс не был явно задан атрибутом `PROC_THREAD_ATTRIBUTE_PARENT_PROCESS`). В таком случае вызывается внутренняя функция `BasepAppXExtension` для получения дополнительной контекстной информации о параметрах современного приложения, которая описывается структурой `APPX_PROCESS_CONTEXT`. В структуре хранится такая информация, как имя пакета (во внутренней терминологии — *моникер пакета*), возможности, связанные с приложением, текущий каталог процесса и флаг полного доверия. Возможность создания современных приложений с полным доверием не предоставляется для открытого использования; она резервируется для приложений, обладающих современным оформлением, но выполняющих операции системного уровня. Классический пример — приложение Параметры (Settings) в Windows 10 (SystemSettings.exe).

9. Если процесс создается как современный, то возможности безопасности (если они предоставляются `PROC_THREAD_ATTRIBUTE_SECURITY_CAPABILITIES`) сохраняются для исходного создания маркера вызовом внутренней функции `BasepCreateLowBox`. Суффикс `LowBox` относится к «песочнице» (контейнеру приложения), в которой будет выполняться процесс. Хотя создание современных процессов прямым вызовом `CreateProcess` не поддерживается (вместо этого следует использовать интерфейсы COM, упоминавшиеся ранее), в Windows SDK и MSDN документирована возможность создания традиционных настольных приложений для контейнера приложения с передачей этого атрибута.

10. Если создается современный процесс, устанавливается флаг, который приказывает ядру пропустить обнаружение встроенного манифеста. У современных процессов встроенного манифеста быть не должно, потому что он просто не нужен. (Современное приложение имеет собственный манифест, не связанный со встроенным манифестом, о котором идет речь.)

11. Если установлен флаг отладки (`DEBUG_PROCESS`), то параметр `Debugger` в разделе реестра `Image File Execution Options` (упоминаемый в следующем разделе) для исполняемого файла помечается для игнорирования. В противном случае отладчик никогда не сможет создать отлаживаемый процесс, так как создание зациклится (попытки создания процесса отладки будут происходить снова и снова).

12. Все окна связаны с рабочими столами — графическими представлениями рабочей области. Если рабочий стол не указан в структуре `STARTUPINFO`, то процесс связывается с текущим рабочим столом вызывающей стороны.

ПРИМЕЧАНИЕ Поддержка виртуального рабочего стола в Windows 10 не использует множественные объекты рабочих столов (в смысле объектов ядра). Рабочий стол только один, но окна отображаются и скрываются по мере необходимости. В этом виртуальный рабочий стол отличается от программы desktop.exe из пакета Sysinternals, которая действительно

создает до четырех объектов рабочего стола. Различия видны при попытке перемещения окна с одного рабочего стола на другой. В случае desktops.exe это сделать не удастся, поскольку в Windows такая операция не поддерживается. С другой стороны, виртуальный рабочий стол Windows 10 позволяет это сделать, так как реальное «перемещение» при этом не происходит.

13. Приложение и аргументы командной строки `CreateProcessInternalW` анализируются, путь к исполняемому файлу преобразуется во внутреннее имя NT (например, c:\temp\a.exe преобразуется в строку вида \device\harddiskvolume1\temp\a.exe), потому что некоторым функциям нужен именно такой формат.
14. Большая часть собранной информации преобразуется в одну большую структуру типа `RTL_USER_PROCESS_PARAMETERS`.

Как только эти действия завершатся, функция `CreateProcessInternalW` выполняет начальный вызов `NtCreateUserProcess` для попытки создания процесса. Поскольку к этому моменту времени Kernel32.dll еще не знает, соответствует ли имя образа приложения Windows-приложению, пакетному файлу (.bat или .cmd) или 16-разрядному приложению (приложению DOS), вызов может окончиться неудачей, в таком случае функция `CreateProcessInternalW` ищет причину ошибки и пытается выправить ситуацию.

Этап 2. Открытие образа, предназначенного для исполнения

На данный момент создающий поток переключился в режим ядра и продолжает работу в реализации системной функции `NtCreateUserProcess`.

1. `NtCreateUserProcess` сначала проверяет аргументы и формирует внутреннюю структуру для хранения всей информации создания. Повторная проверка аргументов гарантирует, что вызов не поступил от вредоносной программы, которой удалось имитировать переход Ntdll.dll в режим ядра при помощи фиктивных аргументов.
2. Как показано на рис. 3.11, далее функция `NtCreateUserProcess` ищет соответствующий Windows-образ, который запустит исполняемый файл, указанный вызывающему коду, и создаст объект раздела, чтобы затем отобразить его на адресное пространство нового процесса. Если по какой-нибудь причине вызов потерпит неудачу, управление вернется функции `CreateProcessInternalW` со статусом сбоя (см. табл. 3.8), что заставит `CreateProcessInternalW` повторить попытку выполнения.
3. Если процесс должен быть создан как защищенный, также проверяется политика подписывания.
4. Если процесс должен быть создан как современный, выполняется проверка, которая подтвердит, что процесс лицензирован и ему разрешено выполнение.

Если приложение было заранее установлено с Windows, ему разрешается выполнение независимо от лицензии. Если разрешена загрузка неопубликованных приложений (настраиваемая в приложении Параметры (Settings)), то будет выполнено любое подписанное приложение, а не только приложение из магазина.

Рис. 3.11. Выбор активируемого Windows-образа

5. Если процесс является трастлетом, то объект раздела должен создаваться со специальным флагом, разрешающим его использование безопасным ядром.

6. Если указанный исполняемый файл является Windows-файлом в формате EXE, функция `NtCreateUserProcess` пытается открыть файл и создать для него объект раздела. Объект пока не отображается на память, но он открывается. Успешное открытие объекта раздела еще не означает, что файл является действительным Windows-образом; это может быть DLL-библиотека или исполняемый POSIX-файл. Если это исполняемый POSIX-файл, вызов завершается неудачей, потому что POSIX более не поддерживается. Если файл является DLL-библиотекой, функция `CreateProcess` также дает сбой.

7. После того как функция `NtCreateUserProcess` нашла настоящий исполняемый Windows-образ, она обращается в раздел реестра HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options в поисках подраздела с именем файла и расширением исполняемого образа (но без каталога и информации о пути, например Notepad.exe). Если такой подраздел существует, функция `PspAllocateProcess` ищет в этом подразделе параметр `Debugger`. Если такой параметр существует, образ, предназначенный для запуска, становится строкой в этом параметре, и функция `CreateProcessInternalW` перезапускается с этапа 1.

СОВЕТ Вы можете воспользоваться таким поведением при создании процесса и отладить пусковой код процессов служб Windows до их запуска — вместо того, чтобы подключать отладчик после запуска службы, что не позволяет отладить пусковой код.

8. Если образ не является Windows-файлом в формате EXE (например, если это приложение MS-DOS или Win16), функция `CreateProcessInternalW` выполняет действия по поиску вспомогательного Windows-образа, чтобы его запустить. Этот процесс необходим, потому что приложения, не являющиеся Windows-приложениями, не запускаются напрямую — вместо этого Windows использует один из нескольких специальных вспомогательных образов, которые отвечают за реальный запуск программ, не являющихся Windows-программами. Например, при попытке запустить приложение MS-DOS или Win16 (только в 32-разрядных версиях Windows) запускаемым образом станет исполняемый Windows-файл Ntvdm.exe. Короче говоря, вам не удастся напрямую создать процесс, не являющийся Windows-процессом. Если Windows не сможет решить, какой образ нужно активировать в качестве Windows-процесса (как показано в табл. 3.8), вызов `CreateProcessInternalW` завершается неудачей.

Таблица 3.8. Дерево принятия решений для стадии 1 функции CreateProcess

Если образ...	Код состояния	Будет выполнен образ...	...со следующим результатом
Приложение MS-DOS с расширением .exe, .com или .pif	`PsCreateFailOnSectionCreate`	Ntvdm.exe	`CreateProcessInternalW` перезапускается с этапа 1
Приложение Win16	`PsCreateFailOnSectionCreate`	Ntvdm.exe	`CreateProcessInternalW` перезапускается с этапа 1
Приложение Win64 на 32-разрядной системе (или двоичный файл PPC, MIPS или Alpha)	`PsCreateFailMachineMismatch`	–	`CreateProcessInternalW` завершается неудачей
Имеет параметр Debugger с другим именем образа	`PsCreateFailExeName`	Имя, указанное в параметре Debugger	`CreateProcessInternalW` перезапускается с этапа 1
Недопустимый или поврежденный Windows EXE	`PsCreateFailExeFormat`	–	`CreateProcessInternalW` завершается неудачей
Не открывается	`PsCreateFailOnFileOpen`	–	`CreateProcessInternalW` завершается неудачей
Командная процедура (приложение с расширением .bat или .cmd)	`PsCreateFailOnSectionCreate`	Cmd.exe	`CreateProcessInternalW` перезапускается с этапа 1

Дерево принятия решений, по которому проходит функция `CreateProcess` при запуске образа, выглядит следующим образом.

- Если образ является приложением MS-DOS с расширением .exe, .com или .pif, а Windows является 32-разрядной версией на платформе x86, подсистеме Windows отправляется сообщение о проверке на наличие созданного для этого сеанса вспомогательного процесса MS-DOS (Ntvdm.exe, указанного в параметре реестра HKLM\SYSTEM\CurrentControlSet\Control\WOW\cmdline). Если вспомогательный процесс был создан, он используется для запуска приложения MS-DOS. (Подсистема Windows отправляет процессу VDM (виртуальной DOS-машины) сообщение о запуске нового образа.) Функция `CreateProcessInternalW` возвращает управление. Если поддерживающий процесс создан не был, запускаемый образ изменяется на Ntvdm.exe, и функция `CreateProcessInternalW` перезапускается с этапа 1.
- Если у запускаемого файла расширение .bat или .cmd, запускаемым образом становится Cmd.exe (окно командной строки Windows), и функция `CreateProcessInternalW` перезапускается с этапа 1. (Имя пакетного файла передается Cmd.exe во втором параметре с ключом `/c`.)
- Если образ является исполняемым файлом Win16 (Windows 3.1) в системе на платформе x86, функция `CreateProcessInternalW` должна решить, должен ли для его запуска быть создан новый VDM-процесс или должен использоваться исходный VDM-процесс, предназначенный для всего сеанса (который к этому моменту может быть еще не создан). Этим решением управляют флаги функции `CreateProcessInternalW` с именами `CREATE_SEPARATE_WOW_VDM` и `CREATE_SHARED_WOW_VDM`. Если эти флаги не определены, поведение по умолчанию диктуется значением параметра HKLM\SYSTEM\CurrentControlSet\Control\WOW\DefaultSeparateVDM. Если приложение должно запускаться на отдельной машине VDM, запускаемый образ меняется на ntvdm.exe, за которым следуют некоторые параметры дополнительной настройки и имя 16-разрядного процесса, и функция `CreateProcessInternalW` перезапускается с этапа 1. В противном случае подсистема Windows отправляет сообщение, чтобы понять, существует ли общий VDM-процесс и может ли он быть использован. (Если VDM-процесс запущен на другом рабочем столе или он не запущен на том же уровне безопасности, что и вызывающий код, он не может быть использован и должен быть создан новый VDM-процесс.) Если может использоваться общий VDM-процесс, подсистема Windows отправляет ему сообщение, чтобы запустить новый образ, и функция `CreateProcessInternalW` возвращает управление. Если VDM-процесс еще не был создан (или если он существует, но не может использоваться), запускаемый образ должен поменяться на образ поддержки VDM, и функция `CreateProcessInternalW` перезапускается с этапа 1.

Этап 3. Создание объекта процесса исполняющей системы Windows

На данный момент функция `NtCreateUserProcess` открыла подходящий исполняемый файл Windows и создала объект раздела для отображения его на адресное пространство нового процесса. Затем она создает объект процесса исполняющей

системы Windows для запуска образа путем вызова внутренней системной функции `PspAllocateProcess`. Создание объекта процесса исполняющей системы (осуществляемое путем создания потока) включает в себя следующие подэтапы:

- **3А** – настройка объекта EPROCESS;
- **3Б** – создание исходного адресного пространства процесса;
- **3В** – инициализация находящейся в ядре структуры процесса (`KPROCESS`);
- **3Г** – завершение настройки адресного пространства процесса;
- **3Д** – настройка PEB;
- **3Е** – завершение настройки объекта процесса.

ПРИМЕЧАНИЕ Единственный случай отсутствия родительского процесса относится к инициализации системы (при создании процесса System). После нее для предоставления контекста безопасности для нового процесса всегда требуется родительский процесс.

Этап 3А. Настройка объекта EPROCESS

Этот подэтап включает в себя следующие действия.

1. Наследование сходства (affinity) родительского процесса, если только оно не было явно установлено в ходе создания процесса (через список атрибутов).
2. Выбор идеального узла NUMA, указанного в списке атрибутов (если он был задан).
3. Наследование приоритета ввода/вывода и приоритета страниц (page priority) от родительского процесса. Если родительского процесса не было, используется исходный приоритет страниц (5) и ввода/вывода (`Normal`).
4. Установка для статуса выхода из нового процесса значения `STATUS_PENDING`.
5. Выбор из списка атрибутов жесткого режима обработки ошибок; в противном случае – наследование родительского режима обработки, если режим не был задан. Если родительского процесса не было – использование режима обработки по умолчанию, при котором выводятся все ошибки.
6. Сохранение идентификатора родительского процесса в поле `InheritedFromUniqueProcessId` объекта нового процесса.
7. Запрос раздела реестра Image File Execution Options с целью проверки, должен ли процесс отображаться с использованием больших страниц (значение `UseLargePages` в разделе IFEO), если только процесс не предназначен для выполнения под управлением WoW64; в этом случае большие страницы не используются. Также запрашиваются данные для проверки того, включена ли библиотека NTDLL в список DLL-библиотек, которые должны отображаться с помощью больших страниц внутри этого процесса.

8. Запрос параметра реестра из IFEO (`PerfOptions`, если он существует), который может состоять из любого количества следующих возможных значений: `IoPriority`, `PagePriority`, `CpuPriorityClass` и `WorkingSetLimitInKB`.

9. Если процесс будет работать в WoW64, выделяется вспомогательная структура WoW64 (`EWOW64PROCESS`), которая присваивается полю `Wow64Process` структуры `EPROCESS`.

10. Если процесс должен создаваться в контейнере приложения (как у большинства современных приложений), проверяется, что маркер был создан в режиме `LowBox`. (Контейнеры приложений более подробно рассматриваются в главе 7.)

11. Попытка получения всех привилегий, необходимых для создания процесса. Выбор для процесса класса приоритета `Real-time`, присваивание новому процессу маркера доступа, отображение процесса с использованием больших страниц и создание процесса в рамках нового сеанса — все эти операции требуют соответствующих привилегий.

12. Создание для процесса первичного маркера доступа (как дубликата первичного маркера доступа родительского процесса). Новые процессы наследуют профиль безопасности у своих родителей. Если для указания других маркеров доступа использовалась функция `CreateProcessAsUser`, то происходит соответствующее изменение маркера доступа. Это изменение может произойти только в том случае, если уровень целостности родительского маркера доступа доминирует над уровнем целостности маркера доступа и если маркер доступа является дочерним или смежным по отношению к родительскому. Следует заметить, что при наличии у родителя привилегии `SeAssignPrimaryToken` эти проверки обходятся.

13. Проверка идентификатора сеанса маркера доступа нового процесса с целью определения, не является ли это создание межсеансовым; в этом случае родительский процесс временно прикрепляется к целевому сеансу для корректной обработки квот и создания адресного пространства.

14. Установка блока квот нового процесса на адрес блока квот его родительского процесса и увеличение показания счетчика ссылок для родительского блока квот. Если процесс был создан с помощью функции `CreateProcessAsUser`, это действие пропускается. Вместо него будет создана квота по умолчанию или квота, соответствующая выбранному профилю пользователя.

15. Установка максимального и минимального значений рабочего набора в соответствии со значениями `PspMinimumWorkingSet` и `PspMaximumWorkingSet`. Эти значения могут быть заменены, если в подразделе `PerfOptions` раздела Image File Execution Options были указаны настройки производительности; в таком случае максимальное значение рабочего набора берется оттуда. Следует заметить, что по умолчанию для рабочего набора устанавливаются мягкие ограничения (soft limits), являющиеся по сути рекомендательными, а максимум рабочего набора в `PerfOptions` является жестким ограничением (т. е. рабочему набору не будет разрешено расти выше указанного числа).

16. Инициализация адресного пространства процесса (см. этап 3Б) и открепление от целевого сеанса, если он отличался от текущего.
17. Выбор для процесса группового сходства, если не использовалось наследование сходства. Исходное групповое сходство либо наследуется от родителя, если ранее было установлено распространение NUMA-узла (будет использована группа, владеющая NUMA-узлом), либо будет назначена по кругу. Если система находится в принудительном режиме осведомленности о группах и алгоритмом выбора была выбрана группа 0, вместо нее выбирается группа 1, если она существует.
18. Инициализация `KPROCESS` как части объекта процесса (см. этап 3В).
19. Назначение процессу маркера доступа.
20. Назначение процессу нормального класса приоритета, если только родительским процессом не использовался класс `Idle` или класс `Below Normal`; в таком случае наследуется класс приоритета родительского процесса.
21. Инициализация таблицы дескрипторов процесса. Если для родительского процесса установлен флаг наследования дескрипторов, все наследуемые дескрипторы копируются из таблицы дескрипторов родительского объекта в новый процесс (см. главу 8 части 2). Также можно использовать атрибуты процесса, но только лишь для указания поднабора дескрипторов; такая возможность может пригодиться при использовании функции `CreateProcessAsUser` для ограничения тех объектов, которые должны быть унаследованы дочерним процессом.
22. Применение настроек производительности, если таковые были указаны в подразделе `PerfOptions`. В подраздел `PerfOptions` включаются переопределения ограничений рабочего набора, приоритета ввода/вывода, приоритета страниц и класса приоритета центрального процессора, применяемые для процесса.
23. Вычисление и установка окончательного класса приоритета процесса и квот по умолчанию для его потоков.
24. Чтение и назначение различных параметров устранения рисков, содержащихся в разделе IFEO (в виде одного 64-разрядного значения с именем `Mitigation`). Если процесс выполняется в контейнере приложения, добавляется флаг `TreatAsAppContainer`.
25. Применяются все остальные флаги устранения рисков.

Этап 3Б. Создание исходного адресного пространства процесса

Исходное адресное пространство процесса состоит из следующих страниц:

- каталога страниц (возможно, и не одного для систем с более чем двухуровневыми каталогами страниц, например для систем x86 в PAE-режиме или для 64-разрядных систем);

- страницы гиперпространства;
- страницы битового массива VAD;
- списка рабочего набора.

Для создания этих страниц предпринимаются следующие действия.

1. Для отображения исходных страниц создаются записи в соответствующих таблицах страниц.
2. Количество страниц вычитается из значения переменной ядра `MmTotalCommittedPages` и добавляется к значению переменной `MmProcessCommit`.
3. Используемый по умолчанию общесистемный размер минимального рабочего набора процесса (`PsMinimumWorkingSet`) вычитается из `MmResidentAvailablePages`.
4. Создание страниц таблицы страниц для глобального системного пространства. (То есть помимо страниц, относящихся к процессу, которые только что были рассмотрены, и за исключением памяти, относящейся к конкретному сеансу.)

Этап 3B. Создание находящейся в ядре структуры процесса

На следующем этапе работы функции `PspAllocateProcess` инициализируется структура `KPROCESS` (из поля `Pcb` структуры `EPROCESS`). Эта работа выполняется функцией `KeInitializeProcess`, которая делает следующее.

1. Инициализирует двусвязный список, который соединяет все потоки, являющиеся частью процесса (изначально пуст).
2. Назначает исходное (или получаемое при сбросе) значение кванта времени процесса, используемое по умолчанию (более подробно рассмотрен далее в разделе «Планирование потоков»), которому жестко задается значение 6 до последующей инициализации с помощью `PspComputeQuantumAndPriority`.

ПРИМЕЧАНИЕ Величина кванта по умолчанию различается для клиентских и серверных систем Windows. Подробнее о квантах потоков см. в разделе «Планирование потоков» главы 4.

3. Базовый приоритет процесса устанавливается на основе вычислений, сделанных на этапе 3A.
4. Задается сходство процессора для потоков процесса в формате группового сходства. Групповое сходство вычисляется на этапе 3A или наследуется от родителя.
5. Процессу назначается резидентное (resident) состояние выгрузки.
6. Затравка (seed) потока назначается в зависимости от идеального процессора, который выбран ядром для этого процесса (этот выбор основывается на

идеальном процессоре ранее созданного процесса, что фактически означает случайный выбор на циклической основе). Создание нового процесса приведет к обновлению затравки в `KeNodeBlock` (в исходном блоке NUMA-узла), чтобы следующий новый процесс получил другую затравку идеального процессора.

7. Если процесс является безопасным (Windows 10 и Server 2016), то его безопасный идентификатор создается вызовом `HvlCreateSecureProcess`.

Этап 3Г. Завершение настройки адресного пространства процесса

Процедура подготовки адресного пространства нового процесса достаточно сложна, поэтому давайте рассмотрим все по порядку. Чтобы извлечь из этого раздела максимум пользы, нужно иметь некоторое представление о внутреннем устройстве диспетчера памяти Windows, описанном в главе 5.

Большую часть работы по настройке адресного пространства выполняет функция `MmInitializeProcessAddressSpace`. Она также поддерживает клонирование адресного пространства из другого процесса. Эта возможность была полезна для реализации системной функции POSIX `fork`; возможно, в будущем она также будет использоваться для поддержки других разновидностей `fork` в стиле UNIX (в частности, так `fork` реализуется в подсистеме Windows для Linux в Redstone 1). Следующее описание не учитывает функциональность клонирования адресного пространства, а ограничивается нормальной инициализацией адресного пространства процесса.

1. Диспетчер виртуальной памяти устанавливает в качестве значения последнего времени подгонки процесса текущее время. Диспетчер рабочего набора (который работает в системном потоке диспетчера набора балансировки) использует это значение для определения, когда происходила точная настройка исходного рабочего набора.
2. Диспетчер памяти инициализирует список рабочего набора процесса — теперь становятся возможными ошибки обращения к страницам.
3. Раздел (созданный при открытии файла образа) теперь отображается на адресное пространство нового процесса, а базовый адрес раздела процесса устанавливается на базовый адрес образа.
4. Создается и инициализируется блок окружения процесса (PEB) — см. описание этапа 3Д.
5. Библиотека Ntdll.dll отображается на процесс; если это процесс WoW64, то отображается также и 32-разрядная библиотека Ntdll.dll.
6. При наличии соответствующего запроса для процесса создается новый сеанс. Это особое действие реализовано главным образом для облегчения работы диспетчера сеанса — Session Manager (Smss) при инициализации нового сеанса.

7. Стандартные дескрипторы дублируются, и новые значения записываются в структуру параметров процесса.
8. Обрабатываются любые фиксированные распределения памяти (memory reservations), перечисленные в списке атрибутов. Кроме того, два флага позволяют зарезервировать первый 1 Мбайт или первые 16 Мбайт адресного пространства. Эти флаги используются внутри системы для отображения, к примеру, кода постоянного запоминающего устройства и векторов реального режима (должны быть в нижних диапазонах виртуального адресного пространства, где обычно располагаются куча и другие структуры процесса).
9. Параметры пользовательского процесса записываются в процесс, копируются и исправляются (т. е. переводятся из абсолютной формы в относительную, чтобы понадобился только один блок памяти).
10. Информация о сходстве записывается в PEB.
11. На процесс отображается набор перенаправления *MinWin* API, а указатель на него сохраняется в PEB.
12. Определяется и сохраняется уникальный идентификатор процесса. Ядро не различает идентификаторы и дескрипторы отдельных процессов и потоков. Идентификаторы процессов и потоков (дескрипторы) хранятся в глобальной таблице дескрипторов (`PspCidTable`), которая не связана ни с каким процессом.
13. Если процесс безопасен (т. е. он выполняется в IUM), то он инициализируется и связывается с объектом ядра, представляющим процесс.

Этап 3Д. Настройка PEB

Функция `NtCreateUserProcess` вызывает функцию `MmCreatePeb`, которая сначала отображает таблицы общесистемной поддержки национальных языков — NLS (National Language Support) — на адресное пространство процесса. Затем эта функция вызывает функцию `MiCreatePebOrTeb` для выделения страницы для PEB с последующей инициализацией нескольких полей, значения большинства из которых основаны на значениях внутренних переменных, которые были настроены через реестр, например через значения параметров `MmHeap*`, `MmCriticalSectionTimeout` и `MmMinimumStackCommitInBytes`. Некоторые из этих полей могут быть заменены настройками связанных исполняемых образов, например Windows-версией PE-заголовка или маской родственности в каталоге конфигурации загрузки PE-заголовка.

Если установлен флаг характеристики заголовка образа `IMAGE_FILE_UP_SYSTEM_ONLY` (указывающий на то, что образ может запускаться только на однопроцессорной системе), для запуска всех потоков нового процесса выбирается один и тот же центральный процессор (`MmRotatingUniprocessorNumber`). Процесс выбора выполняется простым циклическим перебором доступных процессоров, при каждом запуске образа этого типа используется следующий процессор. Так образы этого типа распределяются между процессорами равномерно.

Этап 3E. Завершение настройки объекта процесса исполняющей системы

Перед тем как возвращать дескриптор нового процесса, необходимо завершить ряд финальных действий, выполняемых функцией `PspInsertProcess` и ее вспомогательными функциями.

1. Если включен общесистемный аудит процессов (либо в результате настроек локальной политики, либо в результате настроек групповой политики из контроллера домена), в журнале событий безопасности делается запись о создании процесса.
2. Если родительский процесс входил в задание, то это задание восстанавливается из набора уровня заданий родительского процесса, а затем привязывается к сеансу вновь созданного процесса. И наконец, новый процесс добавляется к заданию.
3. Объект нового процесса вставляется в конец Windows-списка активных процессов (`PsActiveProcessHead`). Теперь процесс становится доступным для таких функций, как `EnumProcess` и `OpenProcess`.
4. Принадлежащий родительскому процессу порт отладки процесса копируется в новый дочерний процесс, если не был установлен флаг `NoDebugInherit` (значение которого может быть запрошено при создании процесса). Если порт отладки был указан, он в этот момент подключается к новому процессу.
5. Поскольку теперь объекты заданий могут определять ограничения, касающиеся той группы или тех групп, в которых могут запускаться потоки внутри процессов, являющихся частью задания, функция `PspInsertProcess` должна убедиться в том, что групповое сходство, связанное с процессом, не нарушает группового сходства, связанного с заданием. Также необходимо учесть еще один интересный второстепенный вопрос: не предоставляют ли разрешения задания доступ к изменению разрешений сходства процесса, поскольку менее привилегированный объект задания может конфликтовать с требованиями сходства более привилегированного процесса.
6. И наконец, функция `PspInsertProcess` создает дескриптор нового процесса, вызывая функцию `ObOpenObjectByPointer`, а затем возвращает этот дескриптор вызывающему коду. Следует заметить, что, пока внутри создаваемого процесса не будет создан первый поток, обратный вызов создания процесса не отправляется, а код всегда отправляет обратные вызовы процесса перед отправкой обратных вызовов, связанных с управлением объектами.

Этап 4. Создание исходного потока, а также его стека и контекста

К этому времени настройка объекта процесса исполняющей системы Windows полностью завершена. Но потоков пока еще нет, поэтому что-либо сделать процесс не может. Пора заняться этой работой. Обычно за все аспекты создания потока от-

вечает функция PspCreateThread, и при создании нового потока именно она и вызывается функцией NtCreateThread. Но поскольку исходный поток создается ядром внутри системы без ввода из пользовательского режима, вместо нее вызываются две вспомогательные процедуры, от которых зависят функции PspCreateThread: PspAllocateThread и PspInsertThread. PspAllocateThread управляет собственно созданием и инициализацией самого объекта потока исполняющей системы, а PspInsertThread управляет созданием дескриптора потока и атрибутов безопасности, а также вызовом функции KeStartThread для превращения объекта исполняющей системы в планируемый поток системы. Но поток пока еще ничего не будет делать, он создан в приостановленном состоянии и не возобновит выполнение, пока процесс не будет полностью проинициализирован (см. описание этапа 5).

ПРИМЕЧАНИЕ Параметром потока (который не может быть указан в CreateProcess, но может быть определен в CreateThread) является адрес PEB. Этот параметр будет использоваться кодом инициализации, который запускается в контексте этого нового потока (см. описание этапа 6).

Функция PspAllocateThread выполняет следующие действия.

1. Запрещает потокам, работа которых планируется в пользовательском режиме UMS (User Mode Scheduling), создаваться в процессах WoW64, а также не позволяет вызывающим процедурам пользовательского режима создавать потоки в системном процессе.
2. Создает и инициализирует объект потока исполняющей системы.
3. Если для системы включена оценка энергопотребления (всегда отключено для XBOX), создает и инициализирует структуру THREAD_ENERGY_VALUES, на которую указывает объект ETHREAD.
4. Инициализирует различные списки, используемые LPC, системой управления вводом/выводом и исполняющей системой.
5. Устанавливает для потока время создания и создает его идентификатор потока TID (Thread ID).
6. Перед тем как поток сможет выполняться, ему нужен стек и контекст, в котором он будет запущен, поэтому она настраивает и то и другое. Размер стека для исходного потока берется из образа, способа указать другой размер не существует. Если процесс является процессом WoW64, будет также инициализирован контекст потока WoW64.
7. Выделяет для нового потока его блок переменных окружения — TEB (Thread Environment Block).
8. Сохраняет стартовый адрес потока пользовательского режима в ETHREAD (в поле StartAddress). Это адрес предоставленной системой функции запуска потока в библиотеке Ntdll.dll (RtlUserThreadStart). Указанный пользователем стартовый адрес Windows сохраняется в ETHREAD в другом месте (поле

Win32StartAddress), чтобы такое средство отладки, как Process Explorer, могло запросить информацию.

9. Вызывает для настройки структуры KTHREAD функцию KeInitThread. Для исходного приоритета потока и текущего базового приоритета устанавливаются значения базового приоритета процесса, а для их сходства и кванта времени устанавливаются такие же значения, как и у процесса. Затем функция KeInitThread выделяет для потока стек ядра и инициализирует для потока аппаратный контекст, зависящий от машины, включая фреймы контекста, системных прерываний и исключений. Контекст потока устанавливается таким образом, чтобы поток запускался в режиме ядра в функции KiThreadStartup. И наконец, функция KeInitThread устанавливает состояние потока в Initialized (Инициализирован) и возвращает управление функции PspAllocateThread.
10. Если это UMS-поток, вызывается функция PspUmsInitThread для инициализации UMS-состояния.

Как только эта работа завершится, функция NtCreateUserProcess вызывает функцию PspInsertThread для выполнения следующих действий.

1. Инициализируется идеальный процессор потока, если он был задан атрибутом.
2. Инициализируется групповое сходство потока, если оно было задано атрибутом.
3. Если процесс является частью задания, проводится проверка того, что групповое сходство потока не нарушает ограничений задания (которые были описаны ранее).
4. Проводится проверка того, что процесс не был уже завершен, поток не был уже завершен или что поток даже еще не был в состоянии приступить к работе. Если имеет место любой из этих случаев, создание потока заканчивается неудачей.
5. Если поток является частью безопасного процесса (IUM), создается и инициализируется объект безопасного потока.
6. Путем вызова функции KeStartThread инициализируется структура KTHREAD как часть объекта потока. Здесь предполагается наследование у процесса-владельца настроек планирования, выбор идеального узла и процессора, обновление группового сходства, назначение базового и динамического приоритета (копированием из процесса), назначение кванта потока и вставка потока в список процесса, который ведется структурой KPROCESS (отдельный список от того, который хранится в EPROCESS).
7. Если процесс находится в состоянии глубокой заморозки (запрещено выполнение любых потоков, включая новые потоки), этот поток также замораживается.
8. В системах, отличных от x86, если поток является первым в процессе (а процесс не является процессом Idle), процесс вставляется в другой общесистемный список процессов, который хранится в глобальной переменной KiProcessListHead.

9. Увеличивается счетчик потоков в объекте процесса и наследуются приоритет ввода/вывода процесса-владельца и приоритет страниц. Если достигнуто наивысшее количество когда-либо имевшихся у процесса потоков, обновляется также наивысший показатель счетчика потоков. Если поток был вторым для процесса, замораживается первичный маркер доступа (т. е. он не может быть больше изменен).
10. Поток вставляется в список потоков процесса, и поток приостанавливается, если создающий его процесс требует этого.
11. Объект потока вставляется в таблицу дескрипторов процесса.
12. Если это был первый поток процесса (и поэтому операции проводились как часть выполнения `CreateProcess*`), вызываются также зарегистрированные функции обратного вызова процесса ядра. Затем вызываются все зарегистрированные функции обратного вызова потока. Если какой-либо обратный вызов блокирует создание, попытка завершается неудачей, а вызывающей стороне возвращается соответствующий код статуса.
13. Если был передан список заданий (с использованием атрибута) и поток является первым в процессе, то процесс включается во все задания из списка.
14. Поток подготавливается к выполнению с помощью вызова функции `KeReadyThread`. Он входит в отложенное состояние (подробнее о состояниях потоков см. в главе 4).

Этап 5. Выполнение инициализации, относящейся к подсистеме Windows

Если функция `NtCreateUserProcess` вернет управление с кодом успешного завершения работы, значит, все необходимые объекты процессов и потоков исполняющей системы были созданы. Затем `CreateProcessInternalW` выполняет различные операции, связанные с операциями, характерными для подсистемы Windows, для завершения инициализации процесса.

1. В первую очередь проводятся различные проверки того, должна ли Windows позволить запуститься исполняемому файлу. В их число входят проверка версии образа в заголовке и проверка, не заблокирован ли процесс сертификацией Windows-приложения (через групповую политику). В специализированных выпусках Windows Server 2012 R2, таких как Windows Storage Server 2012 R2, проводятся дополнительные проверки для определения того, не импортировало ли приложение какие-нибудь запрещенные API-функции.
2. Если в отношении программного обеспечения действуют ограничительные политики, для нового процесса создается маркер с ограничениями. После этого отправляется запрос базе данных совместимости приложения, чтобы определить, есть ли для процесса запись либо в реестре, либо в базе данных

системного приложения. Оболочки совместимости на этой стадии не применяются; информация будет сохранена в PEB, как только начнется выполнение исходного потока (на этапе 6).

3. `CreateProcessInternalW` вызывает некоторые внутренние функции (для незащищенных процессов) для получения SxS-информации (см. раздел «Разрешение имен DLL и перенаправление» далее в этой главе) — такой, как файлы манифестов и пути перенаправлений DLL, а также другой информации (например, является ли носитель, на котором находится EXE-файл, съемным, а также флаги обнаружения установки). Для иммерсивных процессов также возвращается информация версии и целевая платформа из манифеста пакета.
4. На основании собранной информации строится сообщение к подсистеме Windows, предназначенное для отправки Csrss. Сообщение включает в себя следующую информацию:
   - путь и путь SxS;
   - дескрипторы процесса и потока;
   - дескриптор сеанса;
   - дескриптор маркера доступа;
   - информация о носителе;
   - данные AppCompat и оболочки совместимости;
   - информация иммерсивного процесса;
   - адрес PEB;
   - различные флаги (например, является ли процесс защищенным или должен ли он выполняться с повышенными правами);
   - флаг, показывающий, принадлежит ли процесс Windows-приложению (чтобы Csrss мог определить, нужно или нет показывать курсор запуска);
   - информация о языке пользовательского интерфейса;
   - флаги DLL-перенаправления и `.local` (см. раздел «Загрузчик образа» далее в этой главе);
   - информация файла манифеста.

При получении этого сообщения подсистема Windows выполняет следующие действия.

1. Функция `CsrCreateProcess` создает дубликат дескриптора для процесса и потока. Здесь же значение счетчика использования процесса и потока увеличивается со значения 1 (установленного во время создания) до значения 2.
2. Выделяет память для структуры процесса Csrss (`CSR_PROCESS`).
3. Устанавливает порт исключения нового процесса в качестве общего функционального порта для подсистемы Windows, чтобы эта подсистема получала сообщение, когда в процессе возникнет исключение второго шанса. (Подробнее об обработке исключений см. в главе 8 части 2.)

4. Если новая группа процессов создается с новым процессом в качестве корня (флаг `CREATE_NEW_PROCESS_GROUP` в `CreateProcess`), то он задается в `CSR_PROCESS`. Группа процессов удобна для отправки управляющего события совокупности процессов, совместно использующих консоль. За дополнительной информацией обращайтесь к описаниям `CreateProcess` и `GenerateConsoleCtrlEvent` в документации Windows SDK.
5. Выделяет память и инициализирует структуру потока Csrss (`CSR_THREAD`).
6. Функция `CsrCreateThread` вставляет поток в список потоков процесса.
7. Увеличивает значение счетчика процессов в сеансе.
8. Устанавливает для уровня завершения процесса значение 0x280 (уровень завершения процесса по умолчанию, дополнительные сведения можно найти в описании `SetProcessShutdownParameters` в документации Windows SDK).
9. Вставляет структуру нового Csrss-процесса в список процессов, относящихся ко всей подсистеме Windows.

После того как Csrss выполнит эти действия, функция `CreateProcess` проверяет, не был ли процесс запущен с повышенными правами (запущен через функцию `ShellExecute` и повышен в правах службой AppInfo после того, как пользователь ответил на соответствующий запрос). Сюда включена проверка того, не был ли процесс программой установки. Если был, маркер процесса открыт, и флаг виртуализации установлен так, чтобы приложение было виртуализировано (UAC и виртуализация рассматриваются в главе 7). Если приложение содержит оболочки совместимости для повышения прав или имело в своем манифесте запрос на повышение уровня прав, процесс уничтожается, а запрос на повышение прав отправляется службе `AppInfo`.

Следует заметить, что большинство этих проверок не проводится для защищенных процессов, потому что эти процессы должны были предназначаться для Windows Vista или более поздних версий Windows, где не было причин для запроса повышенных прав, виртуализации или проверок совместимости приложений и соответствующей обработки. Кроме того, механизмы, допускающие выполнение, например механизмы оболочек совместимости, использующие свои обычные технологии перехвата и внесения исправлений в память, превращались бы в дефекты безопасности, если кто-нибудь нашел способ вставки произвольных оболочек совместимости, изменяющих поведение защищенных процессов. В дополнение к этому, поскольку механизм оболочек совместимости (Shim Engine) устанавливается родительским процессом, у которого не должно быть доступа к его дочернему защищенному процессу, не может работать даже законное применение оболочек совместимости.

Этап 6. Начало выполнения исходного потока

К этому моменту уже определено окружение процесса, выделены ресурсы для использования его потоками, у процесса есть поток, и подсистема Windows знает

о новом процессе. Если вызывающий код не установил флаг создания приостановленного процесса — CREATE_SUSPENDED, то теперь исходный поток приведен в состояние готовности и может приступить к работе, выполнив оставшуюся часть работы по инициализации процесса, которая осуществляется в контексте нового процесса (этап 7).

Этап 7. Выполнение инициализации процесса в контексте нового процесса

Новый поток начинает свою жизнь с выполнения в режиме ядра процедуры запуска потока KiThreadStartup. Эта процедура снижает IRQL-уровень потока с уровня DPC (Deferred Procedure Call) до уровня APC, а затем вызывает системную процедуру исходного потока PspUserThreadStartup. Этой процедуре в качестве параметра передается определенный пользователем стартовый адрес потока. PspUserThreadStartup выполняет следующие действия.

1. Устанавливает цепочку исключений в архитектуре x86. (Другие архитектуры в этом отношении работают иначе, как показано в главе 8 части 2.)
2. IRQL опускается до уровня PASSIVE_LEVEL (0 — единственный уровень IRQL, на котором разрешена работа пользовательского кода).
3. Отключает возможность замены первичного маркера процесса во время выполнения.
4. Если поток был уничтожен при запуске (независимо от причины), он завершается, и никакие дальнейшие действия не предпринимаются.
5. Сохраняет в TEB идентификатор локального контекста и идеальный процессор на основании информации, присутствующей в структурах данных режима ядра, после чего проверяет, действительно ли создание потока прошло неудачно.
6. Вызывает функцию DbgkCreateThread, которая проверяет, отправлялись ли уведомления образов для нового процесса. Если они не отправлялись, а уведомления включены, сначала уведомление отправляется для процесса, а затем для загрузки образа Ntdll.dll.

ПРИМЕЧАНИЕ Это делается на данном этапе, а не в момент изначального отображения образов, потому что к тому времени идентификатор процесса (необходимый для внешних вызовов ядра) еще не был назначен.

7. После завершения этих проверок выполняется еще одна проверка того, не подвергается ли процесс отладке. Если подвергается и если уведомления отладчика еще не отправлялись, сообщение о создании процесса отправляется через объект отладки (если таковой имеется), чтобы событие отладки запуска процесса (CREATE_PROCESS_DEBUG_INFO) могло быть отправлено соответствующему процессу отладчика. Далее следует аналогичное событие отладки запуска потока и еще одно событие отладки для загрузки образа Ntdll.dll.

Затем функция `DbgkCreateThread` ждет ответа от отладчика (через функцию `ContinueDebugEvent`).

8. Процедура проверяет, разрешена ли в системе предварительная выборка приложения, и, если разрешена, вызывает механизм предварительной выборки (и Superfetch) для обработки файла инструкции предвыборки (если таковой имеется) и осуществления предварительной выборки страниц, на которые были ссылки в течение первых 10 секунд последнего запуска процесса. (Подробнее о предварительной выборке и Superfetch см. в главе 5.)

9. Затем функция `PspUserThreadStartup` проверяет, был ли уже установлен общесистемный объект cookie в структуре `SharedUserData`. Если нет, функция генерирует его на основе хеша системной информации, такой как количество обработанных прерываний, доставок DPC и ошибок обращения к страницам, времени прерывания и на случайном числе. Этот общесистемный объект cookie используется во внутреннем декодировании и кодировании указателей, например, в диспетчере кучи для защиты от определенных классов эксплойтов. (Подробнее о безопасности диспетчера кучи см. в главе 5.)

10. Если процесс является безопасным (процесс IUM), то вызывается функция `HvlStartSecureThread`, которая передает управление безопасному ядру для запуска выполнения потока. Функция возвращает управление только при выходе из потока.

11. И наконец, функция `PspUserThreadStartup` устанавливает контекст исходного преобразователя (initial thunk) для запуска процедуры инициализации загрузчика образа (`LdrInitializeThunk` в Ntdll.dll), а также общесистемной заглушки запуска потока — thread startup stub (`RtlUserThreadStart` в Ntdll.dll). Эти действия совершаются путем редактирования контекста потока на месте с последующим вызовом выхода из операции системной службы, которая загружает специально созданный контекст пользователя. Процедура `LdrInitializeThunk` инициализирует загрузчик, диспетчер кучи, NLS-таблицы, массивы локальной памяти потока — TLS (Thread-Local Storage) и локальной памяти волокна — FLS (Fiber-Local Storage), а также структуры критического раздела. Затем она загружает любые требуемые библиотеки DLL и вызывает с помощью функционального кода `DLL_PROCESS_ATTACH` точки входа DLL.

Как только функция вернет управление, процедура `NtContinue` возвращает контекст нового пользователя и возвращается в пользовательский режим — вот теперь действительно запускается выполнение потока.

Функция `RtlUserThreadStart` использует адрес фактической точки входа образа и пусковой параметр и вызывает код точки входа приложения. Адрес и параметр уже были помещены в стек ядром. Эта довольно сложная череда событий преследует две цели.

- Она позволяет загрузчику образа внутри Ntdll.dll провести настройки процесса закулисно внутри самой системы, чтобы другой код пользовательского режима мог должным образом работать. (В противном случае у него не будет кучи, локальной памяти потока и т. д.)

- Когда все потоки начинаются в общей процедуре, это позволяет заключить их в конструкцию обработки исключений, чтобы при их сбое библиотека Ntdll.dll знала об этом и могла вызвать фильтр необработанного исключения внутри библиотеки Kernel32.dll. Это позволяет также скоординировать выход потока по возвращении из стартовой процедуры потока и выполнение различной завершающей работы. Разработчики приложений могут также вызвать функцию `SetUnhandledExceptionFilter`, чтобы добавить их собственный код обработки необработанных исключений.

ЭКСПЕРИМЕНТ: ОТСЛЕЖИВАНИЕ ЗАПУСКА ПРОЦЕССА

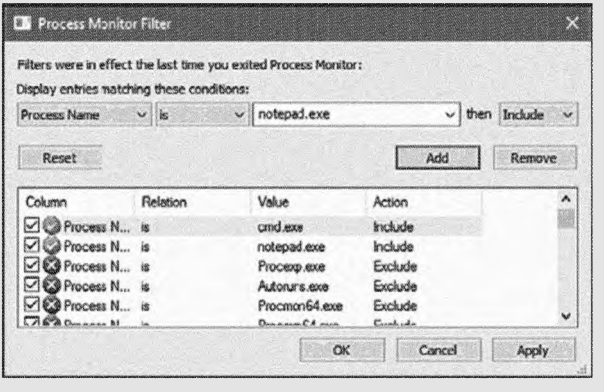
После подробного изучения порядка запуска процесса и различных операций, требуемых для начала выполнения приложения, мы собираемся воспользоваться средством Process Monitor, чтобы посмотреть на файловый ввод/вывод и разделы реестра, к которым было обращение в ходе этого процесса.

Хотя этот эксперимент не дает полной картины всех рассмотренных нами внутренних действий, вы сможете посмотреть на некоторые части системы в действии, а именно: на предвыборку и Superfetch, на варианты выполнения файла-образа и на другие проверки совместимости, а также на отображение DLL-библиотеки загрузчика образа.

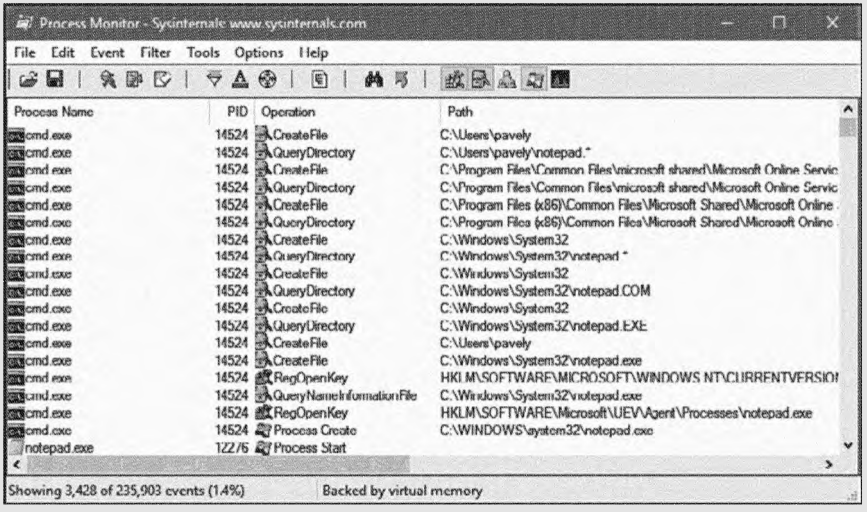
Мы возьмем простой исполняемый файл — Notepad.exe — и запустим его из окна командной строки (Cmd.exe). Существенно то, что мы рассмотрим операции как внутри Cmd.exe, так и внутри Notepad.exe. Вспомним, что большой объем работы в пользовательском режиме выполняется функцией `CreateProcessInternalW`, которая вызывается родительским процессом перед тем, как ядро создаст новый объект процесса.

Чтобы все правильно настроить, выполните следующие действия.

1. Добавьте два фильтра в Process Monitor: один для Cmd.exe и один для Notepad.exe — нужно включить только эти два процесса. Убедитесь в том, что не имеется никаких текущих запущенных экземпляров, чтобы иметь полную уверенность в том, что наблюдаются нужные события. Окно фильтров должно выглядеть так:

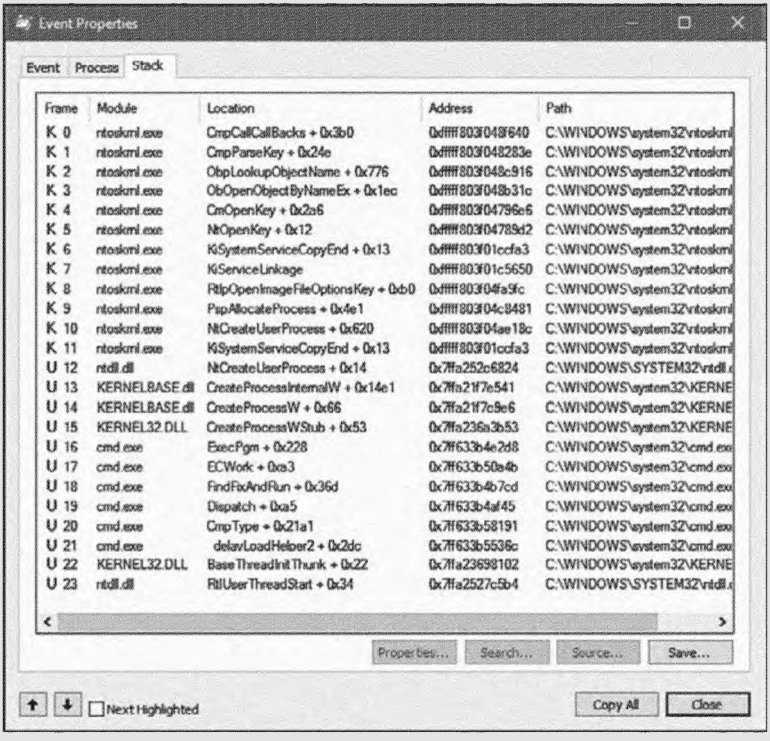


2. Убедитесь в том, что отключена регистрация событий (снят флажок захвата событий — Capture Events в меню File), и запустите окно командной строки.
3. Включите регистрацию событий (откройте меню File и выберите команду Event Logging, или просто нажмите клавиши CTRL+E, или щелкните на значке с увеличительным стеклом на панели инструментов), а затем наберите Notepad.exe и нажмите клавишу Ввод. На обычной Windows-системе вы должны увидеть появление от 500 до 1500 событий.
4. Остановите отслеживание и скройте столбцы Sequence (Последовательность) и Time Of Day (Время дня), чтобы сосредоточиться на интересующих нас столбцах. Окно должно приобрести следующий вид:



Согласно описанию этапа 1 хода выполнения функции CreateProcess, в первую очередь нужно заметить, что перед запуском процесса и созданием первого потока Cmd.exe читает значение параметра реестра HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options\Notepad.exe. Поскольку никаких настроек для образа выполнения, связанного с Notepad.exe, нет, процесс создается в исходном виде.

Как с этим, так и с любым другим событием в журнале Process Monitor, вы можете просмотреть стек событий и увидеть, в каком режиме (пользовательском или режиме ядра) выполнялась каждая часть создания процесса и с помощью каких процедур. Для этого дважды щелкните на событии RegOpenKey и перейдите на вкладку Stack (Стек). На следующей копии экрана показан стандартный стек на 64-разрядной машине с Windows 10:



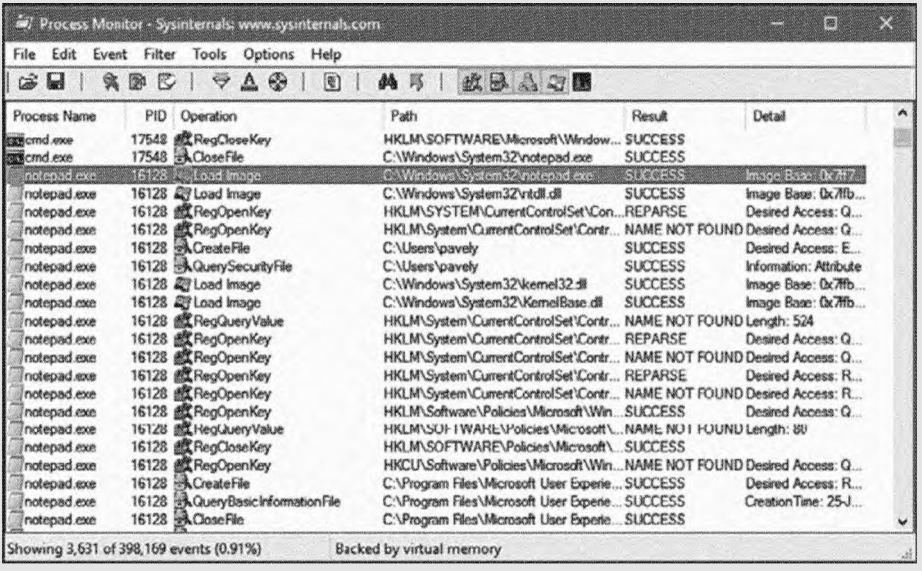
Этот стек показывает, что вы уже достигли той части создания процесса, которая выполняется в режиме ядра (посредством функции `NtCreateUserProcess`), и что за данную проверку отвечает вспомогательная процедура `PspAllocateProcess`.

Спускаясь по списку событий после того, как были созданы потоки и процессы, вы заметите три группы событий:

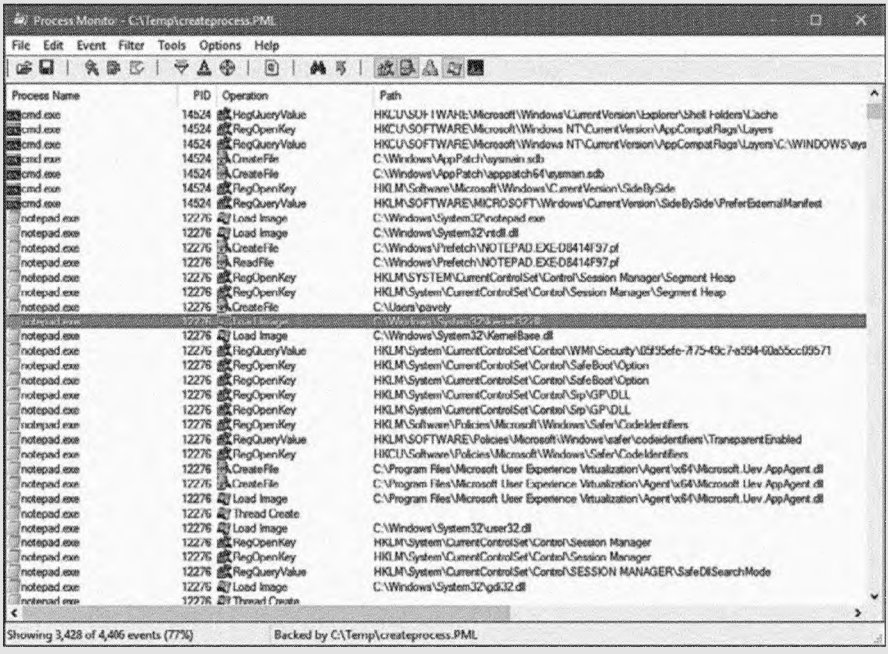
- Простая проверка флагов совместимости приложения, которые должны позволить коду создания процесса, выполняемому в пользовательском режиме, знать, нужно ли через механизм оболочек совместимости проводить проверки внутри базы данных совместимости приложения.
- За этой проверкой следуют несколько считываний разделов Side-By-Side, Manifest и MUI/Language, являющихся частью упомянутой ранее структуры сборки.
- Файловый ввод/вывод в отношении одного или нескольких файлов с расширением .sdb, которые представляют в системе базы данных совместимости приложений. Этот ввод/вывод осуществляется с целью дополнительных проверок, не нужно ли для этого приложения активизировать механизм оболочек совместимости. Поскольку Блокнот (Notepad) является про-

граммой Microsoft с обычным поведением, ему не нужны никакие оболочки совместимости.

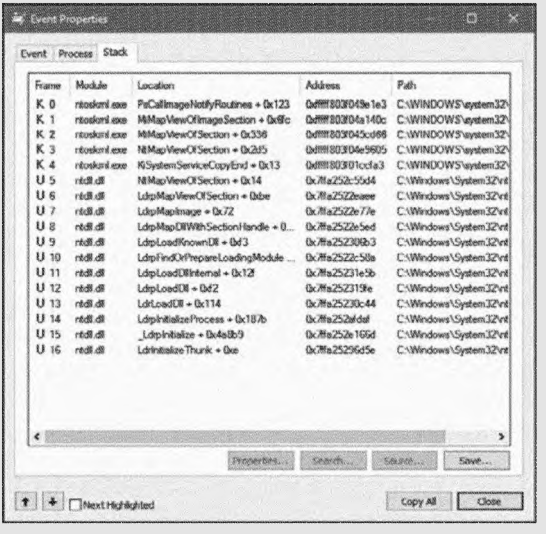
На следующих скриншотах показана дальнейшая череда событий, происходящих внутри самого процесса Notepad. Они инициированы в режиме ядра оболочкой запуска потока в режиме пользователя, которая выполняет ранее рассмотренные действия. Первые два представляют собой отладочные уведомления отладки загрузчика образов Notepad.exe и Ntdll.dll, которые могут быть сгенерированы только сейчас, когда код выполняется внутри контекста процесса Notepad, а не внутри контекста командной строки.



Затем свою лепту вносит механизм предвыборки, который ищет файл своей базы данных, который уже был сгенерирован для Notepad. (Подробнее о предварительной выборке см. в главе 5.) В системе, в которой программа Блокнот уже запускалась хотя бы один раз, эта база данных будет в наличии, и механизм предвыборки начнет выполнение указанных в ней команд. Если это так, опустившись ниже, вы увидите несколько прочитанных и запрошенных DLL-библиотек. В отличие от обычной загрузки DLL-библиотеки, которая осуществляется загрузчиком образа в пользовательском режиме после просмотра таблиц импорта или когда приложение самостоятельно загружает DLL-библиотеку, эти события сгенерированы механизмом предвыборки, который уже знает о тех библиотеках, которые потребуются приложению Блокнот. Обычно за этим следует загрузка образа требуемых DLL-библиотек, и вы увидите события, подобные показанным на следующем скриншоте:



Теперь эти события сгенерированы из кода, запущенного внутри пользовательского режима, который был вызван, как только завершила работу функция оболочки режима ядра. Поэтому это первые события, исходящие из процедуры `LdrpInitializeProcess`, которая вызывается `LdrInitializeThunk` для первого потока в процессе. Вы можете сами в этом убедиться, взглянув на стек этих событий — например, на стек события загрузки образа kernel32.dll, показанный на следующем скриншоте.



Последующие события генерируются этой процедурой и ее вспомогательными функциями до тех пор, пока вы не придете к событиям, сгенерированным функцией `WinMain` внутри Notepad, где будет выполняться код, находящийся под контролем разработчика. Подробное описание всех событий и компонентов пользовательского режима, которые вступают в игру в ходе выполнения процесса, заняло бы всю эту главу, поэтому исследование любых последующих событий остается читателю в качестве самостоятельного упражнения.

Завершение процесса

Процесс одновременно является контейнером и границей. Это означает, что ресурсы, используемые одним процессом, не будут автоматически видны в других процессах, поэтому для передачи информации между процессами должен использоваться некий механизм межпроцессных коммуникаций. А значит, процесс не может случайно перезаписать произвольные байты в памяти другого процесса. Для этого понадобится явный вызов такой функции, как `WriteProcessMemory`. Но чтобы этот способ сработал, необходимо явно открыть дескриптор с подходящей маской доступа (`PROCESS_VM_WRITE`) — а успех этой операции не гарантирован. Естественная изоляция между процессами также означает, что если некоторое исключение происходит в одном процессе, оно никак не повлияет на другие процессы. В худшем случае процесс-источник аварийно завершится, но остальные части системы продолжат работать нормально.

Процесс может корректно завершить свою работу вызовом функции `ExitProcess`. Для многих процессов (в зависимости от настроек компоновщика) стартовый код первого потока в процессе вызывает `ExitProcess` от имени процесса, когда поток возвращает управление из своей главной функции. Под «*корректным завершением*» следует понимать, что DLL-библиотеки, загруженные в процесс, получают возможность выполнить некоторые действия по уведомлению о выходе из процесса, для которого используется вызов их функции `DllMain` с флагом `DLL_PROCESS_DETACH`.

Функция `ExitProcess` может вызываться только самим процессом, запрашивающим свое завершение. Возможно некорректное завершение процесса с использованием функции `TerminateProcess`, которая может вызываться и за пределами процесса. (Например, Process Explorer и диспетчер задач используют эту функцию по запросу пользователя.) Функции `TerminateProcess` передается дескриптор процесса, открытый с маской доступа `PROCESS_TERMINATE`, — может оказаться, что в нем будет отказано. Вот почему бывает нелегко (или невозможно) завершать некоторые процессы (например, Csrss) — дескриптор с необходимой маской доступа не может быть получен пользователем, выдавшим запрос. Под «*некорректностью*» здесь понимается то, что DLL не получает возможности выполнить код (`DLL_PROCESS_DETACH` не отправляется), а все потоки завершаются моментально. В некоторых случаях это может привести к потере данных — например, если файловый кэш не получил возможности записать данные в файл.

Как бы ни прекращалось существование процесса, никаких утечек быть не может. Другими словами, вся закрытая память процесса автоматически освобождается

ядром, адресное пространство уничтожается, закрываются все дескрипторы объектов ядра и т. д. Если открытые дескрипторы процесса все еще существуют (все еще существует структура `EPROCESS`), то другие процессы все равно могут получить доступ к информации управления процессом — например, коду завершения (`GetExitCodeProcess`). После закрытия этих дескрипторов структура `EPROCESS` будет уничтожена, и от процесса действительно ничего не остается.

Несмотря на это, если сторонние драйверы выделяют память в режиме ядра (допустим, из-за IOCTL или просто из-за уведомления процесса), они должны самостоятельно освободить любую такую память. Windows не отслеживает и не зачищает память ядра, принадлежащую процессу (кроме памяти, занимаемой объектами из-за дескрипторов, созданных процессом). Обычно это делается при помощи уведомлений `IRP_MJ_CLOSE` или `IRP_MJ_CLEANUP`, которые сообщают драйверу о закрытии объекта устройства, или через уведомление о завершении процесса. (Подробнее о IOCTL см. в главе 6.)

Загрузчик образов

Когда в системе запускается процесс, ядро создает для его представления объект процесса и выполняет различные задачи инициализации, связанные с ядром. Но эти задачи не приводят к выполнению приложения, они только готовят его контекст и окружение. В действительности, в отличие от драйверов, являющихся кодом режима ядра, приложения выполняются в пользовательском режиме. Поэтому основная часть работы по инициализации проводится вне ядра. Эта работа выполняется *загрузчиком образов*, который во внутренней терминологии обозначается сокращением `Ldr`.

Загрузчик образов находится в системной DLL-библиотеке пользовательского режима Ntdll.dll и в библиотеке ядра не фигурирует. Поэтому он ведет себя как стандартный код, являющийся частью DLL-библиотеки, и на него распространяются те же ограничения относительно доступа к памяти и прав в системе безопасности. Особенность этого кода состоит в том, что он гарантированно всегда находится в запущенном процессе (библиотека Ntdll.dll всегда находится в загруженном состоянии), и в том, что это первый фрагмент кода, запускаемый в пользовательском режиме как часть нового процесса.

Поскольку загрузчик запускается до самого кода приложения, он обычно остается невидимым для пользователей и разработчиков. Кроме того, при всей скрытости инициализационных задач загрузчика программа обычно при своем выполнении взаимодействует с его интерфейсом, например, при каждой загрузке или выгрузке DLL-библиотеки или при запросе базового адреса такой библиотеки. Некоторые важные задачи, за которые отвечает загрузчик:

- инициализация состояния приложения в пользовательском режиме, например создание исходной кучи (динамически размещаемой структуры данных) и подготовка слотов локальной памяти потока (TLS) и локальной памяти волокна (FLS);

- разбор таблицы импорта адресов — IAT (Import Table) — приложения для поиска всех необходимых приложению DLL-библиотек (а затем рекурсивный разбор IAT каждой DLL), за которым следует разбор экспортной таблицы DLL-библиотек, чтобы убедиться в том, что функция уже присутствует (специальные *записи продвижения*, forwarder entries, могут также перенаправить экспорт на другую DLL-библиотеку);
- загрузка и выгрузка DLL-библиотек во время выполнения приложения, а также по требованию, и ведение списка всех загруженных модулей (базы данных модулей);
- обработка файлов манифеста, необходимых для поддержки Windows SxS (Side-by-Side), а также файлов и ресурсов многоязыкового интерфейса (MUI, Multiple Language User Interface);
- чтение базы данных совместимости приложения для любых оболочек совместимости и загрузка, если требуется, DLL-библиотеки механизма оболочек совместимости;
- включение поддержки API-наборов и API-перенаправлений — важной части функциональности OneCore, открывающей возможность создания приложений UWP (Universal Windows Platform);
- разрешение динамического снижения рисков совместимости в процессе выполнения с использованием механизма SwitchBack, а также взаимодействие с механизмами оболочек совместимости и Application Verifier.

Как видите, большинство этих задач играет важную роль в разрешении приложению запускать его код. Без них все, начиная с вызова внешних функций и заканчивая использованием кучи, приведет к немедленному отказу. После того как процесс создан, загрузчик вызывает специальную встроенную API-функцию `NtContinue` для продолжения выполнения на основе фрейма исключения, находящегося в стеке. Этот фрейм исключения, созданный ядром, содержит фактическую точку входа в приложение. Следовательно, поскольку загрузчик не использует стандартный вызов или передачу управления в запущенное приложение, функции инициализации загрузчика никогда не отображаются в дереве вызовов в трассировке стека потока.

ЭКСПЕРИМЕНТ: НАБЛЮДЕНИЕ ЗА РАБОТОЙ ЗАГРУЗЧИКА ОБРАЗОВ

В этом эксперименте глобальные флаги используются для включения функции отладки, которая называется *снимками загрузчика* (loader snaps). Это позволит просмотреть вывод отладки из загрузчика образов при отладке запуска приложения.

1. Из каталога, в котором находится отладчик WinDbg, запустите приложение Gflags.exe, а затем щелкните на вкладке Image File.
2. В поле Image наберите Notepad.exe, а затем нажмите клавишу Tab. Это даст возможность устанавливать флаги. Установите флаг Show Loader Snaps, а затем щелкните на OK или Apply.

3. Запустите WinDbg, откройте меню File, выберите Choose Executable и перейдите к файлу c:\windows\system32\notepad.exe, чтобы запустить его. Вы увидите пару экранов отладочной информации, которые выглядят примерно так:

```
0f64:2090 @ 02405218 - LdrpInitializeProcess - INFO: Beginning execution of
notepad.exe (C:\WINDOWS\notepad.exe)
    Current directory: C:\Program Files (x86)\Windows Kits\10\Debuggers\
    Package directories: (null)
0f64:2090 @ 02405218 - LdrLoadDll - ENTER: DLL name: KERNEL32.DLL
0f64:2090 @ 02405218 - LdrpLoadDllInternal - ENTER: DLL name: KERNEL32.DLL
0f64:2090 @ 02405218 - LdrpFindKnownDll - ENTER: DLL name: KERNEL32.DLL
0f64:2090 @ 02405218 - LdrpFindKnownDll - RETURN: Status: 0x00000000
0f64:2090 @ 02405218 - LdrpMinimalMapModule - ENTER: DLL name: C:\WINDOWS\
System32\KERNEL32.DLL
ModLoad: 00007fff'5b4b0000 00007fff'5b55d000 C:\WINDOWS\System32\KERNEL32.DLL
0f64:2090 @ 02405218 - LdrpMinimalMapModule - RETURN: Status: 0x00000000
0f64:2090 @ 02405218 - LdrpPreprocessDllName - INFO: DLL api-ms-win-core-
rtlsupport-l1-2-0.dll was redirected to C:\WINDOWS\SYSTEM32\ntdll.dll by API set
0f64:2090 @ 02405218 - LdrpFindKnownDll - ENTER: DLL name: KERNELBASE.dll
0f64:2090 @ 02405218 - LdrpFindKnownDll - RETURN: Status: 0x00000000
0f64:2090 @ 02405218 - LdrpMinimalMapModule - ENTER: DLL name: C:\WINDOWS\
System32\KERNELBASE.dll
ModLoad: 00007fff'58b90000 00007fff'58dc6000 C:\WINDOWS\System32\KERNELBASE.dll
0f64:2090 @ 02405218 - LdrpMinimalMapModule - RETURN: Status: 0x00000000
0f64:2090 @ 02405218 - LdrpPreprocessDllName - INFO: DLL api-ms-win-
eventing-provider-l1-1-0.dll was redirected to C:\WINDOWS\SYSTEM32\
kernelbase.dll by API set
0f64:2090 @ 02405218 - LdrpPreprocessDllName - INFO: DLL api-ms-win-core-
apiquery-l1-1-0.dll was redirected to C:\WINDOWS\SYSTEM32\ntdll.dll by API set
```

4. Со временем отладчик прервет выполнение где-то внутри кода загрузчика, в специальном месте, где загрузчик образов проверяет, не присоединен ли к нему отладчик, и сделает из него контрольную точку. Если для продолжения выполнения нажать клавишу g, вы увидите дополнительные сообщения, поступившие от загрузчика, и на экране появится программа Блокнот.

5. Попробуйте поработать в Блокноте, и тогда вы увидите, как те или иные операции приводят к пробуждению загрузчика. Хорошо было бы открыть диалоговое окно сохранения или открытия файла. Тем самым будет продемонстрировано, что загрузчик запускается не только при запуске программы, но и постоянно реагирует на запросы потока, что может привести к *отложенным загрузкам* других модулей (которые затем могут быть выгружены после использования).

Ранняя стадия инициализации процесса

Поскольку загрузчик находится в библиотеке Ntdll.dll, которая является встроенной DLL-библиотекой, не связанной ни с какой конкретной подсистемой, для всех процессов поведение загрузчика практически одинаково (с некоторыми незначительными отличиями). Ранее мы подробно рассмотрели этапы, приводящие к созданию процесса в режиме ядра, а также некоторую работу, производимую Windows-функцией создания процесса `CreateProcess`. А здесь будет рассмотрена

вся остальная работа, совершаемая в пользовательском режиме, независимо от какой-либо подсистемы, как только начнет выполняться первая инструкция пользовательского режима.

При запуске процесса загрузчик выполняет следующие действия.

1. Проверяет, было ли `LdrpProcessInitialized` уже присвоено значение 1 или в TEB установлен флаг `SkipLoaderInit`. В этом случае загрузчик пропускает всю инициализацию и ожидает три секунды, пока кто-нибудь вызовет `LdrpProcessInitializationComplete`. Эта возможность используется в тех случаях, когда отражение процесса используется механизмом Windows Error Reporting, или при других попытках ветвления, при которых инициализация загрузчика не нужна.
2. Флагу `LdrInitState` присваивается значение 0, означающее, что загрузчик не был инициализирован. Также флагам `ProcessInitializing` в PEB и `RanProcessInit` в TEB присваивается значение 1.
3. Инициализирует блокировку загрузчика в PEB.
4. Инициализирует таблицу динамических функций, используемую для поддержки раскрутки стека/исключений в коде JIT.
5. Инициализирует раздел MRDATA (Mutable Read Only Heap), используемый для хранения глобальных переменных, относящихся к безопасности, которые не должны изменяться эксплойтами (подробнее см. в главе 7).
6. Инициализирует базу данных загрузчика в PEB.
7. Инициализирует таблицы поддержки национальных языков (NLS) для процесса.
8. Строит полное имя образа для приложения.
9. Сохраняет обработчики исключений SEH для раздела `.pdata` и строит внутренние таблицы исключений.
10. Сохраняет преобразователи системных вызовов для пяти критических функций загрузчика: `NtCreateSection`, `NtOpenFile`, `NtQueryAttributesFile`, `NtOpenSection` и `NtMapViewOfSection`.
11. Читает параметры устранения рисков для приложения (которые передаются ядром через экспортируемую переменную `LdrSystemDllInitBlock`). Эти параметры более подробно описаны в главе 7.
12. Запрашивает содержимое раздела реестра `Image File Execution Options` (IFEO) для приложения. Здесь хранятся такие параметры, как глобальные флаги (хранящиеся в `GlobalFlags`), параметры отладки кучи (`DisableHeapLookaside`, `ShutdownFlags` и `FrontEndHeapDebugOptions`), параметры загрузчика (`UnloadEventTraceDepth`, `MaxLoaderThreads`, `UseImpersonatedDeviceMap`), параметры ETW (`TracingFlags`), а также ряд других параметров, включая `MinimumStackCommitInBytes` и `MaxDeadActivationContexts`. В процессе этой

работы инициализируется пакет Application Verifier и сопутствующие DLL-библиотеки, а из `CFGOptions` читаются настройки CFG (Control Flow Guard).

13. Проверяет по заголовку исполняемого файла, является ли образ приложением .NET (на что указывает присутствие каталога образов, специфического для .NET) и является ли он 32-разрядным. Также загрузчик обращается с запросом к ядру, чтобы узнать, является ли образ процессом WoW64. При необходимости обрабатываются 32-разрядные образы, содержащие только IL, для которых WoW64 не требуется.

14. Загружает любые конфигурационные настройки, указанные в справочнике конфигурации загрузки образа исполняемого файла. Эти настройки, которые разработчик может определить при компиляции приложения (и которые используются компилятором и компоновщиком для реализации некоторых средств безопасности и устранения рисков — таких, как CFG), управляют поведением исполняемого файла.

15. Выполняет минимальную инициализацию FLS и TLS.

16. Задает параметры отладки для критических разделов, создает базу данных трассировки стека пользовательского режима при установке соответствующего глобального флага и запрашивает значение `StrackTraceDatabaseSizeInMb` из раздела IFEO.

17. Инициализирует диспетчер кучи для процесса и создает первую кучу процесса. При этом для настройки необходимых параметров используются различные параметры конфигурации загрузки, файлов образов, глобальных флагов и параметров заголовка исполняемого файла.

18. Включает защитный режим завершения процесса при повреждении кучи, если соответствующий параметр установлен.

19. Инициализирует журнал диспетчеризации исключений, если эта функция включена соответствующим глобальным флагом.

20. Инициализирует пакет пула потоков, поддерживающий API пула потоков. При этом запрашивается и принимается во внимание информация NUMA.

21. Инициализирует и преобразует блок окружения и блок параметров, особенно при необходимости поддержки процессов WoW64.

22. Открывает каталог объектов \KnownDlls и строит путь к DLL. Для процессов WoW64 используется каталог \KnownDlls32.

23. Для приложения магазина читаются параметры политики модели приложения, закодированные в утверждениях WIN://PKG и WP://SKUID маркера (подробнее см. в разделе «Контейнеры приложений» главы 7).

24. Определяет текущий каталог процесса, системный путь и путь загрузки по умолчанию (используемый при загрузке образов и открытии файлов), а также правил, касающихся порядка поиска DLL по умолчанию. При этом читаются

текущие настройки политики для универсальных приложений (UWP), приложений Desktop Bridge (Centennial) и Silverlight (Windows Phone 8) (или служб).

25. Строит первую запись таблицы данных загрузчика для Ntdll.dll и вставляет ее в базу данных модулей.
26. Строит таблицу истории раскрутки.
27. Инициализирует параллельный загрузчик, который используется для загрузки всех зависимостей (не имеющих перекрестных зависимостей) с использованием пула потоков и параллельных потоков.
28. Строит следующую запись таблицы данных загрузчика для главного исполняемого файла и вставляет ее в базу данных модулей.
29. При необходимости выполняет перемещение главного исполняемого образа.
30. Инициализирует средство Application Verifier, если эта функция включена.
31. Инициализирует ядро WoW64, если это процесс WoW64. В этом случае 64-разрядный загрузчик завершит инициализацию, а 43-разрядный загрузчик перехватит управление и перезапустит многие операции, описанные до настоящего момента.
32. Если это образ .NET, загрузчик проверяет его, загружает Mscoree.dll (оболочка совместимости среды .NET) и получает главную точку входа исполняемого файла (`_CorExeMain`); при этом заменяется запись исключения с назначением этой точки входа вместо обычной функции `main`.
33. Инициализирует слоты TLS процесса.
34. Для приложений подсистемы Windows загрузчик вручную загружает Kernel32.dll и Kernelbase.dll независимо от фактического состояния импортирования процесса. При необходимости эти библиотеки используются для инициализации механизмов SRP/Safer (Software Restriction Policies). Наконец, он разрешает все зависимости набора API-функций, которые существуют конкретно между этими двумя библиотеками.
35. Инициализирует механизм оболочек совместимости и анализирует базу данных оболочек совместимости.
36. Включает параллельный загрузчик образов при условии, что к базовым функциям загрузчика, просканированным ранее, не присоединены никакие перехватчики системных функций. Также учитывается количество потоков загрузчика, заданное в политике и IFEO.
37. Переменной `LdrInitState` присваивается значение 1, означающее «выполняется загрузка импортирования».

Теперь загрузчик образов готов к разбору таблицы импорта исполняемых файлов, принадлежащей приложению, и приступает к загрузке любых DLL-библиотек, которые были динамически скомпонованы во время компиляции приложения. Это

происходит и с образами .NET, данные импорта которых обрабатываются вызовами функций среды .NET, и с обычными образами. Поскольку каждая импортируемая DLL-библиотека может также иметь свою собственную таблицу импорта, эта операция будет продолжаться в рекурсивном режиме до тех пор, пока не будут удовлетворены запросы всех DLL-библиотек и не будут найдены все импортируемые функции. По мере загрузки каждой DLL-библиотеки загрузчик будет сохранять для нее информацию состояния и создавать базу данных модулей.

В более новых версиях Windows загрузчик вместо этого заранее строит карту зависимостей, конкретные узлы которой описывают одну DLL-библиотеку и ее граф зависимостей; таким образом определяются узлы, которые могут загружаться параллельно. В некоторые моменты рабочая очередь пула потоков «сливается», что служит точкой синхронизации. Одна из таких точек располагается перед вызовом всех функций инициализации DLL для всех статических импортируемых функций (одна из последних стадий работы загрузчика). Когда это будет сделано, вызываются все статические инициализаторы TLS. Наконец, для Windows-приложений между этими двумя шагами сначала вызывается функция преобразователя инициализации ядра Kernel32 (`BaseThreadInitThunk`), а потом функция Kernel32, вызываемая в конце инициализации процесса.

Разрешение имен DLL-библиотек и перенаправление

Разрешение (resolution) имен — процесс, с помощью которого система преобразует имя двоичного файла PE-формата в имя физического файла в ситуациях, когда вызывающий модуль не указал файл или не может однозначно его идентифицировать. Поскольку размещение различных каталогов (каталога приложения, системного каталога и т. д.) во время компоновки не может быть жестко задано, этот процесс включает разрешение всех двоичных зависимостей, а также операций `LoadLibrary`, в которых вызывающий модуль не указал полный путь.

При разрешении двоичных зависимостей основная модель Windows-приложений размещает файлы в пути поиска (список тех мест, в которых ведется последовательный поиск файлов с подходящим базовым именем), хотя различные системные компоненты переопределяют механизм пути поиска для расширения стандартной модели приложения. Понятие пути поиска является пережитком эпохи командной строки, когда понятие текущего каталога приложения еще имело смысл; для современных приложений с графическим интерфейсом это представляется анахронизмом.

Но размещение текущего каталога в таком порядке позволяет замещать операции загрузки системных двоичных файлов путем установки вредоносных двоичных файлов с такими же базовыми именами в текущий каталог приложения. Чтобы исключить риски безопасности, связанные с подобным поведением, в механизм поиска был добавлен *безопасный режим поиска DLL-библиотек*, активный по умол-

чанию для всех процессов. В безопасном режиме поиска текущий каталог перемещается в позицию за тремя системными каталогами, что приводит к следующему упорядочению поиска пути.

1. Каталог, из которого было запущено приложение.
2. Основной системный каталог Windows (например, C:\Windows\System32).
3. 16-разрядный системный каталог Windows (например, C:\Windows\System).
4. Каталог Windows (например, C:\Windows).
5. Текущий каталог на момент запуска приложения.
6. Любые каталоги, указанные в переменной среды окружения %PATH%.

Для каждой последующей операции загрузки DLL-библиотеки путь поиска DLL вычисляется заново. Алгоритм вычисления пути поиска аналогичен алгоритму, используемому для вычисления пути поиска по умолчанию, но приложение может изменять конкретные элементы пути, изменяя значение переменной %PATH% с помощью API-функции `SetEnvironmentVariable`, сменяя текущий каталог с помощью API-функции `SetCurrentDirectory` или указывая для процесса каталог DLL-библиотеки с помощью API-функции `SetDllDirectory`. При указании каталога DLL-библиотеки этот каталог заменяет в пути поиска текущий каталог, и загрузчик игнорирует безопасный режим поиска DLL, установленный для процесса.

Вызывающие модули могут также изменять путь поиска DLL для конкретных операций загрузки, вызывая API-функцию `LoadLibraryEx` с флагом `LOAD_WITH_ALTERED_SEARCH_PATH`. Если предоставлен этот флаг и в предоставленном APL-функции имени DLL указывается полная строка поиска, при вычислении пути поиска для операции вместо каталога приложения используется путь, содержащий DLL-файл. Учтите, что для относительных путей такое поведение становится неопределенным и создает потенциальную угрозу. При загрузке приложений Desktop Bridge (Centennial) этот флаг игнорируется.

Также при вызове `LoadLibraryEx` приложения могут указывать флаги `LOAD_LIBRARY_SEARCH_DLL_LOAD_DIR`, `LOAD_LIBRARY_SEARCH_APPLICATION_DIR`, `LOAD_LIBRARY_SEARCH_SYSTEM32` и `LOAD_LIBRARY_SEARCH_USER_DIRS` вместо флага `LOAD_WITH_ALTERED_SEARCH_PATH`. Каждый из этих флагов изменяет порядок поиска так, чтобы поиск производился только в конкретном каталоге (или каталогах), описываемом флагом, или же флаги могут объединяться для поиска в нескольких местах. Например, объединение каталога приложения, System32 и пользовательского каталога дает режим `LOAD_LIBRARY_SEARCH_DEFAULT_DIRS`. Более того, эти флаги могут устанавливаться глобально при помощи API-функции `SetDefaultDllDirectories`, которая влияет на все загрузки библиотек с этого момента.

На порядок поиска также можно повлиять другим способом: если приложение является пакетным или же не является пакетной службой или старым приложением Silverlight 8.0 для Windows Phone. В таких условиях порядок поиска DLL не использует традиционный механизм и API, а ограничивается поиском по пакетному

графу. Это же происходит и при использовании API-функции `LoadPackagedLibrary` вместо обычной функции `LoadLibraryEx`. Пакетный граф вычисляется на основании записей `<PackageDependency>` в разделе `<Dependencies>` файла манифеста приложения UWP и гарантирует, что никакие посторонние DLL-библиотеки не будут случайно загружены в пакете.

Кроме того, при загрузке пакетного приложения, которое не является приложением Desktop Bridge, все API-функции изменения пути поиска DLL (вроде тех, которые были представлены выше) блокируются, и используется только системное поведение по умолчанию (в сочетании с поиском пакетных зависимостей для большинства приложений UWP).

К сожалению, даже с режимом безопасного поиска и стандартными алгоритмами поиска для старых приложений, которые всегда начинают с каталога приложения, двоичный файл все равно может быть скопирован из своего обычного каталога в другой, доступный для пользователя (например, из c:\windows\system32\notepad.exe в c:\temp\notepad.exe; такая операция не требует административных прав). В такой ситуации атакующий может поместить специально созданную DLL-библиотеку в один каталог с приложением, и с учетом описанного порядка такая библиотека подменит системную DLL-библиотеку. После этого подмененная библиотека может использоваться для влияния на работу приложения, которое может быть привилегированным (особенно если пользователь, не подозревая об изменениях, повысит привилегии приложения через UAC). Для защиты от подобных атак процессы и/или администраторы могут использовать политику устранения рисков процесса `Prefer System32 Images` (подробнее см. в главе 7), которая меняет порядок пунктов 1 и 2 в приведенном выше списке.

Перенаправление имен DLL

Перед попыткой разрешения, т. е. преобразования строки имени DLL в имя файла, загрузчик пытается применить правила перенаправления имени DLL. Эти правила перенаправления используются для расширения или замены частей пространства имен DLL, которое обычно соответствует пространству имен файловой системы Win32, с целью расширения модели Windows-приложений. В порядке применения эти правила имеют следующий вид.

- **Перенаправление набора API-функций MinWin.** Механизм набора API-функций разработан с той целью, чтобы дать возможность команде Windows вносить изменения в двоичный файл, экспортирующий системную API-функцию, способом, который был бы прозрачен для приложений. Работа механизма основана на концепции контрактов. Этот механизм был кратко затронут в главе 2, а ниже приводится его более подробное объяснение.
- **.LOCAL-перенаправление.** Механизм .LOCAL-перенаправления позволяет приложениям перенаправлять все загрузки, связанные с указанным базовым

именем DLL, независимо от того, был ли указан полный путь, на локальную копию DLL в каталоге приложения. Это делается либо путем создания копии DLL с таким же базовым именем, за которым следует расширение .local (например, MyLibrary.dll.local), либо путем создания файловой папки с именем .local в каталоге приложения и помещения копии локальной DLL-библиотеки в эту папку (например, C:\Program Files\My App\.LOCAL\MyLibrary.dll). DLL-библиотеки, загрузка которых перенаправлена с помощью механизма .LOCAL, обрабатываются точно так же, как и те, для которых использовался механизм перенаправления SxS. (См. следующий пункт списка.) Загрузчик принимает .LOCAL-перенаправление загрузки DLL-библиотек, только когда исполняемый файл не имеет связанного с ним манифеста (встроенного либо внешнего). По умолчанию этот механизм отключен. Чтобы включить его глобально, добавьте параметр `DevOverrideEnable` с типом `DWORD` в базовый раздел IFEO (HKLM\Software\Microsoft\WindowsNT\CurrentVersion\Image File Execution Options) и присвойте ему значение 1.

- **Перенаправление Fusion (SxS).** Fusion (также SxS) — расширение модели Windows-приложений, которое позволяет компонентам более подробно выразить информацию о зависимостях исполняемых двоичных файлов (обычно она касается версий этих файлов) путем встраивания двоичных ресурсов, называемых *манифестами*. Механизм Fusion был впервые использован с таким расчетом, чтобы приложения могли загрузить правильную версию пакета общих элементов управления Windows (comctl32.dll) после того, как исполняемые двоичные файлы были разбиты на разные версии, которые могут быть параллельно установлены в системе; с тех пор другим двоичным файлам версии стали назначаться аналогичным образом. Начиная с Visual Studio 2005, приложения, созданные с помощью компоновщика Microsoft, стали применять Fusion для нахождения соответствующей версии C-библиотек времени выполнения, тогда как Visual Studio 2015 и выше используют перенаправление наборов API-функций для реализации идеи универсальной среды CRT.

  Инструментарий времени выполнения Fusion читает встроенную информацию о зависимостях из раздела ресурсов исполняемого двоичного файла, используя загрузчик ресурсов Windows, и упаковывает информацию о зависимостях в поисковые структуры, известные как *контексты активации* (activation contexts). Система при своей загрузке и при запуске процесса создает, соответственно, на уровне системы и процесса исходные контексты активации; кроме этого у каждого потока есть связанный с ним стек контекста активации, со структурой контекста активации на вершине того стека, который считается активным. Стек контекста активации, имеющийся у каждого потока, управляется как явным образом посредством API-функций `ActivateActCtx` и `DeactivateActCtx`, так и неявным образом системой в определенные моменты, например, когда вызывается основная процедура DLL, являющаяся исполняемым двоичным файлом со встроенной информацией о зависимостях. Когда в рамках перенаправления

осуществляется Fusion-поиск имени DLL, система ищет информацию о перенаправлении в контексте активации на вершине принадлежащего потоку стека контекста активации, а затем в контекстах активации процесса и системы; при наличии информации о перенаправлении для операции загрузки используется файл, идентифицируемый контекстом активации.

- **Перенаправление известных DLL.** Известные DLL-библиотеки — механизм, который отображает определенные базовые имена DLL на файлы в системном каталоге, препятствуя замене DLL альтернативной версией, находящейся в другом месте.

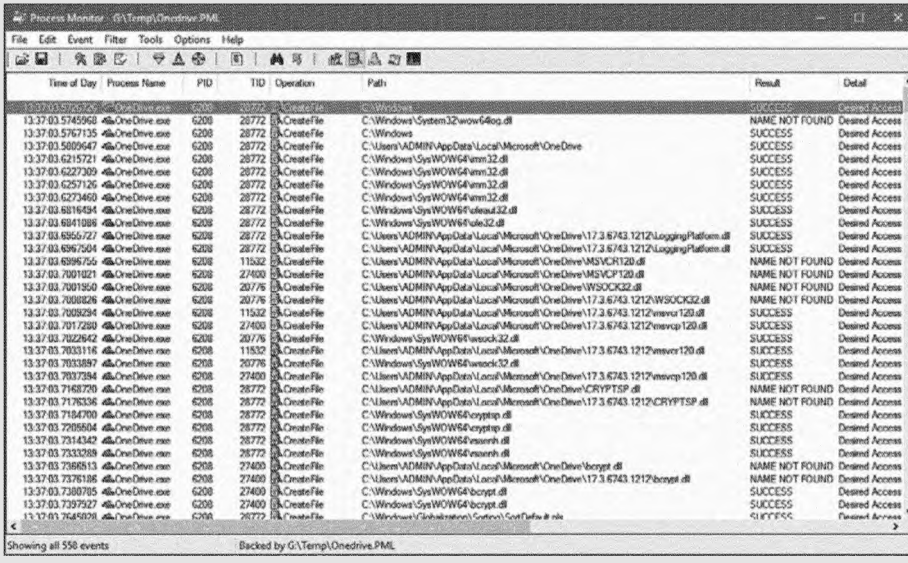
Одним из особых случаев алгоритма поиска пути DLL является проверка версии DLL, осуществляемая в 64-разрядных приложениях и в приложениях WoW64. Если DLL с соответствующим базовым именем обнаружена, но впоследствии определена как скомпилированная для неподходящей машинной архитектуры, например 64-разрядный образ в 32-разрядном приложении, загрузчик игнорирует ошибку и продолжает операцию поиска пути, начиная с элемента пути, который находится после элемента, использовавшегося для обнаружения неподходящего файла. Такой стиль поведения был разработан, чтобы дать возможность приложениям указывать в глобальной переменной среды окружения `%PATH%` записи как для 64-разрядных, так и для 32-разрядных вариантов.

ЭКСПЕРИМЕНТ: НАБЛЮДЕНИЕ ЗА ПОРЯДКОМ ПОИСКА ПРИ ЗАГРУЗКЕ DLL

Для наблюдения за тем, как загрузчик ищет DLL-библиотеки, можно воспользоваться программой Process Monitor из пакета Sysinternals. Когда загрузчик попытается разрешить DLL-зависимость, вы увидите, что он вызывает `CreateFile`, чтобы проверить каждое место поисковой последовательности до тех пор, пока не будет найдена указанная DLL, либо загрузка потерпит неудачу.

Ниже приведен снимок экрана для поиска загрузчиком исполняемого файла OneDrive.exe. Чтобы повторить эксперимент, выполните следующие действия.

1. Если программа OneDrive.exe выполняется, закройте ее со значка системной панели. Не забудьте закрыть все окна Проводника, в которых просматривается содержимое OneDrive.
2. Откройте Process Monitor и добавьте фильтры для отображения только процесса OneDrive.exe. Также можно отобразить только операции `CreateFile`.
3. Перейдите в каталог %LocalAppData%\Microsoft\OneDrive и запустите OneDrive.exe или OneDrive Personal.cmd (который запускает OneDrive для «персонального», а не для «коммерческого» использования). Ниже показан примерный результат (обратите внимание: в данном случае OneDrive — 32-разрядный процесс, который выполняется в 64-разрядной системе):



Некоторые вызовы, относящиеся к описанному выше порядку поиска:

- известные DLL загружаются из системного каталога (ole32.dll на снимке);
- LoggingPlatform.Dll загружается из подкаталога версии — возможно, потому что OneDrive вызывает SetDllDirectory для перенаправления поиска на новейшую версию (17.3.6743.1212 на снимке);
- MSVCR120.dll (исполнительная среда MSVC версии 12) ищется в каталоге исполняемого файла, где ее найти не удается. Затем поиск проводится в подкаталоге версий, и этот вариант поиска оказывается успешным;
- Wsock32.Dll (Winsock) ищется в каталоге исполняемого файла, затем в подкаталоге версии и наконец находится в системном каталоге (SysWow64). Обратите внимание: эта DLL-библиотека не относится к числу известных.

База данных загруженных модулей

Загрузчик ведет список всех модулей (DLL-библиотек, а также основных исполняемых файлов), которые были загружены процессом. Эта информация хранится в PEB — а именно, в подструктуре, имеющей идентификатор `Ldr`, которая называется `PEB_LDR_DATA`. В этой структуре загрузчик ведет три двусвязных списка, в которых содержится одна и та же, но по-разному выстроенная информация: по порядку загрузки, по размещению в памяти или по порядку инициализации. Эти списки содержат структуры, называемые *записями таблицы данных загрузчика* (`LDR_DATA_TABLE_ENTRY`), с которых хранится информация о каждом модуле.

Кроме того, поскольку поиск в связанных списках обходится алгоритмически дорого (за линейное время), загрузчик также поддерживает два красно-черных

дерева — эффективные структуры данных для бинарного поиска. Первое дерево сортируется по базовому адресу, а второе — по хешу имени модуля. С такими деревьями алгоритм поиска может работать за логарифмическое время, что гораздо более эффективно и намного ускоряет создание процессов в Windows 8 и далее. Кроме того, для обеспечения безопасности корни этих двух деревьев (в отличие от связанных списков) недоступны в PEB. Это усложняет их нахождение из кода оболочки, который работает в среде с включенной рандомизацией структуры адресного пространства (ASLR, Address Space Layout Randomization). (Подробнее об ASLR см. в главе 5.)

В табл. 3.9 перечислены различные фрагменты информации, сохраняемой загрузчиком в записи.

Таблица 3.9. Поля в записи таблицы данных загрузчика

Поле	Значение
`BaseAddressIndexNode`	Связывание записи с узлом красно-черного дерева, отсортированного по базовому адресу
`BaseDllName/BaseNameHashValue`	Имя самого модуля без полного пути к нему. Второе поле содержит его хеш, полученный `RtlHashUnicodeString`
`DdagNode/NodeModuleLink`	Указатель на структуру данных распределенного графа зависимостей (DDAG), используемую для параллелизации загрузок зависимостей через пул рабочих потоков. Второе поле связывает структуру с записями `LDR_DATA_TABLE_ENTRY`, ассоциированными с ней (часть того же графа)
`DllBase`	Хранит базовый адрес, по которому был загружен модуль
`EntryPoint`	Содержит начальную процедуру модуля (например, DllMain)
`EntryPointActivationContext`	При вызове инициализаторов содержит контекст активации SxS/Fusion
`Flags`	Флаги состояния загрузчика для этого модуля. (Описание флагов приводится в табл. 3.10)
`ForwarderLinks`	Связанный список модулей, которые были загружены из модуля в результате использования механизма продвижения данных экспортной таблицы
`FullDllName`	Полное имя модуля
`HashLinks`	Связанный список, используемый во время запуска и остановки процесса для более быстрого поиска
`ImplicitPathOptions`	Используется для хранения флагов поиска пути, которые задаются API-функцией `LdrSetImplicitPathOptions` или наследуются на основании пути DLL
`List Entry Links`	Связывает данную запись с каждым из трех упорядоченных списков, являющихся частью базы данных загрузчика
`LoadContext`	Указатель на текущую информацию загрузки DLL. Обычно содержит NULL, если данные активно не записываются

Поле	Значение
ObsoleteLoadCount	Счетчик ссылок для модуля (т. е. количество загрузок). Не содержит полезных данных, которые были перемещены в структуру DDAG
LoadReason	Содержит значение перечисляемого типа, которое объясняет, почему DLL-библиотека была загружена (динамически, статически, при перенаправлении, как зависимость отложенной загрузки и т. д.)
LoadTime	Хранит показания системного времени на момент загрузки этого модуля
MappingInfoIndexNode	Ссылка для узла красно-черного дерева, отсортированного по хешу имени
OriginalBase	Хранит исходный базовый адрес модуля (установленный компоновщиком) до ASLR и перемещения для ускорения обработки записей импорта с модифицируемыми адресами
ParentDllBase	В случае статических зависимостей (или перенаправления, или отложенной загрузки) хранит адрес DLL-библиотеки, которая зависит от этой
SigningLevel	Хранит уровень подписи образа (подробнее об инфраструктуре целостности кода см. в главе 8 части 2)
SizeOfImage	Размер модуля в памяти
SwitchBackContext	Используется технологией SwitchBack (см. далее) для хранения GUID текущего контекста Windows, связанного с этим модулем, и других данных
TimeDateStamp	Отметка времени, записанная компоновщиком при сборке модуля, которую загрузчик получает из PE-заголовка образа модуля
TlsIndex	Слот локального хранилища потока, связанного с данным модулем

Один из способов просмотра базы данных загрузчика процесса — средство WinDbg и его отформатированный блок PEB. В следующем эксперименте показано, как это сделать и как самому посмотреть на структуру `LDR_DATA_TABLE_ENTRY`.

ЭКСПЕРИМЕНТ: ВЫВОД ДАМПА БАЗЫ ДАННЫХ ЗАГРУЖЕННЫХ МОДУЛЕЙ

Перед началом этого эксперимента выполните те же действия, которые производились в двух предыдущих экспериментах по запуску программы Notepad.exe с использованием отладчика WinDbg. Когда дойдете до первого приглашения на ввод данных (там, где до этого момента вам предписывалось ввести команду g), выполните следующие инструкции.

1. PEB-блок текущего процесса можно просмотреть с помощью команды `!peb`. Но теперь нас интересуют только выводимые на экран данные `Ldr`.

```
0:000> !peb
PEB at 000000dd4c901000
    InheritedAddressSpace:    No
    ReadImageFileExecOptions: No
    BeingDebugged:            Yes
    ImageBaseAddress:         00007ff720b60000
    Ldr                       00007ffe855d23a0
    Ldr.Initialized:          Yes
    Ldr.InInitializationOrderModuleList: 0000022815d23d30 . 0000022815d24430
    Ldr.InLoadOrderModuleList:           0000022815d23ee0 . 0000022815d31240
    Ldr.InMemoryOrderModuleList:         0000022815d23ef0 . 0000022815d31250
                    Base TimeStamp                     Module
            7ff720b60000 5789986a Jul 16 05:14:02 2016 C:\Windows\System32\
notepad.exe
            7ffe85480000 5825887f Nov 11 10:59:43 2016 C:\WINDOWS\SYSTEM32\
ntdll.dll
            7ffe84bd0000 57899a29 Jul 16 05:21:29 2016 C:\WINDOWS\System32\
KERNEL32.DLL
            7ffe823c0000 582588e6 Nov 11 11:01:26 2016 C:\WINDOWS\System32\
KERNELBASE.dll
...
```

2. Адрес в строке `Ldr` — указатель на ранее рассмотренную структуру `PEB_LDR_DATA`. Обратите внимание: WinDbg показывает вам адрес трех списков и выводит дамп списка модулей в порядке их инициализации, где показан полный путь, отметка времени и базовый адрес каждого модуля.

3. Можно также проанализировать отдельно запись каждого модуля путем прохода по списку модулей с последующим выводом дампа данных по каждому адресу, отформатированных в виде структуры `LDR_DATA_TABLE_ENTRY`. Но вместо того чтобы делать это для каждой записи, WinDbg может проделать основную часть работы, использовав расширение `!list` и следующий синтаксис:

```
!list -x "dt ntdll!_LDR_DATA_TABLE_ENTRY" @@C++(&@$peb->Ldr-
>InLoadOrderModuleList)
```

Затем вы сможете просмотреть записи для каждого модуля:

```
+0x000 InLoadOrderLinks : _LIST_ENTRY [ 0x00000228'15d23d10 -
0x00007ffe'855d23b0 ]
   +0x010 InMemoryOrderLinks : _LIST_ENTRY [ 0x00000228'15d23d20 -
0x00007ffe'855d23c0 ]
   +0x020 InInitializationOrderLinks : _LIST_ENTRY [ 0x00000000'00000000 -
0x00000000'00000000 ]
   +0x030 DllBase          : 0x00007ff7'20b60000 Void
   +0x038 EntryPoint       : 0x00007ff7'20b787d0 Void
   +0x040 SizeOfImage      : 0x41000
   +0x048 FullDllName      : _UNICODE_STRING "C:\Windows\System32\notepad.
exe"
   +0x058 BaseDllName      : _UNICODE_STRING "notepad.exe"
   +0x068 FlagGroup        : [4]  "???"
   +0x068 Flags            : 0xa2cc
```

Хотя в этом разделе рассматривается загрузчик пользовательского режима из библиотеки Ntdll.dll, обратите внимание на то, что ядро также использует свой

собственный загрузчик для драйверов и зависимых DLL-библиотек, с похожей структурой данных `KLDR_DATA_TABLE_ENTRY`. Точно так же у загрузчика режима ядра есть своя собственная база данных каждой записи, которая напрямую доступна через глобальную переменную `PsLoadedModuleList`. Для вывода дампа базы данных модулей, загруженных в ядро, можно воспользоваться такой же командой `!list`, которая была показана в предыдущем эксперименте, заменив указатель в конце команды строкой `nt!PsLoadedModuleList` и указав новое имя структуры/модуля: `!list-x" dt nt!_KLDR_DATA_TABLE_ENTRY nt!PsLoadedModuleList`.

Просмотр списка в низкоуровневом формате позволяет еще глубже разобраться во внутренностях загрузчика, таких как поле `Flags`, флагов, содержащее информацию о состоянии, которую сама по себе команда `!peb` не показывает. Значения флагов показаны в табл. 3.10. Поскольку эту структуру используют оба загрузчика, и ядра и пользовательского режима, некоторые флаги применимы только к драйверам режима ядра, в то время как другие флаги применимы только к приложениям пользовательского режима (например, связанные с .NET-состоянием).

Таблица 3.10. Флаги записей таблицы данных загрузчика

Флаг	Значение
Пакетный двоичный модуль (0x1)	Модуль является частью пакетного приложения (устанавливается только в главном модуле пакета AppX)
Помечено для удаления (0x2)	Модуль будет выгружен сразу же после того, как будут закрыты все ссылки на него (например, из выполняемого рабочего потока)
DLL-образ (0x4)	Модуль представляет собой DLL образа (а не данных или исполняемого файла)
Уведомления загрузки отправлены (0x8)	Зарегистрированные получатели уже получили уведомления об этом образе
Данные телеметрии обработаны (0x10)	Данные телеметрии уже были обработаны для этого образа
Статический импорт процесса (0x20)	Модуль является статическим импортом главного двоичного модуля приложения
В унаследованных списках (0x40)	Запись образа содержится в двусвязных списках загрузчика
В индексах (0x80)	Запись образа содержится в красно-черных деревьях
DLL оболочки совместимости (0x100)	Запись образа представляет DLL-часть базы данных оболочки совместимости/приложения
В таблице исключений (0x200)	Обработчики исключений .pdata модулей были сохранены в инвертированной таблице функций загрузчика
Выполняется загрузка (0x800)	Модуль загружается в настоящее время
Конфигурация загрузки обработана (0x1000)	Каталог конфигурации загрузки образа был найден и обработан
Запись обработана (0x2000)	Загрузчик завершил обработку этого модуля

Таблица 3.10 (*окончание*)

Флаг	Значение
Защита отложенной загрузки (0x4000)	Механизм Control Flow Guard для этого двоичного модуля запросил защиту IAT отложенной загрузки. За дополнительной информацией обращайтесь к главе 7
Вызов присоединения процесса (0x20000)	Уведомление `DLL_PROCESS_ATTACH` было уже отправлено DLL
Сбой при попытке присоединения процесса (0x40000)	Функция DllMain инициировала сбой уведомления `DLL_PROCESS_ATTACH`
Не вызывать для потоков (0x80000)	Не отправлять DLL уведомления `DLL_THREAD_ATTACH`. Может устанавливаться вызовом `DisableThreadLibraryCalls`
Отложенная проверка COR (0x100000)	COR (Common Object Runtime) проверит образ .NET позднее
Образ COR (0x200000)	Модуль является приложением .NET
Не перемещать (0x400000)	Образ не должен перемещаться или подвергаться рандомизации
Только COR IL (0x800000)	Библиотека содержит только код промежуточного языка (IL) без кода языка ассемблера
База данных совместимости обработана (0x40000000)	Механизм оболочки совместимости обработал DLL

Анализ импорта

Итак, теперь вы знаете, как загрузчик отслеживает все модули, загружаемые для процесса, и мы можем продолжить анализ инициализационных задач, выполняемых загрузчиком при запуске. На этой стадии загрузчик выполняет следующие действия.

1. Загружает каждую DLL, на которую есть ссылка в таблице импорта исполняемого процессом образа.
2. Проверяет, не была ли DLL уже загружена, просматривая для этого базу данных модуля. Если библиотека не будет найдена в списке, загрузчик открывает DLL и отображает ее в памяти.
3. В ходе операции отображения загрузчик сначала просматривает различные пути, где он должен попытаться найти эту DLL, а также выясняет, не является ли эта библиотека «*известной*»; это будет означать, что система уже загрузила ее во время запуска и предоставила для доступа к ней глобальный файл, отображенный в памяти. Также возможны определенные отклонения от стандартного алгоритма поиска, связанные либо с использованием файла с расширением .local (что заставит загрузчик использовать DLL-библиотеки в локальном

пути), либо с файлом манифеста, который может указать на необходимость использования DLL, находящейся в другом месте, чтобы гарантировать использование конкретной версии.

4. После того как DLL найдена на диске и отображена, загрузчик проверяет, не загрузило ли ядро ее в какое-нибудь другое место — это называется *перемещением* (relocation). Если загрузчик обнаружил перемещение, он проводит анализ информации о перемещении в DLL и выполняет требуемые операции. Если информация о перемещении отсутствует, происходит сбой загрузки DLL.
5. Затем загрузчик создает запись таблицы данных загрузчика для этой DLL и вставляет ее в базу данных.
6. После того как DLL была отображена, процесс повторяется для этой DLL с целью анализа ее таблицы импорта и всех ее зависимостей.
7. После загрузки каждой DLL загрузчик проводит анализ IAT с целью поиска конкретных импортированных функций. Обычно это делается по именам, но также может делаться и по порядку (по порядковому номеру). Для каждого имени загрузчик проводит анализ экспортной таблицы импортированной DLL и пытается найти соответствие. Если соответствие найдено не будет, операция прерывается.
8. Таблица импорта, принадлежащая образу, может также быть связанной. Это означает, что в процессе компоновки разработчики уже назначили статические адреса, указывающие на импортируемые функции во внешних DLL-библиотеках. Это снимает необходимость в поиске каждого имени, но предполагает, что те DLL-библиотеки, которые будут использоваться приложением, будут всегда находиться по одним и тем же адресам. Поскольку в Windows используется рандомизация адресного пространства (за информацией об ASLR обращайтесь к главе 5), к системным приложениям и библиотекам это, как правило, не относится.
9. Экспортная таблица импортированной DLL может использовать запись продвижения данных (forwarder entry), означающую, что текущая функция реализована в другой DLL. По сути это должно рассматриваться как импорт или зависимость, поэтому после анализа экспортной таблицы загружается также и каждая DLL, на которую ссылалась запись продвижения данных, и загрузчик возвращается к пункту 1.

После того как будут загружены все импортируемые DLL-библиотеки (и их собственные зависимости или импортируемые данные), найдены все требуемые импортируемые функции и загружены и обработаны все записи продвижения данных, этап завершается: теперь все зависимости, которые были определены приложением и его различными DLL-библиотеками во время компиляции, удовлетворены. В ходе выполнения к загрузчику могут обращаться и, по сути, повторно выполнять те же самые задачи отложенной зависимости (называемые *отложенной загрузкой*), а также операции времени выполнения (например, вызовы функции `LoadLibrary`). Но следует заметить, что отказы на данных этапах, если они вы-

полняются во время запуска процесса, приведут к ошибке запуска приложения. Например, попытка запуска приложения, требующего функцию, отсутствующую в текущей версии операционной системы, может привести к выводу сообщения, подобного показанному на рис. 3.12.

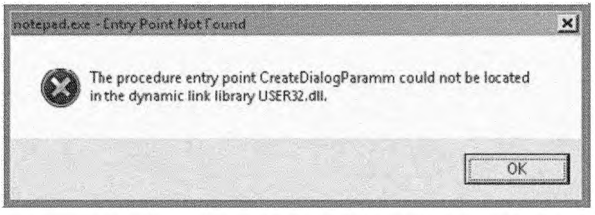


Рис. 3.12. Диалоговое окно, которое отображается при отсутствии необходимой импортированной функции в DLL

Инициализация процесса после импортирования

После загрузки зависимостей для полного завершения запуска приложения должен быть выполнен ряд инициализационных задач. На этой стадии загрузчик должен сделать следующее.

1. Все действия начинаются с присваивания `LdrInitState` значения 2; это означает, что весь импорт был загружен.
2. В этот момент при использовании отладчика (например, WinDbg) будет достигнута исходная контрольная точка отладчика. Именно здесь для продолжения выполнения в ранее рассмотренных экспериментах вам приходилось вводить команду `g`.
3. Проверяет, является ли процесс приложением подсистемы Windows; в этом случае функция `BaseThreadInitThunk` должна быть зафиксирована на ранних этапах инициализации процесса. На этой стадии она вызывается и проверяется на успех. Аналогичным образом вызывается функция `TermsrvGetWindowsDirectoryW`, которая должна быть зафиксирована ранее (в системах с поддержкой терминальных служб), что приводит к сбросу путей к системному каталогу и каталогу Windows.
4. По распределенному графу перебирает все зависимости и запускает инициализаторы для всех статических импортов образов. На этом шаге вызывается функция `DllMain` каждой DLL (чтобы каждая библиотека могла провести свою инициализацию, которая может включать даже загрузку новых DLL во время выполнения), а также обрабатываются инициализаторы TLS всех DLL. Это один из последних этапов, в которых при загрузке приложения может произойти сбой. Все загруженные DLL должны вернуть код успешного завершения после выполнения функций `DllMain`, иначе загрузчик отменяет запуск приложения.
5. Если образ использует слоты TLS, вызывает его инициализатор TLS.

6. Если модуль адаптируется для совместимости приложения, вызывает функцию обратного вызова механизма оболочек совместимости, сообщающую о завершении инициализации.
7. Запускает функцию, вызываемую после завершения инициализации процесса, соответствующей подсистемы, зарегистрированную в PEB. Например, для приложений Windows эта функция выполняет проверки, относящиеся к службам терминалов.
8. Записывает событие ETW, сообщающее об успешной загрузке процесса.
9. Если существует минимальный размер фиксации (commit) стека, то коснитесь стека потоков, чтобы заставить встроенную страницу зафиксировать страницы.
10. Переменной `LdrInitState` присваивается значение 3, которое сообщает о том, что инициализация завершена. Полю `ProcessInitializing` в PEB снова присваивается значение 0, после чего обновляется переменная `LdrpProcessInitialized`.

Технология SwitchBack

По мере внесения исправлений (например, связанных с «ситуациями гонки» или проверкой параметров в существующих API-функциях) в новые версии Windows каждое изменение, даже самое незначительное, создает риск возникновения несовместимости приложений. В Windows используется технология *SwitchBack*, реализованная в загрузчике, которая позволяет разработчикам программного обеспечения встроить в манифест, связанный с исполняемым файлом, GUID-идентификатор для версии Windows, на которую они ориентировались.

Например, если разработчик хочет воспользоваться усовершенствованиями, добавленными в Windows 10 в данные API-функции, он должен включить в свой манифест GUID-идентификатор Windows 10, а если у разработчика имеется устаревшее приложение, сориентированное на поведение, характерное для Windows 7, он должен поместить в манифест GUID-идентификатор Windows 7.

SwitchBack анализирует эту информацию и соотносит ее со встроенной информацией в SwitchBack-совместимых DLL-библиотеках (в разделе образа .sb_data), чтобы решить, какая из версий затронутых API-функций должна вызываться модулем. Поскольку технология SwitchBack работает на уровне загруженных модулей, она позволяет процессу иметь параллельные вызовы одних и тех же API-функций как из устаревших, так и из текущих DLL-библиотек, следуя при этом различным результатам.

Значения GUID для SwitchBack

В настоящее время в Windows определены следующие GUID-идентификаторы, представляющие установки совместимости для всех версий Windows, начиная с Windows Vista:

- {e2011457-1546-43c5-a5fe-008deee3d3f0} для Windows Vista;
- {35138b9a-5d96-4fbd-8e2d-a2440225f93a} для Windows 7;
- {4a2f28e3-53b9-4441-ba9c-d69d4a4a6e38} для Windows 8;
- {1f676c76-80e1-4239-95bb-83d0f6d0da78} для Windows 8.1;
- {8e0f7a12-bfb3-4fe8-b9a5-48fd50a15a9a} для Windows 10.

Эти GUID-идентификаторы должны присутствовать в файле манифеста приложения — в атрибуте `ID` элемента `<SupportedOS>`, присутствующего в записи атрибута совместимости. (Если в манифесте приложения не содержится GUID, в качестве режима совместимости по умолчанию выбирается Windows Vista.) В диспетчере задач можно включить столбец Контекст ОС (Operating System Context) на вкладке Подробности (Details); он покажет, работают ли какие-либо приложения в конкретном контексте ОС (пустое значение в столбце обычно означает, что они работают в режиме Windows 10).

На рис. 3.13 показан пример таких приложений, работающих в режиме Windows Vista и Windows 7 в системе с Windows 10.

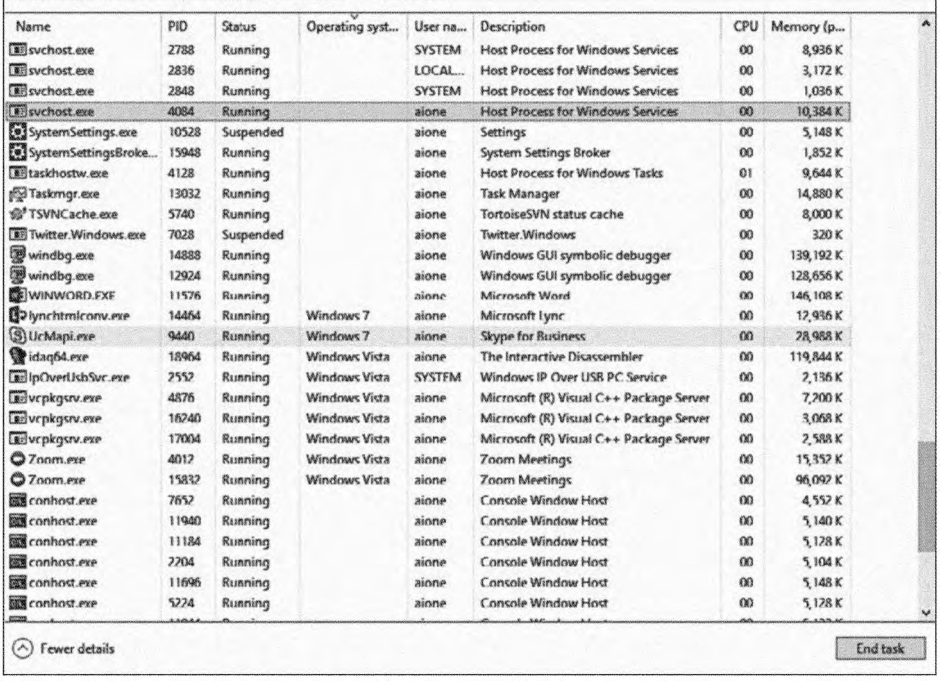


Рис. 3.13. Процессы, работающие в режиме совместимости

Пример записи манифеста, устанавливающей совместимость с Windows 10:

```
<compatibility xmlns="urn:schemas-microsoft-com:compatibility.v1">
  <application>
    <!-- Windows 10 -->
    <supportedOS Id="{8e0f7a12-bfb3-4fe8-b9a5-48fd50a15a9a}" />
  </application>
</compatibility>
```

Режимы совместимости SwitchBack

Рассмотрим несколько примеров того, что делает SwitchBack. При выполнении в контексте Windows 7:

- RPC-компоненты вместо закрытой реализации используют пул потоков Windows;
- блокировка DirectDraw Lock не может быть получена на первичном буфере;
- копирование битового массива (blitting) на рабочем столе не допускается без отсечения (clipping) окна;
- исправляется ситуация гонки в `GetOverlappedResult`;
- вызовам `CreateFile` может передаваться «понижающий» флаг для получения монопольного открытия даже в том случае, если вызывающая сторона не имеет привилегий записи (при этом `NtCreateFile` не передается флаг `FILE_DISALLOW_EXCLUSIVE`).

С другой стороны, запуск в режиме Windows 10 оказывает нетривиальное влияние на работу LFH (Low Fragmentation Heap): подсегменты LFH полностью фиксируются, а все операции выделения памяти дополняются блоком заголовка при отсутствии GUID Windows 10. Кроме того, в Windows 10 при использовании функции устранения рисков «инициировать исключение при закрытии недействительного дескриптора» (см. главу 7) это поведение будет поддерживаться функциями `CloseHandle` и `RegCloseKey`. В более ранних операционных системах, если к процессу не присоединен отладчик, это поведение будет блокироваться перед вызовом `NtClose` и снова включаться после вызова.

Другой пример: механизм проверки правописания будет возвращать `NULL` для языков, для которых такая проверка не поддерживается, тогда как в Windows 8.1 возвращается «пустое» средство проверки. Аналогичным образом реализация функции `IShellLink::Resolve` при передаче относительного пути в режиме совместимости Windows 8 будет возвращать `E_INVALIDARG`, а в режиме Windows 7 эта проверка отсутствует.

Кроме того, вызовы `GetVersionEx` или эквивалентных функций из NtDll (таких, как `RtlVerifyVersionInfo`) будут возвращать максимальный номер версии, соответствующий заданному GUID контекста `SwitchBack`.

ПРИМЕЧАНИЕ Эти API-функции считаются устаревшими. Вызов `GetVersionEx` будет возвращать 6.2 во всех версиях Windows 8 и выше, если не был задан более высокий `SwitchBack` GUID.

Поведение SwitchBack

Когда Windows API подвергается изменениям, которые могут нарушить совместимость, код входа в функции вызывает функцию `SbSwitchProcedure`, чтобы запустить SwitchBack-логику. По указателю управление передается в *таблицу модулей* SwitchBack, в которой содержится информация о механизмах SwitchBack, задействованных в модуле. В таблице также содержится указатель на массив записей для каждой точки SwitchBack. В этой таблице содержится дескриптор каждой точки ветвления, который идентифицирует ее с символическим именем и с полным описанием, а также со связанной меткой устранения рисков совместимости. Обычно в модуле имеется несколько точек ветвления: одна для поведения в Windows Vista, одна для поведения в Windows 7 и т. д.

Для каждой точки ветвления дается требуемый контекст SwitchBack — тот самый контекст, который определяет, какая из двух (или более) ветвей будет выбрана во время выполнения приложения. Наконец, в каждом из этих дескрипторов содержится указатель на функцию с реальным кодом, который должен выполняться при каждом ветвлении. Если приложение запускается с Windows 10 GUID, этот код будет частью его SwitchBack-контекста, и API-функция `SbSelectProcedure` после анализа таблицы модулей выполнит соответствующую операцию. Она находит дескриптор записи модуля для контекста и вызывает функцию по указателю в дескрипторе.

SwitchBack использует ETW для отслеживания выбора SwitchBack-контекстов и точек ветвления, и снабжает данными регистратор Windows AIT (Application Impact Telemetry — телеметрия влияния на приложение). Эти данные периодически собираются компанией Microsoft для определения степени использования каждой записи совместимости, идентификации того приложения, которое ее использует (в журнале предоставляется полная трассировка стека), и для уведомления сторонних поставщиков.

Как уже упоминалось, уровень совместимости приложения хранится в его манифесте. Во время загрузки загрузчик анализирует файл манифеста, создает структуру контекстных данных и кэширует ее в поле `pContextData` блока PEB. В этих данных контекста содержатся соответствующие GUID совместимости, под которыми выполняется данный процесс, и определяется, какая версия точек ветвления у вызванных API-функций, которые будут выполняться используемой технологией SwitchBack.

Наборы API-функций

Несмотря на то что SwitchBack для определенных сценариев совместимости приложений использует API-перенаправление, для всех приложений в Windows используется гораздо более распространенный механизм перенаправления, который

называется *наборами API-функций*. Он предназначен для более точного деления на категории API-функций Windows на DLL-библиотеки вместо создания больших многоцелевых DLL-библиотек, охватывающих тысячи API-функций, которые могут не понадобиться всем типам ныне существующих и будущих Windows-систем. Эта технология, разработанная в основном для поддержки перестройки самых нижних уровней архитектуры Windows для отделения ее от более высоких уровней, идет рука об руку с разбиением Kernel32.dll и Advapi32.dll (наряду со всем остальным) на несколько виртуальных DLL-файлов.

Например, из рис. 3.14 видно, что Kernel32.dll, основная библиотека Windows, ведет импорт из множества других DLL-библиотек, начиная с API-MS-WIN. В каждой из этих DLL-библиотек содержится небольшой поднабор API-функций, которые обычно предоставляются библиотекой Kernel32, но все вместе они составляют все пространство API, предоставляемое библиотекой Kernel32.dll. Например, библиотека `CORE-STRING` содержит только основные строковые функции Windows.

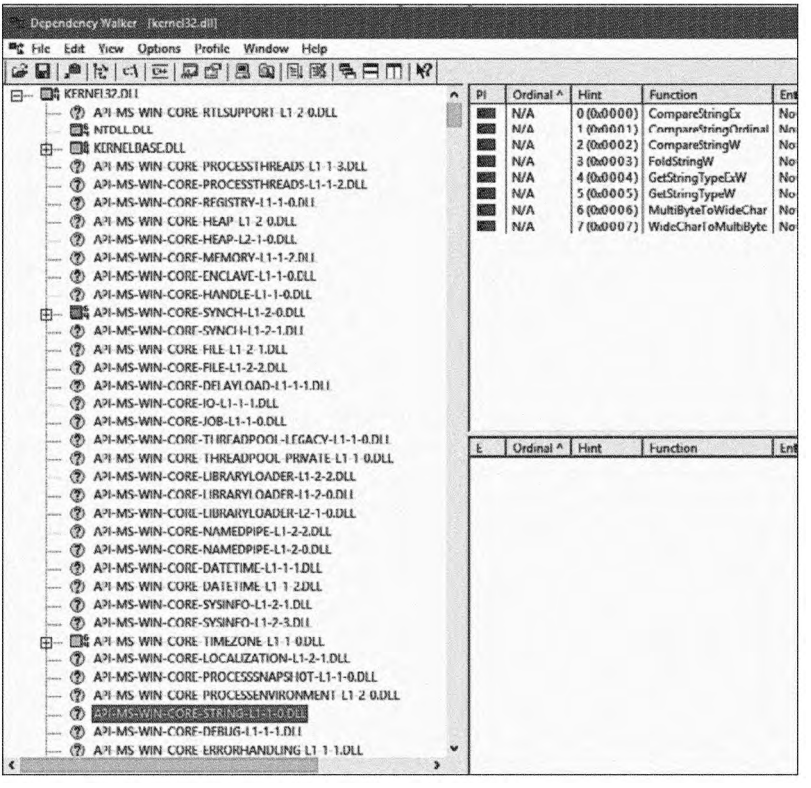


Рис. 3.14. Наборы API-функций для kernel32.dll

Разбиение функций на отдельные файлы достигает двух целей: во-первых, это позволяет будущим приложениям компоноваться только с теми API-библиотеками,

которые предоставляют нужную им функциональность. Во-вторых, если Microsoft создаст версию Windows, не поддерживающую, к примеру, локализацию (скажем, для встроенной системы, предназначенной только для англоязычных стран), она сможет просто удалить соответствующие подчиненные DLL-библиотеки и изменить схему набора API-функций. В результате будет получен более компактный двоичный код Kernel32, а любое приложение, не требующее локализации, будет работать по-прежнему.

В рамках этой технологии была определена (реализована на уровне исходного кода) «базовая» Windows-система под названием «*MinWin*» с минимальным набором сервисных функций, куда было включено ядро и основные драйверы (включая файловую систему, основные системные процессы, такие как CSRSS и диспетчер управления службами (Service Control Manager) и еще небольшой перечень Windows-служб). Семейство Windows Embedded имеет специальный инструмент для построения ядра операционной системы, который называется Platform Builder, который на первый взгляд делает примерно то же: построители системы могут удалять отдельные компоненты Windows (оболочка, сетевой стек и т. д.). Но удаление компонентов из Windows оставляет *зависшие зависимости* — пути выполнения кода, при попытке выполнения которых произойдет сбой, поскольку они зависят от исключенных компонентов. С другой стороны, зависимости MinWin являются полностью самодостаточными.

При инициализации диспетчера процессов вызывается функция `PspInitializeApiSetMap`, отвечающая за создание объекта раздела (используя стандартный объект раздела) для таблицы перенаправления API-набора, которая хранится в %SystemRoot%\System32\ApiSetSchema.dll. Эта библиотека не содержит исполняемого кода, но в ней есть раздел .apiset с данными отображения виртуальных DLL-библиотек API-набора на логические DLL-библиотеки, содержащие реализацию API-функций. При запуске нового процесса диспетчер процессов отображает объект раздела на адресное пространство процесса и записывает в поле `ApiSetMap` в PEB-блоке процесса базовый адрес, куда был отображен объект раздела.

В свою очередь, функция загрузчика `LdrpApplyFileNameRedirection`, которая обычно отвечает за перенаправление рассмотренного ранее манифеста SxS/Fusion и .local, также проверяет перенаправление данных API-набора, когда загружается новая импортируемая библиотека, имя которой начинается с «API-» (как в динамическом, так и в статическом режиме). Таблица API-набора выстраивается библиотекой таким образом, что в каждой записи есть описание, в какой логической DLL-библиотеке может быть найдена та или иная функция; эта DLL-библиотека и будет загружена. Хотя у данных схемы двоичный формат, вы можете вывести дамп строк этих данных с помощью программы Strings из пакета Sysinternals, чтобы посмотреть, какие DLL-библиотеки определены на данный момент:

```
C:\Windows\System32>strings apisetschema.dll
...
api-ms-onecoreuap-print-render-l1-1-0
printrenderapihost.dllapi-ms-onecoreuap-settingsync-status-l1-1-0
```

```
settingsynccore.dll
api-ms-win-appmodel-identity-l1-2-0
kernel.appcore.dllapi-ms-win-appmodel-runtime-internal-l1-1-3
api-ms-win-appmodel-runtime-l1-1-2
api-ms-win-appmodel-state-l1-1-2
api-ms-win-appmodel-state-l1-2-0
api-ms-win-appmodel-unlock-l1-1-0
api-ms-win-base-bootconfig-l1-1-0
advapi32.dllapi-ms-win-base-util-l1-1-0
api-ms-win-composition-redirection-l1-1-0
...
api-ms-win-core-com-midlproxystub-l1-1-0
api-ms-win-core-com-private-l1-1-1
api-ms-win-core-comm-l1-1-0
api-ms-win-core-console-ansi-l2-1-0
api-ms-win-core-console-l1-1-0
api-ms-win-core-console-l2-1-0
api-ms-win-core-crt-l1-1-0
api-ms-win-core-crt-l2-1-0
api-ms-win-core-datetime-l1-1-2
api-ms-win-core-debug-l1-1-2
api-ms-win-core-debug-minidump-l1-1-0
...
api-ms-win-core-firmware-l1-1-0
api-ms-win-core-guard-l1-1-0
api-ms-win-core-handle-l1-1-0
api-ms-win-core-heap-l1-1-0
api-ms-win-core-heap-l1-2-0
api-ms-win-core-heap-l2-1-0
api-ms-win-core-heap-obsolete-l1-1-0
api-ms-win-core-interlocked-l1-1-1
api-ms-win-core-interlocked-l1-2-0
api-ms-win-core-io-l1-1-1
api-ms-win-core-job-l1-1-0
...
```

Задания

Задание (job) представляет собой именованный объект ядра, у которого может быть механизм защиты и механизм совместного использования и который позволяет контролировать один или несколько процессов, сведенных в группу. Основной функцией объекта задания является предоставление возможности управления группой процессов как единым целым и работы с этим объединением. Процесс может входить только в один объект задания. По умолчанию связь процесса с объектом задания не может быть разорвана, и все созданные им процессы и их потомки также связаны с тем же объектом задания — если только дочерние процессы не были созданы с флагом `CREATE_BREAKAWAY_FROM_JOB`, а само задание не ограничило такую возможность. Объект задания также записывает основную учетную информацию для всех процессов, связанных с заданием, и для всех процессов, которые были связаны с заданием, но работа которых уже была завершена.

Задания также могут быть связаны с объектом порта завершения ввода/вывода, в ожидании которого могут находиться другие потоки, с использованием Windows-функции `GetQueuedCompletionStatus` или API пула потоков (функция `TpAllocJobNotification`). Это позволяет заинтересованным сторонам (обычно создателю задания) следить за нарушением лимита и за событиями, которые могут влиять на безопасность задания (например, это может быть создание нового процесса или аварийное завершение процесса).

Задания играют важную роль во многих системных механизмах.

- Они управляют современными приложениями (процессами UWP). За подробностями обращайтесь к главе 9 части 2. Более того, каждое современное приложение работает в составе задания. Вы можете убедиться в этом при помощи программы Process Explorer, как описано в эксперименте «Просмотр объекта задания» далее в этой главе.
- Они используются для реализации поддержки контейнеров Windows через механизм *серверных участков*, рассмотренный позднее в этом разделе.
- Они лежат в основе того, как DAM (Desktop Activity Moderator) управляет регулированием, виртуализацией таймеров, замораживанием таймеров и другим поведением, приводящим к бездействию, в приложениях и службах Win32. DAM описывается в главе 8 части 2.
- Они делают возможным определение и управление группами планирования для планирования DFSS (Dynamic Fair-Share Scheduling), описанного в главе 4.
- Они лежат в основе спецификации нестандартного разбиения памяти, необходимого для использования API секционирования памяти, описанного в главе 5.
- Они становятся одним из ключевых факторов для реализации таких функций, как запуск от имени (вторичный вход), изоляция приложений и помощник совместимости программ.
- Они предоставляют часть песочницы безопасности для таких приложений, как Google Chrome и Microsoft Office Document Converter, а также устранения рисков от атак отказа в обслуживании (DoS) через запросы WMI (Windows Management Instrumentation).

Ограничения заданий

На задание могут накладываться следующие ограничения, связанные с центральным процессором и с памятью.

- **Максимальное количество активных процессов.** Ограничивает количество параллельно выполняемых процессов в задании.
- **Ограничение времени работы центрального процессора в пользовательском режиме на уровне задания.** Ограничения максимального количества времени

работы центрального процессора в пользовательском режиме, которое может быть потреблено процессами в задании (включая те процессы, которые уже выполнены и из них осуществлен выход). По достижении этого лимита все процессы в задании завершаются с кодом ошибки, и никакие новые процессы в задании не могут быть созданы (если только лимит не будет перезапущен). Об объекте задания дается сигнал, поэтому любые потоки, ожидающие задание, будут освобождены. Вы можете изменить поведение по умолчанию вызовом функции `SetInformationJobObject` для установки поля `EndOfJobTimeAction` структуры `JOBOBJECT_END_OF_JOB_TIME_INFORMATION`, переданной с информационным классом `JobObjectEndOfJobTimeInformation` в классе запроса, вместо отправки уведомления через порт завершения задания.

- **Ограничение времени работы центрального процессора в пользовательском режиме на уровне процесса.** Позволяет каждому процессу в задании аккумулировать только фиксированное количество времени работы центрального процессора в пользовательском режиме. По достижении максимума процесс завершает свою работу (не имея возможности выполнить завершающие действия).
- **Сходство процессоров для задания.** Устанавливает маску сходства процессоров для каждого процесса в задании. (Отдельные потоки могут изменять свое сходство на любой поднабор сходства задания, но процессы изменять свои установки сходства процесса не могут.)
- **Групповое сходство для задания.** Устанавливает список групп, которым могут быть назначены процессы в задании. Затем любые изменения сходства сводятся к выбору группы. Это считается устаревшей групповой версией ограничения сходства процессоров для задания, которое не рекомендовано к использованию.
- **Класс приоритета для процессов задания.** Устанавливает класс приоритета для каждого процесса в задании. Потоки не могут повысить свой приоритет относительно класса (обычно они могли это делать). Попытки повышения приоритета потока игнорируются. (Ошибка при вызове функции `SetThreadPriority` не возвращается, но повышение не происходит.)
- **Минимум и максимум рабочего набора по умолчанию.** Определяет указанный минимум и максимум рабочего набора для каждого процесса в задании. (Эта настройка не распространяется сразу на все задание, у каждого процесса есть свой собственный рабочий набор с одинаковыми минимальными и максимальными значениями.)
- **Лимит зафиксированной виртуальной памяти для процесса и задания.** Определяет максимальный размер виртуального адресного пространства, которое может быть зафиксировано либо отдельным процессом, либо всем заданием.
- **Управление процессорным временем.** Определяет максимальную величину процессорного времени, которое разрешено использовать заданию перед тем, как оно столкнется с фиксированным регулированием. Используется как часть поддержки группового планирования, описанного в главе 4.

- **Управление скоростью передачи данных для сетевого канала.** Определяет максимальную скорость передачи данных, после которой в действие вступает регулирование. Также позволяет назначить метку DSCP (Differentiated Services Code Point) в контексте QoS для каждого сетевого пакета, отправляемого заданием. Может задаваться только для одного задания в иерархии, влияет на задание и все дочерние задания.
- **Управление скоростью передачи данных для дискового ввода/вывода.** То же, что и управление скоростью передачи данных для сетевого канала, но относится к дисковому вводу/выводу и может управлять либо самой скоростью передачи данных, либо количеством операций ввода/вывода в секунду (IOPS). Может задаваться как для конкретного тома, так и для всех томов в системе.

Для многих ограничений владелец задания может установить пороговые значения, при которых будут отправляться уведомления (или, если уведомление не зарегистрировано, задание будет просто уничтожаться). Также средства управления скоростью передачи позволяют назначать диапазоны допустимых отклонений — например, разрешить процессу превышать свои 20 % пропускной способности канала каждые 5 минут не более чем на 10 секунд. Эти уведомления осуществляются записью соответствующего сообщения в порт завершения ввода/вывода для задания. (За подробностями обращайтесь к документации Windows SDK.)

Наконец, для процессов в задании можно установить ограничения, касающиеся пользовательского интерфейса. Такие ограничения включают запрет на открытие процессами дескрипторов окон, владельцами которых являются потоки за пределами задания, запрет чтения и/или записи в буфер обмена и на изменения многих параметров системы пользовательского интерфейса через Windows-функцию `SystemParametersInfo`. Этими ограничениями, касающимися пользовательского интерфейса, управляет драйвер GDI/USER Win32k.sys, относящийся к подсистеме Windows, и они приводятся в исполнение через одну из специальных выносок (callouts), зарегистрированных с помощью диспетчера процессов. Вы можете предоставить доступ всем процессам задания к конкретным пользовательским дескрипторам (например, дескрипторам окна) вызовом функции `UserHandleGrantAccess`; эта функция может вызываться только процессом, который не является частью соответствующего задания, что вполне естественно.

Работа с заданиями

Объект задания создается API-функцией `CreateJobObject`. Только что созданное задание не содержит никаких процессов. Чтобы добавить процесс в задание, вызовите функцию `AssignProcessToJobObject`, которая может вызываться многократно для добавления процессов в задание и даже для включения одного процесса в разные задания. В последнем варианте создается вложенное задание (см. следующий раздел). Другой способ добавления процесса в задание основан на ручном указании дескриптора объекта задания при помощи атрибута `PS_CP_JOB_LIST`, упоминавшегося ранее в этой главе. Можно указать один или несколько объектов заданий.

Самая интересная API-функция для заданий, `SetInformationJobObject`, позволяет задавать различные ограничения и параметры, упомянутые в предыдущем разделе, а также содержит внутренние информационные классы, используемые такими механизмами, как контейнеры, DAM или приложения Windows UWP. Эти значения могут читаться функцией `QueryInformationJobObject`, при помощи которой заинтересованные стороны получают информацию об ограничениях, установленных для задания. Также ее необходимо вызывать при установке уведомлений о превышении ограничений (см. предыдущий раздел), чтобы вызывающая сторона точно знала, какие ограничения были нарушены. Также нередко используется функция `TerminateJobObject`, которая завершает все процессы в задании (так, как если бы для каждого процесса была вызвана функция `TerminateProcess`).

Вложенные задания

До выхода Windows 7 и Windows Server 2008 R2 процесс мог быть связан только с одним заданием, отчего задания приносили меньше пользы, чем могли бы, так как в некоторых случаях приложение не может заранее знать, будет ли процесс, которым ему предстоит управлять, принадлежать заданию или нет. Начиная с Windows 8 и Windows Server 2012, процесс может быть связан с несколькими заданиями; фактически это приводит к созданию иерархии заданий.

Дочернее задание содержит подмножество процессов своего родительского задания. После того как процесс будет добавлен более чем в одно задание, система пытается сформировать иерархию, если это возможно. В настоящее время действует ограничение, согласно которому задания не могут формировать иерархию, если какое-либо из них устанавливает любые UI-ограничения (`SetInformationJobObject` с аргументом `JobObjectBasicUIRestrictions`).

Ограничения для дочернего задания не могут быть менее жесткими, чем у родителя, но могут быть более жесткими. Например, если родительское задание устанавливает ограничение по памяти 100 Мбайт, никакое дочернее задание не сможет установить более высокое ограничение (такие попытки будут завершаться неудачей). Однако дочернее задание может установить более жесткое ограничение для своих процессов (и любых дочерних заданий) — например, 80 Мбайт. Любые уведомления, предназначенные для порта завершения ввода/вывода задания, будут отправляться заданию и всем его предкам. (Для отправки уведомлений предкам само задание не обязано иметь порт завершения ввода/вывода.)

Учет ресурсов родительского задания включает агрегированные ресурсы, используемые его непосредственно управляемыми процессами и всеми процессами дочерних заданий. При завершении задания (`TerminateJobObject`) все процессы в задании и его дочерних заданиях завершаются, начиная с дочерних заданий на нижнем уровне иерархии.

На рис. 3.15 показаны четыре процесса, управляемые иерархией заданий.

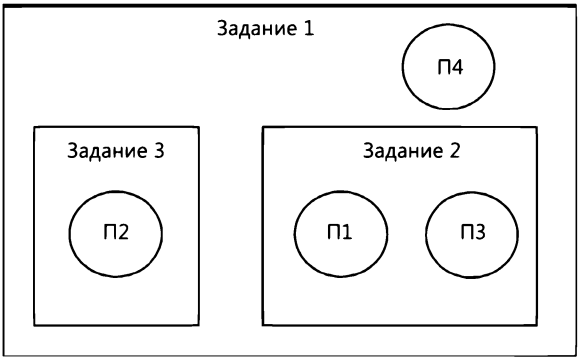


Рис. 3.15. Иерархия заданий

Чтобы создать такую иерархию, добавьте процессы в задания, начиная с корневого задания. Последовательность действий для создания этой иерархии выглядит так:

1. Добавить процесс П1 в задание 1.
2. Добавить процесс П1 в задание 2. При этом создается первое вложение.
3. Добавить процесс П2 в задание 1.
4. Добавить процесс П2 в задание 3. При этом создается второе вложение.
5. Добавить процесс П3 в задание 2.
6. Добавить процесс П4 в задание 1.

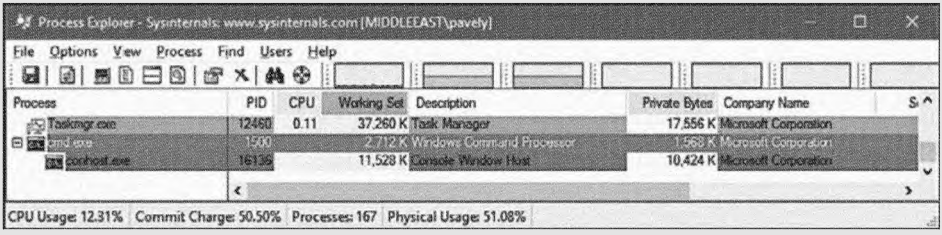
ЭКСПЕРИМЕНТ: ПРОСМОТР ОБЪЕКТА ЗАДАНИЯ

Для просмотра именованных объектов заданий можно воспользоваться Системным монитором (обратитесь к категориям Job Object и Job Object Details). Безымянные задания также можно просмотреть с помощью команд отладчика ядра `!job` или `dt nt!_ejob`.

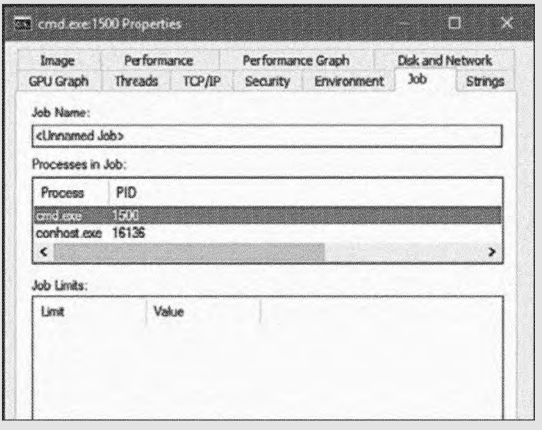
Чтобы увидеть, не связан ли процесс с заданием, можно воспользоваться командой отладчика ядра `!process` или средством Process Explorer. Для создания и просмотра безымянного объекта задания выполните следующие действия.

1. Из окна командной строки воспользуйтесь командой `runas` для создания процесса, запускающего окно командной строки (Cmd.exe). Например, наберите `runas /user:<домен>\<имя_пользователя> cmd`.
2. Команда запросит ваш пароль. Введите свой пароль, и появится окно командной строки. Служба Windows, выполняющая команду `runas`, создает безымянное задание для содержания всех процессов (чтобы она могла завершить все эти процессы во время выхода из системы).
3. Запустите Process Explorer, откройте меню Options, выберите команду Configure Colors и категорию Jobs. Обратите внимание: процесс Cmd.exe

и его дочерний процесс ConHost.exe выделены как часть задания, как показано ниже:



4. Дважды щелкните либо на процессе Cmd.exe, либо на процессе ConHost.exe, чтобы открыть диалоговое окно свойств. Щелкните на вкладке Job, чтобы просмотреть информацию о задании, частью которого является данный процесс:



5. Запустите из окна командной строки программу Блокнот: Notepad.exe.

6. Откройте процесс Блокнота и просмотрите его вкладку Job. Блокнот выполняется в составе того же задания. Это объясняется тем, что cmd.exe не использует флаг создания CREATE_BREAKAWAY_FROM_JOB. В случае вложенных заданий на вкладке Job перечислены процессы задания, которому непосредственно принадлежит процесс, и все процессы в дочерних заданиях.

7. Запустите отладчик ядра в работающей системе и введите команду !process, чтобы найти notepad.exe и вывести основную информацию об этом процессе:

```
lkd> !process 0 1 notepad.exe
PROCESS ffffe001eacf2080
    SessionId: 1  Cid: 3078    Peb: 7f4113b000  ParentCid: 05dc
    DirBase: 4878b3000  ObjectTable: ffffc0015b89fd80  HandleCount: 188.
    Image: notepad.exe
    ...
    BasePriority                      8
    CommitCharge                      671
    Job                               ffffe00189aec460
```

8. Обратите внимание на указатель Job, отличный от нуля. Чтобы получить сводку задания, введите команду отладчика !job:

```
lkd> !job ffffe00189aec460
Job at ffffe00189aec460
  Basic Accounting Information
    TotalUserTime:             0x0
    TotalKernelTime:           0x0
    TotalCycleTime:            0x0
    ThisPeriodTotalUserTime:   0x0
    ThisPeriodTotalKernelTime: 0x0
    TotalPageFaultCount:       0x0
    TotalProcesses:            0x3
    ActiveProcesses:           0x3
    FreezeCount:               0
    BackgroundCount:           0
    TotalTerminatedProcesses:  0x0
    PeakJobMemoryUsed:         0x10db
    PeakProcessMemoryUsed:     0xa56
  Job Flags
  Limit Information (LimitFlags: 0x0)
  Limit Information (EffectiveLimitFlags: 0x0)
```

9. Обратите внимание: поле ActiveProcesses содержит значение 3 (cmd.exe, conhost.exe и notepad.exe). После команды !job можно передать флаг 2, чтобы просмотреть список процессов, входящих в задание:

```
lkd> !job ffffe00189aec460 2
...
Processes assigned to this job:
    PROCESS ffff8188d84dd780
        SessionId: 1  Cid: 5720    Peb: 43bedb6000  ParentCid: 13cc
        DirBase: 707466000  ObjectTable: ffffbe0dc4e3a040  HandleCount:
<Data Not Accessible>
        Image: cmd.exe

    PROCESS ffff8188ea077540
        SessionId: 1  Cid: 30ec    Peb: dd7f17c000  ParentCid: 5720
        DirBase: 75a183000  ObjectTable: ffffbe0dafb79040  HandleCount:
<Data Not Accessible>
        Image: conhost.exe

    PROCESS ffffe001eacf2080
        SessionId: 1  Cid: 3078    Peb: 7f4113b000  ParentCid: 05dc
    DirBase: 4878b3000  ObjectTable: ffffc0015b89fd80  HandleCount: 188.
    Image: notepad.exe
```

10. Чтобы вывести объект задания, можно также воспользоваться командой dt и просмотреть дополнительные поля, относящиеся к заданию, — например, уровень вхождения этого задания и его отношения с другими заданиями в случае вложения (родительское задание, одноуровневые задания, корневое задание):

```
lkd> dt nt!_ejob ffffe00189aec460
   +0x000 Event            : _KEVENT
```

```
   +0x018 JobLinks         : _LIST_ENTRY [ 0xffffe001'8d93e548 -
0xffffe001'df30f8d8 ]
   +0x028 ProcessListHead  : _LIST_ENTRY [ 0xffffe001'8c4924f0 -
0xffffe001'eacf24f0 ]
   +0x038 JobLock          : _ERESOURCE
   +0x0a0 TotalUserTime    : _LARGE_INTEGER 0x0
   +0x0a8 TotalKernelTime  : _LARGE_INTEGER 0x2625a
   +0x0b0 TotalCycleTime   : _LARGE_INTEGER 0xc9e03d
   ...
   +0x0d4 TotalProcesses   : 4
   +0x0d8 ActiveProcesses  : 3
   +0x0dc TotalTerminatedProcesses : 0
...
   +0x428 ParentJob        : (null)
   +0x430 RootJob          : 0xffffe001'89aec460 _EJOB
...
   +0x518 EnergyValues     : 0xffffe001'89aec988 _PROCESS_ENERGY_VALUES
   +0x520 SharedCommitCharge : 0x5e8
```

Контейнеры Windows (серверные участки)

Развитие дешевых облачных вычислений и их широкое распространение привело к следующей интернет-революции, благодаря которой сегодня построение сетевых сервисов и/или служебных серверов для мобильных приложений сводится к щелчку на кнопке в одном из многочисленных поставщиков облачных сервисов. Но с ростом конкуренции между поставщиками облачных сервисов и возможной необходимости перехода с одного сервиса на другой (или даже с облачного сервиса на центр обработки данных, или с центра обработки данных на высокопроизводительный персональный сервер), становится все важнее иметь портируемую служебную подсистему, которая могла бы развертываться и перемещаться по мере необходимости без затрат, связанных с запуском их на виртуальной машине.

Именно для удовлетворения таких потребностей создавались такие технологии, как Docker. Такие технологии, по сути, позволяют развернуть «изолированное приложение» из одного дистрибутива Linux в другом, не беспокоясь о сложном развертывании локальной установки или потреблении ресурсов виртуальной машины. Хотя технология изначально предназначалась только для Linux, компания Microsoft способствовала включению Docker в Windows 10 в составе обновления Anniversary Update. Технология может работать в двух режимах.

- Приложение развертывается в тяжеловесном, но полностью изолированном контейнере Hyper-V, который поддерживается как в клиентском, так и в серверном сценарии.
- Приложение развертывается в легковесном, изолированном на уровне ОС контейнере. В настоящее время данная возможность поддерживается только в серверных сценариях из-за причин, связанных с лицензированием.

Последняя технология, которая будет рассмотрена в этом разделе, вызвала глубокие изменения в операционной системе, необходимые для ее поддержки. Как

упоминалось ранее, возможность создания контейнеров серверных участков (server silos) для клиентских систем существует, но в настоящее время она отключена. В отличие от контейнеров Hyper-V, использующих среду с полноценной виртуализацией, контейнер серверного участка предоставляет второй «экземпляр» всех компонентов пользовательского режима, работающих поверх того же ядра и драйверов. За счет некоторого снижения безопасности формируется гораздо менее затратная среда контейнера.

Объекты заданий и участки

Возможность создания участков связана с некоторыми недокументированными субклассами API-функции `SetJobObjectInformation`. Другими словами, участок по сути является «суперзаданием» с дополнительными правилами и возможностями, выходящими за пределы того, что вы видели ранее. Более того, объект задания может использоваться как для изоляции и управления ресурсами, так и для создания участка. Такие задания называются *гибридными* (hybrid).

На практике объекты заданий могут обслуживать два типа участков: участки приложений (которые в настоящее время используются для реализации технологии Desktop Bridge и не рассматриваются в этом разделе — см. главу 9 части 2) и серверные участки, которые используются поддержкой контейнеров Docker.

Изоляция участков

Первый элемент, определяющий серверный участок — специальный объект корневого каталога (\) диспетчера объектов. (Диспетчер объектов рассматривается в главе 8 части 2.) И хотя этот механизм еще не рассматривался, достаточно сказать, что все именованные объекты, видимые приложению (файлы, разделы реестра, события, мьютексы, порты RPC и т. д.), размещаются в корневом пространстве имен, что позволяет приложениям создавать, находить и совместно использовать эти объекты.

Возможность существования у серверного участка собственного корня означает возможность контроля над обращениями к любому именованному объекту. Контроль может осуществляться тремя разными способами;

- созданием новой копии существующего объекта для предоставления альтернативного доступа к нему из участка;
- созданием символической ссылки на существующий объект для предоставления прямого доступа к нему;
- созданием совершенно нового объекта, который существует только в пределах участка (как объекты, используемые контейнеризованным приложением).

Эта способность объединяется с функциональностью Virtual Machine Compute (Vmcompute) (используемой Docker), которая взаимодействует с другими компонентами для предоставления полного уровня изоляции.

- **Файл с базовым образом Windows (WIM) – базовый образ ОС.** Этот компонент предоставляет отдельную копию операционной системы. На момент написания книги компания Microsoft предоставляет образы Server Core и Nano Server.
- **Библиотека Ntdll.dll ведущей ОС.** Переопределяет библиотеку из базового образа ОС. Это объясняется тем фактом, что серверные участки, как уже было сказано, используют то же ядро и драйверы. Поскольку Ntdll.dll обрабатывает системные функции, это единственный компонент пользовательского режима, который должен использоваться управляющей ОС.
- **Виртуальная файловая система «песочницы», предоставляемая драйвером-фильтром Wcifs.sys.** Позволяет вносить временные изменения в файловой системе контейнера без модификации диска NTFS. Такие изменения могут быть потеряны при закрытии контейнера.
- **Виртуальный реестр «песочницы», предоставляемый компонентом ядра VReg.** Позволяет предоставить временный набор кустов реестра (а также другой уровень изоляции пространства имен, так как корневое пространство имен диспетчера объектов изолирует только корень реестра, но не сами кусты реестра).
- **Диспетчер сеансов (Smss.exe).** Сейчас используется для создания дополнительных сеансов служб или консольных сеансов – новая возможность, необходимая для поддержки контейнеров. При этом Smss расширяется для поддержки не только дополнительных пользовательских сеансов, но и сеансов, необходимых для каждого запущенного контейнера.

На рис. 3.16 изображена архитектура таких контейнеров с этими компонентами.

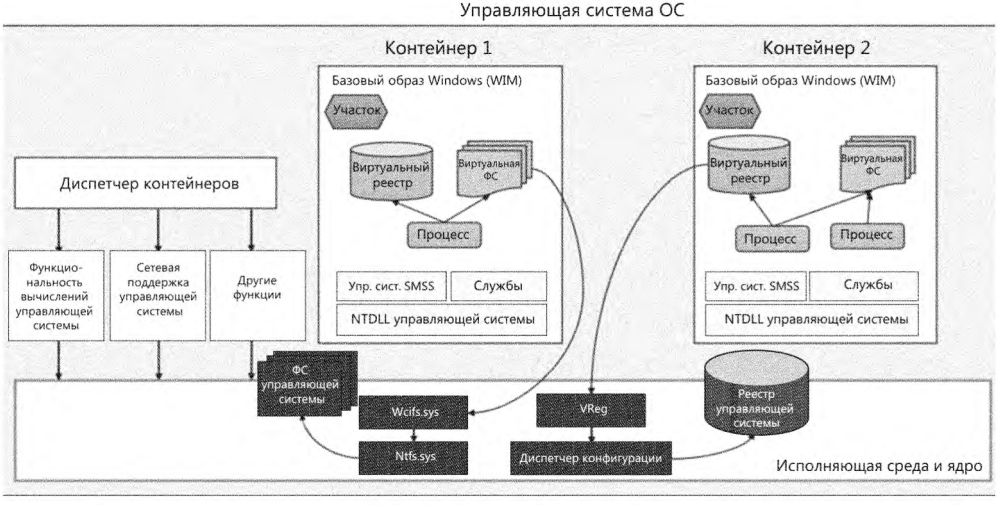


Рис. 3.16. Архитектура контейнеров

Границы изоляции участков

Упомянутые компоненты формируют среду изоляции пользовательского режима. Однако при использовании управляющего компонента Ntdll.dll, взаимодействующего с управляющим ядром и драйверами, важно иметь дополнительные границы изоляции. Ядро предоставляет такие границы для того, чтобы отличать один участок от другого. Соответственно, каждый серверный участок содержит собственные изолированные версии.

- **Общие пользовательские микроданные (SILO_USER_SHARED_DATA в символических именах).** Здесь хранится нестандартный системный путь, идентификатор сеанса, PID переднего плана и тип/семейство продукта. Все эти элементы исходных данных KUSER_SHARED_DATA не могут поступить от управляющей ОС, так как они содержат ссылки на информацию, относящуюся к образу управляющей ОС – вместо образа базовой ОС, которая должна использоваться вместо нее. Различные компоненты и API модифицируются так, чтобы при чтении таких данных использовались совместные общие данные участка (вместо пользовательских общих данных). При этом исходная структура KUSER_SHARED_DATA остается по своему обычному адресу со своим исходным представлением подробной информации хоста; это один из способов, которыми состояние управляющей системы «проникает» в состояние контейнера.

- **Корневое пространство имен каталога объектов.** Участок имеет собственную символическую ссылку \SystemRoot, каталог \Device (через который все компоненты пользовательского режима обращаются к драйверам устройств опосредованно), карту устройства и отображения устройств DOS (например, так приложения пользовательского режима обращаются к сетевым отображенным драйверам), каталог \Sessions и т. д.

- **Отображение набора API.** Базируется на схеме набора API образа базовой ОС, а не на том, который хранится в файловой системе управляющей ОС. Как упоминалось ранее, загрузчик использует отображения наборов API для определения того, какая DLL-библиотека реализует ту или иную функцию. Эти библиотеки могут различаться в зависимости от выпуска ОС, и приложениям должен быть виден выпуск базовой, а не управляющей ОС.

- **Сеанс входа.** Связывается с локально-уникальным идентификатором (LUID) SYSTEM и Anonymous, а также с LUID виртуальной учетной записи, описывающей пользователя в участке. Фактически представляет маркер служб и приложения, которые будут работать в контейнерном сеансе службы, созданном Smss. Подробнее о LUID и сеансах входа см. в главе 7.

- **Контексты трассировки ETW и средства ведения журнала.** Предназначены для изоляции операций ETW с участком без предоставления доступа или утечки данных состояния между контейнерами и/или самой управляющей ОС. (Подробнее о ETW см. в главе 9 части 2.)

Контексты участков

Хотя существуют изоляционные границы, предоставляемые ядром управляющей ОС, другие компоненты внутри ядра, а также драйверы (включая сторонние) могут добавлять контекстные данные в участки с использованием API-функции `PsCreateSiloContext` для записи специализированных данных, связанных с участком, или связыванием существующего объекта с участком. Каждый такой контекст участка использует индекс слота участка, который будет вставляться во все работающие (и будущие) серверные участки, хранящие указатель на контекст. Система предоставляет 32 встроенных общесистемных индекса слотов системного уровня и 256 слотов расширения, предоставляющих множество средств расширения.

При создании серверного участка он получает собственный массив SLS (Silo-Local Storage) по аналогии с тем, как у потоков имеется собственная память TLS (Thread-Local Storage). Внутри массива разные элементы будут соответствовать индексам слотов, выделенным для хранения контекстов участков. Разные участки будут хранить разные указатели на один индекс слота, но по этому индексу всегда хранится один контекст. (Например, драйверу `Foo` всегда принадлежит индекс 5 на всех участках, и он всегда может использовать его для хранения своего указателя/контекста в каждом участке.) В некоторых случаях встроенные компоненты ядра (такие, как диспетчер объектов, монитор безопасности (SRM) и диспетчер конфигурации) используют такие слоты, хотя другие слоты используются встроенными драйверами (такими, как вспомогательный функциональный драйвер для Winsock, `Afd.sys`).

Как и при работе с общими пользовательскими данными серверных участков, различные компоненты и API были обновлены: теперь они обращаются к данным из соответствующих контекстов участков, а не из используемых ранее глобальных переменных ядра. Например, поскольку каждый контейнер теперь содержит собственный процесс `Lsass.exe`, а SRM ядра нужен дескриптор для процесса `Lsass.exe` (подробнее о Lsass и SRM см. в главе 7), он уже не может быть одиночным объектом, хранящимся в глобальной переменной. Соответственно, SRM теперь обращается к дескриптору посредством запроса контекста участка активного серверного участка, а переменная читается из возвращенной структуры данных.

Возникает интересный вопрос: что происходит с процессом `Lsass.exe`, работающим в самой управляющей ОС? Как SRM будет обращаться к дескриптору, если для этого набора процессов и сеанса (т. е. для самого сеанса 0) не существует серверного участка? Чтобы разрешить эту проблему, ядро теперь реализует корневой управляющий участок. Другими словами, сама управляющая ОС теперь также считается частью участка! Она не является участком в полном смысле слова; скорее это хитрый трюк, благодаря которому запросы контекстов для текущего участка работают даже в том случае, если самого текущего участка нет. Задача решается хранением глобальной переменной ядра с именем `PspHostSiloGlobals`, имеющей собственный массив SLS, а также других контекстов участков, используемых встроенными компонентами ядра. При вызове различных API-функций участков с указателем `NULL` этот указатель интерпретируется как «участок отсутствует — использовать участок управляющей ОС».

ЭКСПЕРИМЕНТ: ВЫВОД КОНТЕКСТА УЧАСТКА SRM ДЛЯ УПРАВЛЯЮЩЕГО УЧАСТКА

Итак, хотя система Windows 10 может не управлять серверными участками (особенно если это клиентская система), все равно существует управляющий участок, который содержит изолированные контексты, используемые ядром. У отладчика Windows существует расширение `!silo`, которое может использоваться с параметрами Host в формате `!silo -g Host`. Вывод должен выглядеть примерно так:

```
lkd> !silo -g Host
Server silo globals fffff801b73bc580:
                        Default Error Port: ffffb30f25b48080
                        ServiceSessionId  : 0
                        Root Directory    : 00007fff00000000 ''
                        State             : Running
```

В вашем выводе указатель на глобальные переменные участка должен быть оформлен в виде гиперссылки. Щелчок на ней приводит к выполнению следующей команды:

```
lkd> dx -r1 (*((nt!_ESERVERSILO_GLOBALS *)0xfffff801b73bc580))
(*((nt!_ESERVERSILO_GLOBALS *)0xfffff801b73bc580))                [Type: _
ESERVERSILO_GLOBALS]
[+0x000] ObSiloState       [Type: _OBP_SILODRIVERSTATE]
[+0x2e0] SeSiloState       [Type: _SEP_SILOSTATE]
[+0x310] SeRmSiloState     [Type: _SEP_RM_LSA_CONNECTION_STATE]
[+0x360] CmSiloState       : 0xffffc308870931b0 [Type: _CMP_SILO_CONTEXT *]
[+0x368] EtwSiloState      : 0xffffb30f236c4000 [Type: _ETW_SILODRIVERSTATE *]
...
```

Теперь щелкните на поле `SeRmSiloState`. В открывшейся структуре, среди прочего, содержится указатель на процесс Lsass.exe:

```
lkd> dx -r1 ((ntkrnlmp!_SEP_RM_LSA_CONNECTION_STATE *)0xfffff801b73bc890)
((ntkrnlmp!_SEP_RM_LSA_CONNECTION_STATE *)0xfffff801b73bc890)       :
0xfffff801b73bc890 [Type: _SEP_RM_LSA_CONNECTION_STATE *]
    [+0x000] LsaProcessHandle : 0xffffffff80000870 [Type: void *]
    [+0x008] LsaCommandPortHandle : 0xffffffff8000087c [Type: void *]
    [+0x010] SepRmThreadHandle : 0x0 [Type: void *]
    [+0x018] RmCommandPortHandle : 0xffffffff80000874 [Type: void *]
```

Мониторы участков

Если драйверы ядра обладают возможностью добавлять собственные контексты участков, как они узнают, какие участки выполняются и какие новые участки создаются при запуске контейнеров? Ответ лежит в мониторе участков, предоставляющем набор API-функций для получения уведомлений о создании и/или завершении серверных участков (`PsRegisterSiloMonitor`, `PsStartSiloMonitor`, `PsUnregisterSiloMonitor`), а также уведомлений для любых уже существующих участков. Затем каждый монитор участка может получить свой индекс слота вызовом `PsGetSiloMonitorContextSlot`, который затем может использоваться

функциями PsInsertSiloContext, PsReplaceSiloContext и PsRemoveSiloContext при необходимости. Вы можете создавать дополнительные слоты вызовом PsAllocSiloContextSlot, но это потребуется только в том случае, если компонент по какой-то причине захочет сохранить два контекста. Кроме того, драйверы могут использовать API-функцию PsInsertPermanentSiloContext или PsMakeSiloContextPermanent для создания «долгосрочных» контекстов участков, которые не учитываются при подсчете ссылок и не привязываются к сроку жизни серверного участка или количеству операций чтения контекстов участков. После сохранения такие контексты участков могут читаться методами PsGetSiloContext и/или PsGetPermanentSiloContext.

ЭКСПЕРИМЕНТ: МОНИТОРЫ И КОНТЕКСТЫ УЧАСТКОВ

Чтобы понять, как используются мониторы участков и как хранятся контексты, рассмотрим вспомогательный функциональный драйвер для Winsock (Afd.sys) и его монитор. Сначала выведем структуру данных, представляющую монитор. К сожалению, в файлах с символической информацией ее нет, поэтому данные придется выводить в низкоуровневом формате.

```
lkd> dps poi(afd!AfdPodMonitor)
ffffe387'a79fc120  ffffe387'a7d760c0
ffffe387'a79fc128  ffffe387'a7b54b60
ffffe387'a79fc130  00000009'00000101
ffffe387'a79fc138  fffff807'be4b5b10 afd!AfdPodSiloCreateCallback
ffffe387'a79fc140  fffff807'be4bee40 afd!AfdPodSiloTerminateCallback
```

Обратите внимание на слот (9 в данном примере) из управляющего участка. Участки хранят свои SLS в поле с именем Storage, которое содержит массив структур данных (элементов слотов); каждая структура содержит указатель и некоторые флаги. Для получения смещения правильного элемента слота индекс умножается на 2, после чего мы обращаемся ко второму полю (+1) для получения указателя на следующий указатель:

```
lkd> r? @$t0 = (nt!_ESERVERSILO_GLOBALS*)@@masm(nt!PspHostSiloGlobals)
lkd> ?? ((void***)@$t0->Storage)[9 * 2 + 1]
void ** 0xffff988f'ab815941
```

Обратите внимание: флаг долгосрочного хранения включен в указатель операцией OR. Исключите его при помощи маски, а затем используйте расширение !object для подтверждения того, что это действительно контекст участка.

```
lkd> !object (0xffff988f'ab815941 & -2)
Object: ffff988fab815940 Type: (ffff988faaac9f20) PsSiloContextNonPaged
```

Создание серверного участка

При создании серверного участка сначала используется объект задания, потому что, как упоминалось выше, участки реализуются на основе объектов заданий. Для этого используется стандартная API-функция CreateJobObject, которая была

модифицирована в составе обновления Anniversary Update так, что в нее включается идентификатор задания (JID, Job ID). JID берется из того же пула чисел, что и идентификаторы процесса и потока (PID и TID), т. е. из таблицы клиентских идентификаторов (CID, Client ID). Соответственно, идентификаторы JID уникальны не только между другими заданиями, но также между другими процессами и потоками. Так же автоматически создается GUID контейнера.

Затем используется API-функция `SetInformationJobObject` с классом информации создания участка. Это приводит к установке флага `Silo` в объекте исполняющей системы `EJOB`, представляющем задание, а также к размещению в памяти массива слотов SLS, который вы уже видели ранее в поле `Storage` объекта EJOB. К этому моменту создается *участок приложения*.

Затем создается пространство имен корневого каталога объектов с другим информационным классом и вызовом `SetInformationJobObject`. Новый класс требует привилегии TCB (Trusted Computing Base). Так как участки обычно создаются только службой Vmcompute, это делается для того, чтобы предотвратить злонамеренное использование пространств имен виртуальных приложений для введения приложений в заблуждение и потенциальное нарушение их работы. При создании этого пространства имен диспетчер объектов создает или открывает новый каталог Silos в реальном корне управляющей ОС (\) и присоединяет JIT для создания нового виртуального корня (например, \Silos\148\). Затем создаются объекты `KernelObjects`, `ObjectTypes`, `GLOBALROOT` и `DosDevices`. Далее корень сохраняется как контекст участка с индексом слота из `PsObjectDirectorySiloContextSlot`, выделенным диспетчером объектов при загрузке.

Следующий шаг — преобразование участка в серверный участок, что требует еще одного вызова `SetInformationJobObject` и еще одного информационного класса. Затем выполняется функция `PspConvertSiloToServerSilo` в ядре, которая инициализирует структуру `ESERVERSILO_GLOBALS`, которая была представлена ранее как часть эксперимента с выводом `PspHostSiloGlobals` командой `!silo`. Функция инициализирует пользовательские общие данные участка, отображение набора API, `SystemRoot` и различные контексты участка (например, тот, который использовался SRM для идентификации процесса Lsass.exe). В ходе преобразования мониторы участков, которые зарегистрировали и запустили свои обратные вызовы, будут получать уведомления, что позволит им добавить собственные контекстные данные участка.

Остается сделать последний шаг: «запустить» серверный участок, инициализируя новый сеанс службы для него. Считайте, что это своего рода сеанс 0, но для серверного участка. Для этого используется сообщение ALPC, отправляемое Smss `SmApiPort`, которое содержит дескриптор объекта задания, созданного Vmcompute, который теперь становится объектом задания серверного участка. Как и при создании реального пользовательского сеанса, Smss клонирует собственную копию, но на этот раз копия будет связана с объектом задания в момент создания. Таким образом, новая копия Smss присоединяется ко всем контейнеризованным элементам

серверного участка. Smss будет считать, что это сеанс 0, и выполнит свои обычные действия: запуск Csrss.exe, Wininit.exe, Lsass.exe и т. д. Процесс «запуска» будет продолжаться как обычно: Wininit.exe запустит диспетчер служб (Services.exe), который в свою очередь запустит все службы с автоматическим режимом запуска, и т. д. Теперь в серверном участке могут выполняться новые приложения, которые будут запускаться с сеансом входа, связанным с LUID виртуальной учетной записи службы, как описано выше.

Вспомогательная функциональность

Возможно, вы заметили, что короткое описание, приведенное выше, не приведет к реальному успеху процесса «загрузки». Например, если в процессе инициализации потребуется создать именованный канал с именем `ntsvcs`, это потребует взаимодействия с объектом \Device\NamedPipe или, как его видит Services.exe, — \Silos\JID\Device\NamedPipe. Но такой объект устройства не существует!

Соответственно, для того чтобы драйвер устройства мог функционировать, драйверы должны получить информацию и зарегистрировать собственные мониторы участков, которые затем используют уведомления для создания собственных объектов устройств уровня участка. Ядро предоставляет API-функцию `PsAttachSiloToCurrentThread` (и парную функцию `PsDetachSiloFromCurrentThread`), которая временно сохраняет в поле `Silo` объекта `ETHREAD` переданный объект задания. Это приводит к тому, что все обращения (например, к диспетчеру объектов) будут рассматриваться как исходящие от участка.

Например, драйвер именованного канала может использовать эту функциональность для создания объекта `NamedPipe` в пространстве имен \Device, который становится частью \Silos\JID\.

Другой вопрос: если приложения запускаются фактически в сеансе «службы», как они могут взаимодействовать с пользователем, обрабатывать ввод и вывод? Во-первых, важно заметить, что при запуске в контейнере Windows графический интерфейс невозможен и не разрешен, а попытки использовать удаленный рабочий стол (Remote Desktop, RDP) для обращения к контейнеру тоже невозможны. Соответственно, выполняться могут только приложения командной строки. Но даже таким приложениям обычно нужен «интерактивный» сеанс. Как *они* могут функционировать? Секрет кроется в специальном управляющем процессе CExecSvc.exe, который реализует службу исполнения контейнера. Служба через именованный канал взаимодействует со службами Docker и Vmcompute управляющей ОС и используется для запуска контейнеризованных приложений в сеансе. Она также используется для эмуляции консольной функциональности, которая обычно предоставляется Conhost.exe, передачи ввода и вывода по именованному каналу окну командной строки (или PowerShell), которое изначально использовалось для выполнения команды `docker` в управляющей ОС. Служба также используется при выполнении таких команд, как `docker cp` (передача файлов контейнеру или из контейнера).

Шаблоны контейнеров

Даже если принять во внимание все объекты устройств, которые могут создаваться драйверами при создании участков, есть *другие* бесчисленные объекты, которые создаются ядром и другими компонентами, с которыми должны взаимодействовать службы, работающие в сеансе 0, и наоборот. В пользовательском режиме нет системы мониторов участков, которая позволила бы компонентам поддерживать эту необходимость, а заставлять каждый драйвер всегда создавать специализированный объект устройства для представления каждого участка было бы неразумно.

Если участок хочет воспроизвести музыку на звуковой карте, он не должен использовать отдельный объект устройства для звуковой карты, к которой будут обращаться все остальные участки, а также управляющая ОС. Это было бы необходимо только в том случае, если бы, допустим, требовалось реализовать изоляцию объектов на уровне участков. Другой пример – драйвер AFD. Хотя он использует монитор участка, это делается для идентификации службы пользовательского режима, которая является хостом клиента DNS. Эта информация нужна драйверу для взаимодействия с запросами DNS режима ядра, которая ведется на уровне участка, без создания отдельных объектов \Silos\JID\Device\Afd, так как в системе существует единый стек сети /Winsock.

Кроме драйверов и объектов реестр также содержит различные глобальные сведения, которые должны быть видимыми и существовать во всех участках. На основе этих объектов компонент VReg затем предоставляет изоляцию.

Для поддержки всех этих потребностей пространство имен участка, реестр и файловая система определяются специализированным контейнерным шаблоном, который по умолчанию хранится в файле %SystemRoot%\System32\Containers\wsc.def после включения поддержки контейнеров Windows в диалоговом окне Включение или отключение компонентов Windows (Add/Remove Windows Features). Этот файл описывает пространство имен диспетчера объектов и реестра и связанные с ним правила, позволяющие определять по мере надобности символические ссылки для «настоящих» объектов управляющей ОС. Кроме того, он также описывает, какой объект задания, точки монтирования томов и политики сетевой изоляции должны использоваться. Теоретически в будущем при использовании объектов участков в операционной системе Windows можно будет выбирать другие файлы шаблонов для создания других видов контейнеризованных сред. Ниже приведен фрагмент файла wsc.def в системе, в которой включена поддержка контейнеров:

```
<!-- Файл определения участка для cmdserver.exe -->
<container>
    <namespace>
        <ob shadow="false">
            <symlink name="FileSystem" path="\FileSystem" scope="Global" />
            <symlink name="PdcPort" path="\PdcPort" scope="Global" />
            <symlink name="SeRmCommandPort" path="\SeRmCommandPort" scope="Global"
/>
            <symlink name="Registry" path="\Registry" scope="Global" />
```

```
            <symlink name="Driver" path="\Driver" scope="Global" />
            <objdir name="BaseNamedObjects" clonesd="\BaseNamedObjects"
shadow="false"/>
            <objdir name="GLOBAL??" clonesd="\GLOBAL??" shadow="false">
                <!-- Needed to map directories from the host -->
                <symlink name="ContainerMappedDirectories" path="\
ContainerMappedDirectories" scope="Local" />
                <!-- Допустимые ссылки на \Device -->
                <symlink name="WMIDataDevice" path="\Device\WMIDataDevice"
scope="Local"
/>
                <symlink name="UNC" path="\Device\Mup" scope="Local" />
...
            </objdir>
            <objdir name="Device" clonesd="\Device" shadow="false">
                <symlink name="Afd" path="\Device\Afd" scope="Global" />
                <symlink name="ahcache" path="\Device\ahcache" scope="Global" />
                <symlink name="CNG" path="\Device\CNG" scope="Global" />
                <symlink name="ConDrv" path="\Device\ConDrv" scope="Global" />
...
        <registry>
            <load
                key="$SiloHivesRoot$\Silo$TopLayerName$Software_Base"
                path="$TopLayerPath$\Hives\Software_Base"
                ReadOnly="true"
                />
...
                <mkkey
                    name="ControlSet001"
                    clonesd="\REGISTRY\Machine\SYSTEM\ControlSet001"
                    />
                <mkkey
                    name="ControlSet001\Control"
                    clonesd="\REGISTRY\Machine\SYSTEM\ControlSet001\Control"
                    />
```

Заключение

В данной главе была изучена структура процессов, включая способы их создания и уничтожения. Вы увидели, как задания могут применяться для управления группами процессов как единым целым и как серверные участки способствуют открытию новой эры поддержки контейнеризации в серверных версиях Windows. В следующей главе будут подробно рассмотрены потоки — их структура и принципы работы, принципы их планирования к исполнению, а также различные способы выполнения операций с потоками и использования.

Глава 4
Потоки

В этой главе объясняются структуры данных и алгоритмы, относящиеся к потокам и планированию потоков в Windows. В первом разделе показано, как создавать потоки. Затем описывается внутреннее строение и принципы планирования потоков. Глава завершается описанием пулов потоков.

Создание потоков

Прежде чем обсуждать внутренние структуры, используемые для управления потоками, рассмотрим создание потоков с точки зрения API. Это поможет вам составить представление о действиях и аргументах, задействованных в этом процессе.

Простейшая функция создания потоков в пользовательском режиме — `CreateThread` — создает поток в текущем процессе. При вызове ей передаются следующие аргументы.

- **Необязательная структура с атрибутами безопасности.** Этот аргумент задает дескриптор безопасности, который должен присоединяться к вновь созданному процессу. Он также указывает, должен ли дескриптор потока создаваться как наследуемый. (Наследование дескрипторов обсуждается в главе 8 части 2.)
- **Необязательный размер стека.** Если задан нулевой размер, значение по умолчанию читается из заголовка исполняемого файла. Значение всегда применяется к первому потоку в процессе пользовательского режима. (Стеки потоков рассматриваются в главе 5.)
- **Указатель на функцию.** Точка входа для выполнения нового потока.
- **Необязательный аргумент.** Передается функции потока.
- **Необязательные флаги.** Первый управляет запуском потока в приостановленном режиме (`CREATE_SUSPENDED`). Другой управляет интерпретацией аргумента размера стека (исходный зафиксированный или максимальный зарезервированный размер).

При успешном завершении для нового потока возвращается ненулевой дескриптор и по запросу вызывающей стороны — уникальный идентификатор потока.

Для расширенного создания потоков применяется функция `CreateRemoteThread`. Она получает дополнительный аргумент (первый) с дескриптором процесса, в ко-

тором должен создаваться поток. Функция может использоваться для внедрения потоков в другие процессы. В частности, она применяется для инициирования останова в отлаживаемых процессах. Отладчик внедряет поток, который немедленно прерывает выполнение процесса вызовом `DebugBreak`. Также этот прием часто используется для получения процессом внутренней информации о другом процессе, что проще сделать при выполнении в контексте целевого пространства (например, при этом видимо все адресное пространство). Это может делаться как в законных, так и злонамеренных целях.

Чтобы функция `CreateRemoteThread` работала, дескриптор процесса должен быть получен с достаточными правами доступа, разрешающими такую операцию. Яркий пример такого рода: защищенные процессы не могут внедряться подобным образом, потому что дескрипторы таких процессов могут быть получены только с очень ограниченными правами.

Последняя функция, о которой стоит упомянуть здесь, — `CreateRemoteThreadEx`, она является надмножеством `CreateThread` и `CreateRemoteThread`. Собственно, реализация `CreateThread` и `CreateRemoteThread` просто вызывает `CreateRemoteThreadEx` с соответствующими значениями по умолчанию. `CreateRemoteThreadEx` добавляет возможность передачи списка атрибутов (по аналогии с ролью структуры `STARTUPINFOEX` с дополнительным полем, добавленным в `STARTUPINFO`, при создании процессов). Например, при помощи атрибутов можно задать идеальный процессор и групповое сходство (оба атрибута рассматриваются далее в этой главе).

Если все прошло нормально, функция `CreateRemoteThreadEx` со временем вызывает `NtCreateThreadEx` из Ntdll.dll. При этом происходит обычный переход в режим ядра, где выполнение продолжается в функции `NtCreateThreadEx`. Здесь происходит часть создания потока в режиме ядра (см. раздел «Рождение потока» этой главы).

Создание потока в режиме ядра осуществляется вызовом функции `PsCreateSystemThread` (документированной в WDK). Это может быть полезно для драйверов, которым необходимо выполнять независимую работу в системном процессе (т. е. без связи с каким-либо конкретным процессом). С технической точки зрения функция может использоваться для создания потока в любом процессе, что не так полезно для драйверов.

Выход из функции потока ядра не приводит к автоматическому уничтожению объекта потока. Вместо этого драйверы должны вызвать `PsTerminateSystemThread` из функции потока, чтобы корректно завершить поток. Соответственно, эта функция никогда не возвращает управление.

Внутреннее устройство потоков

В этом разделе обсуждаются внутренние структуры, используемые ядром (а некоторые и в пользовательском режиме) для управления потоками. Если не указано иное, вы можете считать, что все в этом разделе применимо как к потокам пользовательского режима, так и к системным потокам ядра.

Структуры данных

На уровне операционной системы поток Windows представлен создаваемым в исполняющей системе объектом потока. Этот объект инкапсулирует структуру ETHREAD, которая, в свою очередь, содержит в своем первом поле структуру KTHREAD. Это показано на рис. 4.1 (ETHREAD) и рис. 4.2 (KTHREAD). Структура ETHREAD и другие структуры, на которые она указывает, находятся в системном адресном пространстве, за исключением блока переменных окружения потока — TEB (Thread Environment Block), — который находится в адресном пространстве процесса (по аналогии с PEB, потому что он должен быть доступен для компонентов, выполняющихся в пользовательском режиме).

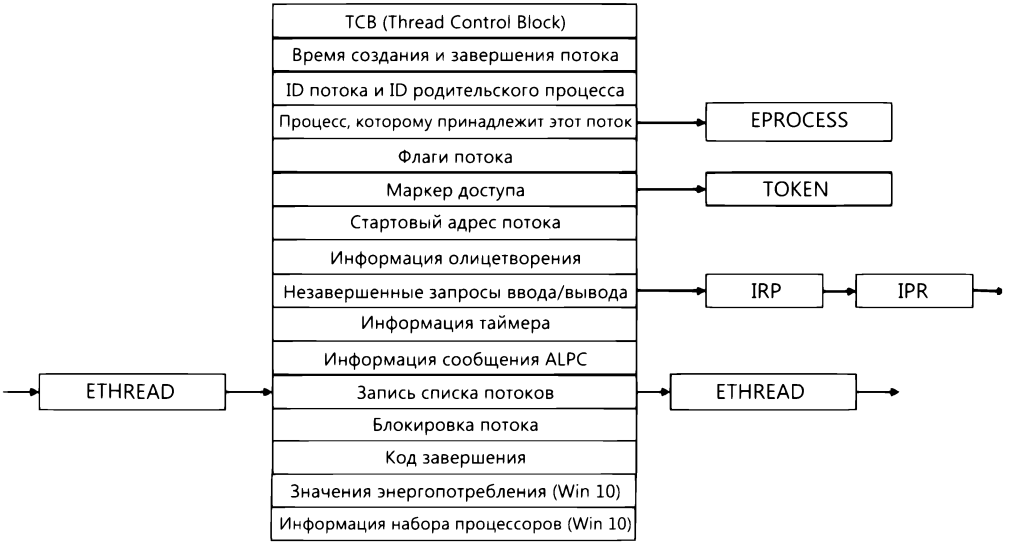


Рис. 4.1. Важные поля структуры потока исполняющей системы ETHREAD

Процесс подсистемы Windows (Csrss) поддерживает параллельную структуру для каждого потока, созданного в приложении этой подсистемы, которая называется CSR_THREAD. Для потоков, вызвавших USER- или GDI-функции подсистемы Windows, та часть этой подсистемы, которая выполняется в режиме ядра (Win32k.sys), поддерживает структуру данных для каждого потока (она называется W32THREAD), на которую указывает структура KTHREAD.

ПРИМЕЧАНИЕ Тот факт, что на исполняющую, высокоуровневую, связанную с графикой структуру потока Win32k указывает KTHREAD, а не ETHREAD, является нарушением системы уровней или упущением в стандартной архитектуре абстракции ядра — планировщик и другие низкоуровневые компоненты этим полем не пользуются.

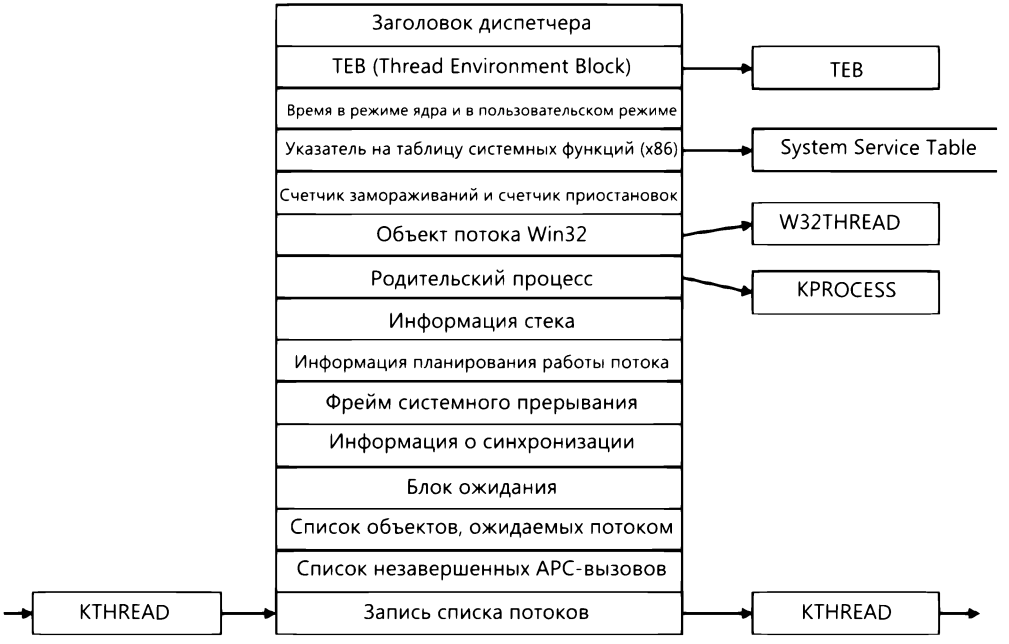


Рис. 4.2. Важные поля структуры потока ядра KTHREAD

Большинство полей, показанных на рис. 4.1, говорят сами за себя. Первое поле структуры `ETHREAD` называется `Tcb` (сокращение от Thread control block — блок управления потоком); оно содержит структуру типа KTHREAD. Далее следует информация идентификации потока; информация идентификации процесса (включая указатель на процесс-владелец, чтобы можно было получить доступ к его информации окружения), информация безопасности в форме указателя на маркер доступа и информации олицетворения (impersonation); поля, относящиеся к сообщениям асинхронного вызова локальных процедур, — Asynchronous Local Procedure Call (ALPC); незавершенные запросы ввода/вывода (IRP); специфические для Win10 поля, относящиеся к управлению питанием (см. главу 6); и наборы процессоров (см. далее в этой главе). Некоторые из этих основных полей более подробно рассматриваются в других разделах данной книги. Для более подробного изучения внутреннего устройства структуры `ETHREAD` можно воспользоваться командой отладчика ядра `dt`, которая выведет ее формат.

Рассмотрим подробнее две основные структуры данных, упомянутые в тексте: `ETHREAD` и `KTHREAD`. Структура `KTHREAD` (хранящаяся в поле `Tcb` структуры `ETHREAD`) содержит информацию, нужную ядру Windows для планирования выполнения потока, синхронизации и хронометража.

ЭКСПЕРИМЕНТ: ВЫВОД СТРУКТУР ETHREAD И KTHREAD

Структуры ETHREAD и KTHREAD можно вывести на экран с помощью команды dt отладчика ядра. В следующем выводе показан формат структуры ETHREAD на 64-разрядной системе Windows 10:

```
lkd> dt nt!_ethread
   +0x000 Tcb              : _KTHREAD
   +0x5d8 CreateTime       : _LARGE_INTEGER
   +0x5e0 ExitTime         : _LARGE_INTEGER
...
   +0x7a0 EnergyValues     : Ptr64 _THREAD_ENERGY_VALUES
   +0x7a8 CmCellReferences : Uint4B
   +0x7b0 SelectedCpuSets  : Uint8B
   +0x7b0 SelectedCpuSetsIndirect : Ptr64 Uint8B
   +0x7b8 Silo             : Ptr64 _EJOB
```

Структуру KTHREAD можно вывести с помощью аналогичной команды или командой dt nt!_ETHREAD Tcb, как было показано ранее в эксперименте со структурами EPROCESS и KPROCESS главы 3:

```
lkd> dt nt!_kthread
   +0x000 Header           : _DISPATCHER_HEADER
   +0x018 SListFaultAddress : Ptr64 Void
   +0x020 QuantumTarget    : Uint8B
   +0x028 InitialStack     : Ptr64 Void
   +0x030 StackLimit       : Ptr64 Void
   +0x038 StackBase        : Ptr64 Void
   +0x040 ThreadLock       : Uint8B
   +0x048 CycleTime        : Uint8B
   +0x050 CurrentRunTime   : Uint4B
...
   +0x5a0 ReadOperationCount : Int8B
   +0x5a8 WriteOperationCount : Int8B
   +0x5b0 OtherOperationCount : Int8B
   +0x5b8 ReadTransferCount : Int8B
   +0x5c0 WriteTransferCount : Int8B
   +0x5c8 OtherTransferCount : Int8B
   +0x5d0 QueuedScb        : Ptr64 _KSCB
```

ЭКСПЕРИМЕНТ: ИСПОЛЬЗОВАНИЕ КОМАНДЫ ОТЛАДЧИКА ЯДРА !THREAD

Команда отладчика ядра !thread выводит дамп поднабора информации в структурах данных потока. Некоторые основные элементы информации, выводимые отладчиком ядра, не могут быть выведены какой-либо утилитой, в том числе:

- внутренние адреса структуры;
- подробные данные о приоритете;
- информация о стеке;
- список отложенных запросов ввода/вывода;

- для потоков, находящихся в состоянии ожидания, — список объектов, ожидаемых потоком.

Для вывода информации о потоке нужно воспользоваться либо командой !process (выводящей все потоки процесса после вывода информации о процессе), либо командой !thread с адресом объекта потока для вывода информации о конкретном потоке.

Найдем все экземпляры explorer.exe:

```
lkd> !process 0 0 explorer.exe
PROCESS ffffe00017f3e7c0
    SessionId: 1 Cid: 0b7c Peb: 00291000 ParentCid: 0c34
    DirBase: 19b264000 ObjectTable: ffffc00007268cc0 HandleCount: 2248.
    Image: explorer.exe

PROCESS ffffe00018c817c0
    SessionId: 1 Cid: 23b0 Peb: 00256000 ParentCid: 03f0
    DirBase: 2d4010000 ObjectTable: ffffc0001aef0480 HandleCount: 2208.
    Image: explorer.exe
```

Найдем один из экземпляров и выведем его потоки:

```
lkd> !process ffffe00018c817c0 2
PROCESS ffffe00018c817c0
    SessionId: 1 Cid: 23b0 Peb: 00256000 ParentCid: 03f0
    DirBase: 2d4010000 ObjectTable: ffffc0001aef0480 HandleCount: 2232.
    Image: explorer.exe

        THREAD ffffe0001ac3c080 Cid 23b0.2b88 Teb: 0000000000257000 Win32Thread:
ffffe0001570ca20 WAIT: (UserRequest) UserMode Non-Alertable
            ffffe0001b6eb470 SynchronizationEvent

        THREAD ffffe0001af10800 Cid 23b0.2f40 Teb: 0000000000265000 Win32Thread:
ffffe000156688a0 WAIT: (UserRequest) UserMode Non-Alertable
            ffffe000172ad4f0 SynchronizationEvent
            ffffe0001ac26420 SynchronizationEvent

        THREAD ffffe0001b69a080 Cid 23b0.2f4c Teb: 0000000000267000 Win32Thread:
ffffe000192c5350 WAIT: (UserRequest) UserMode Non-Alertable
            ffffe00018d83c00 SynchronizationEvent
            ffffe0001552ff40 SynchronizationEvent

...
        THREAD ffffe00023422080 Cid 23b0.3d8c Teb: 00000000003cf000 Win32Thread:
ffffe0001eccd790 WAIT: (WrQueue) UserMode Alertable
            ffffe0001aec9080 QueueObject

        THREAD ffffe00023f23080 Cid 23b0.3af8 Teb: 00000000003d1000 Win32Thread:
0000000000000000 WAIT: (WrQueue) UserMode Alertable
            ffffe0001aec9080 QueueObject

        THREAD ffffe000230bf800 Cid 23b0.2d6c Teb: 00000000003d3000 Win32Thread:
0000000000000000 WAIT: (WrQueue) UserMode Alertable
            ffffe0001aec9080 QueueObject

        THREAD ffffe0001f0b5800 Cid 23b0.3398 Teb: 00000000003e3000 Win32Thread:
0000000000000000 WAIT: (UserRequest) UserMode Alertable
            ffffe0001d19d790 SynchronizationEvent
            ffffe00022b42660 SynchronizationTimer
```

Список потоков усечен для экономии места. Для каждого потока указывается его адрес (ETHREAD), который может передаваться команде `!thread`; его идентификатор клиента (Cid, Client ID) — идентификатор процесса и идентификатор потока (идентификатор процесса для всех предшествующих потоков одинаковый, так как все потоки являются частью одного процесса explorer.exe); блок TEB, который будет рассмотрен ниже; и состояние потока (большинство потоков должно находиться в состоянии Wait с указанием причины в скобках). В следующей строке может выводиться список объектов синхронизации, ожидаемых потоками.

Чтобы получить более подробную информацию о конкретном потоке, передайте его адрес команде `!thread`:

```
lkd> !thread ffffe0001d45d800
THREAD ffffe0001d45d800  Cid 23b0.452c  Teb: 000000000026d000 Win32Thread:
ffffe0001aace630 WAIT: (UserRequest) UserMode Non-Alertable
    ffffe00023678350  NotificationEvent
    ffffe00022aeb370  Semaphore Limit 0xffff
    ffffe000225645b0  SynchronizationEvent
Not impersonating
DeviceMap                 ffffc00004f7ddb0
Owning Process            ffffe00018c817c0       Image:         explorer.exe
Attached Process          N/A            Image:         N/A
Wait Start TickCount      7233205        Ticks: 270 (0:00:00:04.218)
Context Switch Count      6570           IdealProcessor: 7
UserTime                  00:00:00.078
KernelTime                00:00:00.046
Win32 Start Address 0c
Stack Init ffffd000271d4c90 Current ffffd000271d3f80
Base ffffd000271d5000 Limit ffffd000271cf000 Call 0000000000000000
Priority 9 BasePriority 8 PriorityDecrement 0 IoPriority 2 PagePriority 5
GetContextState failed, 0x80004001
Unable to get current machine context, HRESULT 0x80004001
Child-SP          RetAddr           : Args to Child                   : Call Site
ffffd000'271d3fc0 fffff803'bef086ca : 00000000'00000000 00000000'00000001
00000000'00000000 00000000'00000000 : nt!KiSwapContext+0x76
ffffd000'271d4100 fffff803'bef08159 : ffffe000'1d45d800 fffff803'00000000
ffffe000'1aec9080 00000000'0000000f : nt!KiSwapThread+0x15a
ffffd000'271d41b0 fffff803'bef09cfe : 00000000'00000000 00000000'00000000
ffffe000'0000000f 00000000'00000003 : nt!KiCommitThreadWait+0x149
ffffd000'271d4240 fffff803'bf2a445d : ffffd000'00000003 ffffd000'271d43c0
00000000'00000000 fffff960'00000006 : nt!KeWaitForMultipleObjects+0x24e
ffffd000'271d4300 fffff803'bf2fa246 : fffff803'bf1a6b40 ffffd000'271d4810
ffffd000'271d4858 ffffe000'20aeca60 : nt!ObWaitForMultipleObjects+0x2bd
ffffd000'271d4810 fffff803'befdefa3 : 00000000'00000fa0 fffff803'bef02aad
ffffe000'1d45d800 00000000'1e22f198 : nt!NtWaitForMultipleObjects+0xf6
ffffd000'271d4a90 00007ffe'f42b5c24 : 00000000'00000000 00000000'00000000
00000000'00000000 00000000'00000000 : nt!KiSystemServiceCopyEnd+0x13
                                      (TrapFrame @
ffffd000'271d4b00)
00000000'1e22f178 00000000'00000000 : 00000000'00000000 00000000'00000000
00000000'00000000 00000000'00000000 : 0x00007ffe'f42b5c24
```

О потоке выводится много разнообразной информации: его приоритет, информация о стеке, время в пользовательском режиме и режиме ядра и т. д. Многие из этих подробностей рассматриваются в этой главе, в главе 5 «Управление памятью» и в главе 6 «Система ввода/вывода».

ЭКСПЕРИМЕНТ: ПРОСМОТР ИНФОРМАЦИИ ПОТОКА КОМАНДОЙ TLIST

Ниже показан подробный вывод процесса, полученный программой `tlist` из пакета средств отладки для Windows (проследите за тем, чтобы программа `tlist` запускалась из каталога, соответствующего «разрядности» целевого процесса). Обратите внимание: в списке потоков указывается значение `Win32StartAddr`. Это адрес, который передается функции `CreateThread` приложением. Все остальные программы (кроме Process Explorer), выводящие стартовый адрес потока, выводят фактический стартовый адрес (функцию в Ntdll.dll), а не стартовый адрес, заданный приложением.

Ниже приведен (неполный) результат выполнения `tlist` для Word 2016:

```
C:\Dbg\x86>tlist winword
 120 WINWORD.EXE       Chapter04.docm - Word
   CWD:     C:\Users\pavely\Documents\
   CmdLine: "C:\Program Files (x86)\Microsoft Office\Root\Office16\
   WINWORD.EXE" /n
"D:\OneDrive\WindowsInternalsBook\7thEdition\Chapter04\Chapter04.docm
   VirtualSize:   778012 KB   PeakVirtualSize:   832680 KB
   WorkingSetSize:185336 KB   PeakWorkingSetSize:227144 KB
   NumberOfThreads: 45
   12132 Win32StartAddr:0x00921000 LastErr:0x00000000 State:Waiting
   15540 Win32StartAddr:0x6cc2fdd8 LastErr:0x00000000 State:Waiting
   7096 Win32StartAddr:0x6cc3c6b2 LastErr:0x00000006 State:Waiting
   17696 Win32StartAddr:0x77c1c6d0 LastErr:0x00000000 State:Waiting
   17492 Win32StartAddr:0x77c1c6d0 LastErr:0x00000000 State:Waiting
   4052 Win32StartAddr:0x70aa5cf7 LastErr:0x00000000 State:Waiting
   14096 Win32StartAddr:0x70aa41d4 LastErr:0x00000000 State:Waiting
   6220 Win32StartAddr:0x70aa41d4 LastErr:0x00000000 State:Waiting
   7204 Win32StartAddr:0x77c1c6d0 LastErr:0x00000000 State:Waiting
   1196 Win32StartAddr:0x6ea016c0 LastErr:0x00000057 State:Waiting
   8848 Win32StartAddr:0x70aa41d4 LastErr:0x00000000 State:Waiting
   3352 Win32StartAddr:0x77c1c6d0 LastErr:0x00000000 State:Waiting
   11612 Win32StartAddr:0x77c1c6d0 LastErr:0x00000000 State:Waiting
   17420 Win32StartAddr:0x77c1c6d0 LastErr:0x00000000 State:Waiting
   13612 Win32StartAddr:0x77c1c6d0 LastErr:0x00000000 State:Waiting
   15052 Win32StartAddr:0x77c1c6d0 LastErr:0x00000000 State:Waiting
   ...
   12080 Win32StartAddr:0x77c1c6d0 LastErr:0x00000000 State:Waiting
   9456 Win32StartAddr:0x77c1c6d0 LastErr:0x00002f94 State:Waiting
   9808 Win32StartAddr:0x77c1c6d0 LastErr:0x00000000 State:Waiting
   16208 Win32StartAddr:0x77c1c6d0 LastErr:0x00000000 State:Waiting
   9396 Win32StartAddr:0x77c1c6d0 LastErr:0x00000000 State:Waiting
   2688 Win32StartAddr:0x70aa41d4 LastErr:0x00000000 State:Waiting
   9100 Win32StartAddr:0x70aa41d4 LastErr:0x00000000 State:Waiting
   18364 Win32StartAddr:0x70aa41d4 LastErr:0x00000000 State:Waiting
   11180 Win32StartAddr:0x70aa41d4 LastErr:0x00000000 State:Waiting
```

```
 16.0.6741.2037 shp  0x00920000  C:\Program Files (x86)\Microsoft Office\Root\
Office16\WINWORD.EXE
 10.0.10586.122 shp  0x77BF0000  C:\windows\SYSTEM32\ntdll.dll
   10.0.10586.0 shp  0x75540000  C:\windows\SYSTEM32\KERNEL32.DLL
 10.0.10586.162 shp  0x77850000  C:\windows\SYSTEM32\KERNELBASE.dll
  10.0.10586.63 shp  0x75AF0000  C:\windows\SYSTEM32\ADVAPI32.dll
...
   10.0.10586.0 shp  0x68540000  C:\Windows\SYSTEM32\VssTrace.DLL
   10.0.10586.0 shp  0x5C390000  C:\Windows\SYSTEM32\adsldpc.dll
 10.0.10586.122 shp  0x5DE60000  C:\Windows\SYSTEM32\taskschd.dll
   10.0.10586.0 shp  0x5E3F0000  C:\Windows\SYSTEM32\srmstormod.dll
   10.0.10586.0 shp  0x5DCA0000  C:\Windows\SYSTEM32\srmscan.dll
   10.0.10586.0 shp  0x5D2E0000  C:\Windows\SYSTEM32\msdrm.dll
   10.0.10586.0 shp  0x711E0000  C:\Windows\SYSTEM32\srm_ps.dll
   10.0.10586.0 shp  0x56680000  C:\windows\System32\OpcServices.dll
                     0x5D240000  C:\Program Files (x86)\Common Files\Microsoft
Shared\Office16\WXPNSE.DLL
 16.0.6701.1023 shp  0x77E80000  C:\Program Files (x86)\Microsoft Office\Root\
Office16\GROOVEEX.DLL
   10.0.10586.0 shp  0x693F0000  C:\windows\system32\dataexchange.dll
```

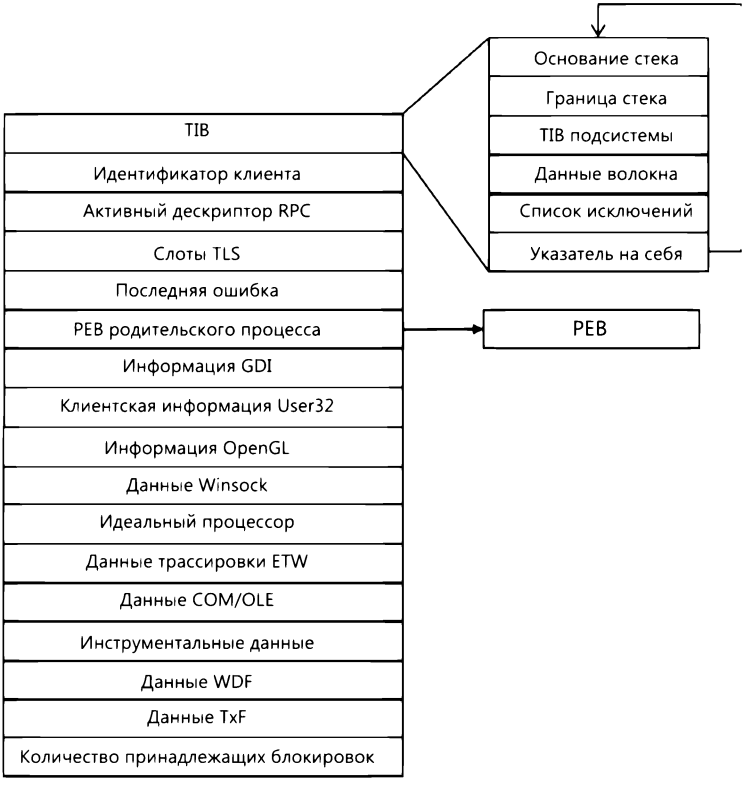


Рис. 4.3. Важные поля блока TEB

Блок TEB (рис. 4.3) — одна из структур данных, описанных в этом разделе, которые существуют в адресном пространстве процесса (в отличие от системного пространства). В TEB включен заголовок *TIB* (Thread Information Block), который в основном существует для совместимости с приложениями OS/2 и Win9x. Также он позволяет хранить информацию об исключениях и стеке в меньшей структуре при создании новых потоков с использованием исходного блока TIB.

В TEB хранится контекстная информация для загрузчика образов и различных DLL-библиотек Windows. Поскольку эти компоненты запускаются в пользовательском режиме, им нужна структура данных, в которую можно вести запись в этом режиме. Поэтому данная структура находится в адресном пространстве процесса, а не в системном пространстве, где запись в нее была бы возможна только из режима ядра. Адрес TEB можно определить с помощью команды отладчика ядра `!thread`.

ЭКСПЕРИМЕНТ: ИЗУЧЕНИЕ TEB

Дамп структуры TEB можно вывести командой `!teb` отладчика ядра или в отладчике пользовательского режима. Команда может использоваться без параметров для вывода текущего потока отладчика или с адресом TEB для вывода содержимого структуры произвольного потока. В случае отладчика ядра для выбора правильного контекста процесса текущий процесс должен быть задан перед выдачей команды с адресом TEB.

Чтобы просмотреть TEB в отладчике пользовательского режима, выполните следующие действия (о том, как просмотреть TEB с использованием отладчика ядра, рассказано в следующем эксперименте).

1. Откройте WinDbg.
2. Откройте меню File и выберите команду Run Executable.
3. Перейдите к файлу c:\windows\system32\Notepad.exe. Отладчик должен прерваться на начальной точке останова.
4. Введите команду `!teb`, чтобы просмотреть TEB единственного потока, существующего на данный момент (пример взят из 64-разрядной версии Windows):

```
0:000> !teb
TEB at 000000ef125c1000
    ExceptionList:          0000000000000000
    StackBase:              000000ef12290000
    StackLimit:             000000ef1227f000
    SubSystemTib:           0000000000000000
    FiberData:              0000000000001e00
    ArbitraryUserPointer:   0000000000000000
    Self:                   000000ef125c1000
    EnvironmentPointer:     0000000000000000
    ClientId:               00000000000021bc . 0000000000001b74
    RpcHandle:              0000000000000000
    Tls Storage:            00000266e572b600
    PEB Address:            000000ef125c0000
    LastErrorValue:         0
```

```
    LastStatusValue:        0
    Count Owned Locks:      0
    HardErrorMode:          0
```

5. Введите команду g или нажмите F5, чтобы продолжить выполнение Блокнота.
6. В Блокноте откройте меню Файл (File) и выберите команду Открыть (Open). Щелкните на кнопке Отмена (Cancel), чтобы закрыть диалоговое окно открытия файла.
7. Нажмите Ctrl+Break или откройте меню Debug и выберите команду Break, чтобы принудительно прервать процесс.
8. Введите команду ~ (тильда), чтобы показать все потоки в процессе. Результат должен выглядеть примерно так:

```
0:005> ~
   0 Id: 21bc.1b74 Suspend: 1 Teb: 000000ef'125c1000 Unfrozen
   1 Id: 21bc.640 Suspend: 1 Teb: 000000ef'125e3000 Unfrozen
   2 Id: 21bc.1a98 Suspend: 1 Teb: 000000ef'125e5000 Unfrozen
   3 Id: 21bc.860 Suspend: 1 Teb: 000000ef'125e7000 Unfrozen
   4 Id: 21bc.28e0 Suspend: 1 Teb: 000000ef'125c9000 Unfrozen
   5 Id: 21bc.23e0 Suspend: 1 Teb: 000000ef'12400000 Unfrozen
   6 Id: 21bc.244c Suspend: 1 Teb: 000000ef'125eb000 Unfrozen
   7 Id: 21bc.168c Suspend: 1 Teb: 000000ef'125ed000 Unfrozen
   8 Id: 21bc.1c90 Suspend: 1 Teb: 000000ef'125ef000 Unfrozen
   9 Id: 21bc.1558 Suspend: 1 Teb: 000000ef'125f1000 Unfrozen
  10 Id: 21bc.a64 Suspend: 1 Teb: 000000ef'125f3000 Unfrozen
  11 Id: 21bc.20c4 Suspend: 1 Teb: 000000ef'125f5000 Unfrozen
  12 Id: 21bc.1524 Suspend: 1 Teb: 000000ef'125f7000 Unfrozen
  13 Id: 21bc.1738 Suspend: 1 Teb: 000000ef'125f9000 Unfrozen
  14 Id: 21bc.f48 Suspend: 1 Teb: 000000ef'125fb000 Unfrozen
  15 Id: 21bc.17bc Suspend: 1 Teb: 000000ef'125fd000 Unfrozen
```

9. Для каждого потока выводится адрес его блока TEB. Чтобы просмотреть конкретный поток, укажите его адрес TEB в команде `!teb`. Пример для потока 9 из приведенного примера:

```
0:005> !teb 000000ef'125f1000
TEB at 000000ef125f1000
    ExceptionList:          0000000000000000
    StackBase:              000000ef13400000
    StackLimit:             000000ef133ef000
    SubSystemTib:           0000000000000000
    FiberData:              0000000000001e00
    ArbitraryUserPointer:   0000000000000000
    Self:                   000000ef125f1000
    EnvironmentPointer:     0000000000000000
    ClientId:               00000000000021bc . 0000000000001558
    RpcHandle:              0000000000000000
    Tls Storage:            00000266ea1af280
    PEB Address:            000000ef125c0000
    LastErrorValue:         0
    LastStatusValue:        c0000034
    Count Owned Locks:      0
    HardErrorMode:          0
```

10. Конечно, вы можете просмотреть саму структуру с адресом TEB (вывод сокращен для экономии места):

```
0:005> dt ntdll!_teb 000000ef'125f1000
   +0x000 NtTib           : _NT_TIB
   +0x038 EnvironmentPointer : (null)
   +0x040 ClientId        : _CLIENT_ID
   +0x050 ActiveRpcHandle : (null)
   +0x058 ThreadLocalStoragePointer : 0x00000266'ea1af280 Void
   +0x060 ProcessEnvironmentBlock : 0x000000ef'125c0000 _PEB
   +0x068 LastErrorValue  : 0
   +0x06c CountOfOwnedCriticalSections : 0
...
   +0x1808 LockCount        : 0
   +0x180c WowTebOffset     : 0n0
   +0x1810 ResourceRetValue : 0x00000266'ea2a5e50 Void
   +0x1818 ReservedForWdf   : (null)
   +0x1820 ReservedForCrt   : 0
   +0x1828 EffectiveContainerId : _GUID {00000000-0000-0000-0000-000000000000}
```

ЭКСПЕРИМЕНТ: ИЗУЧЕНИЕ TEB В ОТЛАДЧИКЕ ЯДРА

Чтобы просмотреть TEB в отладчике ядра, выполните следующие действия.

1. Найдите процесс, TEB потока которого представляет для вас интерес. Например, следующие команды ищут процессы explorer.exe и выводят список его потоков с базовой информацией (сокращенный вывод):

```
lkd> !process 0 2 explorer.exe
PROCESS ffffe0012bea7840
    SessionId: 2 Cid: 10d8 Peb: 00251000 ParentCid: 10bc
    DirBase: 76e12000 ObjectTable: ffffc000e1ca0c80 HandleCount: <Data Not
Accessible>
    Image: explorer.exe

        THREAD ffffe0012bf53080 Cid 10d8.10dc Teb: 0000000000252000
Win32Thread: ffffe0012c1532f0 WAIT: (WrUserRequest) UserMode Non-Alertable
            ffffe0012c257fe0 SynchronizationEvent

        THREAD ffffe0012a30f080 Cid 10d8.114c Teb: 0000000000266000
Win32Thread: ffffe0012c2e9a20 WAIT: (UserRequest) UserMode Alertable
            ffffe0012bab85d0 SynchronizationEvent

        THREAD ffffe0012c8bd080 Cid 10d8.1178 Teb: 000000000026c000
Win32Thread: ffffe0012a801310 WAIT: (UserRequest) UserMode Alertable
            ffffe0012bfd9250 NotificationEvent
            ffffe0012c9512f0 NotificationEvent
            ffffe0012c876b80 NotificationEvent
            ffffe0012c010fe0 NotificationEvent
            ffffe0012d0ba7e0 NotificationEvent
            ffffe0012cf9d1e0 NotificationEvent
...
```

```
        THREAD ffffe0012c8be080 Cid 10d8.1180 Teb: 0000000000270000
Win32Thread: 0000000000000000 WAIT: (UserRequest) UserMode Alertable
            fffff80156946440 NotificationEvent

        THREAD ffffe0012afd4040 Cid 10d8.1184 Teb: 0000000000272000
Win32Thread: ffffe0012c7c53a0 WAIT: (UserRequest) UserMode Non-Alertable
            ffffe0012a3dafe0 NotificationEvent
            ffffe0012c21ee70 Semaphore Limit 0xffff
            ffffe0012c8db6f0 SynchronizationEvent

        THREAD ffffe0012c88a080 Cid 10d8.1188 Teb: 0000000000274000
Win32Thread: 0000000000000000 WAIT: (UserRequest) UserMode Alertable
            ffffe0012afd4920 NotificationEvent
            ffffe0012c87b480 SynchronizationEvent
            ffffe0012c87b400 SynchronizationEvent
...
```

2. Если существует более одного процесса explorer.exe, выберите один произвольный поток для следующих действий.

3. Для каждого потока выводится адрес его блока TEB. Поскольку TEB находится в пользовательском пространстве, адрес имеет смысл только в контексте соответствующего процесса. Необходимо переключиться на процесс/поток, видимый отладчику. Выберите первый поток Проводника, потому что его стек ядра, скорее всего, находится в физической памяти. В противном случае вы получите ошибку.

```
lkd> .thread /p ffffe0012bf53080
Implicit thread is now ffffe001'2bf53080
Implicit process is now ffffe001'2bea7840
```

4. Команда переключает контекст на заданный поток (а следовательно, и процесс). Теперь вы можете ввести команду `!teb` с адресом TEB, указанным для этого процесса:

```
lkd> !teb 0000000000252000
TEB at 0000000000252000
    ExceptionList:        0000000000000000
    StackBase:            00000000000d0000
    StackLimit:           00000000000c2000
    SubSystemTib:         0000000000000000
    FiberData:            0000000000001e00
    ArbitraryUserPointer: 0000000000000000
    Self:                 0000000000252000
    EnvironmentPointer:   0000000000000000
    ClientId:             00000000000010d8 . 00000000000010dc
    RpcHandle:            0000000000000000
    Tls Storage:          0000000009f73f30
    PEB Address:          0000000000251000
    LastErrorValue:       0
    LastStatusValue:      c0150008
    Count Owned Locks :   0
    HardErrorMode:        0
```

Структура CSR_THREAD, показанная на рис. 4.4, аналогична структуре данных CSR_PROCESS, но применяется к потокам. Как вы, наверное, помните, она поддерживается каждым Csrss-процессом внутри сеанса и идентифицирует потоки подсистемы Windows, запущенные внутри этого процесса. В структуре CSR_THREAD хранится дескриптор, который Csrss содержит для потока, различные флаги и указатель для потока на структуру CSR_PROCESS. Там также хранится еще одна копия времени создания потока. Обратите внимание: потоки регистрируются в Csrss в тот момент, когда они отправляют Csrss свое первое сообщение — обычно из-за вызова API-функции, требующей уведомить Csrss о некоторой операции или условии.

Рис. 4.4. Поля структуры CSR_THREAD

ЭКСПЕРИМЕНТ: ИЗУЧЕНИЕ CSR_THREAD

Дамп структуры CSR_THREAD можно вывести с помощью команды отладчика пользовательского режима !dt, подключившись к процессу Csrss. Чтобы выполнить эту операцию, следуйте инструкциям, которые были приведены в главе 3 в разделе «Эксперимент: анализ CSR_PROCESS». Пример вывода в системе Windows 10 x64:

```
lkd> dt csrss!_csr_thread
   +0x000 CreateTime       : _LARGE_INTEGER
   +0x008 Link             : _LIST_ENTRY
   +0x018 HashLinks        : _LIST_ENTRY
   +0x028 ClientId         : _CLIENT_ID
   +0x038 Process          : Ptr64 _CSR_PROCESS
   +0x040 ThreadHandle     : Ptr64 Void
   +0x048 Flags            : Uint4B
   +0x04c ReferenceCount   : Int4B
   +0x050 ImpersonateCount : Uint4B
```

И наконец, структура W32THREAD, показанная на рис. 4.5, является аналогом структуры данных W32PROCESS, но применительно к процессам. Эта структура содержит главным образом информацию, полезную для подсистемы GDI (кисти и DC-атрибуты) и DirectX, а также для среды драйвера принтера пользовательского режима — User Mode Print Driver framework (UMPD), которая используется производителями для написания драйверов принтеров пользовательского режима.

И наконец, она содержит состояние визуализации для компоновки рабочего стола и сглаживания.

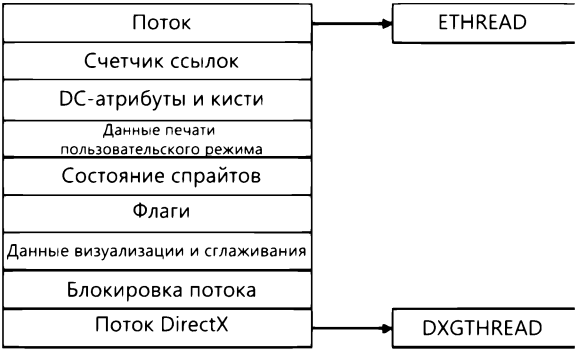


Рис. 4.5. Поля структуры потока Win32k

Рождение потока

Жизненный цикл потока начинается тогда, когда процесс создает новый поток (в контексте некоторого потока — например, потока, в котором выполняется главная функция). Запрос перекочевывает к исполняющей системе Windows, где диспетчер процесса выделяет память под объект потока и вызывает ядро для инициализации блока управления потока (`KTHREAD`). Как упоминалось ранее, различные функции создания потоков в итоге приводят в `CreateRemoteThreadEx`. Действия из следующего списка предпринимаются внутри этой функции в библиотеке Kernel32.dll для создания потока Windows.

1. Функция преобразует параметры Windows API во внутренние флаги и создает соответствующую структуру, описывающую параметры объекта (`OBJECT_ATTRIBUTES` — см. главу 8 части 2).
2. Функция создает список атрибутов с двумя записями: идентификатором клиента (client ID) и адресом TEB. (Подробнее о списках атрибутов см. в разделе «Порядок работы функции CreateProcess» главы 3.)
3. Она определяет, был ли поток создан в вызывающем процессе или в другом процессе, обозначенном переданным дескриптором. Если дескриптор равен псевдодескриптору, возвращаемому `GetCurrentProcess` (со значением −1), то это тот же процесс. Если дескриптор процесса отличается от него, он все равно может быть действительным дескриптором того же процесса, поэтому вызывается функция `NtQueryInformationProcess` (из Ntdll) для проверки этого случая.
4. Вызывается функция `NtCreateThreadEx` (в Ntdll) для перехода к исполняющей системе в режиме ядра, и продолжается выполнение внутри функции с тем же именем и аргументами.

5. `NtCreateThreadEx` (в исполняющей системе) создает и инициализирует контекст потока пользовательского режима (его структура зависит от конкретной архитектуры), а затем вызывает `PspCreateThread` для создания приостановленного объекта потока исполняющей системы. (См. раздел «Порядок работы функции CreateProcess» главы 3, описания этапов 3 и 5.) Затем функция возвращает управление, которое в конечном итоге возвращается в пользовательский режим в `CreateRemoteThreadEx`.
6. Функция `CreateThread` выделяет контекст активации для потока, используемый поддержкой смежной сборки (side-by-side assembly). Затем она запрашивает стек активации, чтобы посмотреть, не требует ли он активации, и в случае необходимости проводит активацию. Указатель на стек активации сохраняется в TEB нового потока.
7. За исключением того случая, когда вызывающий код создал поток с установленным флагом создания приостановленного потока — `CREATE_SUSPENDED`, теперь поток возобновляется и может быть запланирован для выполнения. Когда поток начинает работать, он перед вызовом кода по фактическому стартовому адресу, указанному пользователем, выполняет действия, описанные ранее в разделе «Этап 7. Выполнение инициализации процесса в контексте нового процесса».
8. Дескриптор потока и идентификатор потока возвращаются вызывающей стороне.

Изучение активности потока

Изучение активности потока важно, если вы пытаетесь определить, почему запущен или почему перестал реагировать процесс, в котором выполняются сразу несколько служб (например, Svchost.exe, Dllhost.exe или Lsass.exe).

Различные элементы состояния Windows-потоков могут выводиться несколькими служебными программами: программой WinDbg (в подключенном процессе режима пользователя и в режиме отладки ядра), программой Системный монитор (Performance Monitor) и программой Process Explorer (см. раздел «Планирование потоков»).

Для просмотра потоков в процессе с помощью Process Explorer выберите процесс и сделайте двойной щелчок на нем, чтобы открыть его диалоговое окно Свойства. Также можно щелкнуть правой кнопкой мыши на процессе и выбрать команду Properties. Перейдите на вкладку Threads. В этой вкладке показывается список потоков в процессе и четыре столбца с информацией. Для каждого потока показываются идентификатор (ID) потока, процент потребления ресурса центрального процессора (на основе настроенного интервала обновлений), количество циклов, приходящихся на поток, и стартовый адрес потока. Информацию можно отсортировать по любому из этих четырех столбцов.

Вновь созданные потоки выделены зеленым цветом, а существовавшие потоки — красным. (Чтобы настроить выделение продолжительности, откройте меню Options и выберите команду Difference Highlight Duration.) Это может пригодиться для выявления создания в процессе ненужных потоков. (Обычно потоки должны создаваться при запуске процесса, а не каждый раз, когда запрос обрабатывается внутри процесса.)

При выборе в списке каждого потока Process Explorer показывает его идентификатор, время запуска, состояние, счетчик времени центрального процессора, количество циклов, приходящихся на поток, количество переключений контекста, идеальный процессор и его группу, а также приоритет ввода/вывода, приоритет памяти, базовый и текущий (динамический) приоритет. Есть также кнопка Kill, с помощью которой можно завершить отдельный поток, но ею нужно пользоваться с большой осмотрительностью. Можно также воспользоваться кнопкой Suspend, с помощью которой сдерживается дальнейшее выполнение потока и тем самым предотвращается расход времени центрального процессора со стороны потока, вышедшего из-под контроля. Но это может привести к взаимным блокировкам, поэтому данной кнопкой нужно пользоваться с той же осмотрительностью, что и кнопкой Kill. Наконец, кнопка Permissions позволяет просмотреть дескриптор безопасности потока (подробнее о дескрипторах безопасности см. в главе 7).

В отличие от диспетчера задач и всех остальных средств отслеживания процессов и процессоров, Process Explorer использует счетчик тактовых импульсов, предназначенный для учета времени выполнения потока (см. далее в этой главе), а не интервальный таймер. Поэтому при использовании Process Explorer картина потребления ресурса центрального процессора будет существенно отличаться. Причина в том, что многие потоки запускаются на такой короткий отрезок времени, что они редко являются (если вообще являются) текущим запущенным потоком, когда происходит прерывание от интервального таймера. В результате большая часть их процессорного времени не учитывается, а это приводит к тому, что средства, основанные на использовании таймера, воспринимают использование центрального процессора равным 0 %. С другой стороны, общее количество тактовых циклов представляет фактическое количество циклов процессора, накопленное каждым потоком процесса. Оно не зависит от разрешения интервального таймера, поскольку счетчик поддерживается внутри системы процессором при каждом цикле и обновляется Windows при каждом входе в прерывание. (Финальное накопление выполняется перед переключением контекста.)

Стартовый адрес потока выводится в форме *модуль!функция*, где *модуль* — имя файла с расширением .exe или .dll. Имя функции основывается на доступе к файлам символических имен модуля (см. главу 1, врезка «Эксперимент: просмотр информации о процессах в Process Explorer»). Если вы не уверены в происхождении модуля, щелкните на кнопке Module. В Explorer откроется окно свойств файла того модуля, который содержит стартовый адрес потока (например, .exe или .dll).

ПРИМЕЧАНИЕ Для потоков, созданных Windows-функцией `CreateThread`, Process Explorer показывает функцию, переданную `CreateThread`, а не фактическую стартовую функцию потока. Причина в том, что все потоки Windows стартуют в общей функции-обертке запуска потока (`RtlUserThreadStart` в Ntdll.dll). Если Process Explorer показывал бы фактический стартовый адрес, оказалось бы, что большинство потоков процесса стартовало с одного и того же адреса, что вряд ли помогло бы разобраться, какой код выполнялся в потоке. Но если Process Explorer не может запросить адрес запуска, определенный пользователем (как в случае с защищенным процессом), он покажет функцию-обертку, и вы увидите, что все потоки стартуют в функции `RtlUserThreadStart`.

В выведенном стартовом адресе потока может быть недостаточно информации для точного указания на то, чем занят поток и какой компонент внутри процесса отвечает за ресурс центрального процессора, потребляемый потоком. Это особенно характерно для случая, когда стартовым адресом потока является общая функция запуска (например, если имя функции ничего не говорит о том, чем этот поток занимается). Возможно, исследование стека потока поможет получить нужную информацию. Для просмотра стека потока дважды щелкните на интересующем вас потоке (или выделите этот поток и щелкните на кнопке Stack). Process Explorer покажет стек потока (как пользовательского режима, так и режима ядра, если поток был запущен в режиме ядра).

ПРИМЕЧАНИЕ В то время как отладчики пользовательского режима (WinDbg, Ntsd и Cdb) позволяют подключиться к процессу и вывести для потока стек пользовательского режима, Process Explorer показывает как пользовательский стек, так и стек ядра после простого щелчка на кнопке. Изучить стеки пользовательского потока и потока режима ядра можно также с помощью WinDbg в режиме локальной отладки ядра, как показывают следующие два эксперимента.

При просмотре 32-разрядных процессов, выполняемых в 64-разрядных системах в виде процессов Wow64 (см. главу 8 части 2), Process Explorer выводит для потоков как 32-разрядный, так и 64-разрядный стек. Поскольку на момент реального (64-разрядного) системного вызова поток был переключен на 64-разрядный стек и контекст, простой просмотр 64-разрядного стека потока раскроет только половину картины — 64-разрядную часть процесса с кодом преобразования Wow64. Таким образом, при изучении процессов Wow64 следует обязательно принимать во внимание как 32-разрядные, так и 64-разрядные стеки.

ЭКСПЕРИМЕНТ: ПРОСМОТР СТЕКА ПОТОКА В ОТЛАДЧИКЕ ПОЛЬЗОВАТЕЛЬСКОГО РЕЖИМА

Выполните следующие действия, чтобы присоединить WinDbg к процессу и просмотреть информацию о потоке и его стеке.

1. Запустите notepad.exe и WinDbg.exe.
2. В WinDbg откройте меню File и выберите команду Attach to Process.

3. Найдите экземпляр notepad.exe и щелкните на кнопке OK. Отладчик прерывает выполнение Блокнота.

4. Выведите список существующих потоков в процессе командой ~. Для каждого потока выводится идентификатор отладчика, идентификатор клиента (*ProcessID.ThreadID*), счетчик приостановок (обычно равен 1 из-за точки останова), адрес TEB и признак замораживания командой отладчика.

```
0:005> ~
   0 Id: 612c.5f68 Suspend: 1 Teb: 00000022'41da2000 Unfrozen
   1 Id: 612c.5564 Suspend: 1 Teb: 00000022'41da4000 Unfrozen
   2 Id: 612c.4f88 Suspend: 1 Teb: 00000022'41da6000 Unfrozen
   3 Id: 612c.5608 Suspend: 1 Teb: 00000022'41da8000 Unfrozen
   4 Id: 612c.cf4 Suspend: 1 Teb: 00000022'41daa000 Unfrozen
.  5 Id: 612c.9f8 Suspend: 1 Teb: 00000022'41db0000 Unfrozen
```

5. Обратите внимание на точку перед потоком 5 в выводе. Это текущий поток отладчика. Введите команду k, чтобы просмотреть стек вызовов:

```
0:005> k
 # Child-SP          RetAddr           Call Site
00 00000022'421ff7e8 00007ff8'504d9031 ntdll!DbgBreakPoint
01 00000022'421ff7f0 00007ff8'501b8102 ntdll!DbgUiRemoteBreakin+0x51
02 00000022'421ff820 00007ff8'5046c5b4 KERNEL32!BaseThreadInitThunk+0x22
03 00000022'421ff850 00000000'00000000 ntdll!RtlUserThreadStart+0x34
```

6. Отладчик внедряет в процесс Блокнота поток, выдающий команду останова (`DbgBreakPoint`). Чтобы просмотреть стек вызовов другого потока, используйте команду ~nk, где n — номер потока в `WinDbg`. (Текущий поток отладчика при этом не изменяется.) Пример для потока 2:

```
0:005> ~2k
 # Child-SP          RetAddr               Call Site
00 00000022'41f7f9e8 00007ff8'5043b5e8 ntdll!ZwWaitForWorkViaWorkerFactory+0x14
01 00000022'41f7f9f0 00007ff8'501b8102 ntdll!TppWorkerThread+0x298
02 00000022'41f7fe00 00007ff8'5046c5b4 KERNEL32!BaseThreadInitThunk+0x22
03 00000022'41f7fe30 00000000'00000000 ntdll!RtlUserThreadStart+0x34
```

7. Чтобы переключить отладчик на другой поток, используйте команду ~ns (снова n — номер потока). Переключитесь на поток 0 и выведите его стек:

```
0:005> ~0s
USER32!ZwUserGetMessage+0x14:
00007ff8'502e21d4 c3              ret
0:000> k
 # Child-SP          RetAddr           Call Site
00 00000022'41e7f048 00007ff8'502d3075 USER32!ZwUserGetMessage+0x14
01 00000022'41e7f050 00007ff6'88273bb3 USER32!GetMessageW+0x25
02 00000022'41e7f080 00007ff6'882890b5 notepad!WinMain+0x27b
03 00000022'41e7f180 00007ff8'341229b8 notepad!__mainCRTStartup+0x1ad
04 00000022'41e7f9f0 00007ff8'5046c5b4 KERNEL32!BaseThreadInitThunk+0x22
05 00000022'41e7fa20 00000000'00000000 ntdll!RtlUserThreadStart+0x34
```

8. Обратите внимание: даже при том, что поток может находиться в режиме ядра, отладчик пользовательского режима показывает его последнюю функцию, которая все еще находится в пользовательском режиме (`ZwUserGetMessage` в приведенном выводе).

ЭКСПЕРИМЕНТ: ПРОСМОТР СТЕКА ПОТОКА В ЛОКАЛЬНОМ ОТЛАДЧИКЕ РЕЖИМА ЯДРА

В этом эксперименте вы используете локальный отладчик ядра для просмотра (как пользовательского режима, так и режима ядра). В эксперименте используется один из потоков Explorer, но вы можете опробовать его с другими потоками и процессами.

1. Выведите все процессы, в которых выполняется образ explorer.exe. (Обратите внимание: вы можете увидеть более одного экземпляра Explorer, если в настройках Explorer установлен флажок Launch Folder Windows в Separate Process; один процесс управляет рабочим столом и панелью задач, а другие управляют окнами Explorer.)

```
lkd> !process 0 0 explorer.exe
PROCESS ffffe00197398080
    SessionId: 1 Cid: 18a0 Peb: 00320000 ParentCid: 1840
    DirBase: 17c028000 ObjectTable: ffffc000bd4aa880 HandleCount: <Data
Not Accessible>
    Image: explorer.exe

PROCESS ffffe00196039080
    SessionId: 1 Cid: 1f30 Peb: 00290000 ParentCid: 0238
    DirBase: 24cc7b000 ObjectTable: ffffc000bbbef740 HandleCount: <Data
Not Accessible>
    Image: explorer.exe
```

Выберите один экземпляр и выведите сводку его потоков:

```
lkd> !process ffffe00196039080 2
PROCESS ffffe00196039080
    SessionId: 1 Cid: 1f30 Peb: 00290000 ParentCid: 0238
    DirBase: 24cc7b000 ObjectTable: ffffc000bbbef740 HandleCount: <Data
Not Accessible>
    Image: explorer.exe

        THREAD ffffe0019758f080 Cid 1f30.0718 Teb: 0000000000291000
Win32Thread: ffffe001972e3220 WAIT: (UserRequest) UserMode Non-Alertable
            ffffe00192c08150 SynchronizationEvent

        THREAD ffffe00198911080 Cid 1f30.1aac Teb: 00000000002a1000
Win32Thread: ffffe001926147e0 WAIT: (UserRequest) UserMode Non-Alertable
            ffffe00197d6e150 SynchronizationEvent
            ffffe001987bf9e0 SynchronizationEvent

        THREAD ffffe00199553080 Cid 1f30.1ad4 Teb: 00000000002b1000
Win32Thread: ffffe0019263c740 WAIT: (UserRequest) UserMode Non-Alertable
            ffffe0019ac6b150 NotificationEvent
            ffffe0019a7da5e0 SynchronizationEvent

        THREAD ffffe0019b6b2800 Cid 1f30.1758 Teb: 00000000002bd000
Win32Thread: 0000000000000000 WAIT: (Suspended) KernelMode Non-Alertable
SuspendCount 1
            ffffe0019b6b2ae0 NotificationEvent
...
```

2. Переключитесь на контекст первого потока в процессе (вы можете выбрать другие потоки):

```
lkd> .thread /p /r ffffe0019758f080
Implicit thread is now ffffe001'9758f080
Implicit process is now ffffe001'96039080
Loading User Symbols
.....................................
```

3. Теперь выведите подробную информацию о потоке и его стеке вызовов (адреса сокращены в приведенном выводе):

```
lkd> !thread ffffe0019758f080
THREAD ffffe0019758f080 Cid 1f30.0718 Teb: 0000000000291000 Win32Thread :
ffffe001972e3220 WAIT : (UserRequest)UserMode Non - Alertable
ffffe00192c08150 SynchronizationEvent
Not impersonating
DeviceMap                 ffffc000b77f1f30
Owning Process            ffffe00196039080       Image : explorer.exe
Attached Process          N / A               Image : N / A
Wait Start TickCount      17415276         Ticks : 146 (0:00 : 00 : 02.281)
Context Switch Count      2788             IdealProcessor : 4
UserTime                  00 : 00 : 00.031
KernelTime                00 : 00 : 00.000
*** WARNING : Unable to verify checksum for C : \windows\explorer.exe
Win32 Start Address explorer!wWinMainCRTStartup(0x00007ff7b80de4a0)
Stack Init ffffd0002727cc90 Current ffffd0002727bf80
Base ffffd0002727d000 Limit ffffd00027277000 Call 0000000000000000
Priority 8 BasePriority 8 PriorityDecrement 0 IoPriority 2 PagePriority 5

... Call Site
... nt!KiSwapContext + 0x76
... nt!KiSwapThread + 0x15a
... nt!KiCommitThreadWait + 0x149
... nt!KeWaitForSingleObject + 0x375
... nt!ObWaitForMultipleObjects + 0x2bd
... nt!NtWaitForMultipleObjects + 0xf6
... nt!KiSystemServiceCopyEnd + 0x13 (TrapFrame @ ffffd000'2727cb00)
... ntdll!ZwWaitForMultipleObjects + 0x14
... KERNELBASE!WaitForMultipleObjectsEx + 0xef
... USER32!RealMsgWaitForMultipleObjectsEx + 0xdb
... USER32!MsgWaitForMultipleObjectsEx + 0x152
... explorerframe!SHProcessMessagesUntilEventsEx + 0x8a
... explorerframe!SHProcessMessagesUntilEventEx + 0x22
... explorerframe!CExplorerHostCreator::RunHost + 0x6d
... explorer!wWinMain + 0xa04fd
... explorer!__wmainCRTStartup + 0x1d6
```

Ограничения, накладываемые на потоки защищенного процесса

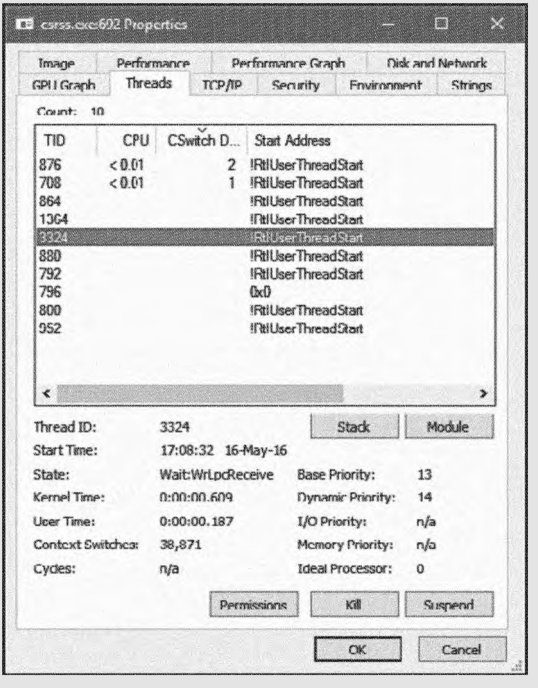
Как упоминалось в главе 3, защищенные процессы (классические защищенные, или PPL) имеют ряд ограничений, накладываемых на права доступа, которые касаются даже пользователей с самыми высокими привилегиями в системе. Эти ограничения

также применяются к потокам внутри такого процесса. Тем самым гарантируется, что код, запущенный внутри защищенного процесса, не будет взломан или каким-то иным образом затронут посредством стандартных Windows-функций, которые запрашивают права доступа, не предоставляемые потокам защищенных процессов. Фактически единственными предоставляемыми правами являются право на приостановку и возобновление выполнения потока — `THREAD_SUSPEND_RESUME` и право на получение по запросу установленной для потока ограниченной информации — `THREAD_SET/QUERY_LIMITED_INFORMATION`.

ЭКСПЕРИМЕНТ: ПРОСМОТР ИНФОРМАЦИИ О ПОТОКЕ ЗАЩИЩЕННОГО ПРОЦЕССА

В этом эксперименте вы просмотрите информацию защищенных потоков. Выполните следующие действия.

1. Найдите в списке процессов любой защищенный процесс или PPL, такой как Audiodg.exe или Csrss.exe.
2. Откройте диалоговое окно свойства и перейдите на вкладку Threads.



3. Process Explorer не выводит стартовый адрес Win32. Вместо этого он выводит адрес стартовой обертки стандартного потока из Ntdll.dll. Щелкнув на кнопке Stack, вы получите ошибку, потому что Process Explorer должен прочитать виртуальную память в защищенном процессе, чего он сделать не может.

4. Обратите внимание: хотя в данных выводятся базовые и динамические приоритеты, приоритеты ввода/вывода и памяти в списке отсутствуют — другой пример ограниченного права доступа `THREAD_QUERY_LIMITED_INFORMATION` по сравнению с полным правом запроса информации (`THREAD_QUERY_INFORMATION`).
5. Попробуйте уничтожить поток в защищенном процессе. Обратите внимание на другую ошибку отказа в доступе из-за отсутствия права `THREAD_TERMINATE`.

Планирование потоков

В этом разделе рассматриваются политика и алгоритмы Windows-планирования. В первом подразделе дается краткое описание порядка планирования работы в Windows и приводятся определения основных понятий. Затем дается описание уровней приоритета как с точки зрения Windows API, так и с точки зрения ядра Windows. После обзора соответствующих утилит и инструментальных средств Windows, имеющих отношение к планированию, будут подробно представлены структуры данных и алгоритмы, составляющие систему планирования Windows, включая описание общих сценариев планирования и порядка выбора потока, а также выбора процессора.

Обзор организации планирования в Windows

В Windows реализуется приоритетная, вытесняющая система планирования, при которой всегда выполняется хотя бы один работоспособный (готовый) поток с самым высоким приоритетом, с той оговоркой, что конкретные, имеющие высокий приоритет и готовые к запуску потоки могут быть ограничены процессами, на которых им разрешено или предпочтительнее всего работать. Это явление называется *сходством процессора*. Это сходство определяется на основе заданной группы процессоров, в которую включается до 64 процессоров. По умолчанию потоки могут запускаться только на доступном процессоре из связанной с процессом группы (для обеспечения совместимости с устаревшими версиями Windows, поддерживающими только 64 процессора), но разработчики могут изменить сходство процессора, воспользовавшись соответствующими API-функциями или настроив маску сходства в заголовке образа, а пользователи могут воспользоваться инструментальными средствами для изменения сходства в ходе выполнения программы или при создании процесса. Хотя несколько потоков в процессе могут быть связаны с разными группами, сам по себе поток запускается только на процессорах, доступных внутри связанной с ним группы. Кроме того, разработчики могут решить создать знающие о группе приложения, использующие расширенные API-функции планирования, для связи логических процессоров из разных групп со сходством их потоков. Это связывает процесс с несколькими группами, что теоретически позволяет ему запускать свои потоки на любом доступном процессоре машины.

После того как поток был выбран для запуска, он запускается на время, называемое *квантом*. Квант — это промежуток времени, в течение которого потоку разрешено работать, пока не настанет очередь запуститься другому потоку с тем же уровнем приоритета. Значение кванта может варьироваться от системы к системе и от процессора к процессору по любой из трех причин:

- настроек конфигурации системы (длинные или короткие кванты, переменные или фиксированные кванты и разнос приоритетов);
- состояния процесса, т. е. является ли он процессом первого плана или фоновым процессом;
- использования объекта задания для изменения кванта.

Более подробно тема кванта будет раскрыта далее в разделе «Кванты времени».

Но поток может и не израсходовать свой квант времени, поскольку в Windows реализуется вытесняющий планировщик: когда готов к запуску другой поток с более высоким приоритетом, текущий выполняемый поток может быть вытеснен еще до окончания его кванта времени. Фактически поток может быть выбран на запуск следующим и вытеснен еще даже до начала своего кванта времени!

Код планировщика Windows реализован в ядре. Но единого модуля или процедуры под названием «планировщик» не существует, код разбросан по ядру, где происходят события, связанные с планированием. Процедуры, выполняющие эти обязанности, обобщенно называются *диспетчером* ядра. Диспетчеризации потоков могут потребовать следующие события.

- Поток становится готовым к выполнению — например, поток был только что создан или только что освобожден от состояния ожидания.
- Поток выходит из состояния выполнения из-за окончания его кванта времени, его работа завершается, ему предоставляется возможность выполнения или он входит в состояние ожидания.
- Изменяется приоритет потока либо по причине вызова системной службы, либо по причине того, что Windows сама изменяет значение приоритета.
- Изменяется сходство процессора потока, и он больше уже не может быть запущен на том процессе, на котором выполнялся.

По совокупности всех этих обстоятельств Windows должна определить, какой из потоков должен быть запущен следующим на логическом процессоре, на котором работал поток, если это приемлемо, или на каком логическом процессоре не должен быть запущен поток. После того как логический процессор выбрал новый поток для выполнения, он выполняет переключение контекста на этот поток. *Переключение контекста* представляет собой процедуру сохранения изменяющегося состояния процессора, связанного с запущенным потоком, загрузки изменяемого состояния другого потока и запуска выполнения нового потока.

Как уже отмечалось, Windows осуществляет планировку потоков на уровне потоков. Такой подход имеет смысл, если учесть, что процессы не запускаются, а только

предоставляют ресурсы и контекст, в котором запускаются их потоки. Поскольку решения по планированию принимаются исключительно на основе потоков, не учитывается, какому процессу какой поток принадлежит. Например, если процесс А имеет 10 работоспособных потоков, процесс Б имеет 2 работоспособных потока и все 12 потоков имеют один и тот же приоритет, каждый поток теоретически получит одну двенадцатую времени центрального процессора, Windows не выделит по 50 % процессорного времени процессу А и процессу Б.

Уровни приоритета

Чтобы разобраться в алгоритмах планирования потоков, сначала нужно понять смысл уровней приоритета в системе Windows. Как показано на рис. 4.6, во внутренней реализации Windows использует 32 уровня приоритета, от 0 до 31 (31 — наивысший). Эти значения разбиваются на категории следующим образом:

- шестнадцать уровней реального времени (от 16 до 31);
- шестнадцать изменяющихся уровней (от 0 до 15), из которых уровень 0 зарезервирован для потока обнуления страниц (см. главу 5).

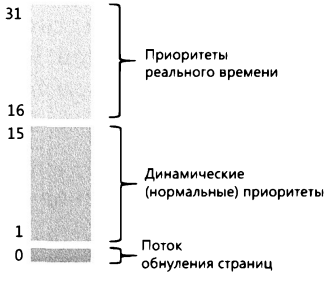


Рис. 4.6. Уровни приоритета потоков

Уровни приоритета потоков назначаются с двух позиций: от Windows API и от ядра Windows. Сначала Windows API систематизирует процессы по классу приоритета, который им присваивается при создании (номера представляют внутренний индекс `PROCESS_PRIORITY_CLASS`, распознаваемый ядром):

- реального времени — Real-time (4);
- высокий — High (3);
- выше обычного — Above Normal (6);
- обычный — Normal (2);
- ниже обычного — Below Normal (5);
- уровень простоя — Idle (1).

API-функция `SetPriorityClass` позволяет изменять класс приоритета процесса до одного из этих уровней.

Затем назначается относительный приоритет отдельных потоков внутри этих процессов. Здесь числа представляют приращение, применяемое к базовому приоритету процесса:

- критичный по времени — Time-critical (15);
- наивысший — Highest (2);
- выше обычного — Above-normal (1);
- обычный — Normal (0);
- ниже обычного — Below-normal (–1);
- самый низший — Lowest (–2);
- уровень простоя — Idle (–15).

Уровень, критичный по времени, и уровень простоя (+15 и –15) называются *уровнями насыщения* и представляют конкретные применяемые уровни вместо смещений. Эти значения могут передаваться API-функции `SetThreadPriority` для изменения относительного приоритета потоков.

Таким образом, в Windows API каждый поток имеет базовый приоритет, являющийся функцией класса приоритета процесса и его относительного приоритета процесса. В ядре класс приоритета процесса преобразуется в базовый приоритет путем использования процедуры `PspPriorityTable` и приведенных ранее индексов `PROCESS_PRIORITY_CLASS`, устанавливающих приоритеты 4, 8, 13, 24, 6 и 10 соответственно. (Это фиксированное отображение, которое не может быть изменено.) Затем применяется относительный приоритет потока в качестве смещения для этого базового приоритета. Например, `Highest`-поток получит базовый приоритет потока на два уровня выше, чем базовый приоритет его процесса.

Это соответствие между Windows-приоритетами и внутренними номерными приоритетами Windows показаны в графическом виде на рис. 4.7, и в текстовом виде в табл. 4.1.

Таблица 4.1. Соответствие между приоритетами ядра Windows и приоритетами Windows API

Класс приоритета/относительный приоритет	Real-time	High	Above-Normal	Normal	Below-Normal	Idle
Time Critical (+ насыщение)	31	15	15	15	15	15
Highest (+2)	26	15	12	10	8	6
Above Normal (+1)	25	14	11	9	7	5
Normal (0)	24	13	10	8	6	4
Below Normal (-1)	23	12	9	7	5	3
Lowest (-2)	22	11	8	6	4	2
Idle (–Saturation)	16	1	1	1	1	1

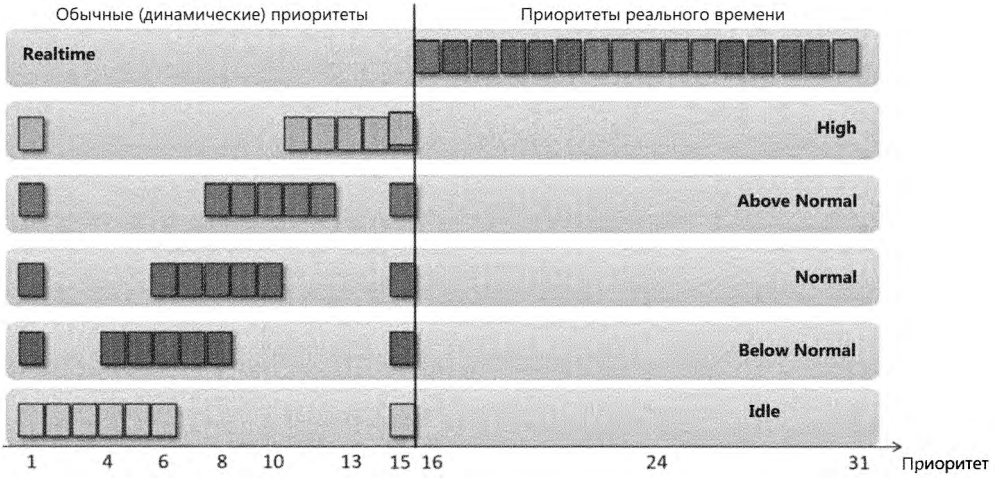


Рис. 4.7. Графическое представление приоритетов потоков с позиций Windows API

Следует заметить, что относительные приоритеты потоков Time-Critical и Idle сохраняют свои значения независимо от класса приоритета процесса (если только это не Real-time). Причина в том, что Windows API запрашивает насыщение (saturation) приоритета от ядра путем передачи 16 или –16 как относительного запрашиваемого приоритета. Эти значения вычисляются по следующей формуле (значение `HIGH_PRIORITY` равно 31):

```
Для Time-Critical: ((HIGH_PRIORITY+1) / 2

Для Idle: -((HIGH_PRIORITY+1) / 2
```

Эти значения распознаются ядром как запрос на насыщение, и в структуре `KTHREAD` устанавливается поле `Saturation`. Это приводит к тому, что при положительном насыщении поток получает наивысший возможный приоритет внутри его класса приоритета (динамического или реального времени), а при отрицательном насыщении — самый низкий из возможных приоритетов. Кроме того, последующие запросы на изменение базового приоритета процесса уже не будут влиять на базовый приоритет этих потоков, потому что насыщенные потоки пропускаются в коде обработки.

Как видно из табл. 4.1, с позиций Windows API потоки имеют семь возможных назначаемых приоритетов (шесть уровней для класса High). Класс приоритета Real-Time позволяет задавать все уровни приоритета от 16 до 31 (как видно из рис. 4.7). Класс приоритета Real-Time фактически позволяет задавать все уровни приоритета от 16 до 31 (см. рис. 4.7). Значения, для которых нет стандартных констант из таблицы, могут задаваться значениями –7, –6, –5, –4, –3, 3, 4, 5 и 6 аргумента `SetThreadPriority`. (Подробнее см. в разделе «Приоритеты реального времени».)

Каким бы образом ни формировался приоритет потока с использованием Windows API (комбинация класса приоритета процесса и относительного приоритета),

с точки зрения планировщика важен только конечный результат. Например, приоритет уровня 10 может быть получен двумя способами: классом приоритета процесса Normal (8) с относительным приоритетом потока Highest (+2) или классом приоритета Above-Normal (10) с относительным приоритетом потока Normal (0). С точки зрения планировщика эти комбинации ведут к одному значению (10), поэтому такие потоки идентичны в отношении своих приоритетов.

В то время как у процесса имеется только одно базовое значение приоритета, у каждого потока имеется два значения приоритета: текущее (динамическое) и базовое. Решения по планированию принимаются исходя из текущего приоритета. Как поясняется в следующем разделе, посвященном повышениям приоритета, система при определенных обстоятельствах на короткие периоды времени повышает приоритет потоков в динамическом диапазоне (от 1 до 15). Windows никогда не регулирует приоритет потоков в диапазоне реального времени (от 16 до 31), поэтому они всегда имеют один и тот же базовый и текущий приоритет.

Исходный базовый приоритет потока наследуется от базового приоритета процесса. Процесс по умолчанию наследует свой базовый приоритет у того процесса, который его создал. Это поведение может быть заменено другим в функции `CreateProcess` или путем использования команды `start` в окне командной строки. Приоритет процесса может быть также изменен после создания процесса путем использования функции `SetPriorityClass` или различных инструментальных средств, предлагающих такую функцию, например диспетчера задач и Process Explorer (щелчок правой кнопкой мыши на имени процесса и выбор нового класса приоритета). Например, можно снизить приоритет процесса, который интенсивно использует центральный процессор, чтобы он не мешал обычным действиям системы. Изменения приоритета процесса изменяют приоритеты потоков, повышая их или снижая, но их относительные установки остаются прежними.

Как правило, пользовательские приложения и службы запускаются с обычным базовым приоритетом (Normal), поэтому их исходный поток чаще всего выполняется с уровнем приоритета 8. Но некоторые системные процессы Windows (например, диспетчер сеансов, диспетчер управления службами и процесс аутентификации локальной безопасности) имеют немного более высокий базовый приоритет, чем тот, который используется по умолчанию для класса Normal (8). Более высокое значение по умолчанию гарантирует, что все потоки в этих процессах будут запускаться с более высоким приоритетом, превышающим значение по умолчанию, равное 8.

Приоритеты реального времени

Повысить или понизить приоритет потока в динамическом диапазоне можно в любом приложении. Однако для этого нужна привилегия повышения приоритета планирования (`SeIncreaseBasePriorityPrivilege`) для входа в диапазон реального времени. Следует иметь в виду, что многие важные системные потоки Windows, выполняемые в режиме ядра, работают в диапазоне приоритетов реального времени, поэтому если потоки тратят слишком много времени, работая в этом диапазоне,

они могут заблокировать критические системные функции (например, диспетчер памяти, диспетчер кэша или какие-нибудь драйверы устройств).

Если вы используете стандартный Windows API, при входе процесса в диапазон реального времени все его потоки (даже потоки Idle) должны выполняться на одном из уровней приоритета реального времени. Таким образом, становится невозможным смешивать потоки реального времени и динамические потоки с использованием стандартных интерфейсов. Причина в том, что API-функция `SetThreadPriority` вызывает низкоуровневую API-функцию `NtSetInformationThread` с информационным классом `ThreadBasePriority`, что позволяет приоритетам оставаться только в том же диапазоне. Кроме того, этот информационный класс позволяет приоритету изменяться только в допустимом диапазоне от –2 до 2 (или Time-Critical/Idle), если только запрос не пришел от CSRSS или от процесса реального времени. Иными словами, это означает, что процесс реального времени имеет возможность подбирать приоритеты потоков в диапазоне от 16 до 31, даже при том, что относительные приоритеты потоков Windows API вроде бы ограничивают его выбор на основе приведенной ранее таблицы.

Как упоминалось ранее, вызов `SetThreadPriority` с одним из специальных значений приводит к вызову функции `NtSetInformationThread` с информационным классом `ThreadActualBasePriority`, что позволяет напрямую задать базовый приоритет ядра, в том числе в динамическом диапазоне, для процессов реального времени.

ПРИМЕЧАНИЕ Термин «*реальное время*» не означает, что Windows является ОС реального времени в общепринятом понимании. Причина в том, что Windows не предоставляет «истинные» средства операционной системы реального времени — например, гарантированные задержки прерывания или способ получения потоками гарантированного времени выполнения. В данном случае «*реальное время*» означает только «выше, чем у других».

Использование инструментальных средств для работы с уровнями приоритета

Для просмотра и изменения базового приоритета процесса можно воспользоваться диспетчером задач или Process Explorer. Process Explorer также позволяет уничтожать отдельные потоки в процессе (конечно, это нужно делать с большой осторожностью).

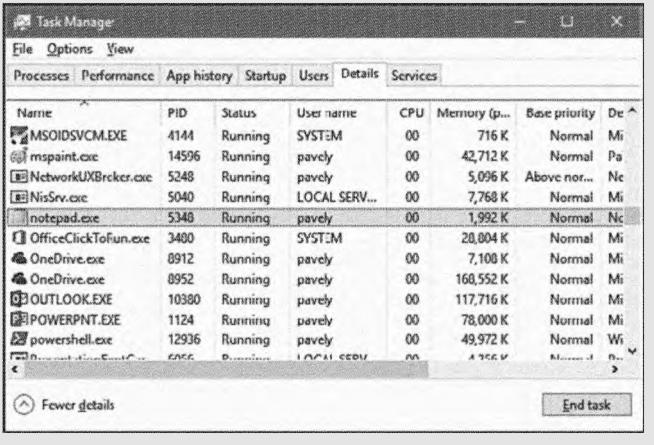
Приоритеты отдельных потоков можно просмотреть с помощью таких средств, как Системный монитор, Process Explorer и WinDbg. Хотя от повышения или понижения приоритета процесса и может быть какая-то польза, обычно нет смысла настраивать приоритеты отдельных потоков внутри процесса, потому что только тот, кто полностью разбирается в логике программы (другими словами, разработчик), поймет относительную важность потоков внутри процесса.

Единственный способ определить для процесса стартового класса приоритет — использовать команду `start` в окне командной строки Windows. Если нужно, чтобы программа каждый раз стартовала с определенным приоритетом, можно определить ярлык, используемый для команды `start`, начиная его командную строку с команды `cmd/c`. Эта команда откроет окно командной строки, выполнит команду и закроет окно командной строки. Например, чтобы запустить Блокнот (Notepad) с приоритетом процесса `Idle`, ярлык должен включать команду `cmd/c start/low Notepad.exe`.

ЭКСПЕРИМЕНТ: ИЗУЧЕНИЕ И ОПРЕДЕЛЕНИЕ ПРИОРИТЕТОВ ПРОЦЕССА И ПОТОКОВ

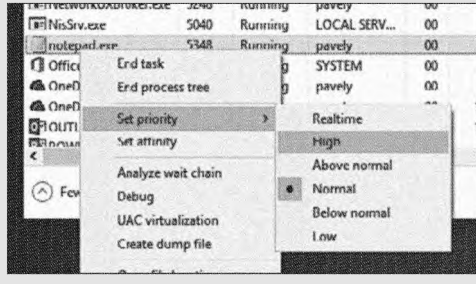
Выполните следующие действия.

1. Запустите Блокнот обычным способом — например, командой `Notepad` в окне командной строки.
2. Откройте диспетчер задач и перейдите на вкладку Подробности (Details).
3. Добавьте столбец с именем Базовый приоритет (Base Priority). Это имя используется диспетчером задач для класса приоритета.
4. Найдите Блокнот в списке. Он выглядит примерно так:

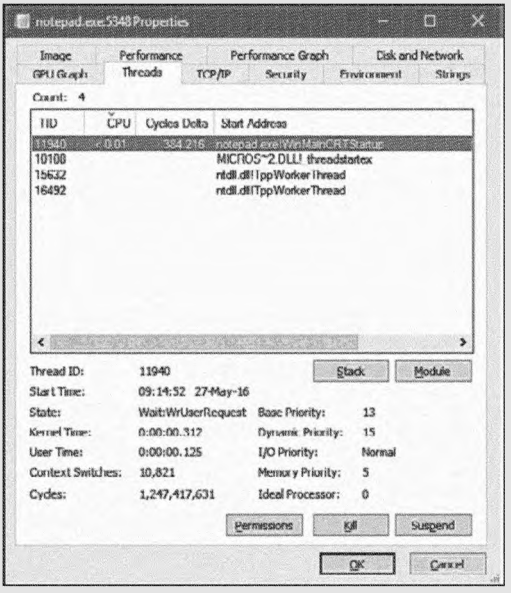


5. Обратите внимание: процесс Notepad.exe выполняется с классом приоритета Normal (8), а в диспетчере задач класс приоритета Idle отображается как Low.
6. Запустите Process Explorer.
7. Дважды щелкните на процессе Notepad.exe, чтобы появилось окно свойств процесса. Щелкните на вкладке Threads.
8. Выберите первый поток (если их несколько). Результат выглядит примерно так:

9. Обратите внимание на приоритеты потока. Его базовый приоритет равен 8, но текущий (динамический) приоритет равен 10. (Причина приращения приоритета рассматривается в разделе «Повышение приоритета».)
10. При желании вы можете приостановить и уничтожить поток. (Конечно, при выполнении обеих операций необходима осторожность.)
11. В диспетчере задач щелкните правой кнопкой мыши на процессе Notepad, выберите команду Приоритет (Set Priority) и задайте значение Высокий (High):

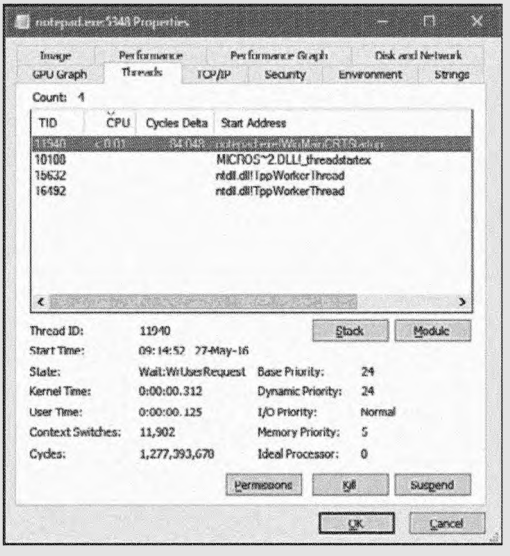


12. Подтвердите выполнение операции в диалоговом окне и вернитесь к Process Explorer. Обратите внимание: приоритет потока поднялся до нового базового значения High (13). Динамический приоритет изменился аналогичным образом:



13. В диспетчере задач выберите класс приоритета Реального времени (Realtime). (Для выполнения этой операции необходимо иметь права администратора на машине.) Это изменение также можно внести в Process Explorer.

14. В Process Manager обратите внимание на то, что базовый и динамический приоритеты потока теперь равны 24. Напомним, что ядро никогда не применяет приращения приоритетов для потоков в диапазоне приоритетов реального времени.



ДИСПЕТЧЕР СИСТЕМНЫХ РЕСУРСОВ WINDOWS

В Windows Server 2012 R2 Standard Edition и последующих выпусках включен дополнительно устанавливаемый компонент, который называется диспетчером системных ресурсов Windows — *Windows System Resource Manager* (WSRM). Он разрешает администратору настраивать политики, определяющие степень использования процессора, настройки сходства и лимиты памяти (как физической, так и виртуальной) для процессов. Кроме того, WSRM может генерировать отчеты об использовании ресурсов, которые могут использоваться для учета и проверки соглашений об уровне обслуживания пользователей.

Политики могут применяться к определенным приложениям (путем поиска соответствия имени образа с определенными аргументами командной строки или без них), пользователям или группам. Эти политики могут планироваться для применения в определенные периоды времени или могут быть включены постоянно.

После установки политики распределения ресурсов для управления определенными процессами служба WSRM отслеживает потребление ресурсов центрального процессора со стороны управляемых процессов и настраивает базовые приоритеты процесса, когда такие процессы не соблюдают целевых требований к затратам процессорного времени.

При накладывании ограничений на физическую память используется функция `SetProcessWorkingSetSizeEx`, с помощью которой устанавливается жестко заданный максимум рабочего набора. Ограничение виртуальной памяти реализовано путем проверки службой закрытой виртуальной памяти, потребляемой процессами (за описанием этих ограничений обращайтесь к главе 5). Если эти лимиты превышены, служба WSRM может быть настроена либо на уничтожение процессов, либо на внесение записи в журнал событий. Это поведение может использоваться для определения процессов с утечками памяти, пока они не растратят всю доступную выделяемую память системы. Следует заметить, что лимиты памяти WSRM не применяются к памяти AWE (Address Windowing Extensions), памяти больших страниц или памяти ядра (невыгружаемого или выгружаемого пула). (За описаниями всех этих терминов обращайтесь к главе 5.)

Состояния потоков

Перед тем как приступить к изучению алгоритмов планирования потоков, нужно разобраться с различными состояниями выполнения, в которых может находиться поток.

- **Готов** (Ready). Поток находится в состоянии готовности, ожидая выполнения (или готовности к возвращению в память после завершения ожидания). При поиске потока для выполнения диспетчер рассматривает только пул потоков, находящихся в состоянии готовности.

- **Готовность с отложенным выполнением** (Deferred ready). Это состояние используется для потоков, выбранных для выполнения на конкретном процессоре, но еще не запущенных на нем. Это состояние существует для того, чтобы ядро могло свести к минимуму период удержания блокировки базы данных планирования на уровне каждого процессора.
- **В повышенной готовности** (Standby). Поток в состоянии повышенной готовности был выбран для запуска следующим на конкретном процессоре. Как только сложатся соответствующие условия, диспетчер выполнит контекстное переключение на этот поток. Для каждого процессора в системе в состоянии повышенной готовности может быть только один поток. Следует заметить, что поток может быть вытеснен из состояния повышенной готовности даже до начала его выполнения (если, к примеру, до того, как начнется выполнение потока, находящегося в повышенной готовности, станет готов к работе поток с более высоким приоритетом).
- **Выполнение** (Running). Как только диспетчер выполняет переключение контекста на поток, тот входит в состояние выполнения. Его выполнение продолжается до тех пор, пока не истечет его квант времени (и не будет готов другой поток с тем же приоритетом), он не будет вытеснен потоком с более высоким приоритетом, он не завершит свою работу, он не уступит выполнение или он самостоятельно не перейдет в состояние ожидания.
- **Ожидание** (Waiting). Поток может войти в состояние ожидания в нескольких случаях: поток может самостоятельно ожидать объекта для синхронизации своего выполнения, операционная система может ждать от имени потока (например, для разрешения страничного ввода/вывода) или подсистема среды может приказать потоку приостановиться. Когда ожидание потока заканчивается, то в зависимости от приоритета поток либо немедленно начинает выполняться, либо возвращается в состояние готовности.
- **Переходное состояние** (Transition). Поток входит в переходное состояние, если он готов к выполнению, но его стек ядра выгружен из памяти. Как только его стек ядра вернется в память, поток войдет в состояние готовности. (Стеки потоков рассматриваются в главе 5.)
- **Завершение** (Terminated). Когда поток заканчивает выполнение, он входит в состояние завершения. Как только поток будет завершен, объект потока исполняющей системы (структура данных с описанием потока в невыгружаемом пуле) может быть освобожден или не освобожден (политику, определяющую время удаления объекта, устанавливает диспетчер объектов). Например, объект остается, если у потока остались открытые дескрипторы. Поток также может перейти в состояние завершения из других состояний, если он будет явно уничтожен другим потоком — скажем, вызовом API-функции `TerminateThread`.
- **Инициализация** (Initialized). Это состояние используется внутри системы при создании потока.

На рис. 4.8 изображена диаграмма переходов между состояниями потока. (Числа являются внутренними кодами состояний, а для их просмотра можно воспользоваться такой программой, как Системный монитор.) Состояния Ready и Deferred Ready представлены вместе. Это отражает тот факт, что состояние Deferred Ready действует в качестве временного заполнителя для процедур планирования (это также относится к состоянию Standby). Эти состояния почти всегда имеют весьма небольшую продолжительность, а потоки в этих состояниях всегда быстро переходят в состояния Ready, Running или Waiting.

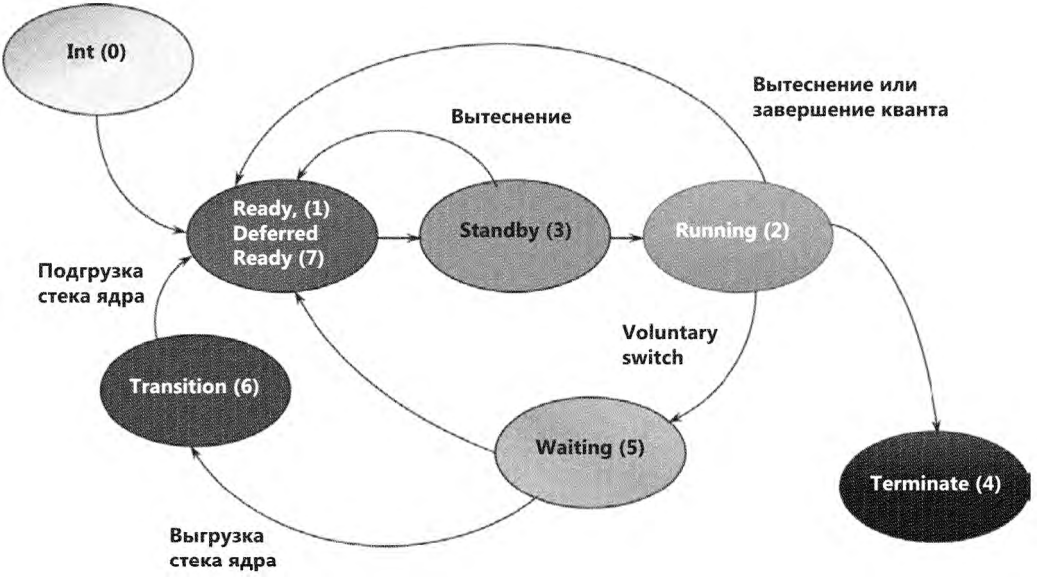
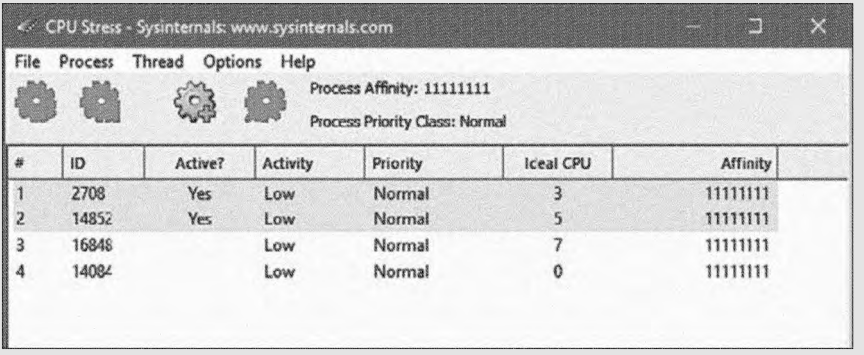


Рис. 4.8. Состояния потока и переходы между ними

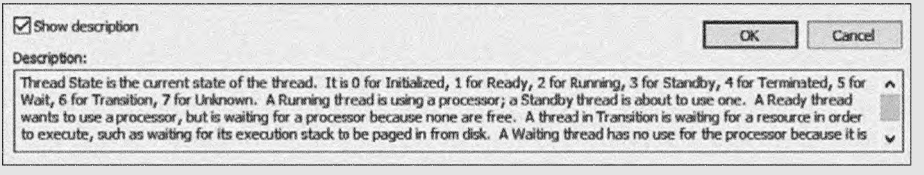
ЭКСПЕРИМЕНТ: ИЗМЕНЕНИЕ СОСТОЯНИЙ ПЛАНИРОВАНИЯ ПОТОКА

Для наблюдения за сменой состояний планирования потока можно воспользоваться Windows-программой Системный монитор. Эта программа может пригодиться при отладке многопоточных приложений, когда вы не уверены, в каком приложении находятся потоки, выполняемые в процессе. Чтобы понаблюдать за изменением состояния потоков в программе Системный монитор, выполните следующие действия.

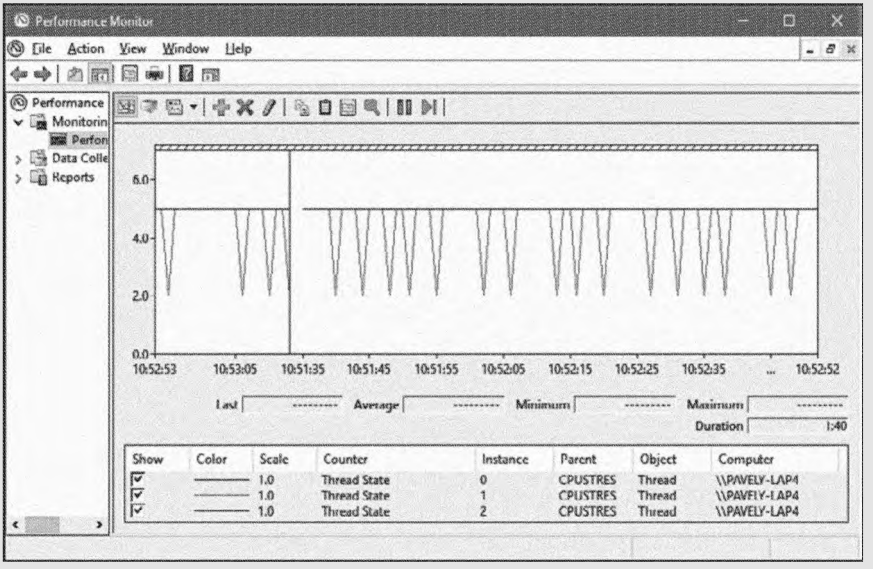
1. Загрузите программу CPU Stress.
2. Запустите программу CPUSTRES.EXE. Активным должен быть поток 1.
3. Активируйте поток 2: выберите его в списке и щелкните на кнопке Activate, или же щелкните на нем правой кнопкой мыши и выберите в контекстном меню команду Activate. Результат выглядит примерно так:



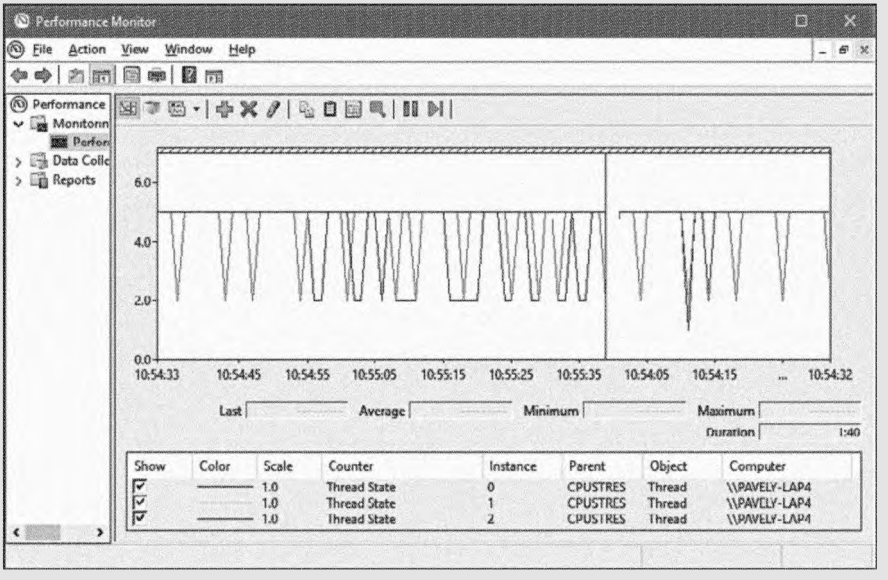
4. Щелкните на кнопке Пуск (Start) и введите команду `perfmon`, чтобы запустить Системный монитор.
5. При необходимости выберите режим диаграммы. Затем удалите существующий счетчик ЦП (CPU.)
6. Щелкните правой кнопкой мыши на графике и выберите команду Свойства (Properties).
7. Перейдите на вкладку График (Graph) и измените значение поля Диапазон значений вертикальной шкалы Максимум (Chart Vertical Scale Maximum) на 7. (Как показано на рис. 4.8, различные состояния обозначаются числами от 0 до 7.) Щелкните на кнопке OK.
8. Щелкните на кнопке Добавить (Add) на панели инструментов. На экране появится диалоговое окно Добавить счетчики (Add Counters).
9. Выберите объект производительности Поток (Thread), а затем выберите счетчик Состояние потока (Thread State).
10. Установите флажок Отображать описание (Show Description), чтобы видеть определения значений.



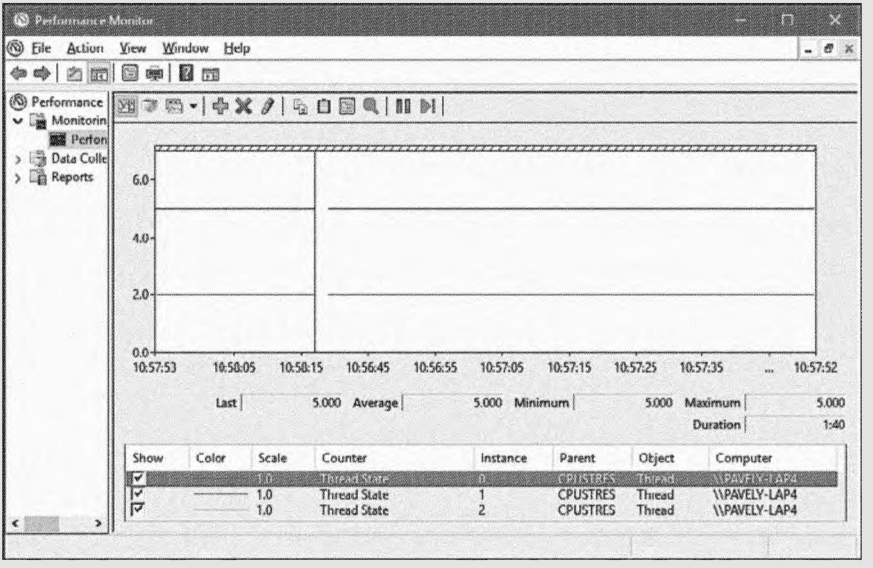
11. В поле Экземпляры выбранного объекта (Instances) выберите значение <все вхождения> (<All Instances>). Введите команду `cpustres` и щелкните на кнопке Поиск (Search).
12. Выберите первые три потока `cpustres` (`cpustres/0`, `cpustres/1` и `cpustres/2`) и щелкните на кнопке Добавить>> (Add>>). Щелкните на кнопке OK. Поток 0 должен находиться в состоянии 5 (Waiting), потому что это поток GUI, ожидающий пользовательского ввода. Потоки 1 и 2 должны переключаться между состояниями 2 и 5 (Running и Waiting). (Поток 1 может скрывать поток 2, так как они выполняются на одном уровне активности и с одним приоритетом.)



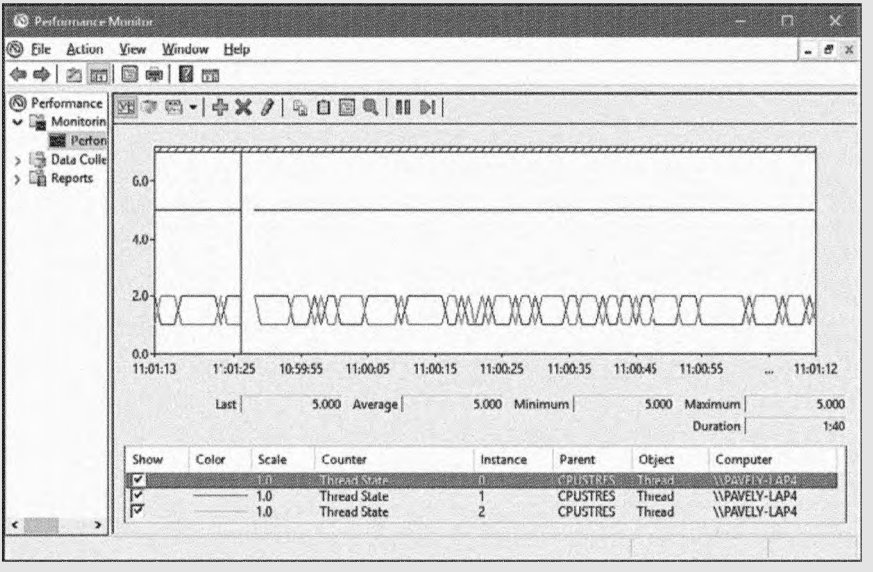
13. Вернитесь к программе CPU Stress, щелкните правой кнопкой мыши на потоке 2 и выберите в контекстном меню команду Busy. Поток 2 должен находиться в состоянии 2 (Running) чаще, чем поток 1:



14. Щелкните правой кнопкой мыши на потоке 1 и выберите уровень активности Maximum. Повторите этот шаг для потока 2. Теперь оба потока должны постоянно находиться в состоянии 2, потому что они фактически выполняются в бесконечном цикле.



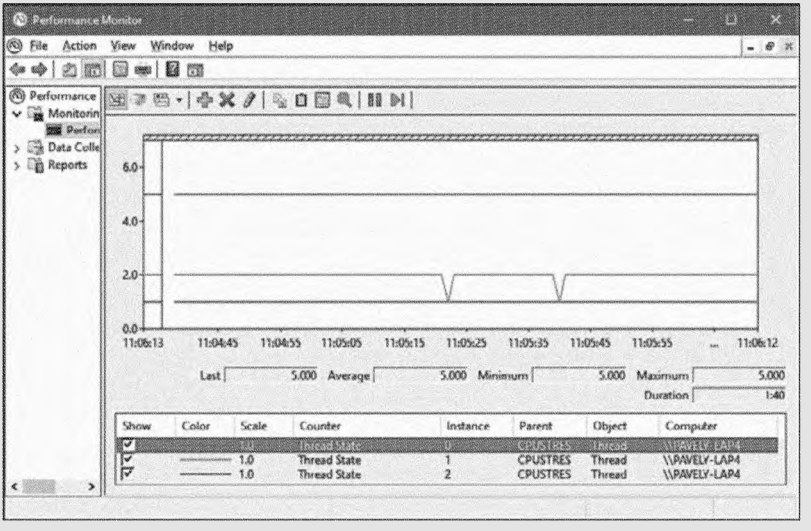
Если вы проводите эксперимент в однопроцессорной системе, результат будет несколько иным. Так как процессор всего один, в любой момент может выполняться один поток, поэтому два потока переключаются между состояниями 1 (Ready) и 2 (Running):



15. Если вы работаете в многопроцессорной системе (что очень вероятно), для получения того же результата можно открыть диспетчер задач, щелкнуть правой кнопкой мыши на процессе CPUSTRES, выбрать команду Задать

сходство (Set Affinity) и выбрать один процессор (неважно, какой именно). (То же самое можно сделать в программе CPU Stress: откройте меню Process и выберите команду Affinity.)

16. Осталось провести еще один эксперимент. Вернитесь к программе CPU Stress, щелкните правой кнопкой мыши на потоке 1 и выберите приоритет Above Normal. Вы увидите, что поток 1 работает постоянно (состояние 2), а поток 2 всегда находится в состоянии готовности (состояние 1). Это связано с тем, что процессор только один, поэтому в общем случае поток с более высоким приоритетом выигрывает. Однако время от времени вы увидите, что поток 1 переходит в состояние готовности. Дело в том, что каждые 4 секунды или около того простаивающий поток получает прирост, который позволяет ему некоторое время поработать. (Часто изменение состояния не отражается на графике, потому что гранулярность Системного монитора ограничивается одной секундой, а это слишком низкая точность.) Эта тема более подробно рассматривается в разделе «Повышение приоритета» этой главы.



База данных диспетчера

Чтобы принимать решения по планированию потоков, ядро поддерживает набор структур данных, собирательно известный как *база данных диспетчера*. В базе данных диспетчера хранится информация о том, какой поток ожидает выполнения и на каких процессорах какие потоки выполняются.

Как показано на рис. 4.9, для улучшения масштабируемости, включая параллельную диспетчеризацию потоков, на многопроцессорных системах Windows у каждого процессора есть для диспетчера очередь готовых потоков. Таким образом, каждый центральный процессор может проверять свои собственные очереди готовых потоков, чтобы запускать следующий поток без блокировки общесистемных очередей готовых потоков.

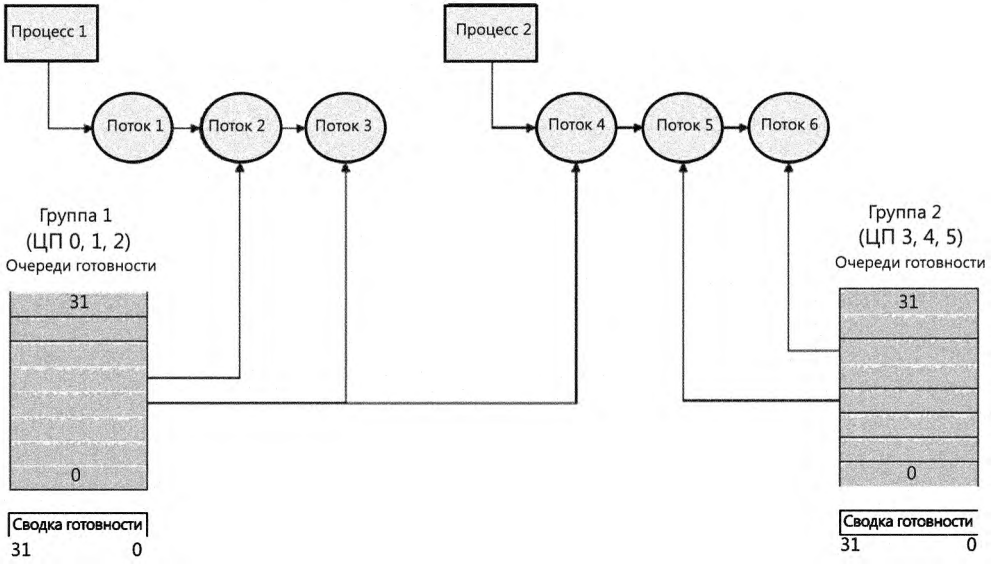


Рис. 4.9. База данных диспетчера многопроцессорной Windows

В версиях до Windows 8 и Windows Server 2012 использовались очереди готовности и сводки готовности на уровне процессора, которые являлись частью структуры блока управления процессора — PRCB (Processor Control Block). (Чтобы увидеть поля в PRCB, введите в отладчике ядра команду `dt nt!_kprcb`.) Начиная с Windows 8 и Windows Server 2012, очередь готовности и сводка готовности используются на уровне группы процессоров. Это позволяет системе принимать более эффективные решения относительно того, какой процессор должен использоваться следующим для группы процессоров. (Очереди готовности на уровне процессоров все еще существуют и используются для потоков с ограничениями сходства.)

ПРИМЕЧАНИЕ Поскольку общие структуры данных должны защищаться (посредством спин-блокировки), группа не должна быть слишком большой. В этом случае проблема конкуренции за очереди остается незначительной. В текущей реализации максимальный размер группы составляет четыре логических процессора. Если количество логических процессоров больше четырех, будет создано более одной группы, а доступные процессоры распределяются равномерно. Например, в шестипроцессорной системе будут созданы две группы из трех процессоров.

Очереди готовности, сводки готовности (см. далее) и еще некоторая информация хранятся в структуре ядра с именем `KSHARED_READY_QUEUE`, которая хранится в PRCB. И хотя она существует для каждого процессора, используется только структура первого процессора в каждой группе; она совместно используется с остальными процессорами этой группы.

Используемые диспетчером очереди готовых потоков (`ReadyListHead` в `KSHARED_READY_QUEUE`) содержат потоки, находящиеся в состоянии готовности в ожидании планирования их выполнения. Для каждого из 32 уровней приоритета существует одна очередь. Для ускорения выбора каждого потока на запуск или вытеснение Windows поддерживает 32-разрядную двоичную маску, которая называется *сводкой готовности* (`ReadySummary`). Каждый установленный бит обозначает один или нескольких потоков в очереди готовых потоков для того или иного уровня приоритета. (Нулевой бит представляет приоритет 0, бит 1 — приоритет 1 и т. д.)

Вместо сканирования каждого списка готовности с целью узнать, пуст он или нет (чтобы принимать решения по планированию в зависимости от количества потоков разного уровня приоритета), низкоуровневая команда процессора сканирует всего один бит и определяет наивысший установленный бит. Независимо от количества потоков в очереди их готовности эта операция выполняется за постоянное время.

База данных диспетчера синхронизируется путем повышения IRQL на `DISPATCH_LEVEL` (2) (см. главу 6). Повышение IRQL таким образом не позволяет другим потокам прерывать диспетчеризацию потоков на процессоре, потому что потоки обычно запускаются с уровнем запроса прерываний IRQL 0 или 1. Но требования одним повышением IRQL не ограничиваются, поскольку другие процессоры могут одновременно поднять уровень до такого же значения IRQL и предпринять попытку работы с базой данных их диспетчера. О том, как Windows синхронизирует обращения к базе данных диспетчера, рассказано далее в разделе «Многопроцессорные системы».

ЭКСПЕРИМЕНТ: ПРОСМОТР ГОТОВЫХ ПОТОКОВ

Для просмотра списка потоков в состоянии готовности используется команда !ready отладчика ядра. Команда выводит поток или список потоков, готовых к выполнению на каждом уровне приоритета. Пример вывода на 32-разрядной машине с четырьмя логическими процессорами:

```
0: kd> !ready
KSHARED_READY_QUEUE 8147e800: (00) ****----------------------------
SharedReadyQueue 8147e800: Ready Threads at priority 8
     THREAD 80af8bc0 Cid 1300.15c4 Teb: 7ffdb000 Win32Thread: 00000000 READY on
```

```
processor 80000002
     THREAD 80b58bc0 Cid 0454.0fc0 Teb: 7f82e000 Win32Thread: 00000000 READY on
processor 80000003
SharedReadyQueue 8147e800: Ready Threads at priority 7
     THREAD a24b4700 Cid 0004.11dc Teb: 00000000 Win32Thread: 00000000 READY on
processor 80000001
     THREAD a1bad040 Cid 0004.096c Teb: 00000000 Win32Thread: 00000000 READY on
processor 80000001
SharedReadyQueue 8147e800: Ready Threads at priority 6
     THREAD a1bad4c0 Cid 0004.0950 Teb: 00000000 Win32Thread: 00000000 READY on
processor 80000002
     THREAD 80b5e040 Cid 0574.12a4 Teb: 7fc33000 Win32Thread: 00000000 READY on
processor 80000000
SharedReadyQueue 8147e800: Ready Threads at priority 4
     THREAD 80b09bc0 Cid 0004.12dc Teb: 00000000 Win32Thread: 00000000 READY on
processor 80000003
SharedReadyQueue 8147e800: Ready Threads at priority 0
     THREAD 82889bc0 Cid 0004.0008 Teb: 00000000 Win32Thread: 00000000 READY on
processor 80000000
Processor 0: No threads in READY state
Processor 1: No threads in READY state
Processor 2: No threads in READY state
Processor 3: No threads in READY state
```

К номерам процессоров добавляется 0x8000000, но фактические номера процессоров хорошо видны. В первой строке выводится адрес KSHARED_READY_QUEUE с номером группы в круглых скобках (00 в выводе), за которым следует графическое представление процессоров этой конкретной группы (четыре звездочки).

Последние четыре строки выглядят немного странно — они показывают, что потоков в состоянии Ready нет, что противоречит предшествующему выводу. В них выводится информация о потоках из старого поля DispatcherReadyListHead структуры PRCB, потому что очереди готовности уровня процессора используются для потоков с ограничениями сходства (настроенными на выполнение на подмножестве процессоров в группе процессоров).

Вы также можете вывести содержимое KSHARED_READY_QUEUE с адресом, выведенным командой !ready:

```
0: kd> dt nt!_KSHARED_READY_QUEUE 8147e800
   +0x000 Lock             : 0
   +0x004 ReadySummary     : 0x1d1
   +0x008 ReadyListHead    : [32] _LIST_ENTRY [ 0x82889c5c - 0x82889c5c ]
   +0x108 RunningSummary   : [32] "???"
   +0x128 Span             : 4
   +0x12c LowProcIndex     : 0
   +0x130 QueueIndex       : 1
   +0x134 ProcCount        : 4
   +0x138 Affinity         : 0xf
```

В поле ProcCount выводится количество процессоров в группе (четыре в данном примере). Также обратите внимание на значение ReadySummary, 0x1d1. Оно соответствует 111010001 в двоичной записи. Читая биты по одному справа налево, мы узнаем, что потоки существуют в приоритетах 0, 4, 6, 7 и 8, что соответствует приведенному выводу.

Кванты времени

Как уже упоминалось ранее, *квант* представляет собой количество времени, предоставляемое потоку для выполнения, до того как Windows проверит наличие другого ожидающего потока с таким же уровнем приоритета. Если поток исчерпал свой квант, а других потоков с его уровнем приоритета нет, Windows позволяет потоку выполняться в течение еще одного кванта времени.

На клиентских версиях Windows потоки по умолчанию выполняются в течение двух интервалов таймера (clock intervals), а на серверных системах по умолчанию поток выполняется в течение 12 интервалов таймера (о том, как изменить эти значения, будет рассказано в разделе «Управление величиной кванта»). Причина более продолжительного значения по умолчанию на серверных системах — стремление к минимизации контекстных переключений. За счет более продолжительного кванта времени серверные приложения, пробуждаемые в результате клиентского запроса, имеют более высокий шанс на завершение обработки запроса и возвращение в состояние ожидания до окончания их кванта времени.

Продолжительность интервала таймера варьируется в зависимости от аппаратной платформы. Частота прерываний таймера зависит от HAL, а не от ядра. Например, интервал таймера на большинстве однопроцессорных систем x86 (они больше не поддерживаются Windows и упоминаются здесь только для примера) составляет около 10 миллисекунд, на большинстве многопроцессорных систем x86 и x64 он составляет порядка 15 миллисекунд. Величина интервала таймера хранится в переменной ядра `KeMaximumIncrement` в сотнях наносекунд.

Хотя время выполнения потока определяется интервалами таймера, система не пользуется подсчетом сигналов таймера для проверки продолжительности выполнения потока и истечения его кванта времени. Дело в том, что учет времени выполнения потока основан на тактах процессора. Вместо этого при запуске системы вычисляется количество тактовых циклов, которому равен каждый квант времени. Для этого тактовая частота процессора в герцах (число тактовых импульсов центрального процессора в секунду) умножается на количество секунд, затрачиваемое на срабатывание одного такта системных часов (на основе `KeMaximumIncrement`). Значение сохраняется в переменной ядра `KiCyclesPerClockQuantum`.

Результат такого метода вычисления состоит в том, что потоки на самом деле запускаются не на количество квантов времени, основанное на тактах системных часов, а на время, определенное квантовой целью, которая представляет собой приблизительный подсчет количества тактовых импульсов центрального процессора, потребленное потоком до того, как он уступит свою очередь другому потоку. Эта цель должна быть равна количеству тактов интервального таймера, поскольку, как вы только что увидели, подсчет тактовых импульсов на квант основан на величине интервала системного таймера, которую можно проверить с помощью следующего эксперимента. С другой стороны, поскольку в потоке не учитываются циклы прерываний, фактический интервал таймера может быть длиннее.

ЭКСПЕРИМЕНТ: ОПРЕДЕЛЕНИЕ ВЕЛИЧИНЫ ИНТЕРВАЛА СИСТЕМНОГО ТАЙМЕРА

Интервал системного таймера возвращает Windows-функция GetSystemTimeAdjustment function. Для определения величины этого интервала нужно запустить программу clockres из пакета Sysinternals. Вывод для четырехъядерной 64-разрядной системы Windows 10 выглядит так:

```
C:\>clockers

ClockRes v2.0 - View the system clock resolution
Copyright (C) 2009 Mark Russinovich
SysInternals - www.sysinternals.com

Maximum timer interval: 15.600 ms
Minimum timer interval: 0.500 ms
Current timer interval: 1.000 ms
```

Текущий интервал может быть ниже максимального (используемого по умолчанию) из-за мультимедийных таймеров. Мультимедийные таймеры применяются с такими функциями, как timeBeginPeriod и timeSetEvent, используемыми для получения обратных вызовов с интервалами не менее 1 миллисекунды в лучшем случае. Это приводит к глобальному перепрограммированию интервального таймера ядра, в результате которого планировщик активизируется с более частыми интервалами, а это может привести к снижению быстродействия системы. В любом случае на длину кванта это не влияет (см. следующий раздел).

Также можно прочитать значение из глобальной переменной ядра KeMaximumIncrement (вывод получен не в той системе, которая использовалась в предыдущем примере):

```
0: kd> dd nt!KeMaximumIncrement L1
814973b4 0002625a
0: kd> ? 0002625a
Evaluate expression: 156250 = 0002625a
```

Это соответствует значению по умолчанию 15,6 мс.

Учет квантового времени

У каждого процесса в его блоке управления (KPROCESS) есть значение перезапуска кванта. Это значение используется при создании новых потоков внутри процесса и дублируется в блок управления потоком (KTHREAD), после чего используется при назначении потоку новой квантовой цели. Значение перезапуска кванта при хранении измеряется в фактических квантовых единицах (вскоре вы узнаете, что это означает), которые затем умножаются на количество тактовых импульсов на квант времени, в результате чего получается квантовая цель.

По мере выполнения потока тактовые циклы центрального процессора тратятся на различные события — переключения контекста, прерывания и конкретные решения

по планированию. Если при возникновении прерывания от интервального таймера количество затраченных тактовых циклов центрального процессора достигает квантовой цели (или уже превышает ее), запускается обработка истечения кванта. Если есть еще один поток, ожидающий запуска и имеющий такой же приоритет, происходит переключение контекста на следующий поток в очереди готовых потоков.

Внутри системы квантовая единица представлена в виде одной трети от такта интервального таймера. (Следовательно, один такт равен трем квантовым единицам.) Это означает, что в клиентских системах Windows у потоков по умолчанию значение перезапуска кванта равно 6 (2×3), а в серверных системах оно равно 36 (12×3). Поэтому значение переменной `KiCyclesPerClockQuantum` в конце ранее рассмотренных вычислений делится на 3, поскольку исходное значение описывает только количество тактовых циклов центрального процессора, приходящееся на один такт интервального таймера.

Квант был сохранен внутри системы в виде доли такта таймера, а не в виде целого такта, чтобы позволить частичный расход кванта на завершение ожидания в версиях Windows, предшествовавших Windows Vista. В предыдущих версиях интервальный таймер использовался для истечения кванта времени. Если бы не эта поправка, потоки могли бы иметь никогда не снижаемый квант времени. Например, если поток выполняется, входит в состояние ожидания, опять выполняется и входит еще в одно состояние ожидания, но при этом он никогда не был текущим выполняемым потоком при запуске интервального таймера, то его квант никогда не будет затрачен на то время, в течение которого он выполнялся. Поскольку теперь у потоков тратятся не кванты, а тактовые циклы центрального процессора и поскольку этот процесс уже не зависит от интервального таймера, такие поправки уже не нужны.

ЭКСПЕРИМЕНТ: ОПРЕДЕЛЕНИЕ КОЛИЧЕСТВА ТАКТОВЫХ ЦИКЛОВ НА ОДИН КВАНТ

В Windows нет функции, которая выдавала бы количество тактовых циклов на один квант. Однако на основе приведенного описания вы сможете легко вычислить его самостоятельно, выполнив следующие действия и использовав такой отладчик ядра, как WinDbg в режиме локальной отладки.

1. Возьмите тактовую частоту вашего процессора, определяемую системой Windows. Можно воспользоваться значением, хранящимся в PRCB-поле MHz, которое можно вывести с помощью команды `!cpuinfo`. На четырехпроцессорной системе Intel, работающей с тактовой частотой 2794 МГц, этот вывод будет выглядеть так:

```
lkd> !cpuinfo

CP F/M/S Manufacturer MHz PRCB Signature MSR 8B Signature Features
 0 6,60,3 GenuineIntel 2794 ffffffff00000000 >ffffffff00000000<a3cd3fff
 1 6,60,3 GenuineIntel 2794 ffffffff00000000                    a3cd3fff
 2 6,60,3 GenuineIntel 2794 ffffffff00000000                    a3cd3fff
 3 6,60,3 GenuineIntel 2794 ffffffff00000000                    a3cd3fff
```

2. Переведите полученное число в герцы (Гц). Получится число тактовых циклов центрального процессора, имеющее место на вашей системе каждую секунду. В данном случае это будет 2 794 000 000 циклов в секунду.
3. Возьмите с помощью программы `clockres` значение интервала таймера на вашей системе. Им оценивается продолжительность периода времени между запусками таймера. На системе, взятой в качестве примера, этот интервал был равен 15,625 мс.
4. Преобразуйте это число в количество запусков интервального таймера в секунду. В одной секунде 1000 мс, поэтому разделите число, полученное при выполнении действия 3, на 1000. В данном случае таймер запускается каждые 0,015625 секунды.
5. Умножьте полученный результат на количество циклов в секунду, полученное при выполнении действия 2. В данном случае на каждый интервал таймера приходится 43 656 250 цикла.
6. Вспомните, что каждая квантовая единица равна одной трети интервала таймера, поэтому разделите количество циклов на три. В нашем примере получится 14 528 083, или 0xDE0C13 в шестнадцатеричном виде. Это количество тактовых циклов, которое должно приходиться на каждую квантовую единицу на системе, запущенной с тактовой частотой 2794 МГц и с интервалом таймера, равным приблизительно 15,6 мс.
7. Чтобы проверить свои вычисления, выведите дамп значения переменной `KiCyclesPerClockQuantum` на своей системе. Это значение должно совпадать с полученным результатом (или почти совпадать с учетом ошибок округления).

```
lkd> dd nt!KiCyclesPerClockQuantum L1
8149755c 00de0c10
```

Управление величиной кванта

Квант потока можно изменить для всех процессов, но вы можете выбрать только одну из двух настроек: короткую (2 такта таймера — значение по умолчанию для клиентских машин) или длинную (12 тактов таймера — значение по умолчанию для серверных систем).

ПРИМЕЧАНИЕ Используя объект задания на системе, работающей с длинными квантами, вы можете выбирать для процесса в задании другие значения кванта.

Чтобы изменить эту настройку, щелкните правой кнопкой мыши на значке Компьютер (This PC) на рабочем столе или выберите в Проводнике Windows команду Свойства (Properties), щелкните на пункте Дополнительные параметры системы (Advanced System Settings), перейдите на вкладку Дополнительно (Advanced), щелкните на кнопке Settings (Параметры) в области Быстродействие (Performance) и, наконец, перейдите на вкладку Дополнительно (Advanced). Вы увидите диалоговое окно, показанное на рис. 4.10.

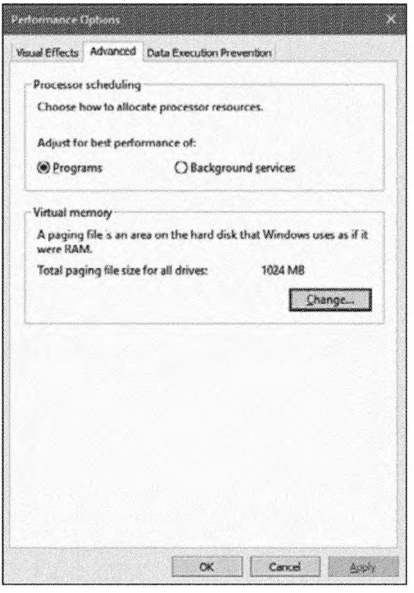


Рис. 4.10. Настройка кванта в диалоговом окне Параметры быстродействия (Performance Options)

Диалоговое окно предоставляет два варианта.

- Режим Оптимизировать работу программ (Programs), определяющий использование коротких, переменных квантов, устанавливается по умолчанию для клиентских версий Windows (и других аналогичных версий: мобильных, XBOX, Hololens и т. д.). Если вы установили службы терминалов (Terminal Services) в серверной системе и настроили сервер в качестве сервера приложений, выбирается эта настройка, чтобы у пользователей сервера терминалов были такие же настройки кванта, которые обычно устанавливаются на настольной или клиентской системе. Можно также выбрать эту настройку самостоятельно, если запускать Windows Server в качестве своей настольной ОС.
- Режим Оптимизировать работу служб, работающих в фоновом режиме (Background Services), с длинными, фиксированными квантами устанавливается по умолчанию для серверных систем. Эту настройку следует выбирать в системе рабочей станции только в том случае, если вы намерены использовать эту рабочую станцию в качестве серверной системы. Но поскольку эффект от изменения этой настройки наступает немедленно, ее использование может быть оправданно для предстоящей работы машины с нагрузкой, напоминающей фоновый режим работы сервера. Например, при длительных вычислениях, кодировании или имитационном моделировании, требующих работы на всю ночь, можно временно выбрать режим Оптимизировать работу служб, работающих в фоновом режиме (Background Services), а утром работу системы можно вернуть в режим Оптимизировать работу программ (Programs).

Переменные кванты

Когда разрешено использование переменных квантов, в таблицу переменных квантов `PspForegroundQuantum`, которая используется функцией `PspComputeQuantum`, загружается таблица `PspVariableQuantums`. Алгоритм выбирает подходящий индекс кванта в зависимости от того, работает процесс на первом плане или нет (т. е. содержится ли в нем поток, владеющий окном первого плана на рабочем столе). Если нет, то выбирается индекс, равный нулю, который соответствует рассмотренному ранее кванту потока по умолчанию. Если это процесс первого плана, индекс кванта соответствует разносу приоритетов.

Значение разноса приоритетов определяет повышение приоритета, которое планировщик будет применять к потокам первого плана (см. раздел «Повышение приоритета»), и он, таким образом, составит пару с соответствующим расширением квоты: для каждого дополнительного уровня приоритета (вплоть до 2) потоку будет дан еще один квант. Например, если поток получил увеличение приоритета на один уровень, он также получает дополнительный квант. По умолчанию Windows устанавливает для потоков первого плана максимально возможное повышение приоритета, это означает, что разнос приоритетов будет равен 2, поэтому выбор индекса кванта 2 в таблице переменных квантов приведет к тому, что поток получит два дополнительных кванта, что в сумме составит три кванта.

В табл. 4.2 приведены точные значения квантов (напомним, что они хранятся в единицах, представляющих 1/3 от такта интервального таймера), которые будут выбраны на основе индекса кванта и используемой конфигурации квантов.

Таблица 4.2. Значения квантов

	Индекс короткого кванта			**Индекс длинного кванта**		
Переменное значение	6	12	18	12	24	36
Фиксированное значение	18	18	18	36	36	36

Таким образом, когда окно выходит на первый план в клиентской системе, все потоки в процессе, содержащем тот поток, который владеет окном первого плана, получают утроенные кванты: потоки в процессе первого плана запускаются с квантом, равным шести тактам интервального таймера, а потоки в других процессах располагают своим клиентским квантом по умолчанию, равным двум тактам таймера. Таким образом, когда происходит обратное переключение из процесса, интенсивно использующего центральный процессор, новый процесс первого плана получит пропорционально больше времени центрального процессора, потому что, когда его потоки запустятся, у них будет более продолжительный период работы, чем у фоновых потоков (опять же при условии, что у потоков как процессов первого плана, так и фоновых процессов одинаковый приоритет).

Настройки кванта в реестре

Рассмотренный ранее пользовательский интерфейс для управления настройками кванта изменяет параметр реестра HKLM\SYSTEM\CurrentControlSet\Control\

PriorityControl\Win32PrioritySeparation. Кроме определения относительной продолжительности кванта потоков (короткий или длинный), этот параметр реестра также определяет, должны ли использоваться переменные кванты, а также приоритетные разделения (что, как вы увидите, определит индекс кванта, используемый при разрешенных переменных квантах). Значение состоит из 6 битов, поделенных на три поля по два бита, показанных на рис. 4.11.

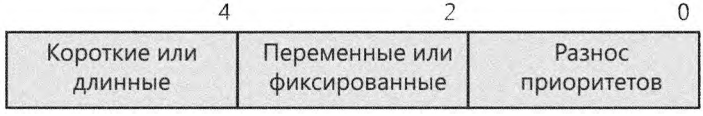


Рис. 4.11. Поля параметра реестра Win32PrioritySeparation

Поля, показанные на рис. 4.11, определяются следующим образом.

- **Короткие или длинные.** Значение 1 определяет длинные кванты, а значение 2 определяет короткие кванты. Значение 0 или 3 показывает, что будут использованы установки по умолчанию, применяемые на данной системе (короткие кванты для клиентских систем, длинные для серверных систем).
- **Переменные или фиксированные.** Значение 1 включает таблицу переменных квантов, использование которой основано на алгоритме, приведенном в разделе «Переменные кванты». Значение 0 или 3 показывает, что будут использованы установки по умолчанию, применяемые на данной системе (переменные кванты для клиентских систем, фиксированные для серверных систем).
- **Разнос приоритетов.** Это поле (значение которого хранится в переменной ядра `PsPrioritySeparation`) определяет приоритетное разделение (вплоть до 2), как объяснялось в разделе «Переменные кванты».

При использовании диалогового окна Параметры быстродействия (Performance Options), показанного на рис. 4.10, можно выбрать только одну из двух комбинаций: короткие кванты, с утроенными квантами первого плана, или длинные кванты без изменения квантов для потоков первого плана. Но можно выбрать другие комбинации, непосредственно изменяя параметр реестра `Win32PrioritySeparation`.

Потоки, входящие в процесс, запущенный с классом приоритета простоя (idle), всегда получают единичный квант потока, игнорируя любые виды настроек конфигурации квантов, будь то настройки по умолчанию или настройки, заданные в реестре.

В системах Windows Server, настроенных для работы в качестве серверов приложений, исходное значение параметра реестра `Win32PrioritySeparation` будет равно шестнадцатеричному числу 26, что идентично значению, установленному выбором в диалоговом окне Параметры быстродействия (Performance Options) параметра Оптимизировать работу программ (Programs). При этом выбирается поведение по определению кванта и повышению приоритета, сходное с поведением клиентских систем Windows; оно подходит для сервера, преимущественно использующегося для размещения пользовательских приложений.

В клиентских системах Windows и на серверах, не настроенных в качестве серверов приложений, исходное значение параметра реестра `Win32PrioritySeparation` будет равно 2. При этом битовые поля Короткие и длинные (Short vs. Long) и Переменные и фиксированные (Variable vs. Fixed) заполняются нулевыми значениями, которые для этих параметров основываются на поведении системы по умолчанию (в зависимости от того, клиентская это система или серверная). Для поля Разнос приоритетов (Priority Separation) обусловливается значение 2. Как только параметр реестра изменяется с помощью диалогового окна Параметры быстродействия (Performance Options), его исходное значение уже не может быть восстановлено кроме как непосредственным изменением реестра.

ЭКСПЕРИМЕНТ: ВЛИЯНИЕ ИЗМЕНЕНИЯ НАСТРОЙКИ КВАНТА

Используя локальный отладчик, можно посмотреть, как две настройки кванта, для программ и для фоновых служб, влияют на таблицы `PsPrioritySeparation` и `PspForegroundQuantum`, а также на изменение значения `QuantumReset` потока в системе. Выполните следующие действия.

1. Откройте приложение Система (System) из панели управления или щелкните правой кнопкой мыши на значке Компьютер (This PC) на рабочем столе и выберите команду Свойства (Properties).
2. Щелкните на пункте Дополнительные параметры системы (Advanced System Settings), перейдите на вкладку Дополнительно (Advanced), щелкните на кнопке Settings (Параметры) в области Быстродействие (Performance) и, наконец, перейдите на вкладку Дополнительно (Advanced).
3. Выберите вариант Оптимизировать работу программ (Programs) и щелкните на кнопке Применить (Apply). Оставьте это диалоговое окно открытым на протяжении всего эксперимента.
4. Выведите значения `PsPrioritySeparation` и `PspForegroundQuantum`, как показано ниже. Выводятся те значения, которые должны выводиться в системе Windows после внесения изменений в шагах 1–3. Обратите внимание на то, что использовалась таблица переменных коротких квантов, а к приложениям первого плана будет применено увеличение приоритета, равное 2:

```
lkd> dd nt!PsPrioritySeparation L1
fffff803'75e0e388 00000002
lkd> db nt!PspForegroundQuantum L3

fffff803'76189d28 06 0c 12
```

5. Теперь взгляните на значение `QuantumReset` любого процесса в системе. Как описано ранее, это значение — полный квант каждого потока в системе при его пополнении. Это значение кэшируется для каждого потока в процессе, но проще будет посмотреть на структуру `KPROCESS`. Обратите внимание, что в данном случае оно равно 6, поскольку WinDbg, как и большинство других приложений, получает квант, установленный в первой записи таблицы `PspForegroundQuantum`:

```
lkd> .process
Implicit process is now ffffe001'4f51f080
```

```
lkd> dt nt!_KPROCESS ffffe001'4f51f080 QuantumReset
+0x1bd QuantumReset : 6 ''
```

6. Теперь в диалоговом окне, открытом в п. 1 и 2, измените настройку в области Быстродействие (Performance), выбрав пункт Оптимизировать работу: служб, работающих в фоновом режиме (Background Services).
7. Повторите команды, показанные в п. 4 и 5. Значения должны измениться в соответствии с нашим обсуждением данного вопроса в этом разделе:

```
lkd> dd nt!PsPrioritySeparation L1
fffff803'75e0e388 00000000
lkd> db nt!PspForegroundQuantum L3
fffff803'76189d28 24 24 24
lkd> dt nt!_KPROCESS ffffe001'4f51f080 QuantumReset
   +0x1bd QuantumReset : 36 '$'
```

Повышение приоритета

Планировщик Windows периодически настраивает текущий приоритет потоков, используя внутренний механизм повышения приоритета. Во многих случаях это делается для уменьшения различных задержек (чтобы потоки быстрее реагировали на события, в ожидании которых они находятся) и ускорения отклика. В других случаях это повышение применяется для предотвращения зависаний и инверсии приоритетов. В данном разделе будет рассмотрен ряд сценариев повышения приоритета, к числу которых относятся следующие сценарии и их причины.

- Повышение вследствие событий планировщика или диспетчера (сокращение задержек).
- Повышение вследствие завершения ввода/вывода (сокращение задержек).
- Повышение вследствие ввода из пользовательского интерфейса (сокращение задержек и времени отклика).
- Повышение вследствие слишком продолжительного ожидания ресурса исполняющей системы (`ERESOURCE`) (предотвращение зависания).
- Повышение в случае, когда готовый к запуску поток не был запущен в течение определенного времени (предотвращение зависания и инверсии приоритетов).

Но, как и любые другие алгоритмы планирования, эти настройки не совершенны, и они могут не принести пользы абсолютно всем приложениям.

ПРИМЕЧАНИЕ Windows никогда не повышает приоритет потоков в диапазоне реального времени (от 16 до 31). Поэтому планирование относительно других потоков в диапазоне реального времени всегда предсказуемо. Windows предполагает, что при использовании приоритетов потоков реального времени вы знаете, что делаете.

В клиентские версии Windows также включены другие псевдоповышающие механизмы, проявляющие себя при проигрывании мультимедиа. В отличие от других повышений приоритета, этими механизмами обычно управляет служба пользо-

вательского режима, которая называется службой планировщика класса мультимедиа — MultiMedia Class Scheduler Service (mmcss.sys), но это не является настоящим повышением; служба просто устанавливает по необходимости новые базовые приоритеты для потоков. Поэтому в данном случае не действует ни одно из правил, относящихся к повышению приоритета. Сначала мы рассмотрим обычные случаи повышения приоритета под управлением ядра, а затем поговорим о MMCSS и о разновидности «повышения», выполняемого этой службой.

Повышение приоритета вследствие событий планировщика или диспетчера

При наступлении события диспетчера вызывается процедура `KiExitDispatcher` с задачей обработки списка отложенных готовых потоков путем вызова процедуры `KiProcessThreadWaitList`, а затем вызова процедуры `KiCheckForThreadDispatch`, чтобы проверить, не должны ли на локальном процессоре быть намечены какие-либо потоки, которые не планируются к выполнению. При каждом наступлении такого события вызывающий код может также указать, какого типа повышение должно быть применено к потоку, а также с каким приращением приоритета должно быть связано это повышение. Следующие сценарии считаются событиями диспетчера `AdjustUnwait`, потому что они имеют дело с объектом диспетчера, входящим в сигнальное состояние, что может стать причиной пробуждения одного или нескольких потоков.

- В очередь потока поставлен APC-вызов (см. главу 6 и более подробное описание в главе 8 части 2).
- Событие установлено или отправило сигнал.
- Таймер установлен, или системное время изменилось, и таймеры должны быть перезапущены.
- Мьютекс был освобожден или ликвидирован.
- Произошел выход из процесса.
- В очередь (`KQUEUE`) была вставлена запись, или очередь была очищена.
- Был освобожден семафор.
- Поток был приведен в состояние готовности, приостановлен, возобновлен, заморожен или разморожен.
- Первичный UMS-поток ожидает переключения на планируемый UMS-поток.

Для планирования событий, связанных с общедоступным API (например, с `SetEvent`), применяемое повышение приращения указывается вызывающим кодом. Windows рекомендует разработчикам использовать конкретные значения, которые будут рассмотрены далее. Для приведения в состояние готовности применяется повышение 2 (если только поток не приводится в состояние готовности вызовом `KeAlertThreadByThreadId` — тогда применяется повышение 1), потому что у API-функций приведения в состояние готовности нет параметра, позволяющего вызывающему коду задать нестандартное приращение.

У планировщика также есть два специальных события диспетчера `AdjustBoost`, являющиеся частью механизма прав собственности на блокировку приоритета. Эти повышения пытаются исправить ситуации, при которых вызывающий код, владеющий блокировкой с приоритетом X, в конечном итоге снимает блокировку для ожидающего потока с приоритетом $\leq X$. В такой ситуации новый поток владельца должен ждать своей очереди (если он запускается с приоритетом X), или, что еще хуже, он может даже вообще не получить возможности, если его приоритет ниже X. В результате освобождаемый поток продолжает выполнение, даже если он должен был разбудить поток нового владельца для получения контроля над процессором. Выдача диспетчером события `AdjustBoost` вызывается следующими двумя событиями диспетчеризации:

- событием, установленным через интерфейс `KeSetEventBoostPriority`, который используется блокировкой ядра по чтению/записи `ERESOURCE`;
- шлюзом, установленным через интерфейс `KeSignalGate`, который используется различными внутренними механизмами при освобождении блокировки шлюза.

Повышения приоритета, связанные с завершением ожидания

Повышения приоритета, связанные с завершением ожидания (unwait boosts), пытаются уменьшить время задержки между потоком, пробуждающимся по сигналу объекта (переходя тем самым в состояние Ready), и потоком, фактически приступившим к своему выполнению в процессе, который не находился в состоянии ожидания (переходя тем самым в состояние Running). В общем случае поток, пробуждающийся из состояния ожидания, должен иметь возможность приступить к выполнению как можно скорее.

В разнообразных файлах заголовков Windows указываются рекомендуемые значения, которые должны использоваться кодом режима ядра, вызывающим такие API-функции, как `KeReleaseMutex`, `KeSetEvent` и `KeReleaseSemaphore`, и которые соответствуют таким определениям, как `MUTANT_INCREMENT`, `SEMAPHORE_INCREMENT` и `EVENT_INCREMENT`. Эти три определения всегда устанавливаются в этих заголовках в 1, поэтому вполне можно предположить, что большинство повышений на этих объектах, связанных с завершением ожидания, кончится повышением 1. В API пользовательского режима приращение указано быть не может; у низкоуровневых системных вызовов, таких как `NtSetEvent`, также не предусмотрены параметры для его указания. Вместо этого, когда эти API-функции вызывают нижележащий `Ke`-интерфейс, они автоматически используют определение `_INCREMENT`. Это также происходит при ликвидации мьютексов и перезапуске таймеров из-за изменения системного времени: система использует повышение по умолчанию, которое обычно будет применяться после того, как мьютекс будет освобожден. И наконец, APC-повышение полностью зависит от вызывающего кода. Скоро будет показано особое применение APC-повышения, связанное с завершением ввода/вывода.

ПРИМЕЧАНИЕ У некоторых объектов диспетчера нет связанных с ними повышений. Например, при установке, или при истечении времени таймера, или при подаче сигнала о процессе повышение не применяется.

Все эти повышения на +1 пытаются решить исходную проблему на основе предположения, что как освобождаемый поток, так и ожидающие потоки запущены с одинаковым приоритетом. Путем повышения приоритета ожидающих потоков на один уровень ожидающий поток должен вытеснить освобождающийся поток, как только операция завершится. К сожалению, на однопроцессорных системах, если это предположение не выполняется, повышение может быть неэффективным: если ожидающий поток ожидает с приоритетом 4, а освобождающийся поток имеет приоритет 8, ожидание с приоритетом 5 не справится с сокращением времени ожидания и принудительным вытеснением. Но на многопроцессорных системах благодаря алгоритмам захвата и балансировки у этого потока с более высоким приоритетом может быть более высокий шанс быть подхваченным другим логическим процессором. Этот факт связан с выбором проектировщиков исходной архитектуры NT, в которой владельцы блокировок не отслеживаются (за исключением нескольких блокировок). Это означает, что планировщик не может быть уверен в том, кто на самом деле владеет событием и на самом ли деле оно будет использоваться в качестве блокировки. Даже если отслеживать владение блокировками, право собственности обычно не передается, чтобы избежать проблем сопровождения, отличающихся от случая с блокировкой `ERESOURCE`, который будет рассмотрен чуть позже.

Но для некоторых видов блокирующих объектов использование событий и шлюзов в качестве их основных объектов синхронизации, повышение приоритета владельцев блокировки решает проблему. Кроме того, благодаря схемам распределения процессоров и балансировки нагрузки, которые будут рассмотрены далее, на многопроцессорной машине готовый поток может быть подобран другим процессором, а его высокий приоритет может повысить шансы на запуск вместо первичного процессора на этом вторичном процессоре.

Повышение приоритета владельца блокировки

Поскольку блокировки ресурсов исполняющей системы (`ERESOURCE`) и блокировки критических разделов используют основные объекты диспетчеризации, в результате освобождения этих блокировок осуществляются повышения приоритета, связанные с завершением ожидания. С другой стороны, поскольку высокоуровневые реализации этих объектов отслеживают владельца блокировки, ядро может принять более взвешенное решение о том, какого вида повышение должно быть применено с помощью `AdjustBoost`. В повышениях этого вида значение `AdjustIncrement` устанавливается на текущий приоритет освобождаемых (или настраиваемых) потоков за минусом любого повышения, связанного с выделением первого плана GUI. Перед тем как будет вызвана функция `KiExitDispatcher`, кодом события и шлюза вызывается функция `KiRemoveBoostThread`, чтобы вернуть освобождаемый поток к его обычному приоритету (через функцию `KiComputeNewPriority`). Это необходимо, чтобы избежать ситуации сопровождения блокировки (lock convoy), в которой два потока, неоднократно передавая блокировку друг другу, получают все большие повышения приоритета.

ПРИМЕЧАНИЕ Следует заметить, что push-блокировки, являющиеся несправедливыми блокировками, потому что владелец блокировки в спорном пути ее приобретения непредсказуем (и, скорее всего, случаен, как и при спин-блокировке), не применяют повышения

приоритета из-за владения блокировкой. Причина в том, что это просто содействовало бы вытеснению и быстрому росту приоритета, что не требуется, поскольку блокировка сразу же после снятия тут же становится свободной (пропуская обычный путь ожидания/прекращения ожидания).

Другие различия между повышением приоритета, связанным с владением блокировкой, и повышением приоритета, связанным с завершением ожидания, проявятся в том способе, которым планировщик обычно применяет повышение приоритета. Эта тема будет рассматриваться в следующем разделе.

Повышение приоритета после завершения ввода/вывода

Windows дает временное повышение приоритета при завершении определенных операций ввода/вывода, при этом потоки, ожидавшие ввода/вывода, имеют больше шансов сразу же запуститься и обработать то, чего они ожидали. Хотя рекомендуемые величины повышения можно найти в заголовочных файлах Windows Driver Kit (WDK) (поищите строку *#define IO* в файле Wdm.h или в файле Ntddk.h), подходящее значение для увеличения зависит от драйвера устройства. (Эти значения перечислены в табл. 4.3.) Именно драйвер устройства указывает повышение при завершении запроса на ввод/вывод в своем вызове функции ядра `IoCompleteRequest`. В табл. 4.3 следует обратить внимание на то, что запросы ввода/вывода к устройствам, гарантирующим ускорение отклика, имеют более высокий прирост приоритета.

Таблица 4.3. Рекомендуемые значения повышения приоритета

Устройство	Повышение
Диск, CD-ROM, параллельный порт, видео	1
Сеть, почтовый слот, именованный канал, последовательный порт	2
Клавиатура, мышь	6
Звуковая карта	8

ПРИМЕЧАНИЕ Казалось бы, от видеокарты или диска следовало бы ожидать большей скорости отклика, чем при повышении 1. Однако ядро в действительности стремится оптимизировать *задержку*, к которой одни устройства более чувствительны, чем другие. Например, для воспроизведения музыки без ощутимых искажений звуковая карта должна получать данные каждую 1 мс, тогда как видеокарта выводит 24 кадра в секунду, или приблизительно каждые 40 мс.

Как указывалось ранее, это повышение после завершения ввода/вывода надеется на повышения, связанные с завершением ожидания, рассмотренные в предыдущем разделе. При этом следует учитывать важную подробность: ядро реализует код подачи сигнала в API-функции `IoCompleteRequest` либо посредством использования APC (для асинхронного ввода/вывода), либо через событие (для синхронного ввода/вывода). Когда драйвер передает в функцию `IoCompleteRequest`, к примеру, параметр `IO_DISK_INCREMENT` для асинхронного чтения диска, ядро вызывает `KeInsertQueueApc` с параметром повышения, установленным на `IO_DISK_INCREMENT`. В свою очередь, когда ожидание потока разрушается из-за APC, он получает повышение, равное 1.

Следует иметь в виду, что значения повышения, которые были даны в предыдущей таблице, это всего лишь рекомендации от компании Microsoft, разработчики драйверов могут при желании спокойно их проигнорировать, и конкретные специализированные драйверы могут использовать свои собственные значения. Например, драйвер, обрабатывающий ультразвуковые данные от медицинского прибора, который должен сообщать приложению визуализации, работающему в пользовательском режиме, о новых данных, может использовать повышение 8, чтобы получить такую же задержку, как у звуковой карты. Но во многих случаях из-за особенностей организации стеков драйверов Windows разработчики драйверов часто создают *мини-драйверы*, обращающиеся к драйверам компании Microsoft, которые снабжают их своим собственным повышением приоритета `IoCompleteRequest`. Например, разработчики карт контроллеров RAID или SATA будут, как правило, вызывать для завершения обработки их запросов функцию `StorPortCompleteRequest`. У этого вызова нет никаких параметров для значения повышения, потому что драйвер Storport.sys вставляет подходящее значение при вызове ядра. Кроме того, в новых версиях Windows при каждом завершении запроса любым драйвером файловой системы (идентифицируемым установкой своего типа устройства в `FILE_DEVICE_DISK_FILE_SYSTEM` или в `FILE_DEVICE_NETWORK_FILE_SYSTEM`), всегда вместо этого применяется повышение `IO_DISK_INCREMENT`, если драйверу передается значение `IO_NO_INCREMENT` (0). Таким образом, этот прирост становится меньше рекомендованного и больше требуемого ядром.

Повышение при ожидании ресурсов исполняющей системы

Когда поток пытается получить ресурс исполняющей системы (`ERESOURCE`; подробнее об объектах синхронизации см. в главе 8 части 2), который уже находится в исключительном владении другого потока, он должен быть в состоянии ожидания до тех пор, пока другой поток не освободит ресурс. Для ограничения риска взаимных исключений исполняющая система выполняет это ожидание, не входя в бесконечное ожидание ресурса, а интервалами по 500 мс. Если по окончании этих 500 мс ресурс все еще находится во владении, исполняющая система пытается предотвратить зависание центрального процессора путем получения блокировки диспетчера, повышения приоритета потока или потоков, владеющих ресурсом, до 15 (если исходный приоритет владельца был меньше, чем у ожидающего, и не был равен 15), перезапуска их квантов и выполнения еще одного ожидания.

Поскольку ресурсы исполняющей системы могут быть либо общими, либо монопольными, ядро сначала повышает приоритет монопольного владельца, а затем проводит проверку общих владельцев и повышает приоритет всех этих владельцев. Когда ожидающий поток опять входит в состояние ожидания, появляется надежда, что планировщик так спланирует работу одного из потоков-владельцев, чтобы у него было достаточно времени для завершения своей работы и освобождения ресурса. Следует учесть, что этот механизм повышения приоритета используется только в случае отсутствия у ресурса установленного флага запрещения повышения приоритета `Disable Boost`, установку которого разработчики могут выбрать при нормальной работе с использованием ресурсов описываемого здесь механизма смены приоритетов.

Следует заметить, что этот механизм несовершенен. Например, если у ресурса есть несколько общих пользователей, исполняющая система повышает приоритет всех потоков-владельцев до значения 14, что приводит к внезапному всплеску в системе количества высокоприоритетных потоков, обладающих полными квантами времени. Хотя исходный поток-владелец будет запущен первым (поскольку был первым при повышении приоритета и поэтому первым в списке готовых потоков), другие владельцы общего ресурса будут запущены следующими, поскольку приоритет ожидающего потока повышен не был. Только после того, как все владельцы общего ресурса воспользуются возможностью запуститься, а их приоритет будет уменьшен до значения ниже приоритета ожидающего потока, этот ожидающий поток наконец-то получит шанс на приобретение ресурса. Поскольку владельцы общего ресурса могут перевести или конвертировать свое владение из общего в монопольное сразу же после освобождения ресурса монопольным владельцем, данный механизм может работать не так, как было задумано.

Повышение приоритета потоков первого плана после ожидания

Как далее будет описано, когда поток в процессе первого плана завершает операцию ожидания объекта ядра, ядро повышает его текущий (но не базовый) приоритет на текущее значение переменной `PsPrioritySeparation`. (Какой процесс отнести к процессам первого плана, решает система управления окнами.) Как было описано в разделе «Управление величиной кванта», в переменной `PsPrioritySeparation` отражается индекс таблицы квантов, используемый для выбора квантов для потоков первого плана. Но в данном случае значение этой переменной будет использовано в качестве значения повышения приоритета.

Смысл этого повышения заключается в улучшении скорости отклика интерактивных приложений: если дать приложениям первого плана небольшое повышение приоритета при завершении ожидания, у них повышаются шансы сразу же приступить к работе, особенно когда другие процессы с таким же базовым приоритетом могут быть запущены в фоновом режиме.

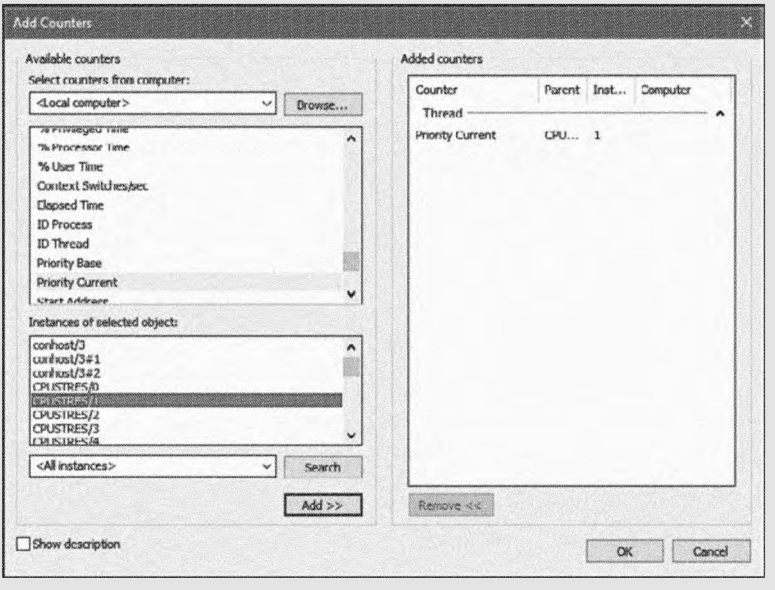
ЭКСПЕРИМЕНТ: НАБЛЮДЕНИЕ ЗА ПОВЫШЕНИЯМИ И СНИЖЕНИЯМИ ПРИОРИТЕТА ПОТОКОВ ПЕРВОГО ПЛАНА

Используя средство CPU Stress, можно вести наблюдение за повышением приоритета в действии. Выполните следующие действия.

1. Откройте оснастку Система (System) в Панели управления (Control Panel) (или щелкните правой кнопкой мыши на значке с именем вашего компьютера на рабочем столе и выберите пункт Свойства (Properties)).
2. Щелкните на пункте Дополнительные параметры системы (Advanced System Settings), перейдите на вкладку Дополнительно (Advanced), щелкните на

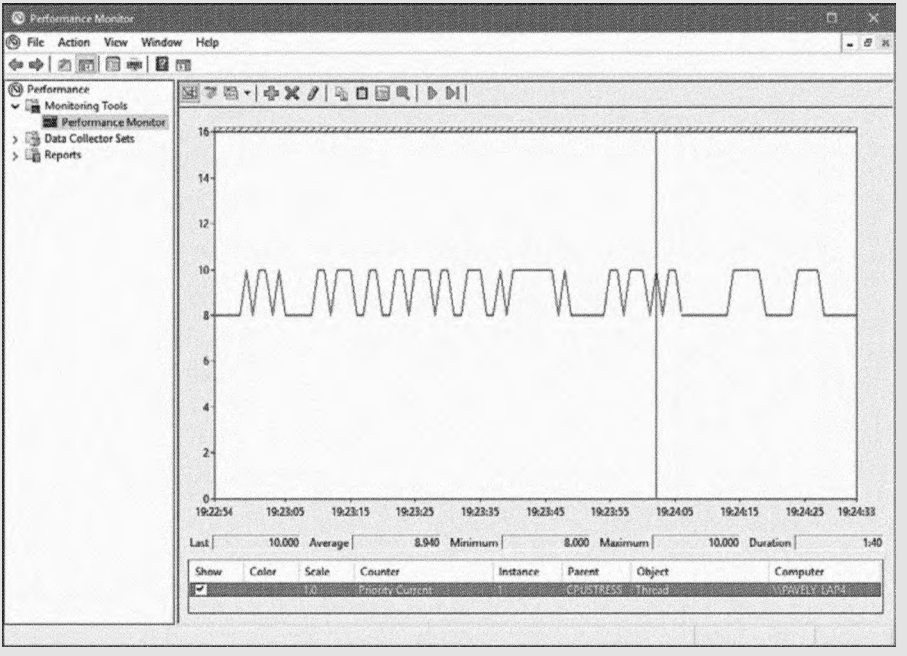
кнопке Параметры (Settings) в области Быстродействие (Performance) и, наконец, щелкните на вкладке Дополнительно (Advanced).

3. Выберите режим Оптимизировать работу программ (Programs). Это приведет к тому, что значение переменной `PsPrioritySeparation` станет равным 2.
4. Запустите программу Cpustres.exe и измените активность потока 1 (thread 1) с низкой — Low на занят — Busy.
5. Запустите Системный монитор.
6. Щелкните на кнопке панели инструментов Добавить (Add Counter) (или нажмите Ctrl+I), чтобы появилось диалоговое окно Добавить счетчики (Add Counters).
7. Выберите объект Поток (Thread), а затем выберите счетчик Текущий приоритет (Priority Current).
8. В поле со списком Экземпляры выбранного объекта (Instances) выберите пункт <все вхождения> (<All Instances>) и щелкните на кнопке Поиск (Search).
9. Прокрутите список, пока не увидите процесс CPUSTRES. Выберите второй поток (поток 1). (Первый поток является потоком GUI.) Щелкните на кнопке Добавить (Add), а затем щелкните на кнопке OK. Результат должен выглядеть примерно так:



10. Щелкните на кнопке OK.
11. Щелкните правой кнопкой мыши на счетчике и выберите команду Свойства (Properties).

12. Перейдите на вкладку Graph (График) и введите диапазон значений вертикальной шкалы 16. Щелкните на кнопке OK.
13. Переведите процесс CPUSTRES на первый план. Приоритет потока CPUSTRES должен повыситься на 2, а затем вернуться к значению базового приоритета. Причина, по которой CPUSTRES периодически получает повышения на 2, заключается в том, что отслеживаемый поток находится примерно 25 % времени в спящем режиме, а затем пробуждается. (Это уровень активности Busy.) Повышение применяется при пробуждении потока. Если установить уровень активности Maximum (Максимальный), вы не увидите никаких повышений, потому что уровень Maximum в программе CPUSTRES помещает поток в бесконечный цикл. Поэтому поток не вызывает никаких функций ожидания, в результате чего не получает никаких повышений приоритета.



14. Когда эксперимент будет завершен, выйдите из Системного монитора и из CPU Stress.

Повышение приоритета после пробуждения GUI-потока

Потоки — владельцы окон получают при пробуждении дополнительное повышение приоритета на 2 из-за активности при работе с окнами, например, при поступлении сообщений от окна. Система работы с окнами (Win32k.sys) применяет это повышение приоритета, когда вызывает функцию `KeSetEvent` для установки события,

используемого для пробуждения GUI-потока. Смысл этого повышения такой же, как у предыдущего повышения, — содействие интерактивным приложениям.

ЭКСПЕРИМЕНТ: НАБЛЮДЕНИЕ ЗА ПОВЫШЕНИЯМИ ПРИОРИТЕТА GUI-ПОТОКОВ

Отслеживая текущий приоритет GUI-приложения и передвигая указатель мыши по окну, можно также наблюдать за повышением на 2 приоритета GUI-потоков, пробуждающихся для обработки сообщений окон. Выполните следующие действия.

1. Откройте приложение Система (System) в Панели управления (Control Panel).
2. Щелкните на пункте Дополнительные параметры системы (Advanced System Settings), щелкните на вкладке Дополнительно (Advanced), щелкните на кнопке Параметры (Settings) в области Быстродействие (Performance) и, наконец, щелкните на вкладке Дополнительно (Advanced).
3. Выберите режим Оптимизировать работу программ (Programs). Это приведет к тому, что значение переменной `PsPrioritySeparation` станет равным 2.
4. Запустите программу Блокнот.
5. Запустите Системный монитор.
6. Щелкните на кнопке панели инструментов Добавить (Add Counter) (или нажмите Ctrl+I) для вызова диалогового окна Добавить счетчики (Add Counters).
7. Выберите объект Поток (Thread), а затем выберите счетчик Текущий приоритет (Priority Current).
8. В поле со списком Экземпляры выбранного объекта (Instances) введите `Notepad`, а затем щелкните на кнопке Поиск (Search).
9. Прокрутите список до строки Notepad/0. Выберите ее, щелкните на кнопке Добавить (Add), а затем щелкните на кнопке OK.
10. Как и в предыдущем эксперименте, установите диапазон значений вертикальной шкалы 16. Вы увидите приоритет потока 0 в Notepad на уровне 8 или 10. (Поскольку Notepad входит в состояние ожидания вскоре после повышения на 2, которое получают потоки процесса первого плана, он еще может не получить снижение с 10 до 8.)
11. После вывода на первый план Системного монитора переместите указатель мыши вдоль окна Блокнота. (Оба окна должны быть видны на Рабочем столе.) Вы увидите, что по только что рассмотренным причинам приоритет иногда остается на 10 единицах, а иногда на 9.

ПРИМЕЧАНИЕ Вам вряд ли удастся увидеть приоритет Блокнота на уровне 8, потому что работа после повышения приоритета GUI-потока на 2 слишком кратковременна и уровень приоритета никогда не успевает снижаться более чем на единицу, прежде чем поток не будет снова пробужден благодаря новой активности работы с окном и новому получению повышения приоритета на 2.

12. Теперь переведите Блокнот на первый план. Вы увидите, что приоритет поднялся до 12 единиц и остался на этом уровне. Это объясняется тем, что поток получает два повышения: повышение на 2, применимое к GUI-потокам при пробуждении процесса, получающего ввод при работе с окном, и дополнительное повышение на 2, поскольку Блокнот находится на первом плане. (Также он может упасть до 11, поскольку происходит снижение приоритета, обычное для потоков с повышенным приоритетом в конце кванта.)
13. Проведите указатель мыши над Блокнотом, пока окно по-прежнему находится на первом плане. Возможно, вы увидите, что приоритет упал до 11 (или даже до 10), поскольку происходит снижение приоритета, которое обычно случается с потоками с повышенным приоритетом при завершении их очереди выполняться. Но применяемое повышение приоритета на 2 единицы обусловлено тем, что процесс остается на первом плане, пока на этом же плане остается само приложение Блокнот.

Выйдите из Системного монитора и из Блокнота.

Повышения приоритета, связанные с перезагруженностью центрального процессора

Представьте себе следующую ситуацию: имеется выполняемый поток с приоритетом 7, который не дает потоку с приоритетом 4 получить процессорное время, но в то же время поток с приоритетом 11 ожидает какой-нибудь ресурс, который заблокирован потоком с приоритетом 4. Но поскольку поток с приоритетом 7 израсходовал только около половины потребляемого времени центрального процесса, процесс с приоритетом 4 никогда не будет выполняться достаточно долго, чтобы завершить свою работу и освободить ресурс, блокирующий поток с приоритетом 11. Этот сценарий называется *инверсией приоритетов*.

Что делает Windows для преодоления подобных ситуаций? Идеальное решение (по крайней мере, теоретически) будет заключаться в отслеживании блокировок и их владельцев и повышении приоритетов соответствующих потоков для продвижения работы. Эта идея реализована в функции *Autoboost* (см. далее раздел «Autoboost»). Однако для общих сценариев перегруженности применяется следующая мера.

Ранее уже было показано, как код исполняющей системы отвечает за управление ресурсами исполняющей системы при таком развитии событий путем повышения приоритета потоков-владельцев, чтобы у них был шанс на выполнение и освобождение ресурса. Но ресурсы исполняющей системы являются только одной из многих конструкций синхронизации, доступной разработчикам, и технология повышения приоритета к любым другим примитивам применяться не будет. Поэтому в Windows также включен общий механизм ослабления загруженности центрального процессора, который называется *диспетчером настройки баланса* и является частью потока (речь идет о системном потоке, который существует главным образом для выполнения функций управления памятью — эта тема более подробно рассматривается в главе 5). Один раз в секунду этот поток сканирует очередь готовых по-

токов в поиске тех из них, которые находятся в состоянии ожидания (т. е. не были запущены) около 4 секунд. Если такой поток будет найден, диспетчер настройки баланса повышает его приоритет до 15 единиц и устанавливает квантовую цель эквивалентной тактовой частоте процессора при подсчете 3 квантовых единиц. Как только квант истекает, приоритет потока тут же снижается до обычного базового приоритета. Если поток не был завершен и есть готовый к запуску поток с более высоким уровнем приоритета, поток с пониженным приоритетом возвращается в очередь готовых потоков, где он опять становится подходящим для еще одного повышения приоритета, если будет оставаться в очереди следующие 4 секунды.

Диспетчер настройки баланса на самом деле при своем запуске сканирует не все потоки, находящиеся в состоянии готовности. Для минимизации затрачиваемого на его работу времени центрального процессора он сканирует только 16 готовых потоков; если на данном уровне приоритета имеется больше потоков, он запоминает то место, на котором остановился, и начинает с него при следующем проходе очереди. Кроме того, он за один проход повысит приоритет только 10 потоков, если найдет 10 потоков, заслуживающих именно этого повышения (что свидетельствует о необычно высоко загруженной системе), он прекратит сканирование на этом месте и начнет его с этого же места при следующем проходе.

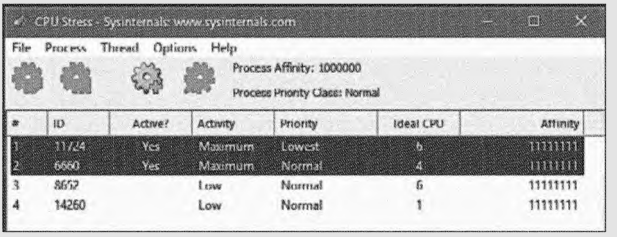
ПРИМЕЧАНИЕ Как уже ранее упоминалось, решения по планированию в Windows не подвержены влиянию со стороны количества потоков и принимаются за постоянный промежуток времени. Поскольку диспетчер настройки баланса нуждается в самостоятельном сканировании очереди готовых потоков, эта операция зависит от количества потоков, имеющихся в системе, и большее количество потоков потребует большего времени сканирования. Но диспетчер настройки баланса не считается частью планировщика или его алгоритмов, это просто расширенный механизм для повышения надежности работы системы. Кроме того, из-за ограничения количества сканируемых потоков и очередей потеря быстродействия сводится к минимуму и остается предсказуемой даже при худшем развитии событий.

ЭКСПЕРИМЕНТ: НАБЛЮДЕНИЕ ЗА ПРИОРИТЕТОМ ПОТОКОВ ДЛЯ ВЫЯВЛЕНИЯ ПЕРЕЗАГРУЖЕННОСТИ ЦЕНТРАЛЬНОГО ПРОЦЕССОРА

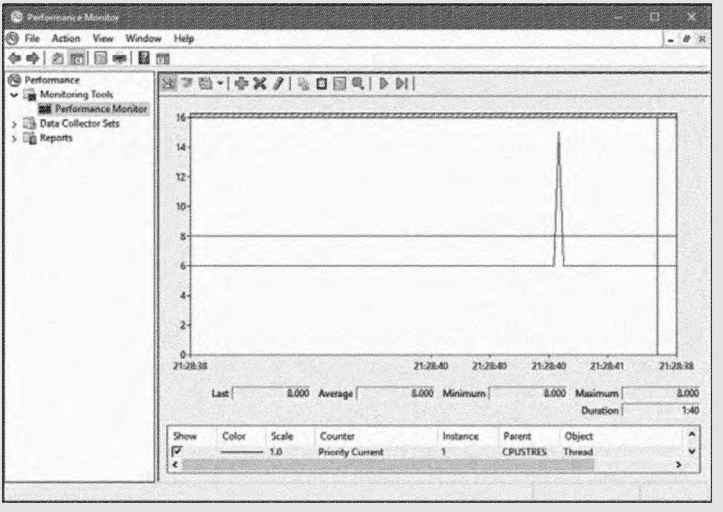
Программа CPU Stress позволяет наблюдать за повышением приоритета в действии. В данном эксперименте вы увидите изменение использования центрального процессора при повышении приоритета потока. Выполните следующие действия.

1. Запустите программу Cpustres.exe.
2. Измените уровень активности потока 1 с Low на Maximum.
3. Измените приоритет потока с Normal на Lowest.
4. Щелкните на потоке 2. Измените его уровень активности с Low на Maximum.
5. Измените маску сходства процесса на один логический процессор. Откройте меню Process и выберите команду Affinity (неважно, для какого процессора).

Также для внесения изменения можно воспользоваться диспетчером задач. Результат должен выглядеть примерно так:



6. Запустите программу Системный монитор.
7. Щелкните на кнопке панели инструментов Добавить (Add Counter) (или нажмите Ctrl+I) для вызова диалогового окна Добавить счетчики (Add Counters).
8. Выберите объект Поток (Thread), а затем выберите счетчик Текущий приоритет (Priority Current).
9. В поле со списком Экземпляры выбранного объекта (Instances) введите CPUSTRES, а затем щелкните на кнопке Поиск (Search).
10. Выберите потоки 1 и 2 (поток 0 является GUI-потоком), щелкните на кнопке Добавить (Add), а затем щелкните на кнопке OK.
11. Измените максимальный диапазон значений вертикальной шкалы на 16.
12. Так как Системный монитор обновляет данные только один раз в секунду, вы можете упустить повышение приоритета. Чтобы решить эту проблему, нажмите Ctrl+F. Если после этого нажать и удерживать Ctrl+U, обновления будут происходить чаще. Если повезет, вы увидите повышение низкоприоритетного потока до уровня 15:



13. Выйдите из Системного монитора и CPU Stress.

Применение повышений приоритета

Вернемся к работе функции KiExitDispatcher: из нее вызывается функция KiProcessThreadWaitList, чтобы обработать любые потоки в списке отложенных готовых потоков. Именно здесь обрабатывается информация о повышении приоритета, переданная со стороны вызова. Это делается путем циклического перебора каждого потока, находящегося в состоянии DeferredReady, с отсоединением его блоков ожидания (отсоединяются только блоки Active и Bypassed) и последующей установкой двух ключевых значений в находящемся в ядре блоке управления потоком: AdjustReason и AdjustIncrement. Причина заключается в одной из двух возможностей настроек (Adjust), показанных ранее, и в приращении, соответствующем значению повышения. Затем вызывается функция KiDeferredReadyThread, которая превращает поток в готовый к выполнению путем запуска двух алгоритмов: алгоритма выбора кванта и приоритета, который вы сейчас увидите, в двух частях, и алгоритма выбора процессора, который будет описан далее в соответствующем разделе.

Сначала посмотрим, когда алгоритм применяет повышение приоритета. Это происходит только в тех случаях, когда поток не относится к диапазону приоритетов реального времени. Что касается повышения приоритета, связанного с завершением ожидания (AdjustUnwait), то оно будет применено, только если поток еще не испытал на себе повышение приоритета и только если поток не был настроен на отключение повышения приоритета вызовом функции SetThreadPriorityBoost, устанавливающей флаг DisableBoost в структуре данных KTHREAD. Также повышение приоритета может быть отключено по другой причине: ядро выясняет, что поток фактически исчерпал свой квант времени (но прерывание от таймера для фиксации этого факта еще не состоялось) и вышел из состояния ожидания, которое длилось менее двух тактов таймера.

Если таких ситуаций в текущий момент не было, новый уровень приоритета потока вычисляется суммированием AdjustIncrement с текущим базовым приоритетом потока. Кроме того, если известно, что поток является частью процесса первого плана (а это означает, что для приоритета памяти установлено значение MEMORY_PRIORITY_FOREGROUND, которое настраивается Win32k.sys при изменении фокуса), то к нему применяется повышение разноса приоритетов (PsPrioritySeparation) путем добавления его значения к верхушке нового приоритета. Это действие известно также как *повышение приоритета потока первого плана*, которое было рассмотрено ранее.

И наконец, ядро проверяет, не превышает ли этот новый вычисленный уровень приоритета текущий приоритет потока, и ограничивает это значение по верхнему значению в 15 единиц, чтобы избежать перехода в диапазон реального времени. Затем значение приоритета потока устанавливается в качестве нового текущего приоритета. Если применялось повышение приоритета на величину разноса приоритета потоков первого плана, это значение устанавливается в поле ForegroundBoost структуры данных KTHREAD, в результате чего значение PriorityDecrement становится равным значению разноса приоритетов.

Для повышений приоритета, связанных с событием AdjustBoost, ядро проверяет, не является ли текущий приоритет потока ниже значения AdjustIncrement (уровень приоритета, задаваемый при настройке потока) и не находится ли текущий приоритет потока ниже уровня в 13 единиц. В этом случае повышение приоритета потока не отключается, приоритет AdjustIncrement используется в качестве нового текущего приоритета, ограниченного максимумом в 13 единиц. Одновременно с этим значение повышения получает поле UnusualBoost структуры KTHREAD, в результате чего значение PriorityDecrement становится равным значению повышения приоритета, связанного с владением блокировкой.

Во всех случаях, где играет роль значение PriorityDecrement, квант времени потока также вычисляется заново, чтобы он был равен одному такту таймера, основанного на значении переменной KiLockQuantumTarget. Тем самым гарантируется, что повышение, превышающее обычное значение, будет утрачено после одного такта таймера, а не после обычных двух (или другого настроенного значения), как будет показано в следующем разделе. Точно так же все происходит и при запросе события AdjustBoost, но поток запускается с приоритетом 13 или 14 или с отключенным повышением.

Когда эта работа завершится, значение AdjustReason устанавливается в AdjustNone.

Удаление повышений

Удаление повышений осуществляется функцией KiDeferredReadyThread при пересчете заново применяемых (как показано в предыдущем разделе) повышений и квантов. Алгоритм начинается с проверки типа выполняемой настройки.

Для сценария AdjustNone это означает, что поток стал готов к выполнению, возможно, из-за вытеснения, квант времени потока будет пересчитан, если он уже достиг своей цели, но прерывания от таймера еще не замечено, поскольку поток был запущен с динамическим уровнем приоритета. Кроме того, будет пересчитан уровень приоритета потока. Для сценария AdjustUnwait или AdjustBoost применительно к потоку, не относящемуся к потокам реального времени, ядро проверяет, не исчерпал ли поток свой квант до прерывания от таймера (точно так же, как при случае из предыдущего раздела). Если так оно и есть, или же если поток был запущен с базовым приоритетом равным 14 или выше, или же если нет переменной PriorityDecrement и поток завершил ожидание, которое продолжалось дольше двух тактов таймера, квант потока пересчитывается, и то же самое происходит с его приоритетом.

Пересчет приоритета происходит в отношении потоков, не являющихся потоками реального времени, и он осуществляется путем вычитания из текущего приоритета его повышения первого плана, вычитания повышения свыше обычного (сочетание этих двух элементов хранится в переменной PriorityDecrement) и, наконец, вычитания 1. Новый приоритет ограничивается базовым приоритетом в качестве низшей границы, и любой существующий декремент приоритета обнуляется (очищая повышения свыше обычного и повышения первого плана). Это означает, что в случае повышения, связанного с владением блокировкой, или любых рассмотренных

повышений свыше обычного значения, все значение повышения утрачивается. С другой стороны, для обычного повышения `AdjustUnwait` приоритет естественным образом снижается на единицу за счет вычитания этой единицы. Это снижение в конечном итоге останавливается, как только при проверке нижней границы будет достигнут базовый приоритет.

Есть еще один случай, когда повышение должно быть удалено с помощью функции `KiRemoveBoostThread`. Это особый случай удаления повышения, который происходит по правилу повышения, связанного с владением блокировкой; оно определяет, что настраиваемый поток должен утратить свое повышение, когда передает свой текущий приоритет пробуждающемуся потоку (чтобы избежать сопровождения блокировки). Это удаление также используется для отмены повышения приоритета из-за целевых DPC-вызовов, а также для отмены повышения, связанного с блокировкой ресурсов исполняющей системы `ERESOURCE`. Единственной особенностью этой процедуры является то, что перед вычислением нового приоритета принимаются особые меры для разделения компонентов `ForegroundBoost` и `UnusualBoost`, составляющих значение `PriorityDecrement` для поддержки любых повышений на величину разноса GUI-потоков, связанных с их выходом на первый план, которые аккумулируются потоком. Такое поведение, появившееся в Windows 7, гарантирует, что потоки, зависящие от повышения, связанного с владением блокировкой, не будут некорректно вести себя при запуске на первом плане или при возвращении на второй план.

На рис. 4.12 показан пример того, как обычные повышения удаляются из потока по истечении его кванта времени.

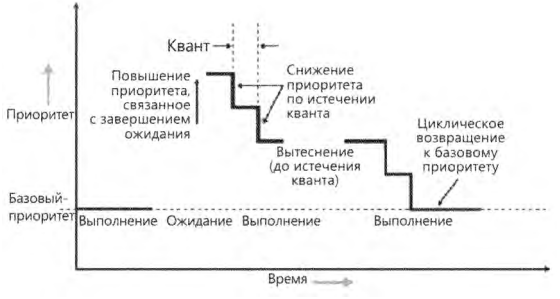


Рис. 4.12. Повышение и снижение приоритета

Повышения приоритетов для мультимедийных приложений и игр

Как было видно из последнего эксперимента, хотя повышения приоритета, связанного с перезагруженностью центрального процессора со стороны Windows, может быть достаточно для выхода потока из аномально долгого состояния ожидания или из потенциальной взаимной блокировки, это повышение просто не может справиться с требованиями к ресурсам, предъявляемыми такими приложениями, как Windows Media Player или компьютерная 3D-игра.

Перескакивания и другие звуковые дефекты были в прошлом обычным источником раздражения со стороны пользователей Windows, а имеющийся в Windows аудиостек пользовательского режима только усугублял ситуацию, поскольку он еще больше повышал шансы на вытеснение потоков. Для решения этой проблемы клиентские версии Windows используют драйвер MMCSS (см. ранее в этой главе), реализованный в %SystemRoot%\System32\Drivers\MMCSS.sys. Этот драйвер предназначен для воспроизведения мультимедийного контента приложений, зарегистрированных с ним, без каких-либо сбоев.

ПРИМЕЧАНИЕ В Windows 7 MMCSS реализуется в виде службы (а не в виде драйвера). Тем не менее такое решение создает потенциальный риск. Если управляющий поток MMCSS по какой-то причине блокируется, то управляемые им потоки сохранят свои приоритеты реального времени, что теоретически приведет к зависанию на системном уровне. Проблема была решена перемещением кода в ядро, где управляющий поток (и другие ресурсы, используемые MMCSS) будет защищен от вмешательства. У драйвера ядра есть и другие преимущества — например, возможность хранения прямого указателя на объекты процесса и потоков (вместо идентификаторов и дескрипторов). Тем самым устраняются потери времени на поиск по идентификаторам или дескрипторам и ускоряется обмен данными с Планировщиком и диспетчером питания.

Клиентские приложения регистрируются у MMCSS, вызывая функцию `AvSetMmThreadCharacteristics` с именем задачи, которое должно соответствовать одному из подразделов HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Multimedia\SystemProfile\Tasks. (OEM-производители могут изменять этот список, включая в него необходимые задачи.) В исходном варианте существуют следующие задачи:

- аудио;
- захват;
- распределение;
- игры;
- низкая задержка;
- воспроизведение;
- аудио профессионального качества;
- задачи администратора многооконного режима.

Каждая из этих задач включает информацию о свойствах, отличающих их друг от друга. Одно из наиболее важных свойств для планирования потоков называется *категорией планирования* (Scheduling Category) — это основной фактор, определяющий приоритет потоков, зарегистрированных с MMCSS. В табл. 4.4 перечислены различные категории планирования.

Таблица 4.4. Категории планирования

Категория	Приоритет	Описание
High (Высокая)	23–26	Потоки Pro Audio, выполняемые с приоритетом выше, чем у других потоков в системе, за исключением критических системных потоков
Medium (Средняя)	16–22	Потоки, являющиеся частью приложений первого плана, например Windows Media Player
Low (Низкая)	8–15	Все остальные потоки, не являющиеся частью предыдущих категорий
Exhausted (Исчерпавших потоков)	1–7	Потоки, исчерпавшие свою долю времени центрального процессора, выполнение которых продолжится, только если не будут готовы к выполнению другие потоки с более высоким уровнем приоритета

Механизм, положенный в основу MMCSS, повышает приоритет потоков внутри зарегистрированного процесса до уровня, соответствующего их категории планирования и относительного приоритета внутри этой категории на гарантированный период. Затем он снижает категорию этих потоков до Exhausted, чтобы другие, не относящиеся к мультимедийным приложениям потоки также получили шанс на выполнение.

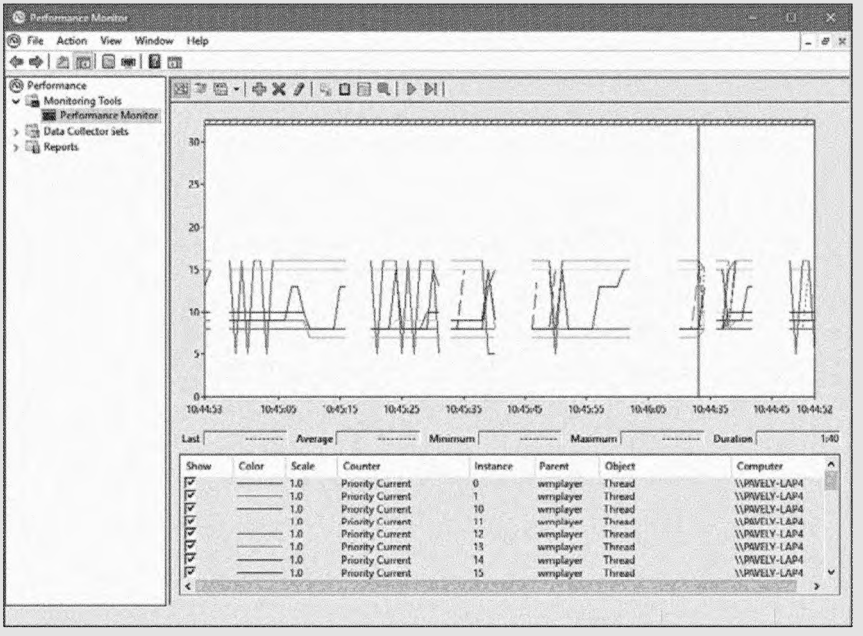
По умолчанию мультимедийные потоки получают 80 % доступного времени центрального процессора, а другие потоки получают 20 % этого времени (если взять в качестве примера 10 мс, это будет 8 мс и 2 мс соответственно). Чтобы изменить это распределение в процентах, измените параметр реестра `SystemResponsiveness` в разделе HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Multimedia\SystemProfile. Значение может изменяться в диапазоне от 10 до 100 % (значение 20 используется по умолчанию; при присваивании значения меньше 10 устанавливается значение 10); оно указывает, какой процент процессорного времени гарантируется системе (а не зарегистрированным аудиоприложениям). Поток планирования MMCSS выполняется с приоритетом 27, поскольку ей нужно вытеснять любые Pro Audio-потоки с целью снижения их приоритета до категории Exhausted.

Как упоминалось ранее, изменять относительные приоритеты потоков внутри процесса обычно бессмысленно, и средств, позволяющих это сделать, не существует, поскольку только разработчики понимают важность различных потоков в их программах. С другой стороны, поскольку приложения должны самостоятельно регистрироваться со службой MMCSS и снабжать эту службу информацией о том, с какого рода потоком она имеет дело, у MMCSS есть необходимые данные для изменения этих относительных приоритетов потоков (и разработчики хорошо осведомлены о том, что это произойдет).

ЭКСПЕРИМЕНТ: ПОВЫШЕНИЕ ПРИОРИТЕТОВ MMCSS

В этом эксперименте вы понаблюдаете за повышением приоритетов MMCSS.

1. Запустите Windows Media Player (wmplayer.exe). (Другие программы проигрывания мультимедиа могут не пользоваться вызовами API-функций, которые требуют регистрации с MMCSS.)
2. Начните воспроизведение какого-нибудь звукового контента.
3. При помощи диспетчера задач или Process Explorer установите сходство процесса Wmplayer.exe, чтобы он выполнялся только на одном процессоре.
4. Запустите программу Системный монитор.
5. В диспетчере задач измените класс приоритета Системного монитора на Realtime, чтобы у процесса было больше шансов для выполнения записи.
6. Щелкните на кнопке панели инструментов Добавить (Add Counter) (или нажмите Ctrl+I), чтобы появилось диалоговое окно Добавить счетчики (Add Counters).
7. Выберите объект Поток (Thread), а затем выберите счетчик Текущий приоритет (Priority Current).
8. В поле со списком Экземпляры выбранного объекта (Instances) введите `Wmplayer`, щелкните на кнопке Поиск (Search) и выберите все потоки.
9. Нажмите Добавить (Add) и OK.
10. Откройте меню Действие (Action) и выберите команду Свойства (Properties).
11. На вкладке График (Graph) установите диапазон значений вертикальной шкалы 32. Вы увидите в Wmplayer один или несколько потоков с приоритетом 16, которые будут постоянно выполняться, если не появится высокоприоритетный поток, требующий времени центрального процессора после того, как они будут опущены до категории Exhausted.
12. Запустите программу CPU Stress.
13. Установите уровень активности потока 1 на максимальный (Maximum).
14. Поднимите приоритет потока 1 с обычного (Normal) до критичного по времени (Time Critical).
15. Измените класс приоритета CPUSTRES на High.
16. Измените сходство CPUSTRES так, чтобы программа использовала тот же процессор, что и Wmplayer. Работа системы существенно замедлится, но воспроизведение музыки продолжится. Время от времени вы сможете получать отклик от других частей системы.
17. В Системном мониторе обратите внимание на то, что потоки Wmplayer с приоритетом 16 время от времени теряют приоритет:



Впрочем, функциональные возможности MMCSS не ограничиваются простым повышением приоритета. Из-за особенностей сетевых драйверов Windows и NDIS-стека отложенные вызовы процедур (DPC) являются довольно распространенным механизмом для задержки работы после получения прерывания от сетевой карты. Так как DPC-вызовы запускаются на IRQL-уровне выше, чем у кода пользовательского режима (см. главу 6), долго работающий код драйвера сетевой карты по-прежнему может прервать проигрывание мультимедиа при сетевых передачах данных или, к примеру, во время игры.

Поэтому MMCSS также отправляет специальную команду сетевому стеку, предписывая ему регулировать сетевые пакеты, пока идет воспроизведение медиаконтента. Это регулирование разработано для максимальной производительности воспроизведения ценой небольших потерь в пропускной способности сети (которая будет незаметна для сетевых операций, обычно осуществляемых во время воспроизведения, например в онлайновой игре). Положенные в основу этой работы конкретные механизмы не имеют никакого отношения к планировщику, поэтому мы их рассматривать не будем.

MMCSS также поддерживает *планирование с критическим сроком*. Идея заключается в том, что программам воспроизведения аудиоконтента не всегда нужен самый высокий уровень приоритета в своей категории. Если такая программа использует буферизацию (получение звуковых данных с диска или из сети) с последующим воспроизведением буфера во время построения следующего буфера, планирование с критическим сроком позволяет клиентскому потоку обозначить время, когда

он должен получить высокий уровень приоритета для предотвращения сбоев, но в настоящее время ему хватает более низкого приоритета (в пределах своей категории). Поток может использовать функцию `AvTaskIndexYield` для обозначения следующего момента времени, когда он должен получить наивысший приоритет в пределах своей категории. До наступления этого момента он получает низший приоритет в своей категории, а система (по крайней мере, теоретически) получает больше процессорного времени.

Autoboost

Инфраструктура Autoboost предназначена для решения проблемы инверсии приоритетов, описанной в предыдущем разделе. Идея заключается в отслеживании владельцев и сторон, ожидающих блокировки, которое позволяло бы повышать приоритеты соответствующих потоков (а также приоритет ввода/вывода при необходимости), чтобы потоки могли двигаться дальше. Информация о блокировках хранится в статическом массиве объектов `KLOCK_ENTRY` в структуре `KTHREAD`. Текущая реализация позволяет использовать не более шести элементов. Каждый объект `KLOCK_ENTRY` содержит два бинарных дерева: для блокировок, принадлежащих потоку, и для блокировок, которые ожидает поток. Эти деревья структурируются по приоритету, чтобы определение наибольшего приоритета, к которому должно применяться повышение, выполнялось за постоянное время. Если повышение необходимо, приоритету ожидающей стороны присваивается приоритет владельца. Он также может повысить приоритет операций ввода/вывода, если они запускаются с низким приоритетом. (За информацией о приоритете ввода/вывода обращайтесь к главе 6.) Как и при всех повышениях приоритета, максимальный приоритет, достигаемый при использовании Autoboost, равен 15. (Приоритет потоков реального времени никогда не повышается.)

Текущая реализация использует Autoboost для push-блокировок и защищенных мьютексов, доступных только для кода ядра. (Подробнее об этих объектах см. в главе 8 части 2.) Autoboost также используется некоторыми компонентами исполняющей среды в особых случаях. Возможно, в будущих версиях Windows Autoboost будет использоваться для объектов, доступных в пользовательском режиме и поддерживающих концепцию принадлежности (например, критических секций).

Переключения контекста

Контекст потока и процедура для переключения контекста сильно зависят от архитектуры процессора. Обычное переключение контекста требует сохранения и перезагрузки следующих данных:

- указателя команд;
- указателя стека ядра;
- указателя на адресное пространство, в котором выполняется поток (каталог таблицы страниц процесса).

Ядро сохраняет эту информацию от старого потока, помещая ее в текущий (принадлежащий старому потоку) стек режима ядра, обновляя этот указатель стека и сохраняя указатель стека в структуре KTHREAD старого потока. Затем указатель стека ядра устанавливается на стек ядра нового потока, и загружается контекст нового потока. Если новый поток принадлежит другому процессу, он загружает свой адрес каталога таблицы страниц в специальный регистр процессора, чтобы его адресное пространство стало доступным. (Преобразование адресов описано в главе 5.) Если APC-вызов ядра, который нуждается в доставке, задерживается, запрашивается прерывание уровня IRQL (см. главу 8 части 2). В противном случае управление передается восстановленному указателю команд нового потока, и новый поток возобновляет выполнение.

Direct Switch

В Windows 8 и Server 2012 появился механизм оптимизации *Direct Switch*. Он позволяет потоку передать свой квант и повышение другому потоку, который немедленно планируется для выполнения на том же процессоре. В синхронных сценариях «клиент/сервер» он позволяет достигнуть существенного повышения быстродействия, потому что потоки не переносятся на другие процессы, которые могут простаивать или быть приостановленными (parked). Также на происходящее можно взглянуть под другим углом: в любой момент времени выполняется либо клиентский, либо серверный поток, поэтому планировщику потоков следует рассматривать их как один логический поток. На рис. 4.13 показан эффект использования Direct Switch.

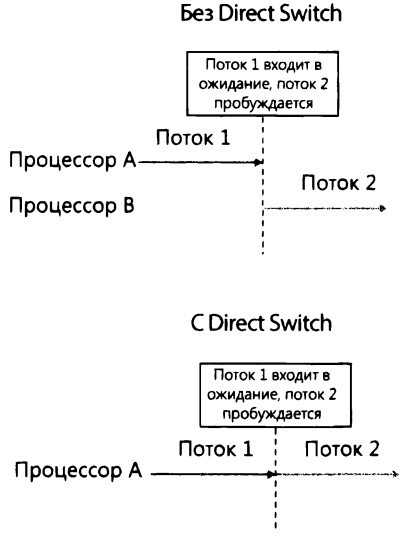


Рис. 4.13. Direct Switch

Планировщик не может знать, что первый поток (Поток 1 на рис. 4.13) собирается войти в состояние ожидания после подачи сигнала по объекту синхронизации, которого ожидает второй поток (Поток 2). Следовательно, должна быть вызвана специальная функция, которая сообщит планировщику о происходящем (атомарная подача сигнала и ожидание).

Функция `KiDirectSwitchThread` выполняет фактическое переключение, если это возможно. Она вызывается функцией `KiExitDispatcher`, если последняя получит флаг, приказывающий использовать Direct Switch (также, если это возможно). Передача приоритета, при которой приоритет первого потока «жертвуется» второму потоку (если приоритет второго ниже, чем у первого), применяется при передаче еще одного флага `KiExitDispatcher`. В текущей реализации эти два флага всегда задаются вместе (или не задается ни один из них); это означает, что при любой попытке применения Direct Switch также делается попытка передачи приоритета. Попытка использования Direct Switch может завершиться неудачей, например, если сходство целевого потока не позволяет ему выполняться на текущем процессоре. Но если она завершается успехом, то квант первого потока передается целевому потоку, а первый поток теряет свой оставшийся квант.

В настоящее время Direct Switch используется в следующих ситуациях:

- если поток вызывает API-функцию `SignalObjectAndWait` (или эквивалентную функцию ядра `NtSignalAndWaitForSingleObject`);
- ALPC (см. главу 8 части 2);
- синхронные вызовы удаленных процедур (RPC);
- удаленные вызовы COM (в настоящее время только между двумя MTA (многопоточные подразделения, Multithreaded Apartment)).

Сценарии планирования

В Windows ответ на вопрос «Кто получит центральный процессор?» основывается на приоритете потока, но как такой подход работает на практике? В следующих разделах будет показано, как именно на уровне потоков работает вытесняющая многозадачность, управляемая приоритетами.

Самостоятельное переключение

Сначала следует заметить, что поток может добровольно отказаться от использования процессора путем входа в состояние ожидания какого-нибудь объекта (например, события, мьютекса, семафора и порта завершения ввода/вывода, процесса, потока, сообщения окна и т. п.) путем вызова одной из Windows-функций ожидания (`WaitForSingleObject` или `WaitForMultipleObjects`). (Ожидание объектов более подробно рассматривается в главе 8 части 2.)

На рис. 4.14 показан поток, входящий в состояние ожидания, и выбор Windows нового потока для выполнения. На этом рисунке верхний блок (поток) самостоятельно уступает процессор, позволяя запуститься следующему потоку из очереди готовых потоков (что показано имеющимся над ним кольцом, полученным при переходе в колонку выполняемых потоков). Хотя при изучении данного рисунка может показаться, что приоритет уступившего процессор потока снизился, на самом деле это не так. Поток просто был перемещен в очередь ожидания того объекта, которого он ожидает.

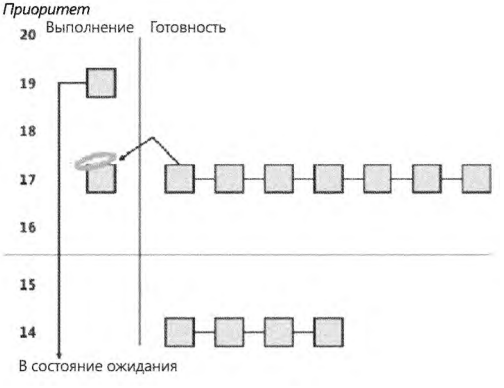


Рис. 4.14. Самостоятельное переключение

Вытеснение

В этом сценарии планировки поток с более низким приоритетом вытесняется, когда становится готовым к выполнению поток с более высоким приоритетом. Такая ситуация может сложиться по двум причинам.

- Поток с более высоким приоритетом завершил ожидание. (Произошло ожидаемое им событие.)
- Произошло повышение или снижение приоритета потока.

В любом из этих случаев Windows должна определить, должен ли текущий выполняемый поток продолжить свое выполнение или он должен быть вытеснен, позволив выполняться потоку с более высоким приоритетом.

ПРИМЕЧАНИЕ Потоки, выполняемые в пользовательском режиме, могут вытеснять потоки, работающие в режиме ядра. Режим, в котором работает поток, роли не играет: определяющим фактором является приоритет потока.

Когда поток вытесняется, он помещается в начало очереди готовых потоков для того уровня приоритета, с которым он выполнялся. Эта ситуация показана на рис. 4.15.

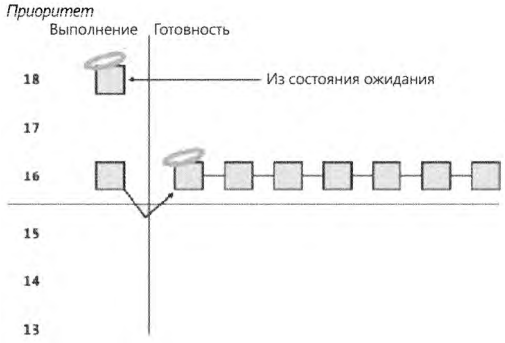


Рис. 4.15. Планирование с вытеснением потоков

На рис. 4.15 поток с уровнем приоритета 18 выходит из состояния ожидания и снова получает в свое распоряжение центральный процессор, выталкивая выполняемый поток (с уровнем приоритета 16) в начало очереди готовых потоков. Следует заметить, что вытолкнутый поток помещается не в конец, а в начало очереди; когда вытеснивший его поток завершит свое выполнение, вытолкнутый поток может завершить свой квант выполнения.

Истечение кванта времени

Когда у выполняемого потока истекает квант времени, Windows должна определить, нужно ли снижать приоритет потока, а затем определить, следует ли спланировать выполнение на процессоре другого потока.

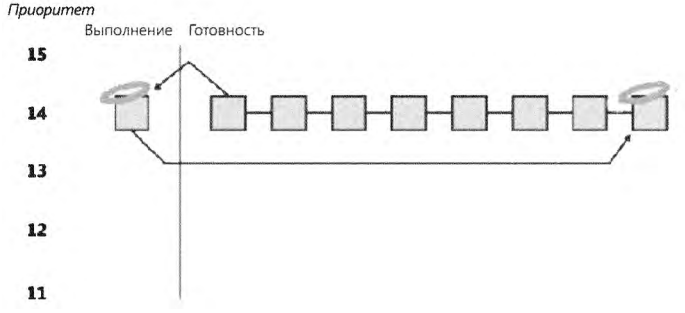


Рис. 4.16. Планирование потоков при истечении кванта времени

Если приоритет потока снижается, Windows ищет для планирования выполнения самый подходящий поток. (Например, наиболее подходящим может быть поток в очереди готовых потоков, имеющих более высокий уровень приоритета, чем новый приоритет для текущего выполняемого потока.) Если приоритет потока не снижается и в очереди готовых потоков имеются потоки с таким же уровнем приоритета, Windows выбирает следующий поток из очереди готовых потоков с таким же уровнем приоритета и перемещает ранее выполнявшийся поток в конец очереди

(задавая ему новое значение кванта и изменяя его состояние с выполняющегося на готовое). Этот случай показан на рис. 4.16. Если другого готового к выполнению потока с таким же уровнем приоритета не имеется, поток получает для выполнения еще один квант времени.

Как уже было показано, вместо того чтобы просто полагаться при планировании выполнения потоков на квант времени, основанный на работе интервального таймера, Windows использует для ведения квантовых целей точный счетчик тактовых циклов центрального процессора. Windows также использует этот счетчик для определения того, применимо ли в данном случае истечение кванта времени к потоку.

Использование модели планирования, которая зависит только от интервального таймера, может привести к следующим ситуациям.

- Потоки А и Б стали готовы к выполнению в середине интервала. (Код планирования запускается не по каждому интервалу времени, поэтому такое часто случается.)
- Поток А начал выполняться, но его выполнение на какое-то время было прервано. Время, затраченное на обработку прерывания, потоку не возмещается.
- Обработка прерывания завершается, и поток А снова запускается на выполнение, но быстро достигает следующего интервала таймера. Планировщик может только предположить, что поток А выполнялся все это время, и теперь переключается на выполнение потока Б.
- Поток Б начинает выполняться и имеет шанс на выполнение в течение полного интервала таймера (если не брать в расчет вытеснение или обработку прерывания).

При таком сценарии развития событий поток А был незаслуженно удален двумя различными способами. Сначала время, затраченное на обработку прерывания от устройства, было зачтено как его собственное время использования центрального процессора — даже при том, что поток, возможно, не имеет никакого отношения к прерыванию. (Следует напомнить, что прерывания обрабатываются в контексте того потока, который выполнялся на данный момент, — см. главу 6.) Он также был незаслуженно наказан на то время, пока система простаивала внутри интервала таймера, до того как он был снова спланирован на выполнение. Такая ситуация показана на рис. 4.17.

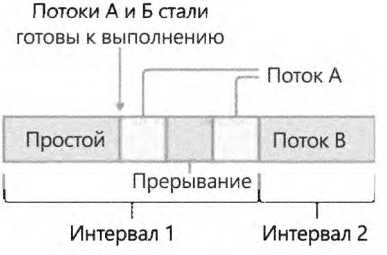


Рис. 4.17. Несправедливое квантование времени в предыдущих версиях Windows

Поскольку Windows ведет точный подсчет количества тактовых циклов центрального процессора, затраченных на выполнение той работы, которая была спланирована для потока (без учета прерываний), и поскольку она хранит квантовую цель в тактовых циклах, которые должны быть потрачены потоком до завершения его кванта времени, оба несправедливых решения, которые могли бы быть приняты в отношении потока А в Windows, не принимаются. Вместо этого происходит следующее.

- Потоки А и Б стали готовы к выполнению в середине интервала.
- Поток А начал выполняться, но его выполнение на какое-то время было прервано. Тактовые циклы центрального процессора, затраченные на обработку прерывания, потоку не возмещаются.
- Обработка прерывания завершается, и поток А снова запускается на выполнение, но быстро достигает следующего интервала таймера. Планировщик смотрит на количество тактовых циклов центрального процессора, выделенных потоку, и сравнивает его с ожидаемым количеством тактовых циклов центрального процессора, которые должны были быть выделены до конца кванта времени.
- Поскольку предшествующее количество намного меньше того, каким оно должно быть, Планировщик предполагает, что поток А начал выполняться в середине интервала таймера и, кроме того, его выполнение могло быть прервано.
- Поток А получает повышение кванта своего времени на еще один интервал таймера, и квантовая цель пересчитывается. Теперь у потока А есть шанс выполняться в течение полного интервала таймера.
- В следующий интервал таймера поток А выберет свой квант времени, и теперь шанс на выполнение получит поток Б.

Эта ситуация представлена на рис. 4.18.

Рис. 4.18. Справедливое квантование времени в современных версиях Windows

Завершение выполнения потока

Когда поток завершает свое выполнение (либо по причине возвращения из своей основной процедуры, называемой `ExitThread`, либо по причине его уничтожения функцией `TerminateThread`), он переходит из состояния выполнения в состояние

завершения. Если у объекта потока нет открытых дескрипторов, поток удаляется из списка потоков процесса, а связанные с ним структуры данных освобождают выделенные им ресурсы.

Потоки простоя

Когда у центрального процессора нет потоков, готовых к выполнению, Windows запускает на этом центральном процессоре поток простоя. У каждого центрального процессора есть свой собственный, выделенный поток простоя, поскольку на многопроцессорных системах один центральный процессора может выполнять поток, в то время как у другого центрального процессор может не быть потоков для выполнения. Каждый поток простоя центрального процессора находится через указатель в `PRCB`-блоке этого процессора.

Все потоки простоя принадлежат процессу простоя. Во многих отношениях процесс простоя и потоки простоя являются особым случаем. Они, конечно же, представлены структурами `EPROCESS/KPROCESS` и `ETHREAD/KTHREAD`, но не являются процессами диспетчера исполняющей системы и объектами потоков. Процесс простоя также не входит в список системных процессов. (Именно поэтому он не появляется в выводе команды отладчика ядра `!process 0 0`.) Но поток или потоки простоя и их процесс можно обнаружить другими способами.

ЭКСПЕРИМЕНТ: ВЫВОД СТРУКТУРЫ ПОТОКОВ ПРОСТОЯ И ИХ ПРОЦЕССА

Структуры потока и процесса простоя могут быть найдены в отладчике ядра с помощью команды `!pcr` (сокращение от «processor control region», т. е. *область управления процессором*). Эта команда выводит поднабор информации из PCR, а также из связанного с этой областью блока PRCB (PRocessor Control Block). Команда `!pcr` получает один числовой аргумент с номером процессора, чья область PCR будет выведена. Загрузочным процессором является процессор с номером 0; он всегда присутствует в системе, поэтому команда `!pcr 0` работает всегда. Ниже показан результат запуска этой команды в виде дампа памяти, взятого из 64-разрядной, восьмипроцессорной системы:

```
lkd> !pcr
KPCR for Processor 0 at fffff80174bd0000:
    Major 1 Minor 1
    NtTib.ExceptionList: fffff80176b4a000
        NtTib.StackBase: fffff80176b4b070
       NtTib.StackLimit: 000000000108e3f8
     NtTib.SubSystemTib: fffff80174bd0000
          NtTib.Version: 0000000074bd0180
      NtTib.UserPointer: fffff80174bd07f0
          NtTib.SelfTib: 00000098af072000

                SelfPcr: 0000000000000000
                   Prcb: fffff80174bd0180
```

```
                Irql: 0000000000000000
                 IRR: 0000000000000000
                 IDR: 0000000000000000
         InterruptMode: 0000000000000000
                 IDT: 0000000000000000
                 GDT: 0000000000000000
                 TSS: 0000000000000000

         CurrentThread: ffffb882fa27c080
            NextThread: 0000000000000000
            IdleThread: fffff80174c4c940

              DpcQueue:
```

Из этого вывода видно, что на время получения дампа памяти CPU 0 выполнял процесс, отличный от своего процесса простоя, поскольку указатели `CurrentThread` и `IdleThread` имеют разные значения. (Если у вас многопроцессорная система, можно попробовать запустить команды `!pcr 1`, `!pcr 2` и т. д., до тех пор пока не закончатся процессоры, наблюдая за тем, что у всех указатели `IdleThread` имеют разные значения.)

Теперь введите команду `!thread` с показанным адресом потока простоя:

```
lkd> !thread fffff80174c4c940
THREAD fffff80174c4c940 Cid 0000.0000 Teb: 0000000000000000 Win32Thread:
0000000000000000 RUNNING on processor 0
Not impersonating
DeviceMap                 ffff800a52e17ce0
Owning Process            fffff80174c4b940         Image: Idle
Attached Process          ffffb882e7ec7640         Image: System
Wait Start TickCount      1637993         Ticks: 30 (0:00:00:00.468)
Context Switch Count      25908837        IdealProcessor: 0
UserTime                  00:00:00.000
KernelTime                05:51:23.796
Win32 Start Address nt!KiIdleLoop (0xfffff801749e0770)
Stack Init fffff80176b52c90 Current fffff80176b52c20
Base fffff80176b53000 Limit fffff80176b4d000 Call 0000000000000000
Priority 0 BasePriority 0 PriorityDecrement 0 IoPriority 0 PagePriority 5
```

Наконец, воспользуйтесь командой `!process` в отношении адреса, указанного для процесса-владельца — «Owning Process», показанного в предыдущем выводе. Для краткости мы добавим еще один аргумент со значением 3, который заставит команду `!process` вывести только минимум информации для каждого потока:

```
lkd> !process fffff80174c4b940 3
PROCESS fffff80174c4b940
    SessionId: none Cid: 0000 Peb: 00000000 ParentCid: 0000
    DirBase: 001aa000 ObjectTable: ffff800a52e14040 HandleCount: 2011.
    Image: Idle
    VadRoot ffffb882e7e1ae70 Vads 1 Clone 0 Private 7. Modified 1627. Locked 0.
    DeviceMap 0000000000000000
    Token                             ffff800a52e17040
    ElapsedTime                       07:07:04.015
```

```
    UserTime                            00:00:00.000
    KernelTime                          00:00:00.000
    QuotaPoolUsage[PagedPool]           0
    QuotaPoolUsage[NonPagedPool]        0
    Working Set Sizes (now,min,max)  (7, 50, 450) (28KB, 200KB, 1800KB)
    PeakWorkingSetSize                  1
    VirtualSize                         0 Mb
    PeakVirtualSize                     0 Mb
    PageFaultCount                      2
    MemoryPriority                      BACKGROUND
    BasePriority                        0
    CommitCharge                        0

        THREAD fffff80174c4c940 Cid 0000.0000 Teb: 0000000000000000
Win32Thread: 0000000000000000 RUNNING on processor 0
        THREAD ffff9d81e230ccc0 Cid 0000.0000 Teb: 0000000000000000
Win32Thread: 0000000000000000 RUNNING on processor 1
        THREAD ffff9d81e1bd9cc0 Cid 0000.0000 Teb: 0000000000000000
Win32Thread: 0000000000000000 RUNNING on processor 2
        THREAD ffff9d81e2062cc0 Cid 0000.0000 Teb: 0000000000000000
Win32Thread: 0000000000000000 RUNNING on processor 3
        THREAD ffff9d81e21a7cc0 Cid 0000.0000 Teb: 0000000000000000
Win32Thread: 0000000000000000 RUNNING on processor 4
        THREAD ffff9d81e22ebcc0 Cid 0000.0000 Teb: 0000000000000000
Win32Thread: 0000000000000000 RUNNING on processor 5
        THREAD ffff9d81e2428cc0 Cid 0000.0000 Teb: 0000000000000000
Win32Thread: 0000000000000000 RUNNING on processor 6
        THREAD ffff9d81e256bcc0 Cid 0000.0000 Teb: 0000000000000000
Win32Thread: 0000000000000000 RUNNING on processor 7
```

Эти адреса процесса и потоков также могут использоваться в командах `dt nt!_EPROCESS`, `dt nt!_KTHREAD` и других подобных командах.

Этот эксперимент показывает ряд аномалий, связанных с процессом простоя и его потоками. Отладчик показывает для него имя образа Idle, которое берется из компонента `ImageFileName` структуры `EPROCESS`. Но различные утилиты Windows в своих отчетах показывают, что процесс простоя использует разные имена. В диспетчере задач и Process Explorer используется имя `System Idle Process`, а в Tlist — имя `System Process`. Идентификатор процесса и идентификаторы потоков (которые в выводе отладчика фигурируют как клиентские идентификаторы, или Cid) равны нулю, так же как указатели на PEB и TEB; есть еще множество других полей в процессе простоя или в его потоках, для которых выводятся нулевые значения. Это происходит потому, что процесс простоя не имеет адресного пространства режима пользователя и его потоки не выполняют кода в режиме пользователя, поэтому им и не нужны различные данные, необходимые для управления средой пользовательского режима. Кроме того, процесс простоя не представлен объектом процесса диспетчера объектов. Вместо этого исходные структуры потока простоя и процесса простоя размещаются в памяти статически и используются для начальной загрузки системы перед тем, как будут инициализированы диспетчер процессов и диспетчер

объектов. Последующие структуры потока простоя размещаются динамически (простым выделением памяти из невыгружаемого пула, в обход диспетчера объектов), как только в дело вступают дополнительные процессоры. После инициализации управления процессами для ссылки на процесс простоя используется специальная переменная `PsIdleProcess`.

Наверное, самой интересной аномалией, относящейся к процессу простоя, является то, что Windows сообщает, что у процесса простоя уровень приоритета равен 0. Но в действительности у приоритетов потоков простоя нет значения, потому что такие потоки выбираются для диспетчеризации, только если нет никаких других потоков, готовых к выполнению. Их приоритет никогда не сравнивается с приоритетами других потоков, не используется для помещения потока простоя в очередь готовых потоков, потоки простоя никогда не стоят ни в каких очередях готовых потоков. (Запомните, только один поток в каждой системе Windows действительно выполняется с приоритетом 0 — это поток обнуления страниц, упоминаемый в главе 5.)

Потоки простоя — не только особый случай в понятиях выбора на выполнение, они также являются особыми случаями для вытеснения. Процедура потоков простоя `KiIdleLoop` выполняет ряд операций, которые препятствуют его вытеснению другими потоками в обычном порядке. Когда на процессоре нет готовых к выполнению потоков, не относящихся к потокам простоя, этот процессор помечается в PRCB как простаивающий. После этого, если поток выбран для выполнения на простаивающем процессоре, адрес потока хранится в указателе следующего потока `NextThread` в PRCB простаивающего процессора. Поток простоя проверяет этот указатель при каждом проходе своего цикла.

Хотя кое-какие детали в последовательности варьируются от архитектуры к архитектуре (это один из блоков кода, написанных на языке ассемблера, а не на C), основная последовательность операций потока простоя выглядит так.

1. Прерывания разрешаются на короткий промежуток времени, позволяющий доставить любые отложенные прерывания, после чего они снова запрещаются (инструкциями STI и CLI — на процессорах x86 и x64). Наличие этой операции обусловлено тем, что значительная часть выполнения потока простоя происходит с отключенными прерываниями.

2. В отладочных версиях некоторых архитектур проверяется, не пытается ли отладчик ядра вклиниться в систему. Если проверка дает положительный результат, ему предоставляется соответствующий доступ.

3. Поток простоя проверяет наличие на процессоре отложенных вызовов процедур DPC, рассмотренных в главе 6. Незавершенные DPC-вызовы могут существовать в том случае, если при постановке их в очередь не было сгенерировано DPC-прерывание. При наличии незавершенных DPC-вызовов цикл простоя вызывает для их доставки функцию `KiRetireDpcList`. Это же действие инициирует истечение интервала таймера, а также обработку отложенных готовых потоков; последнее действие рассматривается в следующем далее разделе, посвященном планированию на многопроцессорных системах.

Вход в функцию `KiRetireDpcList` должен быть осуществлен при отключенных прерываниях, именно поэтому прерывания остаются отключенными после выполнения действия 1. При выходе из функции `KiRetireDpcList` прерывания также остаются отключенными.

4. Поток простоя проверяет, была ли запрошена обработка завершения кванта. Если проверка дает положительный результат, вызывается функция `KiQuantumEnd` для обработки запроса.
5. Поток простоя проверяет, был ли выбран следующий поток для выполнения на процессоре. Если такой поток есть, выполняется его диспетчеризация. Например, это может произойти, если обрабатываемый на шаге 3 DPC-вызов или истечение времени таймера удовлетворили условия ожидающего потока или если другой процессор выбрал поток на выполнение для этого процессора, который в этот момент уже отрабатывал цикл простоя.
6. Поток простоя проверяет наличие потоков, готовых к выполнению на других процессорах, и, если это возможно, выполняет локальное планирование работы одного из них. (Эта операция рассматривается в следующем далее разделе «Планировщик простоя».)
7. Вызывается зарегистрированная процедура управления электропитанием простаивающего процессора (в случае необходимости выполнения любых функций управления электропитанием), которая находится либо в драйвере электропитания процессора (таком, как intelppm.sys), либо в HAL, если такой драйвер недоступен.

Приостановка потока

Явная приостановка и возобновление работы потоков осуществляются вызовами API-функций `SuspendThread` и `ResumeThread` соответственно. У каждого потока есть счетчик приостановок, который увеличивается при приостановке и уменьшается при возобновлении. Если счетчик равен 0, поток может выполняться; в противном случае он выполняться не будет.

Приостановка осуществляется постановкой в очередь APC-вызова ядра. Когда поток переключается на выполнение, сначала выполняется APC. Поток переходит в состояние ожидания для события, которое инициируется при возобновлении работы.

У такого механизма приостановки имеется очевидный недостаток, если поток находится в состоянии ожидания при поступлении запроса на приостановку; это означает, что поток должен пробудиться только для того, чтобы тут же быть приостановленным. Это может привести к подгрузке стека ядра (если стек ядра потока был выгружен). В Windows 8.1 и Server 2012 R2 был добавлен механизм *Lightweight Suspend*, позволяющий приостанавливать потоки, находящиеся в состоянии ожидания, без использования механизма APC, а посредством прямых манипуляций с объектом в памяти и пометки его как приостановленного.

(Глубокое) замораживание

Механизм *замораживания* (freezing) используется процессами для перехода в приостановленное состояние, которое не может быть изменено вызовом `ResumeThread` для потоков процесса. Например, это может быть полезно, когда системе потребуется приостановить приложение UWP. Это происходит, когда приложение Windows переходит в фоновый режим — например, потому, что другое приложение переходит на первый план в планшетном режиме или приложение сворачивается в настольном режиме. В этом случае система дает приложению приблизительно пять секунд на выполнение работы — обычно сохранения состояния приложения. Сохранить состояние важно, потому что приложения Windows могут уничтожаться без каких-либо предварительных уведомлений в случае нехватки ресурсов. Если приложение уничтожается, его состояние может быть заново загружено при следующем запуске, и у пользователя появится впечатление, что приложение вообще никуда не пропадало. Замораживание процесса означает приостановку всех потоков таким образом, что `ResumeThread` не сможет их пробудить. Флаг в структуре `KTHREAD` сообщает, заморожен ли поток. Чтобы поток мог выполняться, его счетчик приостановлений должен быть равен 0, а флаг замораживаний должен быть сброшен.

Глубокое замораживание добавляет еще одно ограничение: вновь созданные потоки в процессе также не могут запускаться. Например, если вызов `CreateRemoteThreadEx` используется для создания нового потока в процессе с глубоким замораживанием, поток будет заморожен до его фактического запуска. Это типичный пример использования механизма замораживания.

Функциональность замораживания процессов и потоков не видна напрямую из пользовательского режима. Она используется во внутренних механизмах службой PSM (Process State Manager), отвечающей за выдачу запросов ядру на глубокое замораживание и размораживание.

Также процессы могут замораживаться с использованием заданий. Возможность замораживания и размораживания заданий открыто не документирована, но она реализуется стандартным системным вызовом `NtSetInformationJobObject`. Обычно она используется для приложений Windows, так как все процессы приложений Windows содержатся в заданиях. Такое задание может содержать один процесс (само приложение Windows), но также может содержать фоновые процессы-хосты, относящиеся к тому же приложению Windows, чтобы замораживание или размораживание всех процессов задания выполнялось за одну операцию. (Подробнее о приложениях Windows см. в главе 8 части 2.)

ЭКСПЕРИМЕНТ: ГЛУБОКОЕ ЗАМОРАЖИВАНИЕ

В этом эксперименте вы понаблюдаете за тем, как происходит глубокое замораживание, на примере отладки виртуальной машины.

1. Откройте WinDbg с административными привилегиями и присоедините отладчик к виртуальной машине с Windows 10.

2. Нажмите Ctrl+Break, чтобы прервать выполнение.
3. Установите точку останова там, где начинается глубокое замораживание, для демонстрации того, что процесс был заморожен:

```
bp nt!PsFreezeProcess "!process -1 0; g"
```

4. Введите команду g (go) или нажмите F5. Вы должны увидеть множественные срабатывания глубокого замораживания.
5. Запустите интерфейс Cortana с панели инструментов, после чего закройте интерфейс. Приблизительно через 5 секунд вы должны получить результат следующего вида:

```
PROCESS 8f518500 SessionId: 2 Cid: 12c8 Peb: 03945000 ParentCid: 02ac
   DirBase: 054007e0 ObjectTable: b0a8a040 HandleCount: 988.
   Image: SearchUI.exe
```

6. Перейдите в отладчик и выведите дополнительную информацию о процессе:

```
1: kd> !process 8f518500 1
PROCESS 8f518500 SessionId: 2 Cid: 12c8     Peb: 03945000   ParentCid: 02ac
DeepFreeze
    DirBase: 054007e0 ObjectTable: b0a8a040    HandleCount: 988.
    Image: SearchUI.exe
    VadRoot 95c1ffd8 Vads 405 Clone 0 Private 7682. Modified 201241. Locked 0.
    DeviceMap a12509c0
    Token                             b0a65bd0
    ElapsedTime                       04:02:33.518
    UserTime                          00:00:06.937
    KernelTime                        00:00:00.703
    QuotaPoolUsage[PagedPool]         562688
    QuotaPoolUsage[NonPagedPool]      34392
    Working Set Sizes (now,min,max)  (20470, 50, 345) (81880KB, 200KB, 1380KB)
    PeakWorkingSetSize                25878
    VirtualSize                       367 Mb
    PeakVirtualSize                   400 Mb
    PageFaultCount                    307764
    MemoryPriority                    BACKGROUND
    BasePriority                      8
    CommitCharge                      8908
    Job                               8f575030
```

7. Обратите внимание на атрибут DeepFreeze, выведенный отладчиком. Также обратите внимание на то, что процесс является частью задания. Запросите дополнительную информацию командой !job:

```
1: kd> !job 8f575030
Job at 8f575030
  Basic Accounting Information
    TotalUserTime:             0x0
    TotalKernelTime:           0x0
    TotalCycleTime:            0x0
    ThisPeriodTotalUserTime:   0x0
    ThisPeriodTotalKernelTime: 0x0
    TotalPageFaultCount:       0x0
    TotalProcesses:            0x1
```

```
    ActiveProcesses:           0x1
    FreezeCount:               1
    BackgroundCount:           0
    TotalTerminatedProcesses:  0x0
    PeakJobMemoryUsed:         0x38e2
    PeakProcessMemoryUsed:     0x38e2
  Job Flags
    [cpu rate control]
    [frozen]
    [wake notification allocated]
    [wake notification enabled]
    [timers virtualized]
    [job swapped]
  Limit Information (LimitFlags: 0x0)
  Limit Information (EffectiveLimitFlags: 0x3000)
  CPU Rate Control
    Rate = 100.00%
    Scheduling Group: a469f330
```

8. Как видно из результатов, задание находится под контролем интенсивности использования процессора (см. далее в этой главе) и заморожено. Отсоединитесь от VM и закройте отладчик.

Выбор потока

Когда логическому процессору нужно выбрать следующий выполняемый процесс, он вызывает функцию планировщика `KiSelectNextThread`. Это может происходить по различным сценариям.

- Произошло изменение жестко заданного сходства, что сделало текущий выполняемый поток или поток, находящийся в состоянии повышенной готовности, неподходящим для выполнения на его выбранном логическом процессоре, поэтому должен быть выбран другой процессор.
- Текущий выполняемый поток исчерпал свой квант, и SMT-набор, на котором он в данный момент выполнялся, теперь занят, в то время как другие SMT-наборы в составе идеального узла полностью простаивают. (SMT (*Symmetric Multithreading*) — технический термин для обозначения технологии гиперпоточности, рассмотренной в главе 2.) Планировщик осуществляет миграцию текущего потока, связанную с истечением его кванта, поэтому должен быть выбран другой поток.
- Операция ожидания завершена, и в регистре статуса ожидания находились незавершенные операции планирования (иными словами, были установлены биты приоритета — Priority и/или сходства — Affinity).

При таких сценариях планировщик ведет себя следующим образом.

- Вызывает функцию `KiSelectReadyThreadEx`, чтобы найти следующий готовый поток, который должен быть выполнен процессором, и проверяет, был ли такой поток найден.

- Если готовый поток не был найден, включается планировщик простоя, и для выполнения выбирается поток простоя. Или, если готовый поток *был* найден, он помещается в состояние готовности в локальной или общей очереди готовности (в зависимости от обстоятельств).

Операция выбора следующего потока `KiSelectNextThread` выполняется только тогда, когда логический процессор нуждается в подборе, но еще не в выполнении следующего планируемого потока (по причине чего поток войдет в состояние Ready). Но в другой раз логический процессор заинтересован в немедленном запуске следующего готового потока или выполнении другого действия, если он не доступен (вместо того, чтобы простаивать), как при следующих обстоятельствах.

- Произошло изменение приоритета, и теперь текущий поток, находящийся в состоянии повышенной готовности, или выполняющийся поток не является готовым потоком, имеющим самый высокий приоритет на выбранном им логическом процессоре, тогда должен быть выполнен поток, имеющий более высокий приоритет.
- Поток явным образом уступил свою очередь с помощью функции `YieldProcessor` или `NtYieldExecution`, и в готовности к выполнению мог бы быть другой поток.
- Квант времени текущего потока истек, и другие потоки с тем же уровнем приоритета должны получить свой шанс на выполнение.
- Поток утратил свое повышение приоритета, вызывая тем самым аналогичное изменение приоритетов в уже рассмотренном сценарии.
- Запущен планировщик простоя, и нужно проверить, не появился ли в интервале между запросом на планирование простоя и запуском планировщика простоя готовый к выполнению поток.

Простой способ запомнить разницу между тем, какая из процедур запускается, заключается в проверке, *должен* ли логический процессор выполнить другой поток (в таком случае вызывается процедура `KiSelectNextThread`) или *может ли он, если это возможно*, выполнить другой поток (в таком случае вызывается процедура `KiSelectReadyThreadEx`). Каждый процессор имеет свою собственную базу данных потоков, готовых к выполнению (принадлежащие базе данных диспетчеры находятся в KPRCB). Процедура `KiSelectReadyThreadEx` может просто проверить очереди текущего логического процессора, забрав первый найденный поток с самым высоким приоритетом, если этот приоритет не ниже, чем у текущего выполняемого потока (в зависимости от того, разрешено ли выполнение текущего потока, что может не относиться к случаю развития событий по сценарию выбора следующего потока — `KiSelectNextThread`). Если потока с более высоким приоритетом нет (или вообще нет готовых к выполнению потоков), то возвращенного потока не будет.

Планировщик простоя

Как только запускается поток простоя, он проверяет, включен ли планировщик простоя, как в одном из сценариев, рассмотренных в предыдущем разделе.

Если планировщик простоя включен, поток простоя путем вызова процедуры `KiSearchForNewThread` начинает сканировать очереди готовых потоков других процессоров в поисках потоков, которые он может запустить. Следует учесть, что расходы на выполнение, связанные с этой операцией, не засчитываются как время потока простоя, а учитываются как время прерывания и DPC-вызова (относятся на счет процессора), поэтому время планирования простоя считается системным временем. Алгоритм процедуры `KiSearchForNewThread`, который основан на функциях, рассмотренных ранее в разделе «Выбор потока», будет рассмотрен в следующем разделе.

Многопроцессорные системы

В однопроцессорной системе планирование осуществляется сравнительно просто: всегда запускается поток с самым высоким приоритетом, требующий выполнения. В многопроцессорной системе планирование осуществляется гораздо сложнее, поскольку Windows предпринимает попытку спланировать выполнение потоков на наиболее оптимальном для потока процессоре, принимая во внимание предпочитаемые потоком процессоры и те процессоры, на которых он уже выполнялся, а также конфигурацию многопроцессорной системы. Поэтому, хотя Windows пытается спланировать работу готовых к выполнению потоков, имеющих самый высокий приоритет на всех доступных центральных процессорах, она гарантирует только то, что где-нибудь ею будет запущен один из потоков, имеющих самый высокий приоритет. С общими очередями готовности (для потоков без ограничений сходства) возможны более сильные гарантии: в каждой общей группе процессоров выполняется как минимум один из потоков с наивысшим приоритетом.

Прежде чем рассматривать конкретные алгоритмы, используемые для выбора, какие потоки когда и где запускать, давайте изучим дополнительную информацию, поддерживаемую системой Windows для отслеживания состояния потока и процессора на многопроцессорных системах и три различных типа многопроцессорных систем, поддерживаемых Windows (SMT, многоядерные и NUMA).

Наборы пакетов и SMT-наборы

Для принятия правильных решений по планированию потоков при работе с топологиями логических процессоров в Windows в блоке KPRCB используется пять полей. Первое поле, `CoresPerPhysicalProcessor`, определяет, является ли этот логический процессор частью многоядерного пакета; его значение вычисляется по идентификатору CPUID, возвращенному процессором, и округляется до степени 2. Второе поле, `LogicalProcessorsPerCore`, определяет, является ли логический процессор частью SMT-набора, например, на процессоре Intel с включенной технологией HyperThreading, и его значение также запрашивается через CPUID и округляется. Перемножение этих двух чисел дает количество логических про-

цессоров в пакете или в реальном физическом процессоре, вставленном в гнездо. Располагая этим количеством, каждый блок PRCB может затем заполнить свое значение `PackageProcessorSet`, которое является маской сходства, дающей описание, какие еще логические процессоры внутри этой группы (поскольку пакеты ограничиваются группами) принадлежат одному и тому же физическому процессору. По аналогии с этим, значение `CoreProcessorSet` объединяет другие логические процессоры одного и того же ядра, что также называется SMT-набором. И наконец, значение `GroupSetMember` определяет, какая битовая маска внутри текущей группы процессоров идентифицирует тот или иной логический процессор. Например, логический процессор 3 обычно имеет значение `GroupSetMember`, равное 8 (2 в степени 3).

ЭКСПЕРИМЕНТ: ПРОСМОТР ИНФОРМАЦИИ О ЛОГИЧЕСКОМ ПРОЦЕССОРЕ

Вы можете просмотреть информацию, предоставляемую Windows для SMT-процессоров, с помощью команды отладчика ядра `!smt`. Следующий вывод получен в четырехъядерной системе Intel Core i7 с SMT (с восемью логическими процессорами):

```
lkd> !smt
SMT Summary:
------------

KeActiveProcessors:
********------------------------------------------------------
(00000000000000ff)
IdleSummary:
-****--*------------------------------------------------------
(000000000000009e)
No PRCB SMT         Set                                    APIC Id
 0 fffff803d7546180 **---------------------------- (0000000000000003)
0x00000000
 1 ffffba01cb31a180 **---------------------------- (0000000000000003)
0x00000001
 2 ffffba01cb3dd180 --**-------------------------- (000000000000000c)
0x00000002
 3 ffffba01cb122180 --**-------------------------- (000000000000000c)
0x00000003
 4 ffffba01cb266180 ----**------------------------ (0000000000000030)
0x00000004
 5 ffffba01cabd6180 ----**------------------------ (0000000000000030)
0x00000005
 6 ffffba01cb491180 ------**---------------------- (00000000000000c0)
0x00000006
 7 ffffba01cb5d4180 ------**---------------------- (00000000000000c0)
0x00000007

Maximum cores per physical processor: 8
Maximum logical processors per core: 2
```

NUMA-системы

Windows поддерживает еще один тип многопроцессорных систем, архитектуру с технологией доступа к неоднородной памяти — NUMA (NonUniform Memory Access). В NUMA-системе процессоры группируются в небольшие модули, которые называются *узлами*. У каждого узла есть свои собственные процессоры и память, и он подключен к более крупной системе через объединительную шину с согласованным кэшем. Эти системы называются *неоднородными*, поскольку у каждого узла имеется своя собственная локальная высокоскоростная память. Хотя любой процессор в любом узле может получить доступ ко всей памяти, доступ к локальной памяти узла осуществляется намного быстрее.

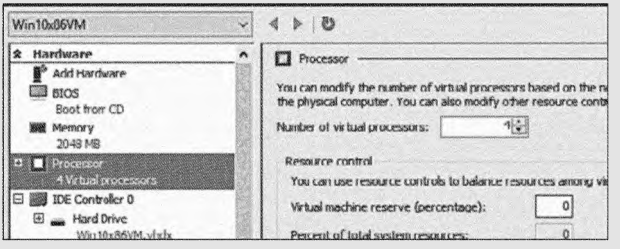
Ядро предоставляет информацию о каждом узле в NUMA-системе в структуре данных, называемой `KNODE`. Переменная ядра `KeNodeBlock` является массивом указателей на `KNODE`-структуры для каждого узла. Для просмотра формата `KNODE`-структуры в отладчике ядра используется команда `dt`:

```
lkd> dt nt!_KNODE
   +0x000 IdleNonParkedCpuSet : Uint8B
   +0x008 IdleSmtSet       : Uint8B
   +0x010 IdleCpuSet       : Uint8B
   +0x040 DeepIdleSet      : Uint8B
   +0x048 IdleConstrainedSet : Uint8B
   +0x050 NonParkedSet     : Uint8B
   +0x058 ParkLock         : Int4B
   +0x05c Seed             : Uint4B
   +0x080 SiblingMask      : Uint4B
   +0x088 Affinity         : _GROUP_AFFINITY
   +0x088 AffinityFill     : [10] UChar
   +0x092 NodeNumber       : Uint2B
   +0x094 PrimaryNodeNumber : Uint2B
   +0x096 Stride           : UChar
   +0x097 Spare0           : UChar
   +0x098 SharedReadyQueueLeaders : Uint8B
   +0x0a0 ProximityId      : Uint4B
   +0x0a4 Lowest           : Uint4B
   +0x0a8 Highest          : Uint4B
   +0x0ac MaximumProcessors : UChar
   +0x0ad Flags            : _flags
   +0x0ae Spare10          : UChar
   +0x0b0 HeteroSets       : [5] _KHETERO_PROCESSOR_SET
```

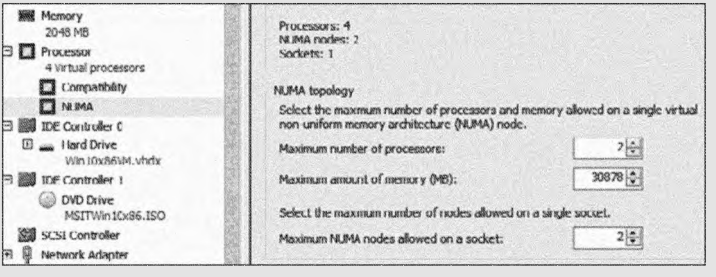
ЭКСПЕРИМЕНТ: ПРОСМОТР NUMA-ИНФОРМАЦИИ

Информацию, поддерживаемую Windows для каждого узла в NUMA-системе, можно просмотреть с помощью команды отладчика ядра `!numa`. Чтобы поэкспериментировать с NUMA-системами даже при отсутствии соответствующего оборудования, вы можете настроить виртуальную машину Hyper-V с включением нескольких узлов NUMA, которые будут использоваться гостевой VM. Чтобы настроить Hyper-V VM для использования NUMA, выполните следующие действия (вам понадобится машина с более чем четырьмя логическими процессорами).

1. Щелкните на кнопке Пуск (Start), введите текст hyper и щелкните на варианте диспетчер Hyper-V (Hyper-V Manager).
2. Убедитесь в том, что виртуальная машина выключена. В противном случае следующие изменения внести не удастся.
3. Щелкните правой кнопкой мыши на VM в диспетчере Hyper-V и выберите команду Параметры (Settings), чтобы получить доступ к настройкам VM.
4. Щелкните на узле Память (Memory) и убедитесь в том, что флажок Dynamic Memory снят.
5. Щелкните на узле Процессор (Processor) и введите 4 в поле Количество виртуальных процессоров (Number of Virtual Processors):



6. Раскройте узел Процессор (Processor) и выберите подузел NUMA.
7. Введите 2 в полях Максимальное количество процессоров (Maximum Number of Processors) и Максимальное число узлов NUMA для сокета (Maximum NUMA Nodes Allowed on a Socket):



8. Щелкните на кнопке OK, чтобы сохранить изменения.
9. Запустите VM.
10. Используйте отладчик ядра для выполнения команды !numa. Пример вывода для ранее настроенной VM:

```
2: kd> !numa
NUMA Summary:
------------
    Number of NUMA nodes : 2
    Number of Processors : 4
unable to get nt!MmAvailablePages
```

```
    MmAvailablePages     : 0x00000000
    KeActiveProcessors   :
    ****--------------------------- (0000000f)

    NODE 0 (FFFFFFFF820510C0):
   Group            : 0 (Assigned, Committed, Assignment Adjustable)
   ProcessorMask    : **---------------------------- (00000003)
   ProximityId      : 0
   Capacity         : 2
   Seed             : 0x00000001
   IdleCpuSet       : 00000003
   IdleSmtSet       : 00000003
   NonParkedSet     : 00000003
Unable to get MiNodeInformation

    NODE 1 (FFFFFFFF8719E0C0):
   Group            : 0 (Assigned, Committed, Assignment Adjustable)
   ProcessorMask    : --**--------------------------- (0000000c)
   ProximityId      : 1
   Capacity         : 2
   Seed             : 0x00000003
   IdleCpuSet       : 00000008
   IdleSmtSet       : 00000008
   NonParkedSet     : 0000000c
Unable to get MiNodeInformation
```

Приложения, желающие добиться максимального быстродействия в NUMA-системах, могут установить маску сходства для ограничения процесса процессорами определенного узла, хотя Windows уже ограничивает чуть ли не все потоки единственным NUMA-узлом из-за своих алгоритмов планирования, ориентированных на NUMA-системы.

Вопрос о том, как алгоритмы планирования учитывают специфику NUMA-систем, рассматривается позже в этой главе в разделе «Выбор процессора». (Оптимизация в диспетчере памяти для использования локальной памяти узлов рассматривается в главе 5.)

Назначение группы процессоров

При запросе топологии системы для построения разнообразных взаимоотношений между логическими процессорами, SMT-наборами, многоядерными пакетами и физическими сокетами Windows назначает процессоры соответствующей группе, которая описывает их сходство (посредством рассмотренной ранее расширенной маски сходства). Эта задача решается процедурой `KePerformGroupConfiguration`, которая вызывается при инициализации до выполнения любой другой работы на этапе 1. Основные этапы этого процесса:

1. Сначала функция запрашивает все найденные узлы (`KeNumberNodes`) и вычисляет емкость каждого узла (т. е. сколько логических процессоров может

быть частью узла). Это значение сохраняется в переменной `MaximumProcessors` блока `KeNodeBlock`, который идентифицирует все NUMA-узлы системы. Если система поддерживает идентификаторы близости NUMA (Proximity ID), для каждого узла запрашивается также идентификатор близости, который сохраняется в блоке узла.

2. Создается массив расстояний NUMA (`KeNodeDistance`) и вычисляется расстояние между всеми NUMA-узлами.

   Следующая череда шагов связана с особенными настройками пользовательской конфигурации, которые заменяют NUMA-назначения, определяемые по умолчанию. Например, на системах с установленной технологией Hyper-V (с гипервизором, настроенным на автозапуск) будет доступна только одна группа процессоров, и все NUMA-узлы (которые могут поместиться) будут связаны с группой 0. Это означает, что сценарии Hyper-V не могут в данный момент воспользоваться машинами с более чем 64 процессорами.

3. Затем функция проверяет наличие любых данных о назначении статических групп, переданных загрузчиком (и поэтому настраиваемых пользователем). Эти данные определяют информацию о близости и групповые назначения для каждого NUMA-узла.

ПРИМЕЧАНИЕ Пользователи, работающие с крупными NUMA-серверами, которые могут потребовать специального управления информацией о близости и групповых назначениях с целью проведения тестирования и проверок, могут ввести эти данные через параметры реестра `Group Assignment` и `Node Distance` в разделе `HKLM\SYSTEM\CurrentControlSet\Control\NUMA`. Точный формат этих данных включает счетчик, за которым следует массив идентификаторов близости и групповых назначений, которые являются 32-разрядными значениями.

---

4. Прежде чем считать эти данные достоверными, ядро запрашивает идентификатор близости, чтобы сопоставить с номером узла, а затем связывает номера групп в соответствии с требованием. Затем оно убеждается в том, что NUMA-узел 0 связан с группой 0, и в том, что емкость всех NUMA-узлов согласуется с размером группы. И наконец, функция проверяет, у скольких групп все еще имеется остаточная емкость.

ПРИМЕЧАНИЕ NUMA-узел 0 при любых обстоятельствах всегда назначается в группу 0.

---

5. Затем ядро пытается динамически назначить NUMA-узлы группам, признавая любые статически настроенные узлы, если они были переданы так, как это описывалось ранее. Обычно ядро пытается минимизировать количество создаваемых групп, объединяя в каждой группе как можно больше NUMA-узлов. Но если такое поведение нежелательно, оно может быть скорректировано с помощью параметра загрузчика `/MAXGROUP`, который настраивается через BCD-элемент `maxgroup`. Включение этого параметра переопределяет поведение по

умолчанию и заставляет алгоритм распространять как можно больше NUMA-узлов в как можно большем количестве групп (с учетом того, что в нынешней реализации максимальное количество групп равно 20). Если имеется только один узел или если все узлы могут поместиться в одной группе (и элемент `maxgroup` отключен), система использует конфигурацию по умолчанию и назначает все узлы группе 0.

6. Если имеется более одного узла, Windows проверяет статические дистанции NUMA-узлов (если таковые имеются), а затем сортирует все узлы по их емкости, чтобы первым стал самый крупный узел. А в режиме минимизации групп ядро путем сложения всех емкостей определяет, какое максимальное количество процессоров может быть в системе. Путем деления результата на количество процессоров в каждой группе ядро предполагает, что на машине будет именно такое общее количество групп (ограниченное максимумом 20). В режиме максимизации групп исходной оценкой будет то, что групп будет столько же, сколько и узлов (опять же ограничивая количество групп числом 20).
7. Теперь ядро приступает к завершающему процессу назначений. Утверждаются все ранее сделанные фиксированные назначения, и для них создаются группы.
8. Затем все NUMA-узлы перегруппировываются для минимизации дистанции между различными узлами в группе. Иными словами, более близкие узлы помещаются в одну группу и сортируются по расстояниям.
9. После этого для любого динамически сконфигурированного узла осуществляются одни и те же процессы группировки.
10. Любые оставшиеся пустые узлы назначаются группе 0.

Логические процессоры, приходящиеся на каждую группу

Обычно, как упоминалось ранее, Windows назначает 64 процессора на каждую группу, но эта конфигурация также может быть адаптирована для различных вариантов нагрузки, например, с помощью параметра `/GROUPSIZE`, который настраивается через BCD-элемент `groupsize`. Указывая число, являющееся степенью двойки, можно ограничить группу количеством процессоров меньше обычного — например, для тестирования поддержки групп в системе (система с 8 логическими процессорами может быть представлена как содержащая 1, 2 или 4 группы). Чтобы принудительно решить данный вопрос, параметр `/FORCEGROUPAWARE` (BCD-элемент `groupaware`), кроме того, заставляет ядро по возможности обойти группу 0, назначая самое большое количество групп, доступное в таких действиях, как выбор потоков и сходства DPC и обработка назначения групп. Избегайте назначения группе размера 1, поскольку это заставит практически все приложения в системе вести себя так, будто они запущены на однопроцессорной машине. Дело в том, что ядро настраивает маску сходства заданного процесса на охват только

одной группы, пока приложение не запросит обратное (чего большинство приложений делать не будет).

В крайнем случае, когда число логических процессоров в пакете не может поместиться в одну группу, Windows подгоняет это количество таким образом, чтобы пакет мог поместиться в одной группе. Для этого сокращается число `CoresPerPhysicalProcessor`, а если SMT не помещается, делает то же самое с `LogicalProcessorsPerCore`. Исключение из этого правила — система, фактически состоящая из нескольких NUMA-узлов в одном пакете (не часто, но встречается). В таких модулях MCM (Multiple-Chip Modules) на одной микросхеме находятся сразу два набора ядер, а также два контроллера памяти. Если в таблице ACPI SRAT MCM-модуль определяется как имеющий два NUMA-узла, то в зависимости от алгоритмов конфигурации групп Windows может связать эти два узла с двумя разными группами (в зависимости от алгоритмов конфигурации групп). В данном сценарии MCM-пакет будет охватывать более одной группы.

Кроме того, что такие конфигурации создают существенные проблемы совместимости драйверов и приложений (которые они должны по замыслу разработчиков обнаруживать и искоренять), они оказывают еще большее влияние на машину: они внедряют NUMA-поведение даже на той машине, которая не обладает NUMA-структурой. Причина в том, что Windows никогда не позволит NUMA-узлу охватывать несколько групп, что следует из алгоритмов назначения. Следовательно, если ядро искусственно создает небольшие группы, у каждой из этих двух групп должен быть свой собственный NUMA-узел. Например, на четырехъядерном процессоре с размером групп 2 будут созданы две группы и, следовательно, два NUMA-узла, которые станут подузлами основного узла. Это повлияет на политики планирования и управления памятью так же, как это было бы на настоящей NUMA-системе, что может пригодиться при тестировании.

Состояние логического процессора

В дополнение к очередям готовых потоков и к сводке готовности Windows поддерживает две битовые маски, в которых отслеживается состояние процессоров в системе (использование этих масок рассмотрено далее в разделе «Выбор процессоров»). Windows поддерживает следующие битовые маски.

- `KeActiveProcessors`. Маска активных процессоров, в которой устанавливается бит для каждого используемого в системе процессора. Число установленных битов может быть меньше количества имеющихся процессоров, если согласно лицензионным ограничениям запущенной версией Windows поддерживается меньше процессоров, чем количество доступных физических процессоров. Для проверки нужно воспользоваться переменной `KeRegisteredProcessors` и посмотреть, сколько процессоров лицензировано на машине. В данном случае под термином «*процессор*» понимаются физические наборы.
- `KeMaximumProcessors`. Максимальное количество логических процессоров (включая все возможные в будущем динамически добавляемые процессоры), ко-

торое ограничено лицензионным соглашением, а также любыми ограничениями платформы, запрашиваемыми путем вызова HAL и проверки с использованием таблицы ACPI SRAT, если таковая имеется.

В данных узла (`KNODE`) хранится набор простаивающих процессоров этого узла (`IdleCpuSet`), набор простаивающих незапаркованных процессоров (`IdleNonParkedCpuSet`) и простаивающие SMT-наборы (`IdleSmtSet`).

Масштабируемость планировщика

Поскольку в многопроцессорных системах одному процессору может понадобиться изменить структуры данных планирования, относящиеся к другому процессору (например, вставить поток, которому нужно выполняться на конкретном процессоре), доступ к этим структурам синхронизирован с помощью выстраиваемых в очередь спин-блокировок, принадлежащих каждому блоку PRCB, которые удерживаются на уровне диспетчеризации `DISPATCH_LEVEL`. Таким образом, выбор потока может произойти только при блокировке одного процессорного PRCB-блока. Если нужно, может быть заблокирован еще один процессорный PRCB-блок, например, при сценариях захвата потоков, которые будут рассмотрены позже. Переключение контекста потока также синхронизируется путем использования более детализированной спин-блокировки, относящейся к конкретному потоку.

У каждого центрального процессора имеется также список потоков, находящихся в состоянии отложенной готовности (`DeferredReadyListHead`). В нем представлены потоки, которые готовы к запуску, но еще не подготовлены к выполнению; фактически операция приведения их в полную готовность отложена до более подходящего момента. Поскольку каждый процессор управляет только своим собственным списком потоков, находящихся в состоянии отложенной готовности, этот список не синхронизируется спин-блокировкой PRCB-блока. Список потоков, находящихся в состоянии отложенной готовности, обрабатывается функцией `KiProcessDeferredReadyList` после того, как функция уже внесла изменения в значения сходства процесса или потока, приоритета (включая все, что относится к повышению приоритета) или кванта времени.

Для каждого потока, присутствующего в списке, эта функция вызывает функцию `KiDeferredReadyThread`, которая выполняет алгоритм, показанный далее в разделе «Выбор процессора». Алгоритм может либо вызвать немедленное выполнение потока, поместив его в список готовых потоков процессора, либо, если процессор недоступен, дать потенциальную возможность потоку быть помещенным в список отложенных готовых потоков другого процессора, перейти в состояние повышенной готовности (standby) или немедленно выполниться. Это свойство используется при работе *механизма парковки ядер* (core parking engine): все потоки помещаются в список отложенных готовых потоков, который затем обрабатывается. Поскольку функция `KiDeferredReadyThread` обходит запаркованные ядра (как будет показано далее), это заставляет все потоки процессора завершать свое выполнение на других процессорах.

Сходство

У каждого потока есть *маска сходства* (affinity mask), которая определяет процессоры, разрешенные для выполнения потока. Маска сходства потока наследуется от маски сходства процесса. По умолчанию сначала у всех процессов (а следовательно, и у всех потоков) маска сходства эквивалентна набору из всех активных процессоров в назначенной для них группе, иными словами, система может свободно планировать выполнение всех потоков на любом доступном процессоре внутри группы, связанной с процессом. Но для оптимизации пропускной способности, распределения рабочей нагрузки на определенном наборе процессоров или решения обеих задач приложения могут выбрать изменение для потока маски сходства. Это может быть сделано на нескольких уровнях.

- Путем вызова функции `SetThreadAffinityMask` для установки сходства для отдельного процесса.
- Путем вызова функции `SetProcessAffinityMask` для установки сходства всех потоков в процессе.
- Диспетчер задач (Task Manager) и Process Explorer предоставляют для этой функции графический интерфейс: щелкните правой кнопкой мыши на имени процесса и выберите пункт Задать соответствие (Set Affinity). Программа Psexec (из пакета Sysinternals) предоставляет для этой функции интерфейс командной строки. (Обратите внимание на ключ `-a` в справке по этому средству.)
- Путем включения процесса в состав задания, у которого имеется общая для всего задания маска сходства, установленная с помощью функции `SetInformationJobObject` (см. главу 3).
- Путем задания маски сходства в заголовке образа при компиляции приложения.

СОВЕТ Если вас заинтересует подробная спецификация формата образов Windows, проведите поиск по условию «*Portable Executable and Common Object File Format Specification*» на сайте *http://msdn.microsoft.com*.

У образа также может быть во время компоновки установлен флаг однопроцессорности (`uniprocessor`). В таком случае во время создания процесса система выбирает один процессор (`MmRotatingProcessorNumber`), отмечает этот выбор в маске сходства, начиная с первого процессора и затем циклически перебирая все процессоры в группе. Например, на системе с двумя процессорами при первом запуске образа, помеченного как однопроцессорный, этот образ назначается центральному процессору 0, во второй раз — центральному процессору 1, в третий раз — центральному процессору 0, в четвертый раз — центральному процессору 1 и т. д. Этот флаг может пригодиться в качестве временного обходного решения для программ, у которых имеются ошибки многопотоковой синхронизации, которые из-за «условий гонки» проявляются в многопроцессорных системах, но не случаются на однопроцессорных. Если образ демонстрирует такие симптомы и не имеет цифровой подписи,

флаг можно установить вручную путем редактирования заголовка образа программами для редактирования образов PE. Более удачное решение (к тому же совместимое с исполняемыми файлами, имеющими цифровую подпись) основано на использовании инструментария совместимости приложений Microsoft Application Compatibility Toolkit и добавлении оболочки совместимости, чтобы база данных совместимости пометила образ на момент его запуска как предназначенный только для однопроцессорной системы.

ЭКСПЕРИМЕНТ: ПРОСМОТР И ИЗМЕНЕНИЕ СХОДСТВА ПРОЦЕССОРА

В этом эксперименте мы изменим настройки сходства для процесса, а также продемонстрируем, как сходство этого процесса наследуется новыми процессами.

1. Запустите окно командной строки (Cmd.exe).
2. Запустите диспетчер задач (Task Manager) или Process Explorer и найдите процесс Cmd.exe в списке процессов.
3. Щелкните правой кнопкой мыши на имени процесса и выберите пункт Задать соответствие (Set Affinity). Будет выведен список процессоров. Например, на двухпроцессорной системе вы увидите следующее окно.

4. Выберите поднабор доступных в системе процессоров и щелкните на кнопке OK. Теперь потоки процесса могут выполняться только на выбранных вами процессорах.
5. В командной строке введите строку `Notepad`, чтобы запустить программу Блокнот.
6. Вернитесь в диспетчер задач (Task Manager) или в Process Explorer и найдите новый процесс Notepad.
7. Щелкните на процессе правой кнопкой мыши и выберите команду Задать соответствие (Set Affinity). Вы увидите тот же самый перечень процессоров, который был выбран для процесса окна командной строки. Дело в том, что процессы наследуют настройки сходства от своих родителей.

Windows не будет перемещать выполняемый поток, который может выполняться на другом процессоре, с одного центрального процессора на другой процессор, чтобы разрешить потоку со сходством, связанным с первым процессором, выполняться на первом процессоре. Рассмотрим, например, следующий сценарий: процессор 0 выполняет поток с приоритетом 8, который может выполняться на любом процессоре, а процессор 1 выполняет поток с приоритетом 4, который тоже может выполняться на любом процессоре. Становится готовым к выполнению поток с уровнем приоритета 6, который может выполняться только на центральном процессоре 0. Что при этом произойдет? Windows не станет перемещать поток с уровнем приоритета 8 с центрального процессора 0 на центральный процессор 1 (вытесняя тем самым поток с уровнем приоритета 4), так что готовый к выполнению поток с уровнем приоритета 6 должен оставаться в состоянии готовности. Следовательно, изменение маски сходства процесса или потока может привести к тому, что потоки получат меньше времени центрального процессора, чем обычно, потому что система Windows ограничивается в запуске потока на конкретном процессоре. Следовательно, при настройке сходства следует действовать крайне осторожно. В большинстве случаев лучше всего позволить системе Windows решать, какой поток где запускать.

Расширенная маска сходства

Для поддержки более 64 процессоров, преодолевая ограничение, накладываемое структурой маски сходства (состоящей из 64 битов на 64-разрядной системе), Windows использует расширенную маску сходства (`KAFFINITY_EX`), представляющую собой массив из масок сходства, по одной для каждой поддерживаемой группы процессоров (на данный момент их число равно 20). Когда планировщику нужно сослаться на процессор в расширенных масках сходства, он сначала выделяет из ссылки нужную битовую маску, используя ее номер группы, а затем напрямую обращается к полученным данным о сходстве. В API-функциях ядра расширенные маски сходства не фигурируют; вместо этого код, вызывающий API-функцию, вводит номер группы в виде аргумента и получает обычную маску сходства для этой группы. Но, с другой стороны, у API-функций Windows обычно можно запросить только информацию об одной группе, которая является группой текущего выполняемого потока (которая имеет фиксированное значение).

Расширенная маска сходства и ее основные функциональные возможности определяют также и способ преодоления процессом границ исходной назначенной группы процессоров. Путем использования API-функций расширенной маски сходства потоки в процессе могут выбрать маски сходства на другой группе процессоров. Например, если процесс содержит 4 потока и у машины имеется 256 процессоров, поток 1 может выполняться на процессоре 4, поток 2 может выполняться на процессоре 68, поток 4 на процессоре 132, а поток 4 на процессоре 196, если для каждого потока будет установлена маска сходства 0x10 (0b10000 в двоичном формате) на группах 0, 1, 2 и 3. В качестве альтернативы для каждого потока может быть установлена маска сходства 0xFFFFFFFF для их исходных групп, и процесс может выполнять свои потоки на любом доступном процессоре системы (с тем лишь

ограничением, что каждый поток выполняется только внутри своей собственной группы).

Расширенная маска сходства должна использоваться путем указания номера группы при создании нового потока в списке его атрибутов (`PROC_THREAD_ATTRIBUTE_GROUP_AFFINITY`) или же вызовом `SetThreadGroupAffinity` для существующего потока.

Маска сходства системы

Поскольку драйверы Windows обычно выполняются в контексте вызывающих потоков или в контексте произвольного потока (т. е. вне границ безопасности), текущий выполняемый код драйвера может быть субъектом правил сходства, установленных разработчиком приложения, которые в настоящее время не соответствуют коду драйвера и могут даже помешать правильной обработке прерываний и другой очередной работе. Поэтому у разработчиков драйверов имеется механизм временного обхода настроек сходства пользовательского потока путем использования API-функций `KeSetSystemAffinityThread(Ex)`/`KeSetSystemGroupAffinityThread` и `KeRevertToUserAffinityThread(Ex)`/`KeRevertToUserGroupAffinityThread`.

Идеальный и последний процессор

У каждого потока есть три номера центрального процессора, хранящиеся в блоке управления процессом, находящемся в ядре:

- **идеальный, или предпочтительный, процессор**, на котором должен выполняться этот поток;
- **последний процессор**, или процессор, на котором поток выполнялся в последний раз;
- **следующий процессор**, или процессор, на котором поток будет выполняться или уже выполняется.

Идеальный процессор выбирается для потока при создании этого потока с использованием *затравки* (seed) в блоке управления процессом. Затравка получает приращение при каждом создании потока, поэтому идеальный процессор для каждого нового потока в процессе выбирается по кругу среди процессоров, доступных в системе. Например, первому потоку в первом процессе системы назначается идеальный процессор 0. Второму потоку в этом процессе назначается идеальный процессор 1. Но следующий процесс в системе назначает своему первому потоку идеальный процессор 1, второй 2 и т. д. Таким образом, потоки каждого процесса распределяются по процессорам. В SMT-системах (гиперпоточность) следующий идеальный процессор выбирается из следующего SMT-набора. Например, в четырехъядерной гиперпоточной системе идеальными процессорами потоков определенного процесса могут быть процессоры 0, 2, 4, 6, 0, ...; 3, 5, 7, 1, 3, ...; и т. д. Таким образом потоки равномерно распределяются между физическими процессорами.

Следует заметить, что при таких назначениях потоков внутри процессов предполагается, что у них будет одинаковый объем работы. Обычно такого в многопо-

токовых процессах не случается, потому что есть один или несколько служебных потоков, а также несколько рабочих потоков. Поэтому многопоточное приложение, которое собирается в полной мере использовать платформу, может для повышения эффективности указывать номера идеальных процессоров для своих потоков путем использования функции `SetThreadIdealProcessor`. Чтобы получить преимущество от групп процессоров, разработчики должны вместо этого вызывать функцию `SetThreadIdealProcessorEx`, которая позволяет выбирать номер группы для указания сходства.

В 64-разрядной версии Windows для балансировки назначений вновь создаваемым потокам в процессе в блоке `KNODE` используется поле прогресса `Stride`. Это поле является скалярным числом, представляющим количество битов сходства в заданном NUMA-узле, которое должно быть пропущено для получения новой независимой логической части процессора. Термин «независимой» означает «расположенной на другом ядре» (для SMT-системы) или «на другом наборе» (для многоядерной системы без использования SMT-технологии). Поскольку 32-разрядная версия Windows не поддерживает системы с большими процессорными конфигурациями, в ней поле `Stride` не используется, и она просто выбирает номер следующего процессора, пытаясь по возможности избежать совместного использования одного и того же SMT-набора.

Идеальный узел

При создании процесса на NUMA-системах для него выбирается идеальный узел. Первый процесс назначается узлу 0, второй процесс — узлу 1, и т. д. Затем из идеального узла процесса для потоков процесса выбираются идеальные процессоры. В качестве идеального процессора для первого потока в процессе назначается первый процессор в узле. По мере создания в процессе дополнительных потоков с тем же самым идеальным узлом для следующего потока в качестве идеального используется следующий процессор и т. д.

Наборы ЦП

Вы уже видели, как сходство (иногда называемое *жестким сходством* — *hard affinity*) может ограничить потоки некоторыми процессорами, которым планировщик всегда отдает предпочтение. Механизм идеального процессора старается выполнять потоки на их идеальных процессорах (иногда это называется *мягким сходством*), обычно ожидая, что состояние потока является частью кэша процессора. Идеальный процессор может использоваться или не использоваться, он не мешает планированию потока на других процессорах. Оба механизма не применяются к системной активности — например, к работе системных потоков. Кроме того, не существует простого способа задать жесткое сходство для всех процессов в системе за одну операцию. Даже обход процессов не сработает. Системные процессы обычно защищаются от внешних изменений сходства, потому что для этого необходимо право доступа `PROCESS_SET_INFORMATION`, не предоставляемое для защищенных процессов.

В Windows 10 и Server 2016 появился механизм *наборов ЦП* (CPU sets). Эта разновидность сходства может назначаться для использования системой в целом

(включая деятельность системных потоков) процессов и даже отдельных потоков. Например, звуковое приложение с низкой задержкой может захотеть использовать процессор монопольно, тогда как остальные компоненты системы направляются для использования других процессоров. Наборы ЦП предоставляют механизм для решения этой задачи.

Документированный API пользовательского режима на момент написания этой книги остается ограниченным. Функция `GetSystemCpuSetInformation` возвращает массив `SYSTEM_CPU_SET_INFORMATION` с данными о каждом наборе ЦП. В текущей реализации набор ЦП эквивалентен одному ЦП. Это означает, что длина возвращаемого массива равна количеству логических процессоров в системе. Каждый набор ЦП задается своим идентификатором, который выбирается равным сумме 256 (0×100) и индекса ЦП (0, 1, ...). Эти идентификаторы должны передаваться функциям `SetProcessDefaultCpuSets` и `SetThreadSelectedCpuSets` для назначения набора ЦП по умолчанию и набора ЦП для заданного потока соответственно.

Например, набор ЦП может назначаться для «важного» потока, выполнение которого по возможности не должно прерываться. Такой поток может иметь набор ЦП, содержащий один процессор, тогда как набор ЦП для процесса по умолчанию включает все остальные процессоры.

Среди функций Windows API отсутствует возможность сокращения системного набора ЦП. Эта задача может быть решена вызовом системной функции `NtSetSystemInformation`. Чтобы попытка завершилась успехом, вызывающая сторона должна обладать привилегией `SeIncreaseBasePriorityPrivilege`.

ЭКСПЕРИМЕНТ: НАБОРЫ ЦП

В этом эксперименте вы просмотрите и измените наборы ЦП, а также ознакомитесь с достигнутым эффектом.

1. Загрузите программу CpuSet.exe из архива загружаемых ресурсов книги.
2. Откройте административное окно командной строки и перейдите в каталог, в котором находится файл CPUSET.exe.
3. В окне командной строки введите команду `cpuset.exe` без аргументов, чтобы просмотреть наборы ЦП текущей системы. Результат должен выглядеть примерно так:

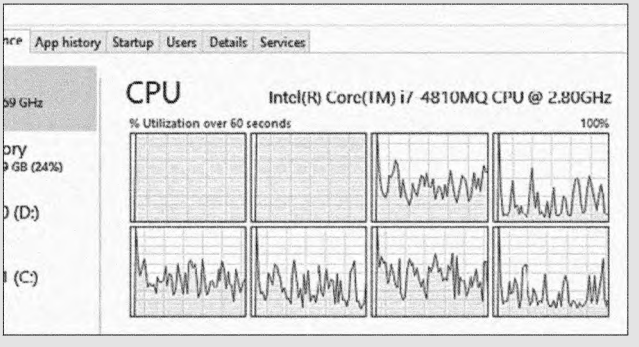
```
System CPU Sets
---------------
Total CPU Sets: 8

CPU Set 0
  Id: 256 (0x100)
  Group: 0
  Logical Processor: 0
  Core: 0
  Last Level Cache: 0
```

```
  NUMA Node: 0
  Flags: 0 (0x0) Parked: False Allocated: False Realtime: False Tag: 0

CPU Set 1
  Id: 257 (0x101)
  Group: 0
  Logical Processor: 1
  Core: 0
  Last Level Cache: 0
  NUMA Node: 0
  Flags: 0 (0x0) Parked: False Allocated: False Realtime: False Tag: 0
...
```

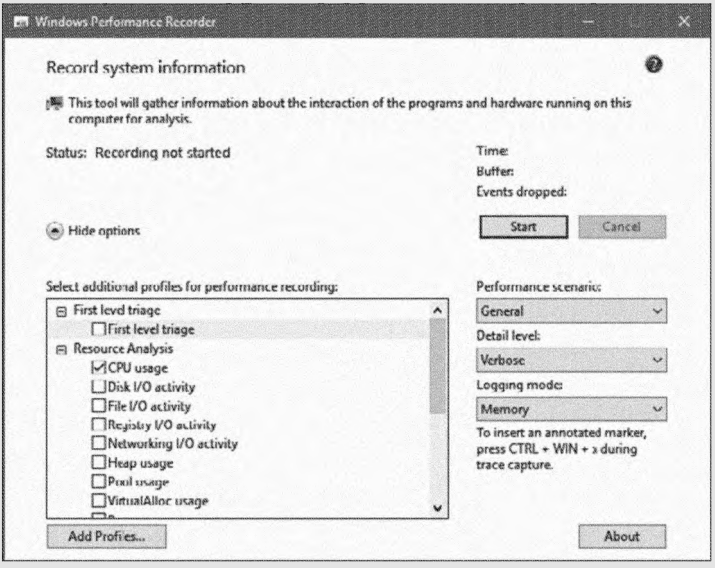
4. Запустите программу CPUSTRES.exe и настройте ее для выполнения одного-двух потоков с максимальным уровнем активности. (Ориентируйтесь приблизительно на 25 % использования процессора.)
5. Откройте диспетчер задач, перейдите на вкладку Быстродействие (Performance) и выберите метку ЦП (CPU).
6. Измените представление графика ЦП так, чтобы на нем отображались отдельные процессоры (если представление настроено для отображения общей загрузки).
7. В командном окне выполните следующую команду (замените число после `-p` идентификатором процесса CPUSTRES в вашей системе):

```
CpuSet.exe -p 18276 -s 3
```

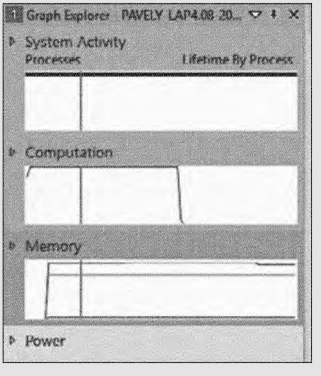
Аргумент `-s` задает маску процессоров, которая назначается по умолчанию для процесса. В данном случае значение 3 означает процессоры 0 и 1. В диспетчере задач видно, что эти процессоры интенсивно загружены работой:



8. Рассмотрим процессор 0 более внимательно, чтобы увидеть, какие потоки на нем выполняются. Для этого можно воспользоваться программами Windows Performance Recorder (WPR) и Windows Performance Analyzer (WPA) из Windows SDK. Щелкните на кнопке Пуск (Start), введите строку `WPR` и выберите Windows Performance Recorder. Подтвердите запрос на повышение прав. На экране должно появиться следующее диалоговое окно:

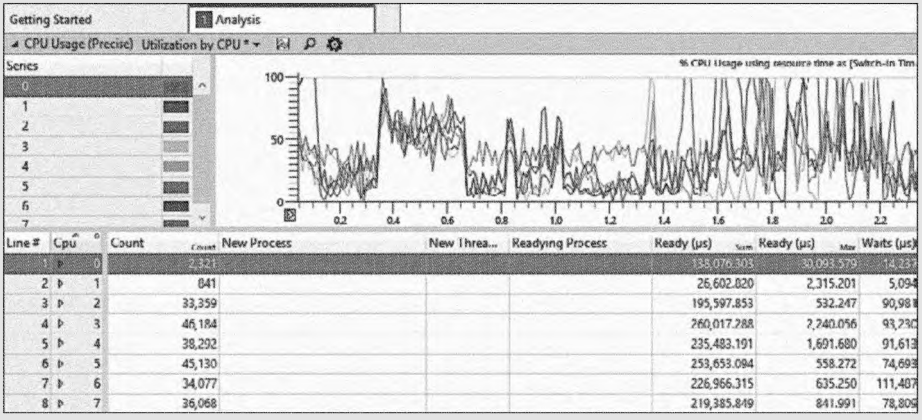


9. По умолчанию записывается информация об использовании процессора — то, что нам нужно. Программа записывает события ETW (Event Tracing for Windows.) (Подробнее о ETW см. в главе 8 части 2.) Щелкните на кнопке Start в диалоговом окне, а через одну-две секунды щелкните на той же кнопке, на которой теперь выводится надпись Save.
10. WPR предложит место для хранения записанных данных. Подтвердите его или выберите другой файл/папку.
11. После того как файл будет сохранен, WPR предложит открыть файл при помощи WPA. Согласитесь на это предложение.
12. Программа WPA открывает и загружает сохраненный файл. (WPA — мощный инструмент, но его обсуждение выходит за рамки темы этой книги.) Слева отображаются различные категории сохраненной информации, которые выглядят примерно так:



13. Раскройте узел Computation, а затем узел CPU Usage (Precise).

14. Сделайте двойной щелчок на графике Utilization by CPU. Он должен открыться на главной панели:



Line #	Cpu	Count	New Process	New Threa...	Readying Process	Ready (µs)	Ready (µs)	Waits (µs)
1	0	2,321				138,076.303	30,093.579	14,237
2	1	841				26,602.820	2,315.201	5,094
3	2	33,359				195,597.853	532.247	90,981
4	3	46,184				260,017.288	2,240.056	93,230
5	4	38,292				235,483.191	1,691.680	91,613
6	5	45,130				253,653.094	558.272	74,693
7	6	34,077				226,966.315	635.250	111,407
8	7	36,068				219,385.849	841.991	78,809

15. На данный момент нас интересует процессор 0. На следующем шаге мы заставим процессор 0 работать только на CPUSTRES. Для начала раскройте узел CPU 0. Вы увидите различные процессы, включая CPUSTRES, — но конечно, не только:

#	Cpu	Count	New Process	Ne
1	0	2,321		
2		1	System (4)	
3		1	RuntimeBroker.exe (7596)	
4		1	System (4)	
5		1	RuntimeBroker.exe (7596)	
6		1	System (4)	
7		1	MsMpEng.exe (3948)	
8		1	System (4)	
9		1	SearchIndexer.exe (10734)	
10		1	System (4)	
11		1	CPUSTRES.exe (18276)	
12		1	conhost.exe (13580)	
13		1	System (4)	
14		1	conhost.exe (13580)	
15		1	System (4)	
16		1	SHSPipeHost.exe (12344)	
17		1	System (4)	
18		1	SHSPipeHost.exe (12344)	
19		1	System (4)	
20		1	SearchIndexer.exe (10734)	
21		1	System (4)	
22		1	SearchIndexer.exe (10734)	

16. Введите приведенную ниже команду, чтобы ограничить систему использованием всех процессоров, кроме первого. В этой системе количество процессоров равно 8, поэтому полная маска равна 255 (0xff). При удалении

процессора 0 получаем 254 (0xfe). Замените маску соответствующей вашей системе:

```
CpuSet.exe -s 0xfe
```

17. Представление в диспетчере задач должно выглядеть примерно так же. А теперь внимательнее присмотримся к процессору 0. Снова запустите WPR и повторите запись в течение одной-двух секунд с теми же настройками.
18. Откройте трассировку в WPA и перейдите к графику Utilization by CPU.
19. Раскройте узел CPU 0. Вы увидите, что CPUSTRES выполняется практически монопольно, с редкими появлениями процесса System.

	Cpu	Count	New Process
1	0	15	
2		1	CPUSTRES.exe (18276)
3		1	CPUSTRES.exe (18276)
4		1	CPUSTRES.exe (18276)
5		1	CPUSTRES.exe (18276)
6		1	System (4)
7		1	CPUSTRES.exe (18276)
8		1	CPUSTRES.exe (18276)
9		1	CPUSTRES.exe (18276)
0		1	System (4)
1		1	CPUSTRES.exe (18276)
2		1	CPUSTRES.exe (18276)
3		1	CPUSTRES.exe (18276)
4		1	CPUSTRES.exe (18276)
5		1	CPUSTRES.exe (18276)

20. Обратите внимание: в столбце CPU Usage (in View) (ms) время, проведенное за процессом System, чрезвычайно мало (несколько микросекунд). Очевидно, процессор 0 выделен под процесс CPUSTRES.
21. Снова выполните CPUSET.exe без аргументов. Первый набор (процессор 0) снабжается пометкой Allocated: True, потому что он теперь выделен под конкретный процесс, а не для общесистемного использования.
22. Закройте программу CPU Stress.
23. Введите следующую команду, чтобы восстановить системный набор ЦП по умолчанию:

```
Cpuset -s 0
```

Выбор потока на многопроцессорных системах

Перед более подробным рассмотрением многопроцессорных систем подведем краткий итог по алгоритмам, рассмотренным в разделе «Выбор потока». По этим алгоритмам либо продолжал выполняться текущий поток (если не был найден новый кандидат), либо запускался поток простоя (если текущий поток блокировался). Но есть еще и третий алгоритм выбора потока, который реализован в функции

KiSearchForNewThread. Этот алгоритм вызывается в одном конкретном случае: когда текущий поток близок к блокировке из-за ожидания объекта, включая ситуацию вызова функции NtDelayExecutionThread, которая так же известна в Windows, как API-функция спячки — Sleep API.

ПРИМЕЧАНИЕ Между часто используемым вызовом Sleep(1), вызывающим блокировку текущего потока до следующего такта таймера, и показанным ранее вызовом SwitchToThread() есть небольшая разница. При вызове Sleep(1) используется алгоритм, который вскоре будет рассмотрен, в то время как при вызове SwitchToThread используется ранее рассмотренная логика.

Функция KiSearchForNewThread сначала проверяет, был ли поток уже выбран для выполнения на этом процессоре (путем чтения поля NextThread); если был, то тут же происходит диспетчеризация этого потока в состояние выполняемого (Running). В противном случае вызывается процедура выбора готового потока KiSelectReadyThreadEx, и, если поток был найден, выполняются такие же действия.

Если же поток не был найден, процессор помечается простаивающим (даже если поток простоя еще не выполняется) и инициируется сканирование очередей другого логического процессора (в отличие от других стандартных алгоритмов, которые теперь отказались бы от дальнейших попыток). С другой стороны, если ядро процессора в данный момент запарковано, алгоритм не будет пытаться проверить другие логические процессоры, поскольку предпочтительнее позволить ядру войти в состояние парковки, нежели держать его загруженным новой работой.

Исключая эти два сценария, теперь запускается цикл захвата работы. Код этого цикла смотрит на текущий NUMA-узел и удаляет все простаивающие процессоры, поскольку они не могут иметь потоков, нуждающихся в захвате. Затем код просматривает общую очередь готовности текущего процессора и вызывает KiSearchForNewThreadOnProcessor в цикле. Если поток не будет найден, сходство изменяется на следующую группу, и функция вызывается снова. Тем не менее на этот раз целевой процессор переводит ее на общую очередь следующей группы вместо текущей, в результате чего процессор находит наилучший готовый поток из очереди готовности группы другого процессора. Если поток и в этом случае не будет найден, происходит аналогичный поиск в локальных очередях процессоров этой группы. Если и на этот раз ничего не найдено и включен планировщик распределенного справедливого долевого планирования (DFSS, Distributed Fair Share Scheduler), то поток, который был в очереди простоя удаленного логического процессора, освобождается на текущем процессоре (если это возможно).

Если кандидаты из готовых потоков найдены не были, попытка повторяется в отношении следующего логического процессора с меньшим номером, и т. д., пока не закончатся все логические процессоры текущего NUMA-узла. В таком случае алгоритм продолжит поиск в следующем ближайшем узле, и т. д., пока не закончатся все узлы в текущей группе (Windows позволяет отдельно взятому потоку иметь связь сходства только с одной группой). Если не будет найден ни один

кандидат, функция возвращает NULL, и процессор в случае ожидания переходит к выполнению потока простоя (при этом планирование простоя пропускается). Если эта работа уже была выполнена планировщиком простоя, процессор входит в состояние спячки.

Выбор процессора

До сих пор мы рассматривали те способы, которые используются Windows при выборе потока, когда такой выбор нужен логическому процессору (или должен быть сделан для заданного логического процессора), и предполагали, что у различных процедур планирования есть существующая база данных готовых потоков, из которых можно сделать этот выбор. Теперь стоит разобраться, как происходит изначальное заполнение этой базы данных — другими словами, как Windows решает, с очередью готовых потоков какого логического процессора должен быть связан заданный готовый поток. После рассмотрения типов многопроцессорных систем, поддерживаемых Windows, а также сходства потоков и настроек идеального процессора, теперь можно рассмотреть порядок использования этой информации для выбора процессора.

Выбор процессора для потока при наличии простаивающих процессоров

Когда поток становится готовым к выполнению, вызывается функция планировщика KiDeferredReadyThread, которая заставляет Windows выполнять две задачи:

- настроить приоритеты и обновить кванты по мере необходимости (об этом уже было рассказано в разделе «Повышение приоритета»);
- выбрать для потока наиболее подходящий логический процессор.

Сначала Windows ищет для потока идеальный процессор, а затем вычисляет набор идеальных процессоров, исходя из принадлежащей потоку жестко заданной маски сходства. Затем этот набор сокращается следующим образом.

1. Удаляются любые простаивающие логические процессоры, запаркованные с помощью механизма парковки ядра. При отсутствии простаивающих процессоров выбор таких процессоров прекращается, и планировщик ведет себя как при отсутствии доступных простаивающих процессоров (см. следующий раздел).
2. Удаляются любые простаивающие логические процессоры, не относящиеся к идеальному узлу (определенному как узел, содержащий идеальный процессор), если только это не вызовет исключения всех простаивающих процессоров.
3. На SMT-системах удаляются любые непростаивающие SMT-наборы, даже если это может вызвать исключение именно того процессора, который считался

идеальным. Иными словами, Windows отдает приоритет неидеальному, простаивающему SMT-набору перед идеальным процессором.

4. Затем Windows проверяет, находится ли идеальный процессор среди оставшихся наборов простаивающих процессоров. Если нет, то система должна затем найти наиболее подходящий простаивающий процессор. Это делается путем предварительной проверки, не является ли процессор, на котором поток выполнялся в последний раз, частью набора оставшихся простаивающих процессоров. Если да, то этот процессор рассматривается как временный идеальный процессор, и выбор останавливается на этом процессоре. (Следует напомнить, что идеальный процессор пытается максимизировать число попаданий в кэш-память процессора, и выбор последнего процессора, на котором выполнялся поток, является неплохим способом добиться этого результата.) Если последний процессор не является частью набора оставшихся простаивающих процессоров, Windows проверяет, не является ли частью этого набора текущий процессор (т. е. процессор, который в данный момент выполняет код планирования). Если он относится к этому набору, то к нему применяется та же самая логика, которая была описана в предыдущем пункте.

5. Если простаивающим не является ни последний, ни текущий процессор, Windows выполняет еще одну операцию сокращения, удаляя любой простаивающий логический процессор, не входящий в тот же SMT-набор, что и идеальный процессор. Если таких процессоров уже не осталось, Windows вместо этого удаляет любые процессоры, не входящие в SMT-набор текущего процессора (если только это не приведет к удалению всех простаивающих процессоров). Иными словами, Windows предпочитает простаивающие процессоры, которые совместно используют тот же самый SMT-набор, что и недоступный идеальный процессор и/или последний процессор, который было бы предпочтительнее выбрать в первую очередь. Поскольку реализации SMT совместно используют кэш-память, имеющуюся в ядре, это будет иметь с точки зрения рационального использования кэш-памяти примерно тот же эффект, что и выбор идеального или последнего процессора.

6. Если после предыдущего шага в идеальном наборе останется более одного процессора, Windows выбирает в качестве текущего процессора потока процессор с меньшим номером.

После выбора процессора, на котором будет выполняться поток, этот поток переводится в состояние повышенной готовности (standby), и PRCB-блок простаивающего процессора обновляется указателем на этот поток. Если процессор простаивает, но не остановлен, ему отправляется DPC-прерывание, чтобы процессор тут же приступил к операции планирования. Каждый раз, когда инициируется такая операция планирования, вызывается функция `KiCheckForThreadDispatch`. Она обнаруживает, что для выполнения на процессоре был спланирован новый поток, что немедленно становится причиной контекстного переключения, если это возможно (а также уведомлением Autoboost о переключении и доставке отложенных APC-вызовов). Если же простаивающих потоков нет, это становится причиной отправки DPC-прерывания.

Выбор процессора для потока при отсутствии простаивающих процессоров

Если при необходимости выполнения потока простаивающие процессоры отсутствуют или если при первом сокращении (с исключением запаркованных простаивающих процессоров) были удалены все простаивающие процессоры, Windows сначала проверяет, не сложилась ли последняя из этих ситуаций. При таком сценарии планировщик вызывает функцию `KiSelectCandidateProcessor`, чтобы спросить у механизма парковки ядра, какой из процессов будет лучшим кандидатом. Механизм парковки выбирает процессор с наибольшим номером, который незапаркован на простаивающем идеальном узле. Если таких процессоров нет, механизм принудительно отменяет состояние парковки идеального процессора и заставляет его распарковаться. По возвращении в планировщик будет проведена проверка, является ли полученный кандидат простаивающим, и если да, планировщик выберет этот процессор для потока, следуя таким же последним шагам, как и в предыдущем сценарии.

При неудаче Windows нужно определить, следует ли вытеснить текущий выполняемый поток. Сначала нужно выбрать целевой процессор. Выбор осуществляется в определенном порядке: идеальный процессор потока, последний процессор, на котором выполнялся поток, первый доступный процессор текущего NUMA-узла, ближайший процессор другого NUMA-узла и все процессоры без ограничений сходства.

После того как процессор будет выбран, следующий вопрос — должен ли новый поток вытеснить текущий поток на этом процессоре? Для этого сравниваются ранги двух потоков. Ранг представляет собой внутренний номер планирования, обозначающий относительную силу потока на основании его группы планирования и других факторов (групповое планирование и ранги рассматриваются в разделе «Групповое планирование» этой главы). Если ранг нового потока равен 0 (высший), или он ниже ранга текущего потока, или ранги равны, но приоритет нового потока выше, чем у текущего, происходит вытеснение. Текущий выполняемый поток помечается для вытеснения, а Windows ставит в очередь DPC-прерывание на целевой процессор для вытеснения текущего выполняемого потока в пользу нашего нового потока.

Если готовый поток не может быть запущен немедленно, он переводится в состояние готового потока и помещается в очередь потоков того уровня приоритета, который соответствует уровню приоритета этого потока, где он будет дожидаться своей очереди на запуск. Как было показано в ранее рассмотренных сценариях планирования, поток будет вставлен либо в начало, либо в конец очереди, в зависимости от того, входил ли он в состояние готовности из-за вытеснения.

ПРИМЕЧАНИЕ Независимо от действующего сценария и различных возможных вариантов, потоки в основном помещаются в индивидуальные очереди готовности своих идеальных процессоров, что гарантирует постоянство алгоритмов, которые определяют, как логический процессор выбирает поток для выполнения.

Неоднородное планирование (big.LITTLE)

Ядро предполагает использование SMP-системы, как описано ранее. Однако некоторые процессоры на базе ARM содержат несколько ядер, которые отличаются друг от друга. Типичный ARM-процессор (например, от Qualcomm) содержит несколько мощных ядер, которые должны работать в течение коротких периодов времени (и потреблять больше энергии), и набор более слабых ядер, которые работают более продолжительное время (и потребляют меньше энергии). Иногда такая конфигурация обозначается термином *big.LITTLE*.

В Windows 10 появилась возможность различать такие ядра и планировать потоки на основании размера и политики ядра, включая статус первого плана потока, его приоритет и ожидаемое время выполнения. Windows инициализирует набор процессоров при инициализации диспетчера электропитания вызовом `PopInitializeHeteroProcessors` (а также при горячем добавлении процессоров в систему). Функция позволяет моделировать разнородные системы (например, для целей тестирования) посредством добавления разделов в раздел реестра HKLM\System\CurrentControlSet\Control\Session Manager\Kernel\KGroups.

Раздел должен содержать две десятичные цифры, идентифицирующие номер группы процессора. (Вспомните, что каждая группа содержит максимум 64 процессора.) Например, 00 — первая группа, 01 — вторая и т. д. (Для большинства систем одной группы будет достаточно.)

- Каждый раздел должен содержать параметр `SmallProcessorMask` типа `DWORD`, в котором должна храниться маска процессоров, считающихся «малыми». Например, если значение равно 3 (установлены первые два бита), а группа содержит шесть процессоров, это будет означать, что процессоры 0 и 1 (3 = 1 или 2) относятся к малым, а другие четыре процессора — к большим. Фактически это то же, что и маска сходства.
- Ядро содержит несколько параметров политики, которые могут настраиваться для разнородных систем; эти параметры хранятся в глобальных переменных. В табл. 4.5 приведены некоторые из этих переменных и их значения.

Таблица 4.5. Переменные ядра для разнородных систем

Имя переменной	Смысл	Значение по умолчанию
`KiHeteroSystem`	Система является разнородной	`False`
`PopHeteroSystem`	Тип разнородной системы: `None` (0) `Simulated` (1) `EfficiencyClass` (2) `FavoredCore` (3)	None (0)

Таблица 4.5 (*окончание*)

Имя переменной	Смысл	Значение по умолчанию
`PpmHeteroPolicy`	Политика планирования: `None` (0) `Manual` (1) `SmallOnly` (2) `LargeOnly` (3) `Dynamic` (4)	Dynamic (4)
`KiDynamicHeteroCpuPolicyMask`	Определяет, что должно учитываться при оценке важности потока	7 (статус первого плана = 1, приоритет = 2, ожидаемое время выполнения = 4)
`KiDefaultDynamicHeteroCpuPolicy`	Поведение разнородной политики Dynamic (см. выше): `All` (0) (все доступные) `Large` (1) `LargeOrIdle` (2) `Small` (3) `SmallOrIdle` (4) Dynamic (5) (решение принимается на основе приоритета и других метрик) `BiasedSmall` (6) (решение принимается на основе приоритета и других метрик, но предпочтение отдается малым процессорам) `BiasedLarge` (7)	`Small` (3)
`KiDynamicHeteroCpuPolicyImportant`	Политика для динамического потока, который считается важным (возможные значения см. выше)	`LargeOrIdle` (2)
`KiDynamicHeterCpuPolicyImportantShort`	Политика для динамического потока, который считается важным, но выполняется короткое время	`Small` (3)
`KiDynamicCpuPolicyExpectedRuntime`	Величина времени выполнения	5,200 мс
`KiDynamicHeteroCpuPolicyImportantPriority`	Приоритет, выше которого потоки считаются важными в случае выбора динамической политики на основе приоритетов	8

Динамические политики (см. табл. 4.5) должны преобразовываться в характеристику важности на основании `KiDynamicHeteroPolicyMask` и состояния потока. Эта задача решается функцией `KiConvertDynamicHeteroPolicy`, которая по порядку проверяет состояние первого плана потока, его приоритет относительно `KiDynamicHeteroCpuPolicyImportantPriority` и его ожидаемое время выполнения. Если поток считается важным (если определяющим фактором является время выполнения, оно также может быть коротким), для принятия решений из области планирования используется политика, основанная на важности. (В табл. 4.5 это `KiDynamicHeteroCpuPolicyImportantShort` или `KiDynamicHeteroCpuPolicyImportant`.)

Групповое планирование

В предыдущем разделе описана стандартная реализация планирования на базе потоков в Windows. С момента своего появления в первом выпуске Windows NT (и с последующими улучшениями масштабируемости в каждой последующей версии) она надежно обеспечивала работу общих пользовательских и серверных сценариев. Однако из-за того, что планирование на базе потоков пытается обеспечить справедливое использование процессора или процессоров только между конкурирующими потоками с одинаковым приоритетом, оно не учитывает высокоуровневые требования — такие, как распределение потоков к пользователям и возможность выделения некоторым пользователям большей доли процессорного времени за счет других пользователей. Такой подход создает проблемы в средах терминальных служб, в которых десятки пользователей конкурируют за процессорное время. Если использовать только потоковое планирование, один высокоприоритетный поток одного пользователя может оставить без ресурсов потоки всех остальных пользователей этой машины.

В Windows 8 и Server 2012 появился механизм группового планирования, построенный на основе концепции групп планирования (`KSCHEDULING_GROUP`). Для группы планирования поддерживается политика, параметры планирования (см. далее) и список управляющих блоков планирования ядра (`KSCB`), по одному на каждый процессор в группе планирования. С другой стороны, поток содержит указатель на группу планирования, к которой он принадлежит. Если этот указатель равен `null`, это означает, что поток находится за пределами групп планирования.

На рис. 4.19 изображена структура группы планирования. На диаграмме потоки П1, П2 и П3 принадлежат группе планирования, а поток 4 — нет.

Несколько терминов, относящихся к групповому планированию:

- **Поколение** — период времени, в течение которого отслеживается использование процессора.
- **Квота** — процессорное время, разрешенное группе на поколение. Исчерпание квоты означает, что группа израсходовала весь свой «бюджет».
- **Вес** — относительная важность группы от 1 до 9 (по умолчанию 5).

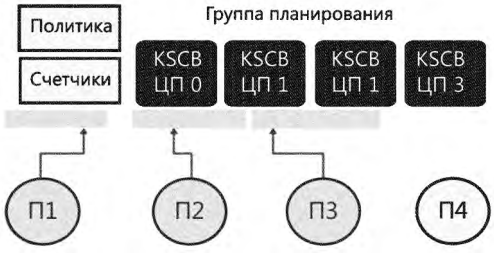


Рис. 4.19. Группа планирования

- **Справедливое долевое планирование** — с этим видом планирования потокам, исчерпавшим квоту, могут выделяться циклы простоя (при отсутствии потоков в пределах квоты, которые желают выполниться).

Структура `KSCB` содержит информацию, относящуюся к процессору:

- Использование циклов для данного поколения.
- Долгосрочное среднее использование (чтобы всплеск активности потока можно было отличить от настоящего поглощения ресурсов).
- Управляющие флаги — например, флаг жесткого ограничения, означающий, что даже при доступности процессорного времени сверх назначенной квоты оно не будет выделяться потоку.
- Очереди готовности, основанные на стандартных приоритетах (только от 0 до 15, потому что потоки реального времени никогда не входят в группы планирования).

Также для групп планирования поддерживается *ранг* — важный параметр, который может рассматриваться как приоритет планирования целой группы потоков. Ранг со значением 0 является наивысшим. Более высокое значение ранга означает, что группа израсходовала больше процессорного времени и поэтому с меньшей вероятностью снова получит процессорное время.

Ранг всегда превосходит приоритет. Это означает, что при наличии двух потоков с разными рангами предпочтение будет отдано потоку с меньшим значением ранга — независимо от приоритета. Потоки с равным рангом сравниваются на основании приоритетов. Ранг периодически обновляется по мере роста использованного процессорного времени.

Ранг 0 является наивысшим, он всегда превосходит большие значения ранга и неявно используется для некоторых потоков. Он может означать следующее:

- Поток не входит ни в одну группу планирования («обычные» потоки).
- Потоки, не израсходовавшие квоту.
- Потоки с приоритетами реального времени (16–31).

С

- Потоки, выполняемые на уровне IRQL APC_LEVEL (1) внутри критической или защищенной области ядра (API и регионы более подробно рассматриваются в главе 8 части 2).

В различных точках выбора, связанных с планированием (например, KiQuantumEnd), решение о том, какой поток нужно спланировать следующим, учитывает группу планирования (если она есть) текущего и готового потока. Если группа планирования существует, более низкий ранг побеждает; за ним следует приоритет (в случае равенства рангов) и первый поступивший поток (при равенстве приоритетов; в конце кванта происходит циклическое смещение).

Распределенное справедливое долевое планирование

Механизм распределенного справедливого долевого планирования (DFSS, Dynamic Fair Share Scheduling) используется для справедливого распределения процессорного времени между сеансами на машине. Он предотвращает возможную монополизацию процессора одним сеансом, если некоторые потоки, выполняемые в пределах этого сеанса, обладают относительно высоким приоритетом и часто выполняются. Этот механизм включается по умолчанию в системах Windows Server с ролью Удаленного рабочего стола. Однако он может быть настроен в любой системе, клиентской или серверной. Его реализация основана на групповом планировании, описанном в предыдущем разделе.

В ходе самой последней части инициализации системы, когда SMSS инициализирует куст реестра SOFTWARE, диспетчер процессов инициирует в функции PsBootPhaseComplete завершающую инициализацию после загрузки, которая вызывает функцию PsInitializeCpuQuota. Именно здесь система принимает решение, какой из двух механизмов квот центрального процессора (DFSS или традиционный) будет задействован. Для включения DFSS следует присвоить ненулевое значение параметру реестра EnableCpuQuota в обоих разделах квот: HKLM\SOFTWARE\Policies\Microsoft\Windows\Session Manager\Quota System для настроек, основанных на политике, а также в системном разделе HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Quota System, в котором определяется, поддерживает ли система эту функциональную возможность (по умолчанию содержит TRUE в Windows Server с ролью Удаленного рабочего стола).

Если планирование DFSS включено, глобальной переменной PsCpuFairShareEnabled присваивается TRUE; это означает, что все потоки принадлежат группам планирования (кроме процессов сеанса 0). Параметры конфигурации DFSS читаются из упомянутого раздела вызовом PspReadDfssConfigurationValues и сохраняются в глобальных переменных. Эти разделы отслеживаются системой. В случае их изменения обратный вызов уведомления снова вызывает PspReadDfssConfigurationValues для обновления конфигурации. В табл. 4.6 представлены значения с краткими описаниями.

Таблица 4.6. Параметры конфигурации DFSS в реестре

Имя параметра	Имя переменной ядра	Смысл	Значение по умолчанию
DfssShortTermSharingMS	PsDfssShortTermSharingMS	Время, необходимое для увеличения ранга группы в цикле	30 мс
DfssLongTermSharingMS	PsDfssLongTermSharingMS	Время, необходимое для перехода от ранга 0 к ненулевому рангу при превышении квоты потоками в цикле	15 мс
DfssGenerationLengthMS	PsDfssGenerationLengthMS	Время отслеживания использования процессора	600 мс
DfssLongTermFraction1024	PsDfssLongTermFraction1024	Значение, используемое в формуле для экспоненциального скользящего целого, используемого при вычислении циклов	512

После включения DFSS каждый раз при создании нового сеанса (кроме сеанса 0) `MiSessionObjectCreate` выделяет группу планирования, связанную с сеансом с весом по умолчанию 5 — средним между минимумом 1 и максимумом 9. Группа планирования управляет либо информацией DFSS, либо информацией долевого использования процессоров (см. следующий раздел) на основании структуры политики (`KSCHEDULING_GROUP_POLICY`), входящей в группу планирования. Поле `Type` указывает, настроена ли она для DFSS (`WeightBased=0`) или долевого использования (`RateControl=1`). `MiSessionObjectCreate` вызывает `KeInsertSchedulingGroup` для вставки группы планирования в глобальный системный список (хранящийся в глобальной переменной `KiSchedulingGroupList` и необходимый для пересчета весов в случае горячего добавления процессоров). Указатель на полученную группу планирования также хранится в структуре `SESSION_OBJECT` конкретного сеанса.

ЭКСПЕРИМЕНТ: DFSS В ДЕЙСТВИИ

В этом эксперименте вы настроите систему для использования DFSS и понаблюдаете за ней в действии.

1. Добавьте разделы и параметры реестра так, как описано в этом разделе, для включения DFSS в системе. (Также этот эксперимент можно провести

с виртуальной машиной.) Затем перезапустите систему, чтобы изменения вступили в силу.

2. Чтобы убедиться в том, что система DFSS активна, откройте сеанс отладки ядра и просмотрите значение `PsCpuFairShareEnabled` следующей командой (значение 1 означает активность DFSS):

```
lkd> db nt!PsCpuFairShareEnabled L1
fffff800'5183722a 01
```

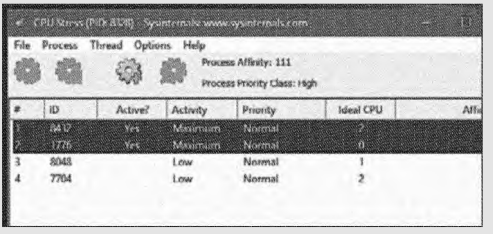
3. В отладчике просмотрите текущий поток. (Это должен быть один из потоков, в которых выполняется WinDbg.) Обратите внимание на то, что поток является частью группы планирования и его поле KSCB отлично от NULL, потому что поток выполнялся на момент отображения информации.

```
lkd> !thread
THREAD ffffd28c07231640 Cid 196c.1a60 Teb: 000000f897f4b000 Win32Thread:
ffffd28c0b9b0b40 RUNNING on processor 1
IRP List:
    ffffd28c06dfac10: (0006,0118) Flags: 00060000 Mdl: 00000000
Not impersonating
DeviceMap                 ffffac0d33668340
Owning Process            ffffd28c071fd080       Image:      windbg.exe
Attached Process          N/A            Image:         N/A
Wait Start TickCount      6146           Ticks: 33 (0:00:00:00.515)
Context Switch Count      877            IdealProcessor: 0
UserTime                  00:00:00.468
KernelTime                00:00:00.156
Win32 Start Address 0x00007ff6ac53bc60
Stack Init ffffbf81ae85fc90 Current ffffbf81ae85f980
Base ffffbf81ae860000 Limit ffffbf81ae85a000 Call 0000000000000000
Priority 8 BasePriority 8 PriorityDecrement 0 IoPriority 2 PagePriority 5
Scheduling Group: ffffd28c089e7a40 KSCB: ffffd28c089e7c68 rank 0
```

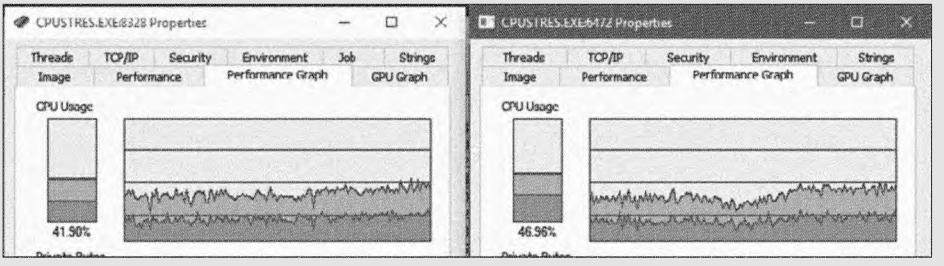
4. Введите команду `dt` для просмотра группы планирования:

```
lkd> dt nt!_kscheduling_group ffffd28c089e7a40
   +0x000 Policy           : _KSCHEDULING_GROUP_POLICY
   +0x008 RelativeWeight   : 0x80
   +0x00c ChildMinRate     : 0x2710
   +0x010 ChildMinWeight   : 0
   +0x014 ChildTotalWeight : 0
   +0x018 QueryHistoryTimeStamp : 0xfed6177
   +0x020 NotificationCycles : 0n0
   +0x028 MaxQuotaLimitCycles : 0n0
   +0x030 MaxQuotaCyclesRemaining : 0n-73125382369
   +0x038 SchedulingGroupList : _LIST_ENTRY [ 0xfffff800'5179b110 -
0xffffd28c'081b7078 ]
   +0x038 Sibling          : _LIST_ENTRY [ 0xfffff800'5179b110 -
0xffffd28c'081b7078 ]
   +0x048 NotificationDpc  : 0x0002eaa8'0000008e _KDPC
   +0x050 ChildList        : _LIST_ENTRY [ 0xffffd28c'062a7ab8 -
0xffffd28c'05c0bab8 ]
   +0x060 Parent           : (null)
   +0x080 PerProcessor     : [1] _KSCB
```

5. Создайте другого локального пользователя на машине.
6. Запустите CPU Stress в текущем сеансе.
7. Запустите несколько потоков на максимальном уровне активности, но не настолько, чтобы перегрузить машину. Например, на следующей иллюстрации показаны два потока с максимальным уровнем активности на трехпроцессорной виртуальной машине:



8. Нажмите Ctrl+Alt+Del и выберите команду Сменить пользователя (Switch User). Затем выполните вход с учетной записью другого пользователя, которую вы создали.
9. Снова запустите CPU Stress с тем же количеством потоков на максимальном уровне активности.
10. Для процесса CPUSTRES откройте меню Process, выберите команду Priority Class и выберите вариант High, чтобы сменить класс приоритета процесса. Без DFSS этот процесс с более высоким приоритетом должен поглощать большую часть процессора. Это связано с тем, что четыре потока конкурируют за три процессора. Один из них проигрывает; этот поток должен быть из процесса с низким приоритетом.
11. Откройте Process Explorer, сделайте двойной щелчок на обоих процессах CPUSTRES и перейдите на вкладку Performance Graph.
12. Разместите оба окна поблизости. Вы должны увидеть, что процессы поглощают процессор приблизительно одинаково, хотя их приоритеты не равны:



13. Отключите DFSS, удалив разделы из реестра. Перезапустите систему.
14. Повторите эксперимент. Различия должны быть хорошо видны: высокоприоритетный процесс получает большую часть процессорного времени.

Ограничения долевого использования процессоров

Работа DFSS основана на автоматическом помещении новых потоков в группу планирования в сеансе. Такое решение хорошо работает в сценарии со службами терминалов, но вряд ли оно подойдет для ограничения процессорного времени потоков или процессов.

Инфраструктура группы планирования может использоваться на более детализированном уровне с объектом задания. Вспомните, о чем говорилось в главе 3: задание может управлять одним или несколькими процессами. Среди ограничений, которые можно установить для задания, — долевое использование процессора, которое устанавливается вызовом `SetInformationJobObject` с классом информации о задании `JobObjectCpuRateControlInformation` и структурой типа `JOBOBJECT_CPU_RATE_CONTROL_INFORMATION`, содержащей фактические данные. Структура содержит набор флагов, которые дают возможность применить одну из трех настроек для ограничения процессорного времени:

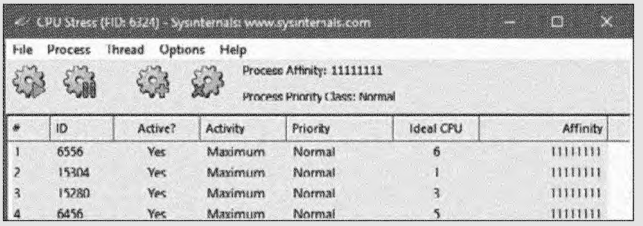
- **Доля использования процессора.** Значение лежит в диапазоне от 1 до 10 000 и представляет процент, умноженный на 100 (например, для 40 % оно равно 4000).
- **Весовой коэффициент.** Значение лежит в диапазоне от 1 до 9 относительно весов других заданий. (DFSS настраивается с этим параметром.)
- **Минимальная и максимальная доля использования процессора.** Эти значения задаются аналогично первому варианту. Когда потоки в задании достигнут максимального процента, заданного для интервала (600 мс по умолчанию), они не смогут получить дополнительное процессорное время до начала следующего интервала. Флаг позволяет указать, нужно ли использовать жесткое ограничение для обеспечения лимита даже в том случае, если доступно свободное процессорное время.

В результате настройки всех этих ограничений все потоки от всех процессов в задании помещаются в новую группу планирования, а группа настраивается в соответствии с инструкциями.

ЭКСПЕРИМЕНТ: ОГРАНИЧЕНИЕ ДОЛЕВОГО ИСПОЛЬЗОВАНИЯ ПРОЦЕССОРА

В этом эксперименте рассматривается ограничение долевого использования процессора с объектом задания. Лучше провести этот эксперимент на виртуальной машине и присоединиться к ее ядру (вместо использования локального ядра) из-за ошибки в отладчике, существующей на момент написания.

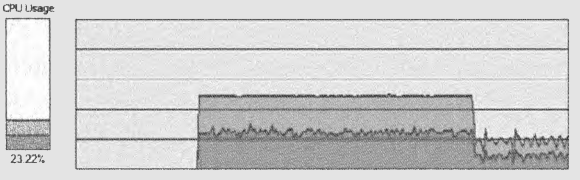
1. Запустите CPU Stress на тестовой VM и настройте несколько потоков для потребления приблизительно 50 % процессорного времени. Например, в восьмипроцессорной системе активизируйте четыре потока с максимальным уровнем активности:



2. Откройте Process Explorer, найдите экземпляр CPUSTRES, откройте его свойства и выберите вкладку Performance Graph. Использование процессора должно составлять приблизительно 50 %.
3. Загрузите программу CPULIMIT из пакета загружаемых ресурсов книги. Эта простая программа позволяет лимитировать использование одного процессора за счет жесткого ограничения.
4. Выполните следующую команду для ограничения использования процессора CPUSTRES 20 %. (Замените число 6324 своим идентификатором процесса.)

   ```
   CpuLimit.exe 6324 20
   ```

5. Снова обратитесь к окну Process Explorer. В нем загрузка должна упасть приблизительно до 20 %:



6. Откройте WinDbg в управляющей системе.
7. Присоедините отладчик к ядру тестовой системы и перейдите к нему.
8. Введите следующую команду для нахождения процесса CPUSTRES:

```
0: kd> !process 0 0 cpustres.exe
PROCESS ffff9e0629528080
    SessionId: 1 Cid: 18b4    Peb: 009e4000 ParentCid: 1c4c
    DirBase: 230803000 ObjectTable: ffffd78d1af6c540 HandleCount: <Data
Not Accessible>
Image: CPUSTRES.exe
```

9. Введите следующую команду, чтобы вывести базовую информацию о процессе:

```
0: kd> !process ffff9e0629528080 1
PROCESS ffff9e0629528080
    SessionId: 1 Cid: 18b4 Peb: 009e4000 ParentCid: 1c4c
    DirBase: 230803000 ObjectTable: ffffd78d1af6c540 HandleCount: <Data
Not Accessible>
    Image: CPUSTRES.exe
    VadRoot ffff9e0626582010 Vads 88 Clone 0 Private 450. Modified 4. Locked 0.
```

```
    DeviceMap ffffd78cd8941640
    Token                             ffffd78cfe3db050
    ElapsedTime                       00:08:38.438
    UserTime                          00:00:00.000
    KernelTime                        00:00:00.000
    QuotaPoolUsage[PagedPool]         209912
    QuotaPoolUsage[NonPagedPool]      11880
    Working Set Sizes (now,min,max) (3296, 50, 345) (13184KB, 200KB, 1380KB)
    PeakWorkingSetSize                3325
    VirtualSize                       108 Mb
    PeakVirtualSize                   128 Mb
    PageFaultCount                    3670
    MemoryPriority                    BACKGROUND
    BasePriority            8
    CommitCharge            568
    Job                     ffff9e06286539a0
```

10. Обратите внимание: объект задания отличен от NULL. Выведите его свойства командой `!job`. Программа создает задание (`CreateJobObject`), добавляет процесс в задание (`AssignProcessToJobObject`) и вызывает `SetInformationJobObject` с классом информации о долевом использовании процессора и значением 2000 (20 %).

```
0: kd> !job ffff9e06286539a0
Job at ffff9e06286539a0
  Basic Accounting Information
    TotalUserTime:             0x0
    TotalKernelTime:           0x0
    TotalCycleTime:            0x0
    ThisPeriodTotalUserTime:   0x0
    ThisPeriodTotalKernelTime: 0x0
    TotalPageFaultCount:       0x0
    TotalProcesses:            0x1
    ActiveProcesses:           0x1
    FreezeCount:               0
    BackgroundCount:           0
    TotalTerminatedProcesses:  0x0
    PeakJobMemoryUsed:         0x248
    PeakProcessMemoryUsed:     0x248
  Job Flags
    [close done]
    [cpu rate control]
Limit Information (LimitFlags: 0x0)
Limit Information (EffectiveLimitFlags: 0x800)
CPU Rate Control
    Rate = 20.00%
    Hard Resource Cap
    Scheduling Group: ffff9e0628d7c1c0
```

11. Снова запустите программу CPULIMIT для того же процесса и снова задайте степень использования процессора на уровне 20 %. Величина потребления процессора программой CPUSTRES падает до 4 %. Это происходит из-за вложения заданий: создается новое задание, которое (как и присоединенный к нему процесс) размещается внутри первого задания. Общий результат составляет 20 % от 20 %, т. е. 4 %.

Динамическое добавление и удаление процессоров

Как вы уже видели, разработчики могут довольно точно настроить процессоры, на которых разрешено выполняться тем или иным процессам (а в случае идеального процессора — на которых они должны выполняться). Все это неплохо работает на системах, имеющих постоянное количество процессоров в ходе всего их рабочего времени. Например, для внесения каких-либо аппаратных изменений в процессор или в количественный состав процессоров настольный компьютер необходимо выключить. Но современные серверные системы не могут позволить себе простаивать, что обычно требуется для замены центрального процессора или для его добавления. Более того, в периоды высокой нагрузки, которая превышает возможности машины при текущем уровне производительности, иногда бывает необходимо добавлять процессоры. Выключение сервера в период пиковой нагрузки полностью противоречит самой цели.

Для решения этой проблемы последнее поколение серверных материнских плат и систем поддерживает возможность добавления (а также удаления) процессоров без прерывания работы машины. ACPI BIOS и соответствующее оборудование машины было специально создано с учетом предоставления такой возможности, но для полной поддержки необходимо участие операционной системы.

Поддержка динамического изменения количества процессоров предоставляется на уровне HAL, который уведомляет ядро о новом процессоре в системе через функцию `KeStartDynamicProcessor`. Эта функция делает примерно то же, что происходит, когда система обнаруживает в ходе запуска более одного процессора и нуждается в инициализации связанных с этим структур. При динамическом добавлении процессора дополнительная работа проделывается различными компонентами системы. Например, диспетчер памяти выделяет новые страницы и структуры памяти, оптимизированные под центральный процессор. Он также инициализирует находящийся в ядре новый стек DPC, в то время как ядро инициализирует глобальную таблицу дескрипторов — GDT (Global Descriptor Table), таблицу диспетчера прерываний — IDT (Interrupt Dispatch Table), область управления процессора — PCR (Processor Control Region), блок управления процессора — PRCB (Process Control Block) и другие связанные с процессором структуры.

Также вызываются и остальные части ядра, относящиеся к исполняющей системе, в основном для инициализации создаваемых для каждого процессора резервных списков (look-aside list), касающихся добавленного процессора. Например, резервные списки, составляемые для каждого процессора, используются диспетчером ввода/вывода, кодом резервных списков исполняющей системы, диспетчером кэша и диспетчером объектов для их часто создаваемых структур.

И наконец, ядро инициализирует потоковую поддержку DPC для процессора и корректирует экспортируемые переменные ядра, чтобы в них отображался новый процессор. Обновляются также различные маски диспетчера памяти и затравки процессов, основанные на количествах процессоров. Также нуждаются в обновле-

нии и свойства процессора, чтобы новый процессор соответствовал всей остальной системе (например, на только что добавленном процессоре включается поддержка виртуализации). Инициализационная последовательность завершается уведомлением компонента архитектуры аппаратных ошибок Windows — WHEA (Windows Hardware Error Architecture) — о том, что новый процессор уже в строю.

HAL также участвует в этом процессе. Он вызывается один раз для запуска динамически добавленного процессора, после того как ядро узнает о нем, и вызывается еще раз после завершения ядром инициализации процессора. Но эти уведомления и функции обратных вызовов всего лишь ставят ядро в известность и отвечают на изменения процессора. Хотя дополнительный процессор повышает пропускную способность ядра, он не делает ничего для помощи драйверам.

Для обработки драйверов в системе есть новый включенный по умолчанию объект функции обратного вызова исполняющей системы, `ProcessorAdd`, который может быть зарегистрирован драйверами для получения уведомлений. Как и те функции обратного вызова, которые уведомляют драйверы о состоянии электропитания системы или об изменении системного времени, эта функция обратного вызова позволяет коду драйвера, к примеру, создавать в случае необходимости новый рабочий поток, чтобы он мог справляться с возросшим объемом работы за то же время.

Как только драйверы будут уведомлены, вызывается последний компонент ядра, называемый диспетчером устройств Plug and Play, который добавляет процессор к узлу устройств системы и проводит перебалансировку прерываний, чтобы новый процессор мог обрабатывать прерывания, которые уже были зарегистрированы для других процессоров. Приложения, требующие повышенных ресурсов центрального процессора, также могут получить преимущества от добавления новых процессоров.

Но внезапное изменение сходства может разрушить изменения для выполняемого приложения (особенно при переходе от однопроцессорной к многопроцессорной среде) из-за появления потенциальных условий гонки или просто нерационального распределения работы (процесс мог вычислить наилучшие соотношения при запуске, основанные на количестве центральных процессоров, о которых ему было известно). В результате приложения не получат никакой выгоды от динамически добавленного процессора автоматически; они должны их запросить.

API-функции Windows `SetProcessAffinityUpdateMode` и `QueryProcessAffinityMode`, использующие недокументированный системный вызов `NtSet/QueryInformationProcess`, сообщают диспетчеру процессов, что этим приложениям нужно обновить свое сходство (флаг `AffinityUpdateEnable` в структуре `EPROCESS`) или что им не требуется иметь дело с обновлениями сходства путем установки флага `AffinityPermanent` в структуре `EPROCESS`. Как только приложение сообщит системе, что его сходство является постоянным, оно не может чуть позже изменить свое мнение и запросить обновление сходства; следовательно, это одноразовое изменение. После того как приложение сообщит системе, что его сходство является постоянным, оно не сможет позже передумать и запросить обновление сходства.

Как часть функции `KeStartDynamicProcessor`, после перебалансировки прерываний был добавлен новый этап, заключающийся в вызове диспетчера процессов для выполнения обновления сходства с помощью функции `PsUpdateActiveProcessAffinity`. Некоторые выполняемые в режиме ядра Windows-процессы и службы уже имеют включенное обновление сходства, а вот программное обеспечение сторонних производителей потребует перекомпиляции для использования новых API-функций. Процесс System, процессы Svchost и Smss совместимы с динамическим добавлением процессоров.

Рабочие фабрики (пулы потоков)

Термин «*рабочие фабрики*» относится к внутреннему механизму, используемому для реализации пулов потоков пользовательского режима. Устаревшие процедуры пула потоков были полностью реализованы в пользовательском режиме внутри библиотеки Ntdll.dll, и Windows API предоставляет различные процедуры для вызова соответствующих подпрограмм, представляющих таймеры ожидания (`CreateTimerQueue`, `CreateTimerQueueTimer` и их аналоги), функции обратного вызова с ожиданием (`RegisterWaitForSingleObject`) и автоматическое создание и удаление потоков (`QueueUserWorkItem`) в зависимости от объема проделываемой работы.

Один из недостатков старой реализации заключался в том, что в процессе можно было создать только один пул потоков, что усложняло реализацию некоторых сценариев. Например, попытки организации приоритетов рабочих элементов за счет построения двух пулов потоков, обслуживающих разные типы запросов, были невозможны. Другая проблема была связана с самой реализацией, которая существовала в пользовательском режиме (в Ntdll.dll.) Поскольку ядро может иметь непосредственный контроль над планированием, созданием и завершением потоков без обычных издержек, связанных с выполнением этих операций из пользовательского режима, большинство функциональных механизмов, требующихся для поддержки реализации в Windows пула потоков пользовательского режима, теперь располагаются в ядре, что также упрощает код, который нужно создавать разработчикам. Например, создание рабочего пула в удаленном процессе может быть осуществлено с помощью единственного API-вызова вместо сложной череды вызовов в виртуальной памяти, которые обычно для этого требовались. В этой модели Ntdll.dll просто предоставляет интерфейсы и высокоуровневые API-функции, необходимые для взаимодействия с кодом рабочей фабрики.

Для управления функциональностью пула потоков ядра используется тип диспетчера объектов, который называется `TpWorkerFactory`, а также четыре системных вызова для управления фабрикой и ее работниками (`NtCreateWorkerFactory`, `NtWorkerFactoryWorkerReady`, `NtReleaseWorkerFactoryWorker`, `NtShutdownWorkerFactory`), двух исходных вызовов запроса/записи информации (`NtQueryInformationWorkerFactory` и `NtSetInformationWorkerFactory`) и вызова

с ожиданием (`NtWaitForWorkViaWorkerFactory`). Как и все другие исходные системные вызовы, эти вызовы предоставляют пользовательскому режиму дескриптор объекта `TpWorkerFactory`, в котором содержится такая информация, как имя и атрибуты объекта, заданная маска доступа и дескриптор безопасности. Но в отличие от других системных вызовов, упакованных Windows API, управление пулом потоков обрабатывается исходным кодом библиотеки Ntdll.dll. Это означает, что разработчики работают с непрозрачными дескрипторами: указателем `TP_POOL` для пула потоков и другими непрозрачными указателями для объектов, созданных на базе пула, включая `TP_WORK` (обратный вызов рабочего элемента), `TP_TIMER` (обратный вызов таймера), `TP_WAIT` (обратные вызовы ожидания) и т. д. В этих структурах содержатся различные фрагменты информации — например, дескриптор объекта `TpWorkerFactory`.

Как следует из названия «рабочая фабрика», реализация этой фабрики отвечает за размещение рабочих потоков (и за вызов кода в заданной точке входа рабочего потока пользовательского режима), поддержку минимального и максимального числа потоков (для постоянных рабочих пулов либо для полностью динамических пулов), а также за другую учетную информацию. Это позволяет выполнять такие операции, как завершение работы пула потоков с помощью единственного обращения к ядру, поскольку ядро является единственным компонентом, отвечающим за создание и завершение потоков.

Поскольку ядро динамически создает новые потоки по мере надобности, основываясь на предоставленных минимальных и максимальных количествах, это также повышает масштабируемость приложений, использующих новую реализацию пула потоков. Рабочая фабрика создаст новый поток при выполнении всех следующих условий:

- Разрешено создание динамических потоков.
- Количество доступных работников ниже максимального количества работников, указанного для фабрики (по умолчанию это количество составляет 500 работников).
- Рабочая фабрика имеет привязанные к ней объекты (например, ALPC-порт, по которому ожидает рабочий поток) или поток, активированный в пуле.
- Существуют отложенные пакеты запросов ввода/вывода (IRP-пакеты; см. главу 6), связанные с рабочим потоком.

Вместе с тем она завершает работу потоков, если те простаивают (т. е. не обрабатывают рабочие элементы) свыше 10 секунд (по умолчанию). Кроме того, хотя при старой реализации разработчики всегда могли воспользоваться максимально возможным количеством потоков, основанным на количестве процессоров в системе, теперь приложения могут использовать пулы потоков для автоматического использования новых процессоров, добавленных в ходе выполнения кода. Эта возможность основана на поддержке динамических процессоров в Windows Server.

Создание рабочей фабрики

Следует заметить, что поддержка рабочей фабрики является всего лишь оберткой для обычных задач, которые в противном случае должны были бы выполняться в пользовательском режиме (с потерей производительности). Основная часть логики нового кода пула потоков остается в той стороне этой архитектуры, которая относится к библиотеке Ntdll.dll. (Теоретически, используя недокументированные функции для рабочих фабрик, можно создавать другую реализацию пула потоков.) К тому же это не код рабочих фабрик предоставляет масштабируемость, внутренние механизмы ожидания и эффективность обработки рабочих заданий. Этим занимается куда более старый компонент Windows: порты завершения ввода/вывода, или, если быть точнее, очереди ядра (`KQUEUE`). Собственно, при создании рабочей фабрики порт завершения ввода/вывода уже должен быть создан кодом пользовательского режима, а его дескриптор должен передаваться при создании.

Именно через этот порт завершения ввода/вывода реализация пользовательского режима будет выстраивать очередь, а также ожидать завершения работы — но путем использования системных вызовов рабочей фабрики вместо API-функций порта завершения ввода/вывода. Но, согласно внутреннему устройству, «освобождающий» вызов рабочей фабрики (который ставит работу в очередь) является оберткой для процедуры `IoSetIoCompletionEx`, которая увеличивает количество отложенной работы, а «ожидающий» вызов является оберткой для процедуры `IoRemoveIoCompletion`. Обе эти процедуры совершают вызов в код реализации очереди ядра. Задачи кода рабочей фабрики следующие: управление либо постоянными, статическими, либо динамическими пулами потоков; инкапсуляция модели порта завершения ввода/вывода в интерфейсах, пытающихся предотвратить остановку продвижения рабочих очередей путем автоматического создания динамических потоков; упрощение глобальной очистки и завершение операций в ходе запроса на завершение работы фабрики; и беспрепятственная блокировка новых запросов к фабрике при таком развитии событий.

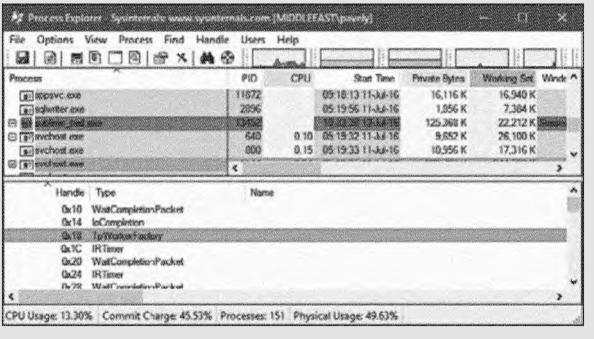
Функция исполняющей подсистемы, которая создает рабочую фабрику `NtCreateWorkerFactory`, получает несколько аргументов для настройки пула потоков (например, максимальное количество создаваемых потоков и исходные размеры фиксируемого и резервируемого стека). Однако API-функция `CreateThreadpool` использует размеры стека по умолчанию, встроенные в исполняемый образ (как и стандартная функция `CreateThread`). Но Windows API не предоставляет способа для переопределения значений по умолчанию, и об этом можно только пожалеть, так как во многих случаях потоки из пула не требуют глубоких стеков вызовов, и было бы эффективнее создать стеки меньшего размера.

Структура данных, используемая реализацией рабочей фабрики, не входит в набор общедоступных символических имен, но, как показано в следующем эксперименте, возможность посмотреть на некоторые рабочие пулы все же есть. Кроме того, API-функция `NtQueryInformationWorkerFactory` выводит дамп почти каждого поля в структуре рабочей фабрики.

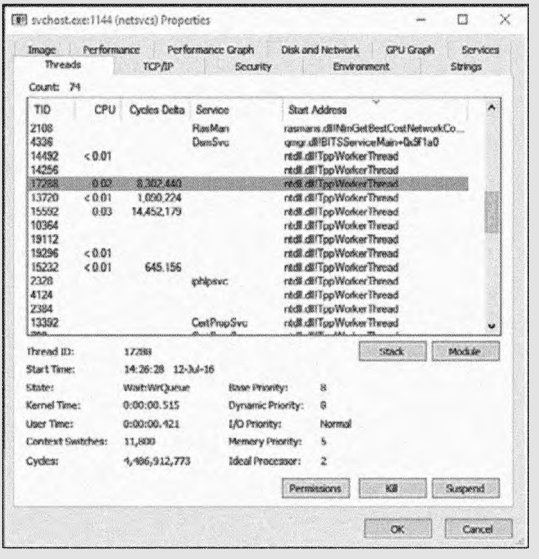
ЭКСПЕРИМЕНТ: ПРОСМОТР ПУЛОВ ПОТОКОВ

Из-за преимуществ использования механизма пула потоков ими пользуются многие основные системные компоненты и приложения, особенно когда они работают с такими ресурсами, как ALPC-порты (для динамической обработки входящих сообщений на соответствующем и масштабируемом уровне). Одним из способов определения тех процессов, которые используют рабочую фабрику, является просмотр списка дескрипторов в Process Explorer. Чтобы посмотреть на некоторые присущие им особенности, выполните следующие действия.

1. Запустите Process Explorer.
2. В меню View установите флажок Show Unnamed Handles And Mappings. (К сожалению, библиотека Ntdll.dll не дает имена рабочим фабрикам, чем и объясняется необходимость данного действия с целью просмотра дескрипторов.)
3. Выберите экземпляр svchost.exe в списке процессов.
4. Откройте меню View и выберите команду Show Lower Pane, чтобы отображать нижнюю панель таблицы дескрипторов.
5. Откройте меню View, выберите команду Lower Pane View и пункт Handles, чтобы таблица отображалась в режиме дескрипторов.
6. Щелкните правой кнопкой мыши на заголовках столбцов панели. Выберите команду Select Columns.
7. Убедитесь в том, что столбцы Type и Handle Value выбраны.
8. Щелкните на заголовке Type, чтобы отсортировать список по типам.
9. Прокрутите список дескрипторов и следите за содержимым столбца Type, пока не найдете дескриптор типа `TpWorkerFactory`.
10. Щелкните на заголовке Handle, чтобы отсортировать список по значению дескриптора. Результат должен выглядеть примерно так, как показано ниже. Обратите внимание: дескриптору `TpWorkerFactory` предшествует дескриптор `IoCompletion`. Как упоминалось ранее, это происходит из-за того, что дескриптор порта завершения ввода/вывода, через который будет отправляться работа, должен быть создан перед созданием рабочей фабрики.



11. Дважды щелкните на выбранном процессе в списке процессов, затем на вкладке Threads, затем на столбце Start Address. Результат должен выглядеть примерно так, как показано ниже. Потоки рабочих фабрик легко узнать по точке входа Ntdll.dll, TppWorkerThread. (Tpp — сокращение от «Thread Pool Private», т. е. «*закрытый пул потоков*».)



Если посмотреть на другие рабочие потоки, можно увидеть, что некоторые из них ожидают объектов (например, событий). У процесса может быть несколько пулов потоков, и каждый пул потоков может иметь различные потоки, выполняющие совершенно не связанные друг с другом задачи. Ту или иную работу назначает разработчик, который вызывает API-функции пула потоков для регистрации этой работы посредством Ntdll.dll.

Заключение

В данной главе была изучена структура процессов, потоков и заданий, рассмотрены способы их создания и то, как система Windows решает, какой поток следует выполнять и насколько долго, а также на каком процессоре или процессорах. Следующая глава посвящена одному из самых важных аспектов любой ОС — управлению памятью.

Глава 5
Управление памятью

В этой главе вы узнаете, как система Windows реализует виртуальную память и как она управляет подмножеством виртуальной памяти, хранящимся в физической памяти. Также будет описана внутренняя структура и компоненты, образующие диспетчер памяти, включая ключевые структуры данных и алгоритмы. Но прежде чем переходить к изучению этих механизмов, мы рассмотрим основные виды сервиса, предоставляемые диспетчером памяти, и ключевые концепции (такие, как зарезервированная память, подтвержденная память и общая память).

Знакомство с диспетчером памяти

По умолчанию виртуальный размер процесса в 32-разрядной версии Windows составляет 2 Гбайт. Если образ помечен как поддерживающий большое адресное пространство, а система загружена в специальном режиме (описанном в разделе «Структура адресных пространств x86» этой главы), 32-разрядный процесс может достигать 3 Гбайт в 32-разрядных версиях Windows и 4 Гбайт в 64-разрядных версиях. Размер виртуального адресного пространства процесса в 64-разрядных версиях Windows 8 и Server 2012 составляет 8192 Гбайт (8 Тбайт), а в 64-разрядной Windows 8.1 (и выше) и Server 2012 R2 (и выше) он увеличивается до 128 Тбайт.

Как было показано в главе 2 (особенно в табл. 2.2), максимальный объем физической памяти, поддерживаемый в Windows в настоящее время, лежит в диапазоне от 2 Гбайт до 24 Тбайт в зависимости от версии и редакции Windows. Так как виртуальное адресное пространство может быть больше или меньше объема физической памяти на машине, у диспетчера памяти есть две основные задачи.

- Трансляция (или отображение) виртуального адресного пространства процесса в физическую память, чтобы при чтении и записи в виртуальном адресном пространстве потоком, выполняющимся в контексте этого процесса, происходило обращение к правильному физическому адресу. (Подмножество виртуального адресного пространства процесса, находящееся в физической памяти, называется *рабочим набором*. Рабочие наборы более подробно рассматриваются в разделе «Рабочие наборы» этой главы.)
- Выгрузка части содержимого памяти на диск при превышении выделения — т. е. когда выполняемые потоки пытаются использовать больше физической памяти,

чем доступно, — и перенос выгруженных данных в физическую память в случае необходимости.

Помимо управления виртуальной памятью, диспетчер памяти предоставляет базовый набор сервисных функций, на основе которых строятся различные подсистемы среды Windows. К числу таких сервисных функций относятся файлы, отображенные в память (во внутренней реализации называемые *объектами секций*), память с копированием при записи и поддержка приложений, использующих большое разреженное адресное пространство. Диспетчер памяти также предоставляет средства для выделения и использования больших объемов физической памяти, которая может одновременно отображаться в виртуальное адресное пространство процесса — например, в 32-разрядных системах с более чем 3 Гбайт физической памяти. Эта тема объясняется далее в разделе «Address Windowing Extensions».

ПРИМЕЧАНИЕ Приложение панели управления — Система (System) — позволяет управлять размером, количеством и местонахождением страничных файлов. Используемая в нем терминология создает впечатление, что виртуальная память и страничный файл — это одно и то же. На самом деле это не так. Страничный файл всего лишь один из аспектов виртуальной памяти. Собственно, даже если страничного файла вообще не будет, Windows все равно будет использовать виртуальную память. Это различие более подробно рассматривается позднее в этой главе.

Компоненты диспетчера памяти

Диспетчер памяти является частью исполняющей среды Windows, а следовательно, существует в файле Ntoskrnl.exe. Это самый большой компонент исполняющей системы, что указывает на его важность и сложность. Никакие части диспетчера памяти не находятся в HAL. Диспетчер памяти состоит из следующих компонентов:

- Набор сервисных функций исполняющей системы для выделения, освобождения виртуальной памяти и управления ею, большинство из которых доступно через Windows API или интерфейсы драйверов устройств режима ядра.
- Обработчик некорректного преобразования и ошибок доступа для обработки исключений управления памятью, обнаруженных на аппаратном уровне, и размещения виртуальных страниц в памяти от имени процесса.
- Шесть ключевых функций верхнего уровня, каждая из которых выполняется в одном из шести потоков режима ядра в процессе System:
  - Диспетчер набора балансировки (`KeBalanceSetManager`, приоритет 17). Вызывает внутреннюю функцию диспетчера рабочих наборов (`MmWorkingSetManager`) каждую секунду, а также при падении объема свободной памяти ниже некоторого порога. Диспетчер рабочих наборов управляет общими политиками управления памятью: усечением рабочих наборов, устареванием и записью измененных страниц.

- Управление подгрузкой процесса/стека (`KeSwapProcessOrStack`, приоритет 23). Выполняет подгрузку и выгрузку стека процессов и потоков ядра. Диспетчер наборов балансировки и код планирования потоков в ядре активизируют этот поток, когда потребуется выполинть операцию подгрузки или выгрузки.
- Запись измененных страниц (`MiModifiedPageWriter`, приоритет 18). Записывает «грязные» страницы из списка модификаций в соответствующие страничные файлы. Поток активизируется, когда требуется сократить размер списка измененных страниц.
- Запись отображенных страниц (`MiMappedPageWriter`, приоритет 18). Записывает «грязные» страницы из списка модификаций на диск или в удаленное хранилище. Поток активизируется, когда требуется сократить размер списка измененных страниц или страницы отображенных файлов находились в списке модификаций более 5 минут. Второй поток записи измененных страниц необходим из-за того, что он может генерировать ошибки страниц, приводящие к запросам свободных страниц. Если бы существовал только один поток записи измененных страниц, то при отсутствии свободных страниц в системе возникла бы взаимная блокировка с ожиданием свободных страниц.
- Поток разыменования сегментов (`MiDereferenceSegmentThread`, приоритет 19). Отвечает за сокращение кэша, а также за рост и сокращение страничного файла. Например, если не остается виртуального адресного пространства для роста выгружаемого пула, этот поток сокращает кэш страниц, чтобы выгружаемый пул мог быть освобожден для повторного использования.
- Поток обнуления страниц (`MiZeroPageThread`, приоритет 0). Заполняет нулями страницы из свободного списка, чтобы для удовлетворения будущих запросов обнуления был доступен кэш нулевых страниц. В некоторых случаях обнуление памяти осуществляется более быстрой функцией `MiZeroInParallel`. За дополнительной информацией обращайтесь к разделу «Динамика списков страниц» далее в этой главе.

Каждый из этих компонентов более подробно рассматривается позднее в этой главе. Исключение составляет поток разыменования сегментов — он рассматривается в главе 14 «Диспетчер кэша» части 2.

Большие и малые страницы

Управление памятью осуществляется на уровне блоков, называемых *страницами* (pages.) Это объясняется тем, что подсистема аппаратного управления памятью преобразует виртуальные адреса в физические на уровне страниц. Таким образом, страница является минимальной единицей защиты на аппаратном уровне. (Различные варианты защиты страниц описаны в разделе «Защита памяти» далее в этой главе.) Процессоры, на которых работает Windows, поддерживают два раз-

мера страниц: малые и большие. Фактические размеры зависят от архитектуры процессора; они перечислены в табл. 5.1.

Таблица 5.1. Размеры страниц

Архитектура	Размер малой страницы	Размер большой страницы	Количество малых страниц в большой странице
x86 (PAE)	4 Кбайт	2 Мбайт	512
x64	4 Кбайт	2 Мбайт	512
ARM	4 Кбайт	4 Мбайт	1024

ПРИМЕЧАНИЕ Некоторые процессоры поддерживают возможность настройки размеров страниц, но в Windows эта возможность не используется.

Главным преимуществом больших страниц является скорость преобразования адресов при обращениях к данным в больших страницах. Это преимущество возникает из-за того, что первое обращение к любому байту в большой странице приводит к тому, что в кэш буфера TLB (Translation Look-aside Buffer) (см. раздел «Преобразование адресов» далее в этой главе) заносится информация, необходимая для преобразования ссылок на любой другой байт в большой странице. Если используются малые страницы, для одного диапазона виртуальных адресов потребуется больше записей TLB; это потребует замены записей по мере появления новых виртуальных адресов, требующих преобразования.

В свою очередь, это означает возврат к структурам таблиц страниц при обращении к виртуальным адресам за пределами малой страницы, для которой было кэшировано преобразование. Кэш TLB очень мал; следовательно, большие страницы обеспечивают более эффективное использование этого ограниченного ресурса.

Чтобы использовать большие страницы в системе, оснащенной более чем 2 Гбайт оперативной памяти, Windows отображает на большие страницы основные образы операционной системы (Ntoskrnl.exe и Hal.dll) и основные данные операционной системы (например, начальную часть невыгружаемого пула и структуры данных, описывающие состояние каждой страницы физической памяти). Windows также автоматически отображает запросы пространства ввода/вывода (вызовы драйверов устройств к `MmMapIoSpace`) на большие страницы, если запрос имеет достаточно большую длину страницы и выравнивание. Кроме того, Windows позволяет приложениям отображать свои образы, закрытую память и секции на базе страничных файлов с большими страницами (см. флаг `MEM_LARGE_PAGES` функций `VirtualAlloc`, `VirtualAllocEx` и `VirtualAllocExNuma`). Вы также можете указать другие драйверы устройств, отображаемые на большие страницы, — добавьте многострочный параметр реестра `LargePageDrivers` в раздел HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management и укажите имена драйверов в виде строк, разделенных нуль-символами.

Попытки выделения больших страниц могут закончиться неудачей, если операционная система отработала в течение продолжительного времени, потому что

физическая память каждой большой страницы должна занимать значительное количество (см. табл. 5.1) физически смежных малых страниц. Следовательно, этот блок физических страниц должен начинаться на границе больших страниц. Например, физические страницы 0–511 могут использоваться как большая страница в системе x64, как и физические страницы 512–1023, но со страницами 10–521 это невозможно. Свободная физическая память фрагментируется в процессе работы системы. Это не создает проблем для выделения памяти малыми страницами, но попытка выделения большой страницы может завершиться неудачей.

Память также всегда невыгружаема, потому что система страничных файлов не поддерживает большие файлы. Так как память невыгружаема, вызывающая сторона должна обладать привилегией `SeLockMemoryPrivilege` для выделения памяти с использованием больших страниц. Кроме того, выделенная память не считается частью рабочего набора процесса (см. раздел «Рабочие наборы» далее в этой главе); не все выделения больших страниц подвержены ограничениям уровня задания на использование виртуальной памяти.

В Windows 10 версии 1607 x64 и Server 2016 большие страницы также могут отображаться с использованием огромных (huge) страниц, размер которых составляет 1 Гбайт. Это происходит автоматически, если запрошенный размер выделяемого блока превышает 1 Гбайт, но он не обязан быть кратным 1 Гбайт. Например, выделение блока размером 1040 Мбайт приведет к использованию одной огромной страницы (1024 Мбайт) и 8 «нормальных» больших страниц (16 Мбайт делятся на 2 Мбайт).

У больших страниц есть один неприятный побочный эффект. Каждая страница (огромная, большая или малая) должна отображаться с режимом защиты, применяемым ко всей странице. Это происходит из-за того, что аппаратная защита памяти работает на уровне страниц. Если большая страница содержит, например, код только для чтения и данные для чтения/записи, эта страница должна быть помечена как доступная для чтения/записи; это означает, что код будет доступен для записи. В результате драйверы устройств и другой код режима ядра могут (злонамеренно или из-за ошибки) модифицировать код, который должен быть доступным только для чтения кодом операционной системы или драйвера, без нарушения доступа к памяти.

Если использовать малые страницы для отображения кода режима ядра операционной системы, части `Ntoskrnl.exe` и `Hal.dll`, доступные только для чтения, могут отображаться как доступные только для чтения. Использование малых страниц сокращает эффективность преобразования адресов, но если другой драйвер (или другой код режима ядра) пытается изменить часть операционной системы, доступную только для чтения, в системе немедленно произойдет фатальный сбой; при этом в информации исключения содержится указатель на инструкцию-нарушителя в драйвере. Если бы запись была возможной, аварийный сбой в системе, скорее всего, произошел бы позднее (и диагностировать эту ошибку будет намного сложнее), когда другой компонент попытается использовать поврежденные данные.

Если вы подозреваете, что столкнулись с повреждением кода ядра, включите функцию Driver Verifier (см. главу 6 «Подсистема ввода/вывода»), которая запрещает использование больших страниц.

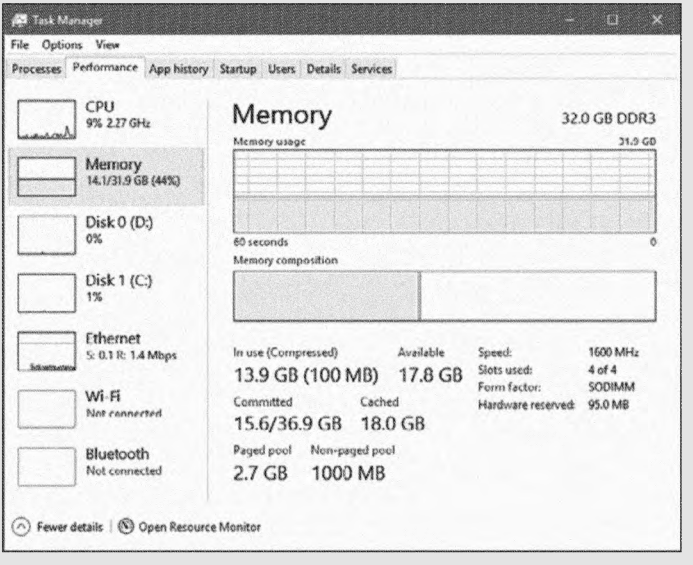
ПРИМЕЧАНИЕ Термин «страница», используемый в этой и следующих главах, обозначает малую страницу, если только обратное не указано явно или не следует из контекста.

Анализ использования памяти

Категории счетчиков производительности Память (Memory) и Процесс (Process) предоставляют доступ к основной информации об использовании памяти системой и процессами. В этой главе мы приведем ссылки на конкретные счетчики производительности с информацией об описываемых компонентах. Также здесь приводятся примеры и описания экспериментов. Одно предупреждение: при выводе информации о памяти в разных программах используются разные, порой непоследовательные или невразумительные имена. Следующий эксперимент демонстрирует этот факт. (Используемые термины будут поясняться в следующих разделах.)

ЭКСПЕРИМЕНТ: ПРОСМОТР ИНФОРМАЦИИ О СИСТЕМНОЙ ПАМЯТИ

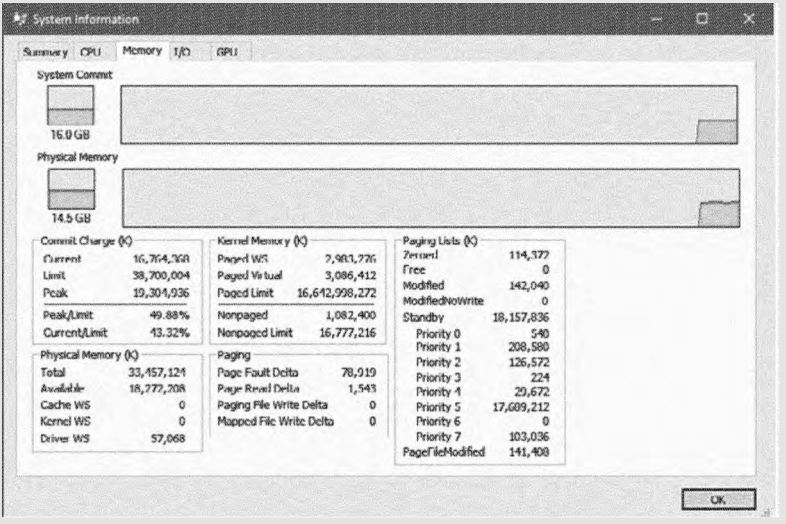
На вкладке Быстродействие (Performance) диспетчера задач Windows, изображенной на следующем снимке экрана из Windows 10 версии 1607 (щелкните на вкладке Память (Memory) слева от вкладки Быстродействие (Performance)), выводится основная информация о системной памяти. Она составляет подмножество подробной информации о памяти, которая может быть получена через счетчики производительности. Она включает данные об использовании как физической, так и виртуальной памяти. В следующей таблице приведены значения всех величин, связанных с памятью.



Значение в диспетчере задач	Определение
Использование памяти (Memory Usage)	Высота линии на графике представляет объем физической памяти, используемой Windows (не может быть получена через счетчик производительности). Область над линией соответствует значению Доступно (Available) на нижней панели. Общая высота графика равна значению, выведенному в правой верхней части графика (31,9 Гбайт в данном примере). Она представляет общий объем оперативной памяти, который может использоваться операционной системой (без учета теневых страниц BIOS, памяти устройств и т. д.)
Структура памяти (Memory Composition)	Соотношение активно используемой, находящейся в ожидании, измененной и свободной памяти с обнуленными страницами (см. далее в этой главе)
Общая физическая память (правый верхний угол графика)	Физическая память, которая может использоваться Windows
Используется (со сжатием) (In Use (Compressed))	Физическая память, используемая в настоящее время. Объем сжатой физической памяти указан в скобках. Если навести указатель мыши на это значение, вы увидите объем памяти, сэкономленной за счет сжатия. (См. раздел «Сжатие памяти» этой главы.)
Кэшировано (Cached)	Сумма следующих счетчиков производительности из категории Память: Cache Bytes, Modified Page List Bytes, Standby Cache Core Bytes, Standby Cache Normal Priority Bytes и Standby Cache Reserve Bytes
Доступно (Available)	Объем памяти, доступной для непосредственного использования операционной системой, процессами и драйверами. Равен суммарному размеру ожидающей памяти, свободной памяти и списков обнуления страниц
Свободно (Free)	Объем свободной памяти и списков обнуления страниц. Чтобы просмотреть информацию, наведите указатель мыши на правый край диаграммы Структура памяти (Memory Composition)
Выделено (Committed)	Эти два числа равны значениям счетчиков производительности Committed Bytes и Commit Limit соответственно
Выгружаемый пул (Paged pool)	Общий размер выгружаемого пула, включая свободные и распределенные области
Невыгружаемый пул (Non-paged pool)	Общий размер невыгружаемого пула, включая свободные и распределенные области

Для просмотра фактического использования выгружаемого и невыгружаемого пула используйте программу Poolmon, описанную в разделе «Контроль за использованием пулов» этой главы.

Программа Process Explorer из пакета Sysinternals выводит намного больше информации о физической и виртуальной памяти. На главном экране откройте меню View, выберите команду System Information и перейдите на вкладку Memory. Пример вывода в 64-разрядной системе Windows 10 (большинство счетчиков описано в соответствующих разделах этой главы):



Для вывода расширенной информации об использовании памяти используются две другие программы из пакета Sysinternals:

- **VMMap** — выводит информацию об использовании виртуальной памяти процессом на более высоком уровне детализации.
- **RAMMap** — подробная информация об использовании физической памяти.

Эти программы будут представлены в экспериментах, описанных далее в этой главе.

Наконец, команда `!vm` отладчика ядра выводит основную информацию управления памятью, доступную через соответствующие счетчики производительности. Команда может пригодиться при просмотре аварийного дампа или содержимого памяти зависшей системы. Пример вывода для 64-разрядной системы Windows 10 с 32 Гбайт памяти:

```
lkd> !vm
Page File: \??\C:\pagefile.sys
  Current:   1048576 Kb  Free Space:   1034696 Kb
  Minimum:   1048576 Kb  Maximum:      4194304 Kb
Page File: \??\C:\swapfile.sys
  Current:     16384 Kb  Free Space:     16376 Kb
  Minimum:     16384 Kb  Maximum:     24908388 Kb
No Name for Paging File
  Current:  58622948 Kb  Free Space:  57828340 Kb
  Minimum:  58622948 Kb  Maximum:     58622948 Kb
```

```
Physical Memory:          8364281 (   33457124 Kb)
Available Pages:          4627325 (   18509300 Kb)
ResAvail Pages:           7215930 (   28863720 Kb)
Locked IO Pages:                0 (          0 Kb)
Free System PTEs:      4295013448 (17180053792 Kb)
Modified Pages:             68167 (     272668 Kb)
Modified PF Pages:          68158 (     272632 Kb)
Modified No Write Pages:        0 (          0 Kb)
NonPagedPool Usage:           495 (       1980 Kb)
NonPagedPoolNx Usage:      269858 (    1079432 Kb)
NonPagedPool Max:      4294967296 (17179869184 Kb)
PagedPool 0 Usage:         371703 (    1486812 Kb)
PagedPool 1 Usage:          99970 (     399880 Kb)
PagedPool 2 Usage:         100021 (     400084 Kb)
PagedPool 3 Usage:          99916 (     399664 Kb)
PagedPool 4 Usage:          99983 (     399932 Kb)
PagedPool Usage:           771593 (    3086372 Kb)
PagedPool Maximum:     4160749568 (16642998272 Kb)
Session Commit:             12210 (      48840 Kb)
Shared Commit:             344197 (    1376788 Kb)
Special Pool:                   0 (          0 Kb)
Shared Process:             19244 (      76976 Kb)
Pages For MDLs:            419675 (    1678700 Kb)
Pages For AWE:                  0 (          0 Kb)
NonPagedPool Commit:       270387 (    1081548 Kb)
PagedPool Commit:          771593 (    3086372 Kb)
Driver Commit:              24984 (      99936 Kb)
Boot Commit:               100044 (     400176 Kb)
System PageTables:           5948 (      23792 Kb)
VAD/PageTable Bitmaps:      18202 (      72808 Kb)
ProcessLockedFilePages:       299 (       1196 Kb)
Pagefile Hash Pages:           33 (        132 Kb)
Sum System Commit:         1986816 (    7947264 Kb)
Total Private:             2126069 (    8504276 Kb)
Misc/Transient Commit:       18422 (      73688 Kb)
Committed pages:           4131307 (   16525228 Kb)
Commit limit:              9675001 (   38700004 Kb)
...
```

Значения перед скобками измеряются в малых страницах (4 Кбайт.) Многие подробности вывода этой команды описаны далее в этой главе.

Внутренняя синхронизация

Диспетчер памяти, как и все остальные компоненты исполняющей среды Windows, обладает полной реентерабельностью и поддерживает одновременное исполнение в многопроцессорных системах. Иначе говоря, он позволяет двум потокам захватывать ресурсы таким образом, чтобы они не повредили данные друг друга. Для обеспечения полной реентерабельности диспетчер памяти использует различные внутренние механизмы синхронизации (такие, как спин-блокировки и взаимосвязанные инструкции) для управления доступом к своим внутренним структурам данных. (Объекты синхронизации рассматриваются в главе 8 части 2.)

Некоторые общесистемные ресурсы, доступ к которым должен синхронизироваться диспетчером памяти:

- Динамически выделенные части системного виртуального адресного пространства.
- Системные рабочие наборы.
- Пулы памяти ядра.
- Список загруженных драйверов.
- Список выгружаемых файлов.
- Списки физической памяти.
- Структуры ASLR (Address Space Layout Randomization).
- Каждая отдельная запись в базе данных PFN (Page Frame Number).

К числу структур данных управления памятью уровня процесса, требующих синхронизации, относятся:

- **Блокировка рабочего набора.** Удерживается во время внесения изменений в список рабочих наборов.
- **Блокировка адресного пространства.** Удерживается во время изменения адресного пространства.

Обе блокировки реализуются с использованием push-блокировок (см. главу 8 части 2).

Сервисные функции, предоставляемые диспетчером памяти

Диспетчер памяти предоставляет набор сервисных функций для выделения и освобождения виртуальной памяти, совместного использования памяти между процессами, отображения файлов в память, записи виртуальных страниц на диск, получения информации о диапазоне виртуальных страниц, изменения защиты виртуальных страниц и блокировки виртуальных страниц в памяти.

Сервисные функции управления памятью, как и другие сервисы исполняющей среды Windows, позволяют вызывающей стороне передать дескриптор процесса для обозначения конкретного процесса, с виртуальной памятью которого вы собираетесь работать. Вызывающая сторона может работать либо со своей памятью, либо (при наличии необходимых разрешений) с памятью другого процесса. Например, если процесс создает дочерний процесс, по умолчанию ему предоставляется право работать с виртуальной памятью дочернего процесса. Это означает, что родительский процесс может выделять, освобождать, выполнять операции чтения и записи в памяти дочернего процесса, вызывая сервисные функции виртуальной памяти

и передавая дескриптор дочернего процесса в аргументе. Эта возможность используется подсистемами для управления памятью своих клиентских процессов. Также она играет важную роль в реализации отладчиков, поскольку отладчики должны иметь возможность читать и записывать данные в памяти отлаживаемого процесса.

Многие сервисные функции предоставляются через Windows API. Как видно из рис. 5.1, Windows API содержат четыре группы функций для управления памятью в приложениях.

- **API Virtual.** API нижнего уровня для общих операций выделения и освобождения памяти. Он всегда работает на уровне страниц. Этот API также является самым мощным: он поддерживает полную функциональность диспетчера памяти. К этой группе относятся функции `VirtualAlloc`, `VirtualFree`, `VirtualProtect`, `VirtualLock` и др.
- **API Heap.** Функции для выделения малых блоков памяти (обычно менее страницы). Во внутренней реализации используется API Virtual, но поверх него добавляются средства управления. К этой группе относятся функции `HeapAlloc`, `HeapFree`, `HeapCreate`, `HeapReAlloc` и др. Диспетчер кучи рассматривается в разделе «Диспетчер кучи» этой главы.
- **Локальные/глобальные API.** Пережитки эпохи 16-разрядной версии Windows; сейчас реализуются с использованием API Heap.
- **Файлы, отображенные в память.** Эти функции позволяют отображать файлы в память и/или организовывать совместное использование памяти между взаимодействующими процессами. К этой группе относятся функции `CreateFileMapping`, `OpenFileMapping`, `MapViewOfFile` и др.

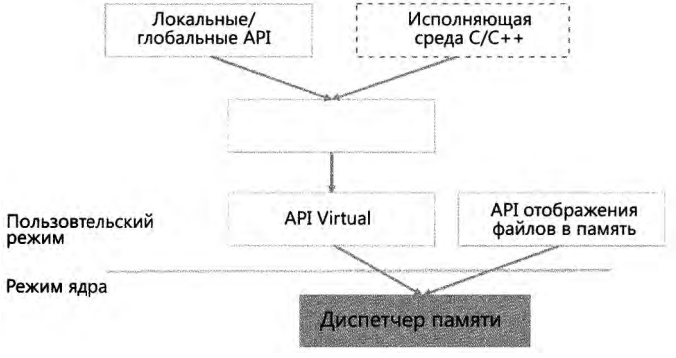


Рис. 5.1. Группы API-функций для работы с памятью в пользовательском режиме

В пунктирном прямоугольнике содержится типичная реализация управления памятью в исполняющей среде C/C++ (такие функции, как `malloc`, `free`, `realloc`, оператор C++ `new` и `delete`) с использованием API Heap. Прямоугольник заключен в пунктирный контур, потому что эта реализация зависит от компилятора и, безус-

ловно, не является строго обязательной (хотя и довольно распространенной). Аналоги исполняющей среды C реализованы в Ntdll.dll с использованием API Heap.

Диспетчер памяти также предоставляет ряд сервисных функций другим компонентам режима ядра исполнительной среды и драйверам устройств. К их числу принадлежит выделение и освобождение физической памяти и блокировка страниц в физической памяти для передачи данных через DMA (Direct Memory Access). Имена таких функций начинаются с префикса `xMm`. Кроме того, некоторые функции исполняющей среды с именами, начинающимися с `Ex` (хотя формально они не являются частью диспетчера памяти), используются для выделения и освобождения памяти из системных куч (выгружаемого и невыгружаемого пула), а также для операций с резервными списками. Все эти темы более подробно рассматриваются в разделе «Кучи режима ядра (пулы системной памяти)» этой главы.

Состояние страниц и выделение памяти

Страницы в виртуальном адресном пространстве процесса могут быть свободными, зарезервированными, подтвержденными или разделяемыми. *Подтвержденные* (committed) и *разделяемые* (shareable) страницы при обращении в конечном итоге преобразуются в действительные страницы в физической памяти. Подтвержденные страницы также называются *закрытыми*; это связано с тем, что другие процессы не могут получить к ним доступ, тогда как с разделяемыми страницами это возможно (хотя использоваться они могут только одним процессом).

Для выделения подтвержденных страниц используются Windows-функции `VirtualAlloc`, `VirtualAllocEx` и `VirtualAllocExNuma`, которые в конечном счете передают управление исполняющей среде в функции `NtAllocateVirtualMemory` внутри диспетчера памяти. Эти функции могут как выделять, так и резервировать память. При резервировании памяти система фактически «откладывает в сторону» диапазон смежных виртуальных адресов для будущего использования (например, в массиве) с незначительными затратами системных ресурсов и последующим подтверждением частей зарезервированного пространства по мере надобности во время работы приложения. Или, если требования к размеру выделяемой памяти известны заранее, процесс может зарезервировать и подтвердить память в одном вызове функции. В любом случае полученные подтвержденные страницы доступны для любого потока в процессе. Попытка обратиться к свободной или зарезервированной памяти приводит к исключению по нарушению прав доступа, потому что страница не связана ни с какой областью памяти, которая могла бы использоваться для разрешения ссылки.

Если к подтвержденным (закрытым) страницам еще не было ни одного обращения, они создаются при первом обращении как *обнуленные* страницы. Позднее операционная система может автоматически записать закрытые подтвержденные страницы в страничный файл, если это будет необходимо из-за потребности в физической памяти. Термин «закрытость» в данном случае указывает на тот факт, что эти страницы обычно недоступны для других процессов.

ПРИМЕЧАНИЕ Некоторые функции (такие, как `ReadProcessMemory` и `WriteProcessMemory`) на первый взгляд предоставляют межпроцессный доступ к памяти, но они реализуются выполнением кода режима ядра в контексте целевого процесса (это называется *присоединением к процессу*). Они также требуют, чтобы дескриптор безопасности целевого процесса предоставлял обращающейся стороне право `PROCESS_VM_READ` или `PROCESS_VM_WRITE` соответственно или чтобы обращающаяся сторона обладала привилегией `SeDebugPrivilege`, которая по умолчанию предоставляется только участникам административной группы.

Подтвержденные страницы обычно отображаются на представление секции. В свою очередь, секция представляет часть файла (или полный файл), но вместо этого может представлять часть пространства страничного файла. Все разделяемые страницы могут использоваться совместно с другими процессами. Секции в Windows API представлены объектами отображенных файлов.

Когда к разделяемой странице впервые обращается любой процесс, она читается из связанного с ней отображенного файла — если только секция не связана со страничным файлом; в этом случае она создается в виде страницы, инициализированной нулями. Позднее, если страница все еще находится в физической памяти, второй обращающийся к ней процесс (как и все последующие) может просто использовать содержимое страницы, уже находящееся в памяти. Система может применить к разделяемым страницам предварительную выборку.

В двух последующих разделах этой главы, «Общая память и отображенные файлы» и «Объекты секций», разделяемые страницы рассматриваются намного более подробно. Для записи страниц на диск используется механизм *записи измененных страниц*. Это происходит при перемещении страниц из рабочего набора процесса в общесистемный список, называемый списком измененных страниц. После этого они записываются на диск или в удаленное хранилище. (Рабочие наборы и списки измененных страниц рассматриваются далее в этой главе.) Страницы отображенных файлов также могут быть записаны обратно в свои исходные файлы на диске явным вызовом `FlushViewOfFile` или механизмом записи отображенных страниц в соответствии с потребностями в памяти.

Функция `VirtualFree` или `VirtualFreeEx` позволяет отменить подтверждение закрытых страниц и/или освободить адресное пространство. Различия между отменой подтверждения и освобождением аналогичны различиям между резервированием и подтверждением. Память с отмененным подтверждением остается зарезервированной, а освобожденная память просто освобождается, не являясь ни подтвержденной, ни зарезервированной.

Двухшаговый процесс резервирования с последующим подтверждением виртуальной памяти откладывает момент подтверждения страниц, но при этом сохраняет удобство целостности виртуальной памяти. Резервирование памяти — относительно малозатратная операция, так как она расходует очень мало фактической памяти. Для ее выполнения достаточно обновить или построить относительно небольшие внутренние структуры данных, представляющие состояние адресного пространства процесса. Эти структуры данных, называемые *таблицами страниц* и дескриптора-

ми виртуальных адресов, или *VAD* (Virtual Address Descriptors), рассматриваются позднее в этой главе.

Чрезвычайно распространенный пример с резервированием больших пространств и подтверждением их частей по мере надобности встречается при работе со стеком пользовательского режима каждого потока. При создании потока стек создается резервированием смежной части адресного пространства процесса (размер по умолчанию составляет 1 Мбайт, но это значение можно переопределить вызовами функций `CreateThread` и `CreateRemoteThread(Ex)` или же изменить его на уровне исполняемого образа при помощи флага компоновщика `/STACK`). По умолчанию исходная страница в стеке подтверждается, а следующая страница помечается как *сторожевая страница* (не подтвержденная); она перехватывает обращения, выходящие за пределы подтвержденной части стека и расширяет его.

ЭКСПЕРИМЕНТ: ЗАРЕЗЕРВИРОВАННЫЕ И ПОДТВЕРЖДЕННЫЕ СТРАНИЦЫ

Для выделения больших объемов зарезервированной или закрытой подтвержденной виртуальной памяти можно воспользоваться программой TestLimit из пакета Sysinternals. Далее программа Process Explorer поможет понаблюдать за различиями между этими видами памяти. Выполните следующие действия:

1. Откройте два окна командной строки.
2. Запустите в одном из окон командной строки TestLimit и создайте большой объем зарезервированной памяти:

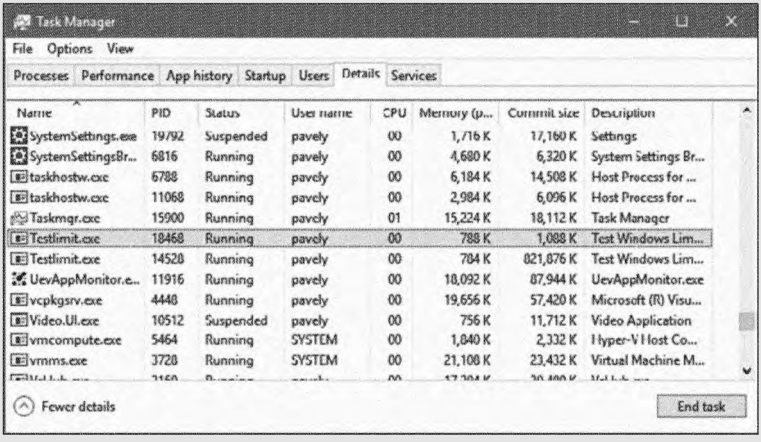
```
C:\temp>testlimit -r 1 -c 800

Testlimit v5.24 - test Windows limits
Copyright (C) 2012-2015 Mark Russinovich
Sysinternals - wwww.sysinternals.com

Process ID: 18468

Reserving private bytes 1 MB at a time ...
Leaked 800 MB of reserved memory (800 MB total leaked). Lasterror: 0
The operation completed successfully.
```

3. В другом окне командной строки создайте аналогичный объем подтвержденной памяти:

```
C:\temp>testlimit -m 1 -c 800

Testlimit v5.24 - test Windows limits
Copyright (C) 2012-2015 Mark Russinovich
Sysinternals - wwww.sysinternals.com

Process ID: 14528

Leaking private bytes 1 KB at a time ...
Leaked 800 MB of private memory (800 MB total leaked). Lasterror: 0
The operation completed successfully.
```

4. Откройте диспетчер задач, перейдите на вкладку Подробности (Details) и добавьте столбец Commit Size.

5. Найдите в списке два экземпляра TestLimit.exe. Они должны выглядеть примерно так:

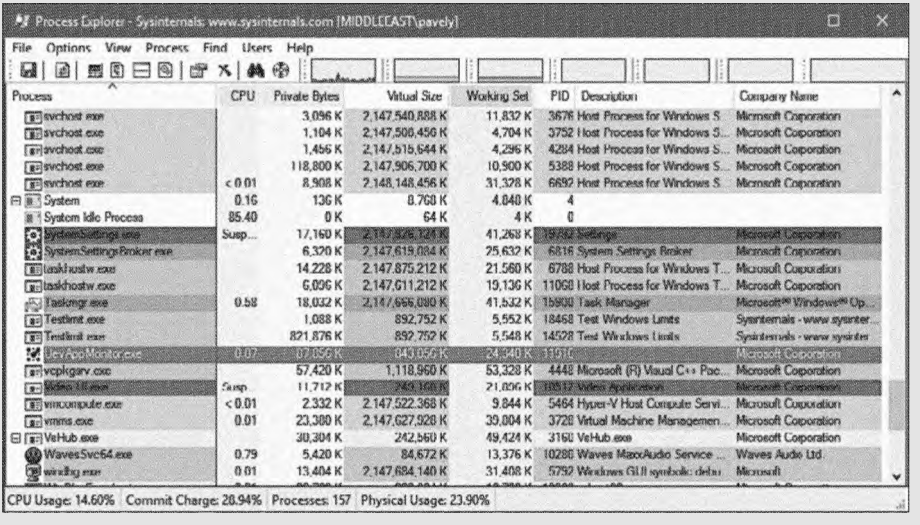


6. Обратите внимание: в диспетчере задач выводится подтвержденный размер, но в нем нет счетчиков для вывода зарезервированной памяти в другом процессе TestLimit.

7. Откройте Process Explorer.

8. Перейдите на вкладку Process Memory и выберите столбцы Private Bytes и Virtual Size.

9. Найдите два процесса TestLimit.exe в главном окне:



10. Обратите внимание: виртуальные размеры двух процессов совпадают, но только для одного выводится значение Private Bytes, сравнимое со значением Virtual Size. Сильно различающиеся значения в другом процессе, TestLimit (идентификатор 18468) обусловлены зарезервированной памятью. Вы можете провести то же сравнение в Системном мониторе, обратившись к счетчикам Virtual Bytes и Private Bytes из категории Процесс (Process).

Нагрузка подтверждения памяти и предел подтверждения

На вкладке Быстродействие (Performance) в диспетчере задач, на странице Память (Memory) имеется надпись Выделено (Committed), под которой расположены два числа. Диспетчер памяти отслеживает использование закрытой подтвержденной памяти на глобальной основе; этот показатель называется *нагрузкой подтверждения*. Это первое из двух чисел; оно представляет суммарный объем всей закрепленной виртуальной памяти в системе.

На уровне системы установлен предел, называемый *системным пределом подтверждения памяти* (или просто пределом подтверждения), для объема подтвержденной виртуальной памяти, существующей в любой момент времени. Этот предел соответствует суммарному размеру всех страничных файлов и объема ОЗУ, используемого операционной системой. Он представлен вторым из двух чисел под надписью Выделено (Committed). Диспетчер памяти может автоматически увеличить предел подтверждения, расширив один или несколько страничных файлов, словно они еще не достигли настроенного максимального размера.

Нагрузка подтверждения и системный лимит подтверждения более подробно рассмотрены далее в этой главе.

Блокировка памяти

В общем случае лучше доверить диспетчеру памяти принятие решений о том, какие страницы должны находиться в физической памяти. Однако в некоторых особых обстоятельствах приложение или драйвер устройства должны заблокировать страницы в физической памяти. Блокировка страниц в памяти может осуществляться двумя способами:

- Приложения Windows могут вызвать функцию `VirtualLock` для блокировки страниц в рабочем наборе процесса. Страницы, заблокированные этим механизмом, остаются в памяти до тех пор, пока не будут явно разблокированы или не завершится заблокировавший их процесс. Количество страниц, которые могут быть заблокированы процессом, не может превысить минимальный размер рабочего набора без восьми страниц. Если процессу понадобится заблокировать большее количество страниц, он может увеличить минимум своего рабочего

набора функцией `SetProcessWorkingSetSizeEx` (эта функция рассматривается далее в разделе «Управление рабочим набором»).

- Драйверы устройств могут вызывать функции режима ядра `MmProbeAndLockPages`, `MmLockPagableCodeSection`, `MmLockPagableDataSection` и `MmLockPagableSectionByHandle`. Страницы, заблокированные этим механизмом, остаются в памяти вплоть до явной разблокировки. Последние три из этих API-функций не соблюдают квоту на количество страниц, которые могут быть заблокированы в памяти, потому что значение нагрузки резидентных доступных страниц определяется при первой загрузке драйвера. Тем самым предотвращаются системные сбои из-за избыточной блокировки. Для первой API-функции необходимо получение расхода квоты, в противном случае функция вернет статус ошибки.

Гранулярность выделения памяти

Windows выравнивает каждую область зарезервированного адресного пространства процесса так, чтобы она начиналась с целочисленной границы, определяемой гранулярностью выделения памяти; для получения этой величины можно воспользоваться Windows-функцией `GetSystemInfo` или `GetNativeSystemInfo`. Эта величина составляет 64 Кбайт; эта гранулярность используется диспетчером памяти для эффективного выделения метаданных (например, VAD, битовых карт и т. д.) для поддержки различных операций процессов. Кроме того, если в системе появится поддержка будущих процессоров с увеличенным размером страницы (например, до 64 Кбайт) или кэшей с виртуальным индексированием, требующих выравнивания между физическими и виртуальными страницами на уровне системы, это снизит риск необходимости изменений в приложениях, предполагающих определенный способ выравнивания при выделении памяти.

ПРИМЕЧАНИЕ На код режима ядра Windows эти ограничения не распространяются. Он может резервировать память с одностраничной гранулярностью (хотя драйверам устройств такая возможность не предоставляется по причинам, описанным выше). Такой уровень гранулярности используется прежде всего для более плотной упаковки выделения памяти TEB. Поскольку этот механизм используется исключительно во внутренней реализации, код может быть легко изменен, если для будущей платформы потребуются другие значения. Кроме того, для поддержки 16-разрядных приложений и приложений MS-DOS только для систем x86 диспетчер памяти предоставляет флаг `MEM_DOS_LIM` функции `MapViewOfFileEx`, который принудительно включает одностраничную гранулярность.

Наконец, при резервировании области адресного пространства Windows следит за тем, чтобы размер и база области были кратны размеру страницы в системе, каким бы он ни был. Например, так как в системах x86 используются страницы с размером 4 Кбайт, при попытке зарезервировать область памяти с размером 18 Кбайт фактический зарезервированный объем в системе x86 составит 20 Кбайт. Если же указать для этой области базовый адрес 3 Кбайт, то фактически зарезервированный объем

составит 24 Кбайт. Обратите внимание: VAD для этого выделения памяти также округляется для границы/длины 64 Кбайт, вследствие чего остаток будет недоступен.

Общая память и отображенные файлы

Как и в большинстве современных операционных систем, в Windows существует механизм совместного использования памяти между процессами и операционной системой. *Общую память* можно определить как память, которая видна более чем одному процессу или которая присутствует в виртуальном адресном пространстве более чем одного процесса. Например, если два процесса используют одну DLL-библиотеку, будет разумно загрузить используемые кодовые страницы DLL в физическую память только один раз, а потом использовать эти страницы во всех процессах, отображающих эту библиотеку (рис. 5.2).

Рис. 5.2. Совместное использование памяти между процессами

Каждый процесс поддерживает собственные области памяти для хранения закрытых данных, но код DLL и неизменяемые страницы данных могут находиться в общем доступе без какого-либо вреда. Как будет показано позднее, такой способ совместного использования реализуется автоматически, потому что кодовые страницы исполняемых образов (файлов EXE и DLL, некоторых других разновидностей — например, экранных заставок (SCR), которые фактически представляют собой DLL с другими именами) отображаются как доступные только для исполнения, а страницы с возможностью записи отображаются в режиме копирования при записи. (За дополнительной информацией обращайтесь к разделу «Копирование при записи» этой главы.)

На рис. 5.2 изображены два процесса с разными образами, которые совместно используют DLL-библиотеку, однократно отображенную в физическую память. Код самих образов (EXE) в данном случае не используется совместно, потому что в двух процессах выполняются разные образы. Код EXE может совместно использоваться между процессами, в которых выполняется один образ (например, двумя и более процессами, в которых выполняется Notepad.exe).

Примитивы, на основе которых диспетчер памяти реализует общую память, называются *объектами секций*; в Windows API они представлены объектами отображенных файлов. Внутренняя структура и реализация объектов секций описана позднее в этой главе, в разделе «Объекты секций». Этот важнейший примитив диспетчера памяти используется для отображения виртуальных адресов в основную память, в страничный файл или в любой другой файл, к содержимому которого приложение захочет обращаться так, как если бы оно находилось в памяти. Секция может быть открыта как одним, так и многими процессами. Иначе говоря, объекты секций не обязательно соответствуют общей памяти.

Объект секции может быть связан с открытым файлом на диске (этот файл называется *отображенным*) или с подтвержденной памятью (для реализации общей памяти). Секции, отображаемые на подтвержденную память, называются *секциями на базе страничных файлов*, потому что страницы записываются в страничный файл (вместо отображенного файла), если возникнет необходимость в физической памяти. (Так как Windows может работать вообще без страничного файла, секции на базе страничных файлов могут работать «на базе» только физической памяти.) Как и любая другая пустая страница, видимая в пользовательском режиме (как закрытые подтвержденные страницы), общие подтвержденные страницы всегда заполняются нулями при первом обращении; таким образом предотвращается возможная утечка конфиденциальных данных.

Чтобы создать объект секции, вызовите функцию Windows `CreateFileMapping`, `CreateFileMappingFromApp` или `CreateFileMappingNuma(Ex)` и передайте ей ранее открытый дескриптор файла для отображения (или `INVALID_HANDLE_VALUE` для секции на базе страничного файла); также можно указать имя и дескриптор безопасности. Если у секции есть имя, другие процессы смогут открыть ее при помощи функции `OpenFileMapping` или `CreateFileMapping*`. Также доступ к объектам секций может быть предоставлен через наследование дескрипторов (для этого следует указать, что дескриптор должен наследоваться при его открытии или создании) или дублирование дескрипторов (с использованием `DuplicateHandle`). Драйверы устройств также могут манипулировать объектами секций при помощи функций `ZwOpenSection`, `ZwMapViewOfSection` и `ZwUnmapViewOfSection`.

Объект секции может ссылаться на файлы гораздо большего размера, чем может поместиться в адресном пространстве процесса. (Если объект секции существует на базе страничного файла, то в страничном файле и/или ОЗУ должно существовать достаточно пространства для его создания.) Чтобы обратиться к очень большому объекту секции, процесс может отобразить только нужную часть объекта секции (называемую *представлением секции*) вызовом функции `MapViewOfFile(Ex)`, `MapViewOfFileFromApp` или `MapViewOfFileExNuma` и указанием отображенного диапазона. Отображение представлений позволяет процессам экономить адресное пространство, потому что в память отображаются только представления объекта секции, необходимого в данный момент.

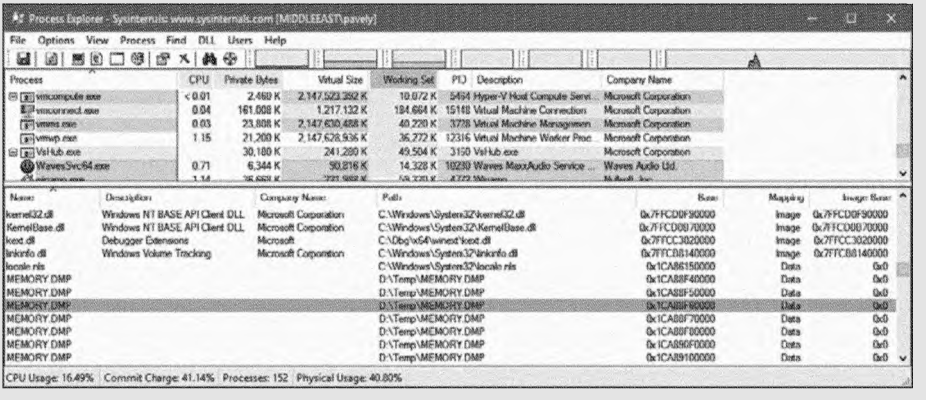
Приложения Windows могут использовать отображенные файлы для удобного выполнения операций ввода/вывода с файлами; для этого достаточно просто

разместить данные в памяти в границах их адресного пространства. Пользовательские приложения — не единственные потребители объектов секций; загрузчик образов использует объекты секций для отображения исполняемых образов, DLL-библиотек и драйверов устройств в память, а диспетчер кэша использует их для обращения к данным в кэшированных файлах. (За информацией о том, как диспетчер кэша интегрируется с диспетчером памяти, обращайтесь к главе 14 части 2.) Реализация общих секций в памяти — в контексте преобразования как адресов, так и внутренних структур данных — объясняется в разделе «Объекты секций» этой главы.

ЭКСПЕРИМЕНТ: ПРОСМОТР ФАЙЛОВ, ОТОБРАЖЕННЫХ В ПАМЯТЬ

Для просмотра списка файлов процесса, отображенных в память, можно воспользоваться программой Process Explorer. Настройте нижнюю панель для вывода представления DLL (откройте меню View, выберите команду Lower Panel View и пункт DLLs). Обратите внимание: перед вами не просто список DLL-библиотек — в нем представлены все файлы, отображенные в память, в адресном пространстве процесса. Часть из них — DLL-библиотеки, один элемент — выполняемый файл образа (EXE), а дополнительные элементы могут представлять файлы данных, отображенные в память.

На следующем снимке экрана Process Explorer показан процесс WinDbg с несколькими разными отображениями памяти для обращения к анализируемому файлу дампа памяти. Как и большинство Windows-программ, Process Explorer (или одна из используемых ею DLL-библиотек Windows) также использует отображение в память для обращения к файлу данных Windows с именем Locale.nls. Этот файл является частью поддержки интернационализации в Windows.



Также можно провести поиск файлов, отображенных в память; откройте меню Find и выберите команду Find Handle or DLL (или нажмите клавиши Ctrl+F). Данная возможность может пригодиться для определения того, какой процесс (или процессы) использует DLL или файл, отображенный в память, который вы пытаетесь заменить.

Защита памяти

Как объяснялось в главе 1 «Концепции и средства», Windows предоставляет средства защиты памяти, чтобы пользовательские процессы не могли случайно или намеренно повредить адресное пространство другого процесса или операционной системы. В Windows существует четыре основных механизма защиты.

- Все общесистемные структуры данных и пулы памяти, используемые системными компонентами режима ядра, доступны только в режиме ядра. Потоки пользовательского режима не могут обращаться к этим страницам. Если они попытаются это сделать, то оборудование сгенерирует ошибку, о которой диспетчер памяти сообщает потоку как о нарушении прав доступа.
- У каждого процесса имеется отдельное закрытое адресное пространство, защищенное от обращений любых потоков, принадлежащих другим процессам. Даже общая память на самом деле не является исключением, потому что каждый процесс обращается к общим областям по адресам, которые являются частью его собственного виртуального адресного пространства. Единственное исключение встречается в том случае, если другой процесс обладает доступом для чтения/записи виртуальной памяти к объекту процесса (или обладает привилегией `SeDebugPrivilege`), а следовательно, может использовать функцию `ReadProcessMemory` или `WriteProcessMemory`. Каждый раз, когда поток обращается к некоторому адресу, аппаратная поддержка виртуальной памяти в сочетании с диспетчером памяти вмешивается в происходящее и преобразует виртуальный адрес в физический. Управляя преобразованием виртуальных адресов, Windows может гарантировать, что потоки, выполняемые в одном процессе, не попытаются некорректно обратиться к странице, принадлежащей другому процессу.
- Кроме неявной защиты, предоставляемой механизмом преобразования виртуальных адресов в физические, все процессоры, поддерживаемые Windows, предоставляют некоторую разновидность аппаратной защиты памяти — чтение/запись, только чтение и т. д. (Конкретные аспекты каждого вида защиты зависят от процессора.) Например, кодовые страницы в адресном пространстве процесса помечаются как доступные только для чтения, а следовательно, защищаются от изменений со стороны других потоков. В табл. 5.2 перечислены средства защиты памяти, определенные в Windows API. (За информацией о функциях `VirtualProtect`, `VirtualProtectEx`, `VirtualQuery` и `VirtualQueryEx` обращайтесь к документации.)
- Объекты секций общей памяти содержат стандартные списки управления доступом Windows (ACL), которые проверяются при попытке открытия их процессом; таким образом доступ к общей памяти может осуществляться только со стороны процессов, обладающих необходимыми правами. Управление доступом также вступает в действие, когда поток создает секцию для отображенного файла. Чтобы создать секцию, поток должен обладать как минимум доступом для чтения к объекту файла, в противном случае операция завершится неудачей.

Таблица 5.2. Средства защиты памяти, определенные в Windows API

Атрибут	Описание
PAGE_NOACCESS	Любая попытка чтения, записи или выполнения кода в этой области памяти приводит к нарушению прав доступа
PAGE_READONLY	Любая попытка записи (а на процессорах без поддержки выполнения — попытка выполнения кода) в этой области памяти приводит к нарушению прав доступа, но операции чтения разрешены
PAGE_READWRITE	Страница доступна для чтения и записи, но не для выполнения
PAGE_EXECUTE	Любая попытка записи в код, хранящийся в этой области памяти, приведет к нарушению прав доступа, но выполнение (и операции чтения на всех существующих процессорах) разрешено
PAGE_EXECUTE_READ*	Любая попытка записи в этой области памяти приведет к нарушению прав доступа, но выполнение (и операции чтения на всех существующих процессорах) разрешено
PAGE_EXECUTE_READWRITE*	Страница доступна для чтения, записи и выполнения. Любая попытка обращения завершается успехом
PAGE_WRITECOPY	При любой попытке записи в эту область памяти система предоставляет процессу закрытую копию страницы. На процессорах без соответствующей поддержки попытки выполнения кода в этой области приведут к нарушению прав доступа
PAGE_EXECUTE_WRITECOPY	При любой попытке записи в эту область памяти система предоставляет процессу закрытую копию страницы. Чтение и выполнение кода в этой области разрешено (копия в этом случае не создается)
PAGE_GUARD	При любой попытке чтения или записи в страницу защиты происходит исключение EXCEPTION_GUARD_PAGE, а статус страницы защиты отключается. Таким образом, страница защиты срабатывает как одноразовый сигнал. Обратите внимание: флаг может быть установлен одновременно с любой формой защиты страниц из этой таблицы, кроме PAGE_NOACCESS
PAGE_NOCACHE	Использует физическую память, которая не кэшируется. Не рекомендуется для общего использования. Флаг может использоваться для драйверов устройств — например, для отображения буфера видеокадра без кэширования
PAGE_WRITECOMBINE	Включает обращения к памяти в режиме комбинированной записи. При включении этого режима процессор не кэширует запись в память (что может привести к существенному росту трафика в памяти по сравнению с режимом кэширования), но пытается объединять запросы на запись для повышения быстродействия. Например, при нескольких операциях записи, сделанных на один и тот же адрес, может быть выполнена только последняя операция. Аналогичным образом разная запись по смежным адресам может быть свернута в одну большую операцию записи. В обычных приложениях этот режим обычно не используется, но может быть полезен для буферов устройств — например, для отображения буфера видеокадра в режиме комбинированной записи

Атрибут	Описание
PAGE_TARGETS_INVALID и PAGE_TARGETS_NO_UPDATE (Windows 10 и Windows Server 2016)	Значения управляют поведением CFG (Control Flow Guard) для выполняемого кода в этих страницах. Обе константы имеют одинаковое значение, но используются в разных вызовах. PAGE_TARGETS_INVALID означает, что косвенные вызовы должны привести к сбою CFG и аварийному завершению процесса. PAGE_TARGETS_NO_UPDATE позволяет совершать вызовы VirtualProtect, которые изменяют диапазон страниц, чтобы выполнение не приводило к обновлению состояния CFG. За дополнительной информацией о CFG обращайтесь к главе 7 «Безопасность»

* Защита с запретом поддерживается на процессорах, обладающих соответствующей аппаратной поддержкой (например, на всех процессорах x64), но не на старых процессорах x86. При отсутствии поддержки операция выполнения преобразуется в операцию чтения.

После того как поток успешно откроет дескриптор секции, его действия по-прежнему ограничиваются диспетчером памяти и аппаратной защитой страниц, о которой было рассказано выше. Поток может изменить защиту уровня страниц для виртуальных страниц секции, если это изменение не нарушает разрешений ACL для этого объекта секции. Например, диспетчер памяти позволяет потоку изменять страницы секции, доступной только для чтения, на доступ копирования при записи — но не на доступ для чтения/записи. Доступ копирования при записи разрешен, потому что он не отражается на других процессах, работающих с теми же данными.

Предотвращение выполнения данных

Механизм DEP (Data Execution Prevention), или защита страниц NX (No-eXecute), приводит к тому, что при попытке передачи управления команде в странице, для которой запрещено выполнение, происходит нарушение прав доступа. Таким образом предотвращается использование уязвимостей в системе некоторыми видами вредоносных программ через выполнение кода, размещенного в странице данных (например, в стеке). DEP также может выявлять плохо написанные программы, которые некорректно устанавливают разрешения для страниц, предназначенных для выполнения кода. Если в режиме ядра совершается попытка выполнения кода в странице, для которой запрещено выполнение, в системе происходит фатальный сбой с кодом ошибки ATTEMPTED_EXECUTE_OF_NOEXECUTE_MEMORY (0xFC). (За описаниями этих кодов обращайтесь к главе 15 «Анализ аварийных дампов» части 2.) Если это происходит в пользовательском режиме, потоку, пытающемуся использовать некорректную ссылку, передается исключение STATUS_ACCESS_VIOLATION (0xC0000005). Если процесс выделяет память, которая должна использоваться для выполнения, такие страницы должны явно помечаться флагом PAGE_EXECUTE, PAGE_EXECUTE_READ, PAGE_EXECUTE_READWRITE или PAGE_EXECUTE_WRITECOPY при вызове функций выделения памяти уровня страниц.

В 32-разрядных системах x86 с поддержкой DEP бит 63 записи PTE (Page Table Entry) используется для пометки страниц как невыполняемых. Следовательно,

функциональность DEP доступна только при работе процессора в режиме PAE (Physical Address Extension), без которого длина записей PTE равна всего 32 битам. (См. раздел «Преобразование виртуальных адресов на платформе x86» этой главы.) Таким образом, аппаратная поддержка DEP в 32-разрядных системах требует загрузки ядра PAE (%SystemRoot%\System32\Ntkrnlpa.exe), которое в настоящее время является единственным поддерживаемым ядром для систем x86.

В ARM-системах DEP присваивается значение `AlwaysOn`.

В 64-разрядных версиях Windows защита выполнения всегда применяется ко всем 64-разрядным процессам и драйверам устройств, а отключить ее можно только присваиванием параметру `nx` BCD значения `AlwaysOff`. Защита выполнения для 32-разрядных программ зависит от настроек конфигурации системы, которые будут описаны ниже. В 64-разрядных версиях Windows защита выполнения применяется к стекам потоков (как пользовательского режима, так и режима ядра), страницам пользовательского режима, не помеченным как выполняемые, выгружаемому пулу ядра и сеансовому пулу ядра. Пулы памяти ядра описаны в разделе «Кучи режима ядра (пулы системной памяти)». С другой стороны, в 32-разрядных версиях Windows защита выполнения применяется только к стекам потоков и страницам пользовательского режима, но не к выгружаемым пулам и сеансовым пулам.

Применение защиты выполнения в 32-разрядных процессах зависит от значения параметра BCD `nx`. Чтобы изменить настройки, откройте вкладку Предотвращение выполнения данных (Data Execution Prevention) в диалоговом окне Параметры

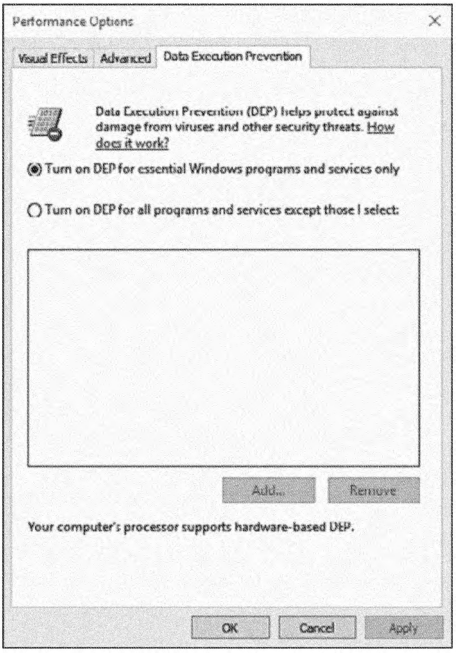


Рис. 5.3. Настройки вкладки «Предотвращение выполнения данных»

быстродействия (Performance Options) (рис. 5.3). Чтобы открыть это диалоговое окно, щелкните правой кнопкой мыши на значке Компьютер (Computer), выберите команду Свойства (Properties), щелкните на кнопке Advanced System Settings и выберите команду Performance Settings. Если вы настроите защиту запрета на выполнение в диалоговом окне Параметры быстродействия (Performance Options), то параметру BCD nx будет присвоено соответствующее значение. В табл. 5.3 перечислены значения и их соответствие на вкладке Предотвращение выполнения данных (Data Execution Prevention). В разделе реестра HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\AppCompatFlags\Layers перечислены 32-разрядные приложения, исключенные из защиты выполнения; имя параметра представляет полный путь к исполняемому файлу, и ему присваивается значение `DisableNXShowUI`.

Таблица 5.3. Значения BCD nx

Значение BCD nx	Параметр на вкладке Предотвращение выполнения данных (Data Execution Prevention)	Объяснение
`OptIt`	Включить DEP только для основных программ и служб Windows (Turn on DEP for Essential Windows Programs and Services Only)	DEP включается для основных системных образов Windows. Режим позволяет 32-разрядным процессам динамически настраивать DEP на протяжении их жизненного цикла
`OptOut`	Включить DEP для всех служб и программ, кроме выбранных ниже (Turn on DEP for All Programs and Services Except Those I Select)	DEP включается для всех исполняемых файлов, кроме перечисленных. Режим позволяет 32-разрядным процессам динамически настраивать DEP на протяжении их жизненного цикла. Также включаются исправления системной совместимости для DEP
`AlwaysOn`	Нет аналога в диалоговом окне	Включает DEP для всех компонентов без возможности исключения отдельных приложений. Режим запрещает динамическую конфигурацию для 32-разрядных процессов и отключает исправления системной совместимости
`AlwaysOff`	Нет аналога в диалоговом окне	Отключает DEP (не рекомендуется). Также отключает динамическую конфигурацию для 32-разрядных процессов

В клиентских версиях Windows (как 64-разрядных, так и 32-разрядных) защита выполнения для 32-разрядных процессов настраивается по умолчанию только для основных исполняемых файлов операционной системы Windows. Иначе говоря, параметру BCD nx присваивается значение `OptInt`. Это делается для того, чтобы не нарушать работу 32-разрядных приложений, которые могут зависеть от возможности исполнения кода в страницах, не помеченных как исполняемые, — например, в самораспаковывающихся или упакованных приложениях. В серверных версиях Windows защита выполнения для 32-разрядных приложений настраивается по умолчанию для всех 32-разрядных программ (другими словами, параметру BCD nx присваивается значение `OptOut`).

Даже если вы принудительно включите DEP, существуют другие способы, при помощи которых приложения могут отключить DEP для своих образов. Например, независимо от настройки параметров защиты выполнения загрузчик образов проверяет сигнатуру исполняемого файла по известным механизмам защиты авторских прав (таких, как SafeDisc и SecuROM) и отключает защиту выполнения для обеспечения совместимости со старыми защищенными программами — такими, как компьютерные игры (за дополнительной информацией о загрузчике образов обращайтесь к главе 3).

ЭКСПЕРИМЕНТ: ПРОСМОТР ИНФОРМАЦИИ О ЗАЩИТЕ DEP ДЛЯ ПРОЦЕССОВ

Process Explorer может вывести текущее состояние DEP для всех процессов вашей системы, включая информацию о том, пользуется ли процесс постоянной защитой. Чтобы просмотреть статус DEP для процессов, щелкните правой кнопкой мыши на любом столбце в дереве процессов, выберите команду Select Columns, а затем выберите DEP Status на вкладке Process Image. Возможны три значения:

- **DEP (permanent).** Означает, что для процесса включается DEP, потому что он относится к числу «основных программ и служб Windows».
- **DEP.** Означает, что процесс дал согласие на использование DEP. Это может быть обусловлено общесистемной политикой для всех 32-разрядных процессов (из-за вызова API-функции `SetProcessDEPPolicy` или из-за установки флага `/NXCOMPAT` при построении образа).
- **Ничего.** Если в столбце не выводится информация о процессе, DEP, поддержка DEP отключена из-за политики системного уровня, явного вызова API-функции или оболочки совместимости.

Кроме того, для обеспечения совместимости с более старыми версиями фреймворка ATL (Active Template Library) (версия 7.1 и ранее) ядро Windows предоставляет среду эмуляции ATL. Эта среда обнаруживает последовательности кодовых инструкций преобразователя ATL, вызвавшие исключение DEP, и эмулирует ожидаемую операцию. Разработчики приложений могут потребовать, чтобы эмуляция ATL не применялась — для этого они используют новейшую версию компилятора Microsoft C++ и указывают флаг `/NXCOMPAT` (это приводит к установке флага `IMAGE_DLL-CHARACTERISTICS_NX_COMPAT` в заголовке PE); тем самым они сообщают системе, что исполняемый образ полностью поддерживает DEP. Учтите, что эмуляция ATL полностью отключается при установке значения `AlwaysOn`.

Наконец, если система работает в режиме `OptIn` или `OptOut` и выполняет 32-разрядный процесс, функция `SetProcessDEPPolicy` позволяет процессу динамически отключить DEP или включить его на постоянной основе. Если механизм DEP включается этой API-функцией, его не удастся отключить на программном уровне на протяжении жизненного цикла процесса. Функция также может использоваться

для динамического отключения эмуляции ATL, если образ не был откомпилирован с флагом `/NXCOMPAT`. В 64-разрядных процессах или в системах, загруженных в режиме `AlwaysOff` или `AlwaysOn`, функция всегда возвращает признак неудачи. Функция `GetProcessDEPPolicy` возвращает 32-разрядную политику DEP уровня процесса (в 64-разрядных системах, где политика всегда одна — включенная, возвращается признак ошибки), тогда как `GetSystemDEPPolicy` может использоваться для возвращения значений, соответствующих политикам из табл. 5.3.

Копирование при записи

Защита страниц методом копирования при записи представляет собой оптимизацию, которая применяется диспетчером памяти для экономии физической памяти. Когда процесс отображает в режиме копирования при записи представление объекта секции, содержащее страницы для чтения/записи, диспетчер памяти откладывает копирование страниц до момента записи страницы (вместо того, чтобы создавать закрытую копию для процесса в момент отображения). Например, на рис. 5.4 два процесса совместно используют три страницы, каждая из которых помечена режимом копирования при записи, но ни один из двух процессов еще не пытался изменить данные страниц.

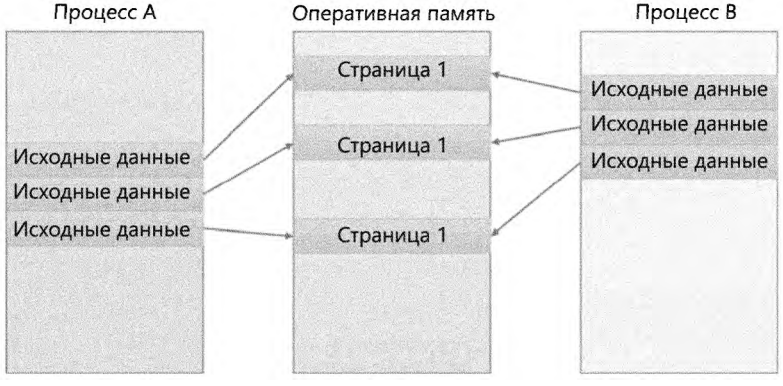


Рис. 5.4. Состояние до копирования при записи

Если поток одного из процессов выполняет запись в страницу, генерируется ошибка управления памятью. Диспетчер памяти видит, что запись осуществляется в страницу, для которой назначен режим копирования при записи, поэтому вместо сообщения об ошибке как о нарушении прав доступа происходит следующее:

1. Диспетчер памяти создает новую страницу чтения/записи в физической памяти.
2. Содержимое исходной страницы копируется в новую страницу.

3. Соответствующая информация отображения страницы обновляется (см. далее в этой главе) в этом процессе ссылкой на новый адрес.
4. Исключение игнорируется, а инструкция, сгенерировавшая ошибку, выполняется повторно.

На этот раз операция записи выполняется успешно. Но как показано на рис. 5.5, только что скопированная страница становится закрытой для процесса, выполнившего запись, и остается невидимой для другого процесса, который продолжает использовать общую страницу с копированием при записи. Каждый новый процесс, который записывает данные в эту общую страницу, также получает собственную закрытую копию.

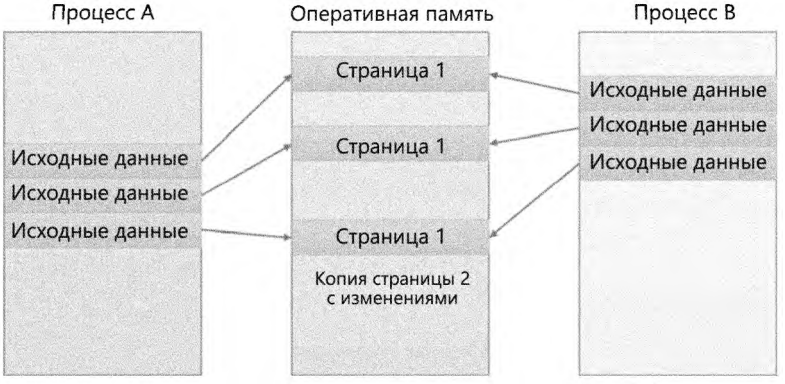


Рис. 5.5. Состояние после копирования при записи

Одно из применений копирования при записи — реализация точек останова в отладчиках. Например, по умолчанию кодовые страницы начинаются в состоянии доступа только для выполнения. Если программист устанавливает точку останова в процессе отладки программы, отладчик должен добавить в код инструкцию точки останова. Для этого сначала режим защиты страницы меняется на `PAGE_EXECUTE_READWRITE`, после чего изменяется поток инструкций. Так как кодовая страница является частью отображенной секции, диспетчер памяти создает закрытую копию процесса с установленной точкой останова, а другие процессы продолжают использовать немодифицированную кодовую страницу.

Копирование при записи — один из примеров механизма *отложенного вычисления*, который применяется диспетчером памяти настолько часто, насколько это возможно. Алгоритмы отложенного вычисления откладывают выполнение высокозатратных операций до того момента, когда это станет абсолютно необходимо. Если выполнять операцию не потребуется, то время на нее не тратится.

Чтобы проанализировать пропорцию ошибок копирования при записи, просмотрите счетчик быстродействия Write Copies/Sec в категории Память (Memory) Системного монитора.

Address Windowing Extensions

Хотя 32-разрядная версия Windows может поддерживать до 64 Гбайт физической памяти (см. табл. 2.2), каждый 32-разрядный пользовательский процесс по умолчанию содержит не более 2 Гбайт виртуального адресного пространства. (Вы можете повысить этот порог до 3 Гбайт при помощи параметра BCD `increaseuserva`, описанного в разделе «Структура виртуальных адресных пространств».) Приложение, которому потребуется легко использовать более 2 Гбайт (или 3 Гбайт) данных в одном процессе, может воспользоваться механизмом отображения файлов, переключающим часть своего адресного пространства на разные части большого файла. Тем не менее каждое повторное отображение будет сопровождаться значительной подкачкой данных.

Для повышения быстродействия (и еще более точного управления) Windows предоставляет набор функций, объединенных в категорию *AWE* (Address Windowing Extensions). Эти функции позволяют процессу выделить объем памяти, превышающий тот, который может быть представлен в виртуальном адресном пространстве. Затем приложение может обратиться к физической памяти, отображая часть своего виртуального адресного пространства в отдельные части физической памяти в разные моменты времени. Выделение и использование памяти функциями AWE осуществляется за три этапа.

1. Выделение физической памяти для использования. Приложение использует Windows-функции `AllocateUserPhysicalPages` или `AllocateUserPhysicalPagesNuma`. (Для этого необходима привилегия `SeLockMemoryPrivilege`.)
2. Одна или несколько областей виртуального адресного пространства используются в качестве «окон» для отображения представлений физической памяти. Приложение использует Win32-функции `VirtualAlloc`, `VirtualAllocEx` и `VirtualAllocExNuma` с флагом `MEM_PHYSICAL`.
3. Вообще говоря, на этапах 1 и 2 происходит инициализация. Чтобы реально использовать память, приложение использует функцию `MapUserPhysicalPages` или `MapUserPhysicalPagesScatter` для отображения части физической области, выделенной на этапе 1, в одну из виртуальных областей («окон»), выделенных на этапе 2.

Пример показан на рис. 5.6. Приложение создало 256-мегабайтное окно в своем адресном пространстве и выделило 4 Гбайт физической памяти. После этого оно может использовать функцию `MapUserPhysicalPages` или `MapUserPhysicalPagesScatter` для обращения к любой части физической памяти, для чего нужная часть памяти отображается в 256-мегабайтное окно. Размер виртуального адресного пространства приложения определяет объем физической памяти, с которым может работать приложение в каждом конкретном отображении. Чтобы обратиться к другой части выделенной памяти, приложение может просто отобразить область заново.

Функции AWE существуют во всех версиях Windows и могут использоваться независимо от объема физической памяти. Тем не менее AWE приносит наибольшую пользу в 32-разрядных системах, оснащенных более чем 2 Гбайт физической памяти,

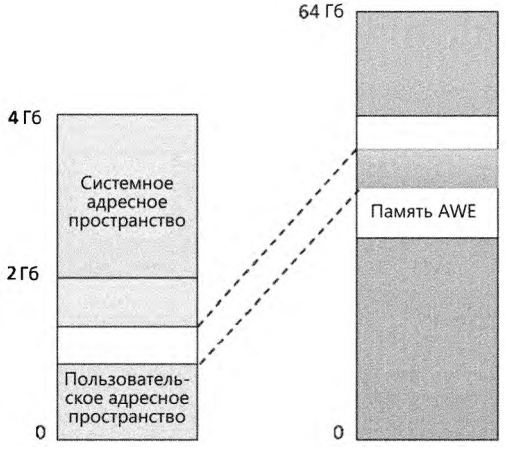


Рис. 5.6. Использование AWE для отображения физической памяти

потому что AWE позволяет 32-разрядным процессам работать с большим объемом памяти, чем позволило бы виртуальное адресное пространство. Также AWE может применяться для обеспечения безопасности. Так как память AWE никогда не выгружается, данные в памяти AWE никогда не копируются в страничный файл, содержимое которого может быть проанализировано при загрузке другой операционной системы. (VirtualLock предоставляет аналогичные гарантии для страниц вообще.)

Наконец, существуют некоторые ограничения на выделение и отображение памяти функциями AWE:

- Страницы не могут совместно использоваться процессами.
- Одна физическая страница не может отображаться более чем на один виртуальный адрес.
- Защита страниц ограничивается доступом для чтения/записи, доступом только для чтения и запретом доступа.

В 64-разрядных версиях Windows механизм AWE уже не столь эффективен, потому что эти системы поддерживают до 128 Тбайт виртуального адресного пространства на процесс, тогда как объем ОЗУ не может превышать 24 Тбайт (в системах Windows Server 2016). А значит, приложение может использовать оперативную память в объеме, превышающем объем виртуального адресного пространства, без помощи AWE; объем оперативной памяти в системе всегда меньше виртуального адресного пространства процесса. Впрочем, AWE все равно может пригодиться для настройки невыгружаемых областей адресного пространства процесса — этот механизм обладает более высокой гранулярностью, чем API отображения файлов. (Системный размер страницы составляет 4 Кбайт вместо 64 Кбайт.)

За описанием структур данных таблиц страниц, используемых для отображения памяти в системах с физической памятью более 4 Гбайт, обращайтесь к разделу «Преобразование виртуальных адресов x86».

Кучи режима ядра (пулы системной памяти)

На стадии инициализации системы диспетчер памяти создает два пула памяти, или *кучи* (heap), с динамически изменяемым размером; они используются большинством компонентов режима ядра для выделения системной памяти:

- **Невыгружаемый пул** состоит из диапазонов системных виртуальных адресов, которые заведомо будут постоянно находиться в физической памяти. Таким образом, к этим адресам можно обратиться в любой момент без возникновения *ошибки страницы* (page fault) — это означает, что к ним можно обращаться на любом уровне IRQL. Одна из причин необходимости невыгружаемого пула заключается в том, что ошибки страниц не могут обслуживаться на уровне DPC/Dispatch и выше. Таким образом, весь код и данные, выполнение/обращение к которым может происходить на уровне DPC/Dispatch и выше, должны находиться в невыгружаемой памяти.
- **Выгружаемый пул** — область виртуальной памяти в системном пространстве, содержимое которой может подгружаться в систему и выгружаться из нее. Драйверы устройств, которым не нужно обращаться к памяти с уровня DPC/Dispatch и выше, могут использовать выгружаемый пул. Он доступен из любого контекста процесса.

Оба пула памяти располагаются в системной части адресного пространства и отображаются в виртуальное адресное пространство каждого процесса. Исполняющая среда предоставляет функции выделения и освобождения этих пулов. За информацией обращайтесь к описаниям функций, имена которых начинаются с `ExAllocatePool`, `ExAllocatePoolWithTag` и `ExFreePool`, в документации WDK (Windows Development Kit).

Сначала система использует четыре выгружаемых пула, которые объединяются в общий системный выгружаемый пул, и два невыгружаемых пула. Дополнительные пулы — до 64 — создаются в зависимости от количества NUMA-узлов в системе. Наличие нескольких выгружаемых пулов сокращает частоту блокировки системного кода из-за одновременных вызовов функций пулов. Кроме того, разные созданные пулы отображаются в разные виртуальные адресные пространства, соответствующие разным NUMA-узлам в системе. Также разные структуры данных (например, большие резервные списки страниц) для описания выделенных пулов отображаются между разными NUMA-узлами.

Кроме выгружаемых и невыгружаемых пулов, существуют и другие пулы со специальными атрибутами и сценариями использования. Например, в пространстве сеанса имеется область пула, которая используется для данных, общих для всех процессов в сеансе. Память, выделенная из другого пула — *специального пула* (special pool), — окружается страницами, доступ к которым запрещен; это помогает изолировать проблемы в коде, который обращается к памяти до или после выделенной области.

Размеры пулов

Исходный размер невыгружаемого пула определяется объемом физической памяти в системе, а затем пул расширяется по мере надобности. Для невыгружаемого пула исходный размер составляет 3 % системной оперативной памяти. Если он менее 40 Мбайт, система использует 40-мегабайтный пул при условии, что 10 % оперативной памяти превышают 40 Мбайт. В противном случае 10 % оперативной памяти выбираются в качестве минимума. Windows динамически выбирает максимальный размер пулов и разрешает пулу увеличиваться от исходного размера до максимумов, указанных в табл. 5.4.

Таблица 5.4. Максимальные размеры пулов

Тип пула	Максимум в 32-разрядных системах	Максимум в 64-разрядных системах (Windows 8, Server 2012)	Максимум в 64-разрядных системах (Windows 8.1, 10, Server 2012 R2, 2016)
Невыгружаемый	75 % физической памяти или 2 Гбайт (меньшее из двух)	75 % физической памяти или 128 Гбайт (меньшее из двух)	16 Тбайт
Выгружаемый	2 Гбайт	384 Гбайт	15,5 Тбайт

Таблица 5.5. Системные переменные размера пула и счетчики производительности

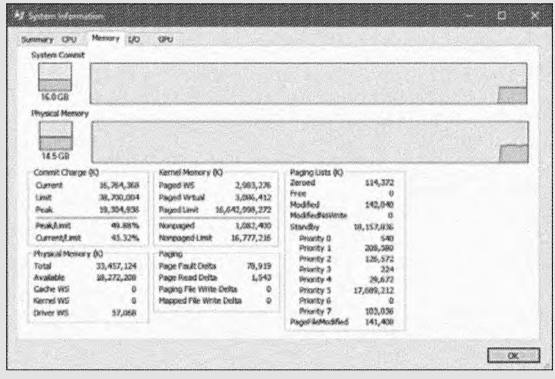
Переменная ядра	Счетчик производительности	Описание
`MmSizeOfNonPagedPoolInBytes`	Memory: Pool nonpaged bytes	Исходный размер невыгружаемого пула; может автоматически сокращаться или увеличиваться системой в соответствии с требованиями к памяти. Эти изменения не отражаются в переменной ядра, но отражаются в счетчике производительности
`MmMaximumNonPagedPoolInBytes` (Windows 8.x и Server 2012/R2)	–	Максимальный размер невыгружаемого пула
`MiVisibleState->MaximumNonPagePool InBytes` (Windows 10 и Server 2016)	–	Максимальный размер невыгружаемого пула
–	Memory: Pool paged bytes	Текущий общий размер выгружаемого пула
`WorkingSetSize` (количество страниц в структуре `MmPagedPoolWs struct` (тип `MMSUPPORT`) (Windows 8.x и Server 2012/R2)	Memory: Pool paged resident bytes	Текущий физический (резидентный) размер выгружаемого пула
`MmSizeOfPagedPoolInBytes` (Windows 8.x и Server 2012/R2)	–	Максимальный (виртуальный) размер выгружаемого пула

Переменная ядра	Счетчик производительности	Описание
`MiState.Vs.SizeOfPagedPoolIn Bytes` (Windows 10 и Server 2016)	–	Максимальный (виртуальный) размер выгружаемого пула

Четыре из этих вычисленных размеров хранятся в переменных ядра в Windows 8.x и Server 2012/R2. Три из них предоставляются в виде счетчиков производительности, а один вычисляется только как значение счетчика производительности. В Windows 10 и Server 2016 глобальные переменные были перемещены в поля глобальной структуры управления памятью (`MI_SYSTEM_INFORMATION`) с именем `MiState`. В ней находится переменная с именем `Vs` (и типом `_MI_VISIBLE_STATE`), в которой хранится эта информация. Глобальная переменная `MiVisibleState` также указывает на поле `Vs`. Переменные и счетчики перечислены в табл. 5.5.

ЭКСПЕРИМЕНТ: ОПРЕДЕЛЕНИЕ МАКСИМАЛЬНЫХ РАЗМЕРОВ ПУЛОВ

Максимальные размеры пулов можно узнать в программе Process Explorer или в процессе «живой» отладки ядра (см. главу 1). Чтобы просмотреть данные в Process Explorer, откройте меню View, выберите команду System Information и перейдите на вкладку Memory. Предельные значения отображаются в разделе Kernel Memory:



ПРИМЕЧАНИЕ Для получения этой информации Process Explorer должен иметь доступ к символическим именам ядра вашей системы. За информацией о том, как настроить Process Explorer для получения символической информации, обращайтесь к врезке «Эксперимент: просмотр информации о процессах в Process Explorer» главы 1.

Для просмотра той же информации в отладчике ядра можно воспользоваться командой !vm, как было показано ранее в этой главе.

Анализ использования пулов

Объект счетчиков производительности Память (Memory) имеет разные счетчики для невыгружаемого и выгружаемого пула (как виртуальных, так и физических). Кроме того, программа Poolmon (из каталога WDK Tools) позволяет получать подробную информацию об использовании невыгружаемого и выгружаемого пула. При запуске Poolmon вы увидите окно, похожее на изображенное на рис. 5.7.

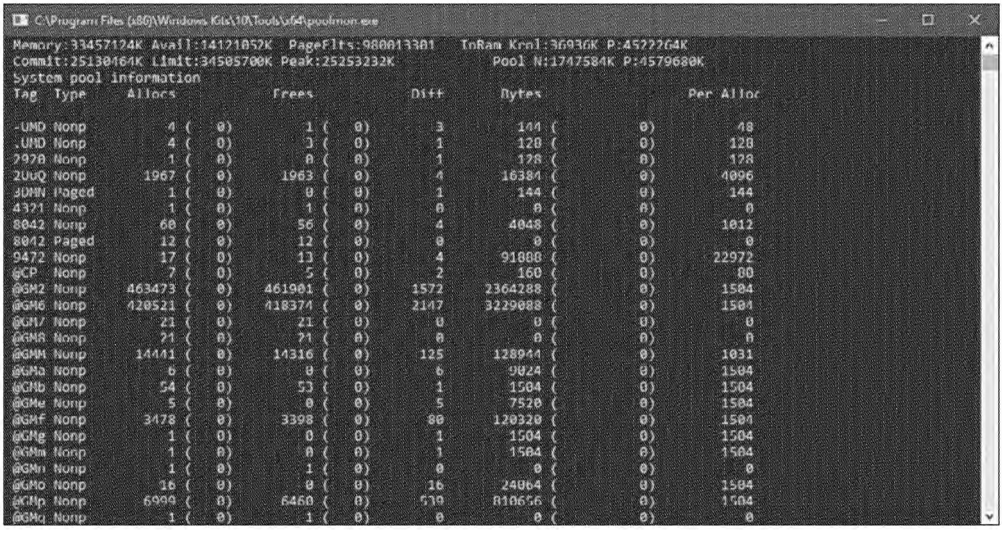


Рис. 5.7. Вывод Poolmon

Выделенные строки представляют изменения в выводимой информации. (Чтобы отключить цветовое выделение, нажмите клавишу / во время работы Poolmon; повторное нажатие / снова включает выделение.) Клавиша ? во время выполнения Poolmon вызывает экран со справочной информацией. Вы можете настроить отслеживаемые пулы (выгружаемый и/или невыгружаемый) и порядок сортировки. Например, если вы нажмете клавишу P, пока выводятся только невыгружаемые пулы, а затем нажмете клавишу D для сортировки по столбцу Diff (разность), можно узнать, какие структуры наиболее многочисленны в невыгружаемом пуле. Кроме того, выводятся параметры командной строки для отслеживания конкретных тегов (или всех тегов, кроме заданного). Например, команда `poolmon -iCM` будет отслеживать только теги CM (выделение памяти диспетчером конфигурации, который управляет реестром). Столбцы выходных данных перечислены в табл. 5.6.

За описаниями тегов пулов, используемых Windows, обращайтесь к файлу Pooltag.txt в подкаталоге Triage каталога, в котором размещается инструментарий отладки для Windows. Поскольку теги пулов сторонних драйверов устройств в файле не указаны, вы можете воспользоваться ключом `-c` в 32-разрядной версии Poolmon, поставляемой с WDK, для генерирования локального файла тегов пулов (Localtag.

txt). Этот файл содержит теги пулов, которые используются драйверами, найденными в вашей системе, включая сторонние драйверы. (Обратите внимание: если двоичный файл драйвера был удален после загрузки, его теги пулов распознаваться не будут.)

Таблица 5.6. Столбцы Poolmon

Столбец	Объяснение
`Tag`	Четырехбайтовый тег, назначенный выделяемой памяти из пула
`Type`	Тип пула (выгружаемый или невыгружаемый)
`Allocs`	Количество всех операций выделения. В скобках выводится разность со значением столбца Allocs с момента последнего обновления
`Frees`	Количество всех операций освобождения. В скобках выводится разность со значением столбца Frees с момента последнего обновления
`Diff`	Разность между количеством выделений и освобождений памяти
`Bytes`	Общее количество байтов, потребляемых тегом. В скобках выводится разность со значением столбца Bytes с момента последнего обновления
`Per Alloc`	Размер одного экземпляра данного тега в байтах

Также можно провести поиск драйверов устройств в вашей системе по тегам пулов программой Strings.exe из пакета Sysinternals. Например, следующая команда выводит драйверы, содержащие строку `"abcd"`:

```
strings %SYSTEMROOT%\system32\drivers\*.sys | findstr /i "abcd"
```

Драйверы устройств не обязаны находиться в папке %SystemRoot%\System32\Drivers — папка может быть любой. Чтобы вывести полный путь всех загруженных драйверов, выполните следующие действия:

1. Откройте меню Пуск (Start) и введите команду `Msinfo32` (должна появиться команда Сведения о системе (System Information)).
2. Выполните команду Сведения о системе (System Information).
3. Выберите узел Программная среда (Software Environment).
4. Выберите пункт Системные драйверы (System Drivers). Если драйвер устройства был загружен, а затем удален из системы, он не будет присутствовать в списке.

Другой способ просмотра информации об использовании пулов по драйверам устройств основан на включении функциональности отслеживания пулов Driver Verifier (см. главу 6). Хотя в этом варианте отображение тегов пулов на драйверы устройств становится излишним, он требует перезагрузки (для включения Driver Verifier с нужными драйверами). После перезагрузки с включенным отслеживанием пулов либо запустите графическую программу Driver Verifier Manager (%SystemRoot%\System32\Verifier.exe), либо выполните команду `Verifier /Log` для сохранения информации в файле.

Наконец, для просмотра информации использования пулов также можно воспользоваться командой отладчика ядра `!poolused`. Команда `!poolused 2` выводит

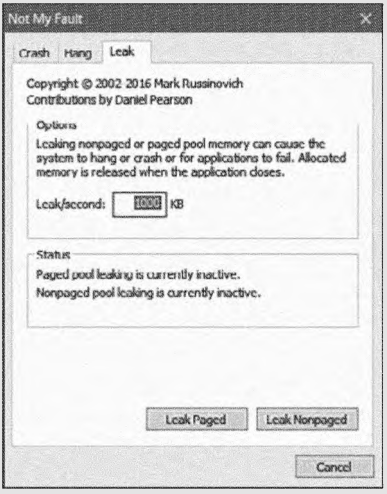
информацию об использовании невыгружаемых пулов, отсортированных по тегам пулов. Команда `!poolused 4` выводит информацию об использовании выгружаемых пулов, также отсортированную по тегам пулов. Следующий пример демонстрирует неполный вывод этих двух команд:

```
lkd> !poolused 2
........
Sorting by NonPaged Pool Consumed
               NonPaged                Paged
Tag     Allocs         Used   Allocs       Used
 File    626381    260524032        0          0      File objects
 Ntfx    733204    227105872        0          0      General Allocation , Binary:
                                                      ntfs.sys
MmCa    513713    148086336        0          0      Mm control areas for mapped
                                                      files , Binary: nt!mm
FMsl    732490    140638080        0          0      STREAM_LIST_CTRL structure ,
                                                      Binary: fltmgr.sys
 CcSc    104420     56804480        0          0      Cache Manager Shared Cache Map
                                                      , Binary: nt!cc
SQSF    283749     45409984        0          0      UNKNOWN pooltag 'SQSF', please
                                                      update pooltag.txt
 FMfz    382318     42819616        0          0      FILE_LIST_CTRL structure ,
                                                      Binary: fltmgr.sys
 FMsc     36130     32950560        0          0      SECTION_CONTEXT structure ,
                                                      Binary: fltmgr.sys
 EtwB       517     31297568      107  105119744      Etw Buffer , Binary: nt!etw
 DFmF    382318     30585440   382318   91756320      UNKNOWN pooltag 'DFmF', please
                                                      update pooltag.txt
 DFmE    382318     18351264        0          0      UNKNOWN pooltag 'DFmE', please
                                                      update pooltag.txt
 FSfc    382318     18351264        0          0      Unrecoginzed File System Run
                                                      Time allocations (update
                                                      pooltag.w) , Binary: nt!fsrtl
 smNp      4295     17592320        0          0      ReadyBoost store node pool
                                                      allocations , Binary: nt!store
                                                      or rdyboost.sys
 Thre      5780     12837376        0          0      Thread objects , Binary: nt!ps
 Pool         8     12834368        0          0      Pool tables, etc.
```

ЭКСПЕРИМЕНТ: ДИАГНОСТИКА УТЕЧКИ ПАМЯТИ В ПУЛАХ

В этом эксперименте мы исправим реальную утечку памяти из выгружаемого пула в вашей системе; для выявления утечки будут использованы средства, описанные в предыдущем разделе. Утечка генерируется программой Notmyfault из пакета Sysinternals. Выполните следующие действия:

1. Запустите программу Notmyfault.exe для разрядности вашей ОС (например, 64-разрядную версию для 64-разрядной системы).
2. Notmyfault.exe загружает драйвер устройства Myfault.sys и выводит диалоговое окно Not My Fault с выбранной вкладкой Crash. Щелкните на вкладке Leak. Окно должно выглядеть примерно так:



3. Введите в поле Leak/second значение 1000 Кбайт.
4. Щелкните на кнопке Leak Paged. Программа Notmyfault начинает отправлять драйверу Myfault запросы на выделение памяти из выгружаемого пула. Notmyfault продолжает отправлять запросы, пока вы не нажмете кнопку Stop Paged. Выгружаемый пул обычно не освобождается даже при закрытии программы, которая привела к утечке (за счет взаимодействия с драйвером устройства, содержащим ошибку). Утечка из пула идет постоянно, вплоть до момента перезагрузки системы. Впрочем, для упрощения тестирования драйвер устройства Myfault обнаруживает, что процесс был закрыт, и освобождает выделенную память.
5. Пока в пуле существует утечка, откройте диспетчер задач, перейдите на вкладку Быстродействие (Performance) и выберите категорию Память (Memory). Обратите внимание: значение Выгружаемый пул (Paged Pool) постепенно растет. В этом также можно убедиться на панели System Information программы Process Explorer (откройте меню View, выберите команду System Information и перейдите на вкладку Memory).
6. Чтобы определить, с каким тегом пулов связана утечка, запустите программу Poolmon и нажмите клавишу B, чтобы отсортировать информацию по количеству байтов.
7. Дважды нажмите клавишу P, чтобы в Poolmon отображался только выгружаемый пул. Обратите внимание: тег пула Leak поднимается в начало списка. (Poolmon помечает изменения в выделенной памяти подсветкой изменяемых строк.)
8. Щелкните на кнопке Stop Paged, чтобы не исчерпать выгружаемый пул в вашей системе.
9. Запустите программу Strings (из пакета Sysinternals), чтобы найти исполняемые файлы драйверов с тегом пула Leak. В результате поиска должен

быть найден файл Myfault.sys; тем самым подтверждается, что этот драйвер использует тег пула Leak.

```
Strings %SystemRoot%\system32\drivers\*.sys | findstr Leak
```

Резервные списки

Windows предоставляет быстрый механизм выделения памяти — так называемые *резервные списки* (look-aside lists). Основное различие между пулами и резервными списками заключается в том, что из пула могут выделяться блоки памяти переменного размера, тогда как резервные списки содержат только блоки фиксированного размера. И хотя обобщенные пулы памяти более гибки в отношении того, что они могут поддерживать, резервные списки работают быстрее, потому что они не используют спин-блокировки.

Компоненты исполняющей среды и драйверы устройств могут создавать резервные списки, размер которых соответствует размеру часто создаваемых структур данных, функциями `ExInitializeNPagedLookasideList` (для невыгружаемой памяти) и `ExInitializePagedLookasideList` (для выгружаемой памяти) — за описаниями обращайтесь к документации WDK. Чтобы свести к минимуму затраты на многопроцессорную синхронизацию, некоторые подсистемы исполняющей среды — диспетчер ввода/вывода, диспетчер кэша, диспетчер объектов — создают раздельные для каждого процессора резервные списки для всех структур данных, к которым происходят частые обращения. Исполняющая среда также создает общий выгружаемый и невыгружаемый резервный список для выделений малых блоков (256 байт и менее) для каждого процессора.

Если резервный список пуст (как в момент создания), система должна выделять память из выгружаемого и невыгружаемого пула. Но если он содержит освобожденный блок, запрос на выделение памяти может быть удовлетворен очень быстро. (Список растет по мере возвращения в него блоков.) Функции выделения памяти автоматически настраивают количество свободных буферов, которые сохраняются резервными списками, в зависимости от того, насколько часто драйвер устройства или исполняющая подсистема выделяет память из списка. Чем чаще выделяется память, тем больше блоков сохраняется в списке. Если из резервного списка не выделяется память, то его размер автоматически сокращается (эта проверка выполняется один раз в секунду, когда системный поток диспетчера набора балансировки пробуждается и вызывает функцию `ExAdjustLookasideDepth`).

ЭКСПЕРИМЕНТ: ПРОСМОТР СИСТЕМНЫХ РЕЗЕРВНЫХ СПИСКОВ

Для просмотра содержимого и размеров различных системных резервных списков можно воспользоваться командой отладчика ядра `!lookaside`. Ниже приведен фрагмент вывода этой команды:

```
lkd> !lookaside

Lookaside "nt!CcTwilightLookasideList" @ 0xfffff800c6f54300  Tag(hex):
          0x6b576343 "CcWk"
    Type           =       0200  NonPagedPoolNx
    Current Depth  =          0  Max Depth  =          4
    Size           =        128  Max Alloc  =        512
    AllocateMisses =     728323  FreeMisses =     728271
    TotalAllocates =    1030842  TotalFrees =    1030766
    Hit Rate       =         29% Hit Rate   =         29%

Lookaside "nt!IopSmallIrpLookasideList" @ 0xfffff800c6f54500  Tag(hex):
          0x73707249 "Irps"
    Type           =       0200  NonPagedPoolNx
    Current Depth  =          0  Max Depth  =          4
    Size           =        280  Max Alloc  =       1120
    AllocateMisses =      44683  FreeMisses =      43576
    TotalAllocates =     232027  TotalFrees =     230903
    Hit Rate       =         80% Hit Rate   =         81%

Lookaside "nt!IopLargeIrpLookasideList" @ 0xfffff800c6f54600  Tag(hex):
          0x6c707249 "Irpl"
    Type           =       0200  NonPagedPoolNx
    Current Depth  =          0  Max Depth  =          4
    Size           =       1216  Max Alloc  =       4864
    AllocateMisses =     143708  FreeMisses =     142551
    TotalAllocates =     317297  TotalFrees =     316131
    Hit Rate       =         54% Hit Rate   =         54%
...

Total NonPaged currently allocated for above lists =        0
Total NonPaged potential for above lists           =    13232
Total Paged currently allocated for above lists    =        0
Total Paged potential for above lists              =     4176
```

Диспетчер кучи

Многие приложения выделяют блоки, размер которых меньше минимальной гранулярности, возможной при использовании страничных функций — таких, как `VirtualAlloc` (64 Кбайт). Выделение такой большой области для относительно малых блоков не оптимально с позиций использования памяти и быстродействия. Для решения этой проблемы Windows предоставляет компонент, называемый *диспетчером кучи* (heap dispatcher); он управляет выделением памяти внутри областей большего размера, зарезервированных функциями выделения памяти со страничной гранулярностью. Гранулярность выделения в диспетчере кучи относительно невелика: 8 байт в 32-разрядных системах, 16 байт в 64-разрядных системах. Диспетчер кучи был спроектирован для оптимизации использования памяти и быстродействия этих меньших операций.

Диспетчер кучи существует в двух местах: Ntdll.dll и Ntoskrnl.exe. API подсистем (например, API кучи Windows) вызывают функции из Ntdll.dll, а различные компонен-

ты исполняющей среды и драйверы устройств вызывают функции из Ntoskrnl.exe. Его собственные интерфейсы (с префиксом `Rtl`) доступны только для внутренних компонентов Windows и драйверов устройств режима ядра. Документированные интерфейсы кучи Windows API (с префиксом `Heap`) передают управление внутренним функциям из Ntdll.dll. Кроме того, предоставляются старые API (с префиксом `Local` или `Global`) для поддержки более старых Windows-приложений. В своей внутренней реализации они также обращаются с вызовами к диспетчеру кучи, используя некоторые из его специализированных интерфейсов для поддержки старого поведения. Наиболее часто используемые функции кучи Windows:

- `HeapCreate` или `HeapDestroy` — создают или удаляют кучу соответственно. Исходный зарезервированный и подтвержденный размер может задаваться при создании.
- `HeapAlloc` — функция выделяет блок из кучи. Вызов перенаправляется `RtlAllocateHeap` в Ntdll.dll.
- `HeapFree` — функция освобождает блок, ранее выделенный функцией `HeapAlloc`.
- `HeapReAlloc` — функция изменяет размер существующего выделенного буфера с сокращением или увеличением существующего блока. Вызов перенаправляется функции `RtlReAllocateHeap` в Ntdll.dll.
- `HeapLock` и `HeapUnlock` — функции управляют блокировками при операциях с кучей.
- `HeapWalk` — обход элементов и областей в куче.

Кучи процессов

У каждого процесса имеется как минимум одна куча: *куча процесса по умолчанию*. Куча по умолчанию создается при запуске процесса и никогда не удаляется на протяжении его жизненного цикла. По умолчанию ее размер составляет 1 Мбайт, но вы можете увеличить ее, указав начальный размер в файле образа при помощи флага компоновщика `/HEAP`. Тем не менее этот размер всего лишь является начальным — куча будет расширяться автоматически по мере необходимости. Также в файле образа можно задать исходный подтвержденный размер.

Куча по умолчанию может использоваться в программах явно или же косвенно некоторыми внутренними функциями Windows. Приложение может запросить кучу процесса по умолчанию, обращаясь с вызовом к Windows-функции `GetProcessHeap`. Процессы также могут создавать дополнительные закрытые кучи функцией `HeapCreate`. Когда необходимость в закрытой куче отпадает, процесс может вернуть виртуальное адресное пространство вызовом `HeapDestroy`. В каждом процессе поддерживается массив со всеми кучами, а поток может запросить их Windows-функцией `GetProcessHeaps`.

Процесс приложений UWP (Universal Windows Platform) включает как минимум три кучи. !

- Только что описанная куча процесса по умолчанию.
- Общая куча, используемая для передачи больших аргументов экземпляру сеанса Csrss.exe процесса; создается функцией `CsrClientConnectToServer` из Ntdll.dll, которая выполняется на ранней стадии инициализации процесса, выполняемой в Ntdll.dll. Дескриптор кучи доступен в глобальной переменной `CsrPortHeap` (в Ntdll.dll).
- Куча, созданная библиотекой времени выполнения Microsoft C. Ее дескриптор хранится в глобальной переменной `_crtheap` (в модуле `msvcrt`). Эта куча используется во внутренней реализации функциями выделения памяти C/C++ — такими, как `malloc`, `free`, оператор `new/delete` и т. д.

Куча может управлять выделением памяти либо в больших областях памяти, зарезервированных из диспетчера памяти функцией `VirtualAlloc`, либо из объекта файла, отображенного в память, в адресном пространстве процесса. Второй метод редко применяется на практике (и не предоставляется в Windows API), но подходит для сценариев, в которых содержимое блоков должно совместно использоваться двумя процессами или компонентами режима ядра и пользовательского режима. Драйвер GUI-подсистемы Win32 (Win32k.sys) использует такую кучу для совместного использования объектов GDI и USER с пользовательским режимом. Если куча строится на базе файла, отображенного в память, действуют определенные ограничения в отношении компонентов, которые могут вызывать функции кучи:

- Внутренние структуры кучи используют указатели и поэтому не допускают повторного отображения на другие адреса других процессов.
- Синхронизация между разными процессами или между компонентом ядра и пользовательским процессом не поддерживается функциями ядра.
- В случае совместного использования кучи между пользовательским режимом и режимом ядра отображение пользовательского режима должно быть доступно только для чтения, чтобы код пользовательского режима не мог повредить внутренние структуры кучи (что привело бы к системному фатальному сбою). Драйвер режима ядра также отвечает за то, чтобы никакие конфиденциальные данные не оказались в общей куче, чтобы не произошла утечка с выводом их в пользовательский режим.

Типы куч

До выхода Windows 10 и Server 2016 существовала только одна разновидность куч, которую мы будем называть *кучей NT*. Куча NT расширяется дополнительной интерфейсной прослойкой, которая в случае использования состоит из низкофрагментированной кучи LFH (Low-Fragmentation Heap).

В Windows 10 вводится новая разновидность кучи — *сегментная куча* (segment heap). Две разновидности кучи включают общие элементы, но имеют разную структуру и реализацию. По умолчанию сегментная куча используется всеми при-

ложениями UWP и некоторыми системными процессами, а куча NT используется всеми остальными процессами. Эту конфигурацию можно изменить в реестре, как описано в разделе «Сегментная куча» далее в этой главе.

Куча NT

Как показано на рис. 5.8, куча NT в пользовательском режиме состоит из двух прослоек: интерфейсной и внутренней (иногда называемой *ядром кучи*). Внутренняя прослойка обеспечивает основную функциональность и включает управление блоками внутри сегментов, управление сегментами, политиками для расширения кучи, подтверждения и отмены подтверждения памяти и управления большими блоками.

Рис. 5.8. Прослойки кучи NT в пользовательском режиме

Для куч пользовательского режима интерфейсная прослойка кучи может существовать поверх базовой функциональности. Windows поддерживает одну необязательную интерфейсную прослойку LFH, описанную в разделе «Низкофрагментированная куча».

Синхронизация кучи

По умолчанию диспетчер кучи поддерживает параллельный доступ из нескольких потоков. Тем не менее, если процесс является однопоточным или использует внешний механизм для синхронизации, он может приказать диспетчеру кучи отказаться от затрат на синхронизацию, указав флаг `HEAP_NO_SERIALIZE` либо при создании кучи, либо на уровне отдельных выделений памяти. Если синхронизация кучи разрешена, на каждую кучу создается одна блокировка, которая защищает все внутренние структуры кучи.

Процесс также может установить блокировку всей кучи и запретить всем потокам выполнение с кучей операций, требующих согласованности состояний между

вызовами. Например, перебор блоков кучи с использованием Windows-функции `HeapWalk` требует установления блокировки кучи, если нескольким потокам потребуется одновременно выполнить операции с кучей. Установление и снятие блокировки с кучи выполняется функциями `HeapLock` и `HeapUnlock` соответственно.

Низкофрагментированная куча

Многие приложения, работающие в Windows, относительно слабо используют память из кучи — обычно менее 1 Мбайт. В этом классе приложений политика оптимального размещения помогает поддерживать низкие объемы требуемой памяти для каждого процесса. Тем не менее эта стратегия плохо масштабируется для больших процессов и многопроцессорных машин. В таких случаях память, доступная для использования в куче, может сокращаться из-за фрагментации кучи.

В сценариях, в которых только некоторые размеры часто используются параллельно из разных потоков, запланированных для выполнения на разных процессорах, может пострадать быстродействие. Это объясняется тем, что разные процессоры могут пытаться изменить одну область памяти (например, начало резервного списка для этого конкретного размера) одновременно, что создаст значительную конкуренцию за соответствующую линию кэша.

LFH избегает фрагментации за счет управления выделенными блоками с заранее определенными диапазонами размеров блоков, называемых *гнездами* (buckets). Когда процесс выделяет память из кучи, LFH выбирает гнездо, которое отображается на наименьший блок, размер которого достаточен для необходимого размера. (Наименьший размер блока равен 8 байтам.)

Первое гнездо используется для выделения памяти в диапазонах от 1 до 8 байт, второе — для выделения от 9 до 16 байт и т. д., вплоть до 32-го гнезда, которое используется для выделения памяти от 249 до 256 байт. Далее идет 33-е гнездо, которое используется для выделения памяти от 257 до 272 байт и т. д. Наконец, 128-е гнездо (последнее) используется для выделения от 15 873 до 16 384 байт. Если размер выделяемого блока превышает 16 384 байт, то LFH просто передает его используемой внутренней прослойке кучи. В табл. 5.7 перечислены различные гнезда с указанием их гранулярности и диапазонов размеров.

Таблица 5.7. Гнезда LFH

Гнезда	Гранулярность	Диапазон
1–32	8	1–256
33–48	16	257–512
49–64	32	513–1024
65–80	64	1025–2048
81–96	128	2049–4096
97–112	256	4097–8192
113–128	512	8193–16 384

Для решения этих проблем LFH использует диспетчер ядра кучи и резервные кучи. Диспетчер кучи Windows реализует алгоритм автоматической настройки, который может включить LFH по умолчанию в некоторых ситуациях — например, при конкуренции за блокировку или присутствии популярных размеров выделяемых блоков, для которых включение LFH повышает быстродействие. Для больших куч значительный процент выделений часто группируется в относительно небольшом количестве гнезд. Стратегия выделения памяти, используемая LFH, направлена на оптимизацию использования таких ситуаций за счет эффективной обработки блоков одинакового размера.

Для решения проблем масштабируемости LFH расширяет часто используемые внутренние структуры до слотов, количество которых вдвое больше текущего количества процессоров на машине. Распределение потоков между слотами осуществляется компонентом LFH, называемым *диспетчером сходства* (affinity manager). В исходном состоянии LFH использует первый слот для выделения памяти из кучи; однако при обнаружении конкуренции при обращении к внутренним данным LFH переключает текущий поток на использование другого слота. Дальнейшая конкуренция распределяет потоки по большему количеству слотов. Управление слотами осуществляется по отдельности для каждого гнезда, чтобы улучшить локализацию обращений и минимизировать общие затраты памяти.

Даже при включении кучи LFH менее частые размеры выделяемых блоков могут продолжать использование основных функций кучи для выделения памяти, тогда как для самых популярных категорий память будет выделяться из LFH. После того как поддержка LFH будет включена для конкретной кучи, отключить ее уже не удастся. API-функция `HeapSetInformation` с классом `HeapCompatibilityInformation`, которая могла отключать прослойку LFH в Windows 7 и более ранних версиях Windows, теперь игнорируется.

Сегментная куча

На рис. 5.9 показана архитектура сегментной кучи, появившейся в Windows 10.

Прослойка, которая непосредственно управляет выделением памяти, зависит от размера выделяемых блоков:

- Для малых размеров (не более 16 368 байт) используется система выделения памяти LFH, но только если размер определяется как часто используемый. Логика похожа на логику интерфейсной прослойки LFH кучи NT. Если механизм LFH еще не вступил в дело, вместо этого используется механизм выделения блоков переменного размера (VS).
- Для размеров, меньших либо равных 128 Кбайт (не обслуживаемых LFH), используется механизм выделения памяти VS. Механизмы выделения памяти VS и LFH используют внутреннюю прослойку для создания нужных подсегментов кучи по мере надобности.

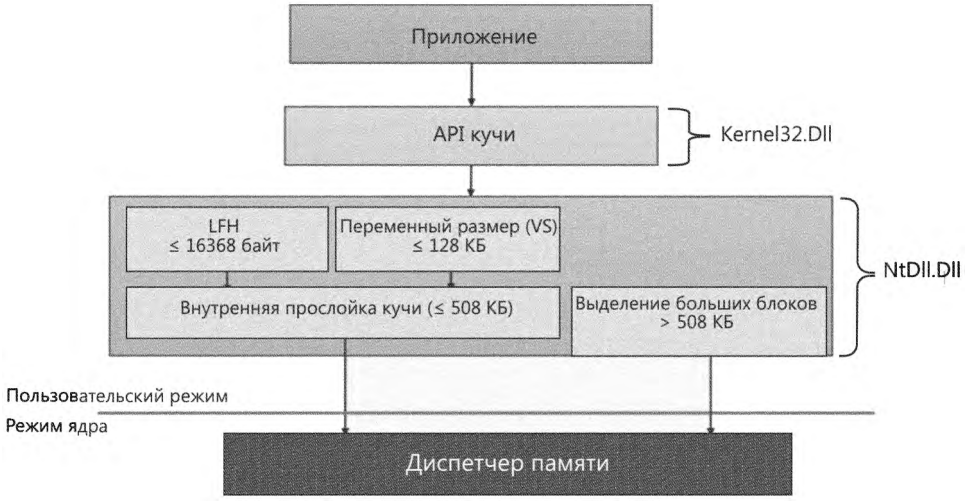


Рис. 5.9. Сегментная куча

- Блоки размером 128 Кбайт, а также размером менее или равным 508 Кбайт, обслуживаются напрямую внутренней прослойкой кучи.
- Блоки размером более 508 Кбайт обслуживаются прямым вызовом диспетчера памяти (`VirtualAlloc`) — они достаточно велики, чтобы выделение памяти со стандартной гранулярностью 64 Кбайт (и округлением до ближайшего размера страницы) считалось достаточно эффективным.

Краткое сравнение двух реализаций кучи:

- В некоторых сценариях сегментная куча работает чуть медленнее кучи NT. Тем не менее, по всей вероятности, в будущих версиях Windows она не будет уступать куче NT.
- Сегментная куча обладает меньшими затратами памяти для хранения метаданных, из-за чего она лучше подходит для устройств с низким объемом памяти (например, телефонов).
- Метаданные сегментной кучи отделяются от фактических данных, тогда как метаданные кучи NT переплетаются с самими данными. Сегментная куча обладает большей надежностью, потому что становится сложнее получить метаданные выделенной области только по адресу блока.
- Сегментная куча может использоваться только для кучи с возможностью увеличения размера. Она не может использоваться с файлом, отображенным в память, полученным от пользователя. При попытке создания такой сегментной кучи вместо этого будет создана куча NT.
- Обе кучи поддерживают выделение памяти по типу LFH, но их внутренние реализации полностью различны. Сегментная куча использует более эффективную реализацию в отношении потребления памяти и быстродействия.

Как упоминалось ранее, приложения UWP по умолчанию используют сегментные кучи. Это объясняется в основном их пониженными затратами памяти, лучше подходящими для устройств с малыми объемами памяти. Они также используются с некоторыми системными процессами, которые определяются по имени исполняемого образа: csrss.exe, lsass.exe, runtimebroker.exe, services.exe, smss.exe и svchost.exe.

Сегментная куча не используется по умолчанию для настольных приложений из-за некоторых проблем совместимости, которые могут повлиять на работу существующих приложений. Впрочем, не исключено, что в будущих версиях она будет использоваться по умолчанию. Чтобы разрешить или запретить использование сегментной кучи для конкретного исполняемого файла, задайте в параметрах Image File Execution Options значение с именем `FrontEndHeapDebugOptions` (тип `DWORD`):

- Бит 2 (4) для запрета сегментной кучи.
- Бит 3 (8) для разрешения сегментной кучи.

Также сегментную кучу можно разрешить или запретить глобально — добавьте параметр с именем `Enabled` (тип `DWORD`) в раздел реестра HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Segment Heap. Нулевое значение отключает сегментную кучу, а ненулевое разрешает ее использование.

ЭКСПЕРИМЕНТ: ПРОСМОТР ОСНОВНОЙ ИНФОРМАЦИИ О КУЧАХ

В этом эксперименте мы рассмотрим некоторые кучи процесса UWP.

1. Запустите в системе Windows 10 программу Калькулятор. (Щелкните на кнопке Пуск (Start) и введите строку Калькулятор.)
2. Калькулятор в Windows 10 был преобразован в приложение UWP (Calculator.exe). Запустите WinDbg и присоедините отладчик к процессу калькулятора.
3. После присоединения WinDbg передает управление в процесс. Введите команду `!heap`, чтобы получить краткую сводку куч процесса:

```
0:033> !heap
        Heap Address      NT/Segment Heap

          2531eb90000         Segment Heap
          2531e980000              NT Heap
          2531eb10000         Segment Heap
          25320a40000         Segment Heap
          253215a0000         Segment Heap
          253214f0000         Segment Heap
          2531eb70000         Segment Heap
          25326920000         Segment Heap
          253215d0000              NT Heap
```

4. Обратите внимание на разные кучи с дескрипторами и типами (segment или NT). На первом месте стоит куча процесса по умолчанию. Так как она может увеличиваться и не использует никакие заранее существующие блоки памяти, она создается в формате сегментной кучи. Вторая куча используется

с определенным пользователем блоком памяти (см. ранее в разделе «Кучи процесса»). Поскольку эта функциональность в настоящее время не поддерживается сегментной кучей, она создается как куча NT.

5. Кучей NT управляет структура `NtDll!_HEAP`. Рассмотрим эту структуру для второй кучи:

```
0:033> dt ntdll!_heap 2531e980000
   +0x000 Segment          : _HEAP_SEGMENT
   +0x000 Entry            : _HEAP_ENTRY
   +0x010 SegmentSignature : 0xffeeffee
   +0x014 SegmentFlags     : 1
   +0x018 SegmentListEntry : _LIST_ENTRY [ 0x00000253'1e980120 -
0x00000253'1e980120 ]
   +0x028 Heap             : 0x00000253'1e980000 _HEAP
   +0x030 BaseAddress      : 0x00000253'1e980000 Void
   +0x038 NumberOfPages    : 0x10
   +0x040 FirstEntry       : 0x00000253'1e980720 _HEAP_ENTRY
   +0x048 LastValidEntry   : 0x00000253'1e990000 _HEAP_ENTRY
   +0x050 NumberOfUnCommittedPages : 0xf
   +0x054 NumberOfUnCommittedRanges : 1
   +0x058 SegmentAllocatorBackTraceIndex : 0
+0x05a Reserved         : 0
   +0x060 UCRSegmentList   : _LIST_ENTRY [ 0x00000253'1e980fe0 -
0x00000253'1e980fe0 ]
   +0x070 Flags            : 0x8000
   +0x074 ForceFlags       : 0
   +0x078 CompatibilityFlags : 0
   +0x07c EncodeFlagMask   : 0x100000
   +0x080 Encoding         : _HEAP_ENTRY
   +0x090 Interceptor      : 0
   +0x094 VirtualMemoryThreshold : 0xff00
   +0x098 Signature        : 0xeeffeeff
   +0x0a0 SegmentReserve   : 0x100000
   +0x0a8 SegmentCommit    : 0x2000
   +0x0b0 DeCommitFreeBlockThreshold : 0x100
   +0x0b8 DeCommitTotalFreeThreshold : 0x1000
   +0x0c0 TotalFreeSize    : 0x8a
   +0x0c8 MaximumAllocationSize : 0x00007fff'fffdefff
   +0x0d0 ProcessHeapsListIndex : 2
   ...
   +0x178 FrontEndHeap     : (null)
   +0x180 FrontHeapLockCount : 0
   +0x182 FrontEndHeapType : 0 ''
   +0x183 RequestedFrontEndHeapType : 0 ''
   +0x188 FrontEndHeapUsageData : (null)
   +0x190 FrontEndHeapMaximumIndex : 0
   +0x192 FrontEndHeapStatusBitmap : [129]  ""
   +0x218 Counters         : _HEAP_COUNTERS
   +0x290 TuningParameters : _HEAP_TUNING_PARAMETERS
```

6. Обратите внимание на поле FrontEndHeap — признак существования интерфейсной прослойки. В приведенном выводе оно содержит null; это означает, что прослойки нет. Значение, отличное от `null`, указывает на существование интерфейсной прослойки LFH (как единственной определенной).

7. Сегментная куча определяется структурой NtDll!_SEGMENT_HEAP. Куча процесса по умолчанию:

```
0:033> dt ntdll!_segment_heap 2531eb90000
   +0x000 TotalReservedPages : 0x815
   +0x008 TotalCommittedPages : 0x6ac
   +0x010 Signature          : 0xddeeddee
   +0x014 GlobalFlags        : 0
   +0x018 FreeCommittedPages : 0
   +0x020 Interceptor        : 0
   +0x024 ProcessHeapListIndex : 1
   +0x026 GlobalLockCount    : 0
   +0x028 GlobalLockOwner    : 0
   +0x030 LargeMetadataLock : _RTL_SRWLOCK
   +0x038 LargeAllocMetadata : _RTL_RB_TREE
   +0x048 LargeReservedPages : 0
   +0x050 LargeCommittedPages : 0
   +0x058 SegmentAllocatorLock : _RTL_SRWLOCK
   +0x060 SegmentListHead    : _LIST_ENTRY [ 0x00000253'1ec00000 -
0x00000253'28a00000 ]
   +0x070 SegmentCount       : 8
   +0x078 FreePageRanges     : _RTL_RB_TREE
   +0x088 StackTraceInitVar : _RTL_RUN_ONCE
   +0x090 ContextExtendLock : _RTL_SRWLOCK
   +0x098 AllocatedBase      : 0x00000253'1eb93200  ""
   +0x0a0 UncommittedBase    : 0x00000253'1eb94000  "--- memory read error at
address 0x00000253'1eb94000 ---"
   +0x0a8 ReservedLimit      : 0x00000253'1eba5000  "--- memory read error at
address 0x00000253'1eba5000 ---"
   +0x0b0 VsContext          : _HEAP_VS_CONTEXT
   +0x120 LfhContext         : _HEAP_LFH_CONTEXT
```

8. Обратите внимание на поле Signature: по нему можно различать два типа куч.

9. Обратите внимание на поле SegmentSignature структуры _HEAP: оно имеет такое же смещение (0x10). По этому факту такие функции, как RtlAllocateHeap, определяют, какую реализацию следует выбрать на основании одного лишь дескриптора кучи (адреса).

10. Два последних поля в структуре _SEGMENT_HEAP содержат информацию механизмов выделения памяти VS и LFH.

11. Чтобы получить подробную информацию о каждой куче, введите команду !heap -s:

```
0:033> !heap -s
                                    Process     Total      Total
                           Global      Heap Reserved  Committed
    Heap Address  Signature   Flags     List    Bytes      Bytes
                                     Index      (K)        (K)
      2531eb90000   ddeeddee        0        1     8276       6832
      2531eb10000   ddeeddee        0        3     1108        868
      25320a40000   ddeeddee        0        4     1108         16
      253215a0000   ddeeddee        0        5     1108         20
      253214f0000   ddeeddee        0        6     3156        816
```

```
    2531eb70000   ddeeddee        0        7     1108         24
    25326920000   ddeeddee        0        8     1108         32

************************************************************************
*************
                        NT HEAP STATS BELOW
************************************************************************
*************
LFH Key                   : 0xd7b666e8f56a4b98
Termination on corruption : ENABLED
Affinity manager status:
   - Virtual affinity limit 8
   - Current entries in use 0
   - Statistics:  Swaps=0, Resets=0, Allocs=0

          Heap     Flags   Reserv  Commit  Virt   Free  List   UCR  Virt
Lock  Fast
                            (k)     (k)    (k)     (k) length      blocks
cont. heap
------------------------------------------------------------------------
----------
000002531e980000 00008000      64      4     64      2     1     1    0
0
00000253215d0000 00000001      16     16     16     10     1     1    0
N/A
------------------------------------------------------------------------
----------
```

12. Обратите внимание на первую часть вывода: в ней содержится расширенная информация о сегментных кучах (если они есть). Во второй части содержится расширенная информация о кучах NT в процессе.

Команда отладчика `!heap` предоставляет различные средства для просмотра, анализа и поиска в кучах. За дополнительной информацией обращайтесь к документации «Debugger Tools for Windows».

Безопасность кучи в Windows

По мере своего развития диспетчер кучи стал играть более важную роль в раннем обнаружении ошибок использования кучи и сокращения эффекта потенциальных эксплойтов, основанных на использовании кучи. Эти меры существуют для того, чтобы снизить риск от потенциальных уязвимостей в приложениях. В реализациях как кучи NT, так и сегментной кучи присутствуют механизмы, сокращающие опасность эксплойтов в памяти.

Метаданные, используемые кучами для внутреннего управления, упакованы с высокой степенью рандомизации; это делается для того, чтобы усложнить потенциальную модификацию внутренних структур для предотвращения сбоев или маскировки попыток атаки. Механизм проверки целостности проверяет эти блоки для обнаружения простых повреждений (таких, как переполнения буфера). Наконец, куча использует небольшую степень рандомизации базовых адресов или

дескрипторов. Используя API `HeapSetInformation` с классом `HeapEnableTerminationOnCorruption`, процессы могут согласиться на автоматическое завершение в случае обнаружения повреждений, чтобы избежать выполнения неизвестного кода.

Из-за рандомизации метаданных блоков попытка просто вывести дамп заголовка блока области памяти в отладчике особой пользы не принесет. Например, найти в обычном дампе данные о размере блока и о том, занят он в настоящий момент или нет, не так уж просто. То же можно сказать о блоках LFH. Они хранят в заголовке метаданные другого типа, также частично рандомизированные. Для вывода дампа этой информации используется команда `!heap -i` в отладчике; она читает поля метаданных из блока и помечает все несоответствия контрольных сумм или свободных списков, если таковые будут найдены. Команда работает как для LFH, так и для блоков обычных куч. В выходных данных указывается общий размер блоков, запрошенный пользователем размер, сегмент-владелец блока и частичная контрольная сумма заголовка, как показано в следующем примере. Так как алгоритм рандомизации использует гранулярность кучи, команда `!heap -i` должна использоваться только в правильном контексте кучи, содержащей блок. В этом примере дескриптор кучи равен 0x001a0000. Если бы текущий контекст кучи был другим, то декодирование заголовка прошло бы неверно. Чтобы назначить правильный контекст, сначала необходимо выполнить ту же команду `!heap -i` с дескриптором кучи в качестве аргумента.

```
0:004> !heap -i 000001f72a5e0000
Heap context set to the heap 0x000001f72a5e0000

0:004> !heap -i 000001f72a5eb180
Detailed information for block entry 000001f72a5eb180
Assumed heap       : 0x000001f72a5e0000 (Use !heap -i NewHeapHandle to change)
Header content     : 0x2FB544DC 0x1000021F (decoded : 0x7F01007E 0x10000048)
Owning segment     : 0x000001f72a5e0000 (offset 0)
Block flags        : 0x1 (busy )
Total block size   : 0x7e units (0x7e0 bytes)
Requested size     : 0x7d0 bytes (unused 0x10 bytes)
Previous block size: 0x48 units (0x480 bytes)
Block CRC          : OK - 0x7f
Previous block     : 0x000001f72a5ead00
Next block         : 0x000001f72a5eb960
```

Средства безопасности, специфические для сегментной кучи

В реализации сегментной кучи используются многие механизмы безопасности, усложняющие повреждение памяти или внедрение кода в ходе атак. Некоторые из этих механизмов:

- **Быстрое прекращение при повреждении узлов связанного списка.** Сегментная куча использует связанные списки для отслеживания сегментов и подсегментов. Как и в случае с кучей NT, при вставке и удалении узлов добавляются проверки, предотвращающие произвольную запись в память из-за поврежденных узлов списка. При обнаружении поврежденного узла процесс завершается вызовом `RtlFailFast`.

- **Быстрое прекращение при повреждении узлов красно-черного дерева (RB).** Сегментная куча использует деревья RB для отслеживания свободной памяти и выделения памяти VS. Функции вставки и удаления узлов проверяют задействованные узлы, а в случае повреждения активизируют механизм быстрого прекращения.
- **Декодирование указателей на функции.** Некоторые аспекты сегментной кучи позволяют использовать обратные вызовы (в структурах `VsContext` и `LfhContext`, части структуры `_SEGMENT_HEAP`). Атакующий может переопределить эти обратные вызовы ссылками на свой код. Однако указатели на функции кодируются с использованием функции XOR с внутренним случайным ключом и адресом контекста; ни то ни другое невозможно предугадать заранее.
- **Страницы защиты.** При выделении подсегментов VS и LFH и больших блоков в конце добавляется страница защиты. Она помогает обнаруживать переполнения и повреждения близлежащих данных. За дополнительной информацией о страницах защиты обращайтесь к разделу «Стеки» этой главы.

Средства отладки кучи

Диспетчер кучи включает ряд средств, упрощающих выявление ошибок, с использованием следующих настроек:

- **Включение конечной проверки.** В конце каждого блока располагается сигнатура, которая проверяется при освобождении блока. Если переполнение буфера приводит к полному или частичному уничтожению сигнатуры, диспетчер кучи сообщает об ошибке.
- **Включение проверки свободных блоков.** Свободный блок заполняется последовательностью байтов, которая проверяется в различные моменты, когда диспетчеру кучи требуется обратиться к блоку (например, при удалении из списка свободной памяти для удовлетворения запросов на выделение памяти). Если процесс продолжает запись в блок после его освобождения, диспетчер кучи обнаруживает изменения в сигнатуре и сообщает об ошибке.
- **Проверка параметров.** Функция подразумевает обширную проверку параметров, передаваемых функциям кучи.
- **Проверка кучи.** Вся куча проверяется при каждом вызове.
- **Пометка кучи и поддержка трассировки стеков.** Функция поддерживает назначение тегов для выделения памяти и/или сохраняет трассировку стеков пользовательского режима для вызовов, чтобы сократить набор возможных причин ошибок кучи.

Первые три средства включаются по умолчанию, если загрузчик обнаруживает, что процесс запущен под контролем отладчика. (Отладчик может переопределить это поведение и отключить эти функции.) Вы можете определить средства отладки кучи для исполняемого образа, устанавливая различные флаги отладки в заголовке образа

при помощи программы Gflags. (См. описание следующего эксперимента и раздел «Глобальные флаги Windows» главы 2 в части 2.) Также для включения средств отладки кучи можно воспользоваться командой `!heap` в стандартных отладчиках Windows. (За дополнительной информацией обращайтесь к справке отладчика.)

Включение средств отладки кучи влияет на все кучи в процессе. Кроме того, при включении каких-либо средств отладки куча LFH будет автоматически отключена, и будет использоваться ядро кучи (с включением необходимых средств отладки). LFH также не используется для куч, не поддерживающих расширение (из-за дополнительных затрат, добавляемых к существующим структурам кучи), или для куч, не поддерживающих сериализацию.

Pageheap

Так как средства конечной проверки и проверки свободных блоков могут выявить повреждения, произошедшие до обнаружения проблемы, существует дополнительная функция отладки кучи *Pageheap*. Pageheap направляет вызовы функций кучи (все или часть) другому диспетчеру кучи.

Для включения Pageheap можно воспользоваться программой Gflags (входящей в набор средств отладки для Windows). В этом случае диспетчер кучи помещает выделяемые блоки в конец страниц и резервирует непосредственно следующую страницу. Так как зарезервированные страницы недоступны, любые переполнения буфера приводят к нарушению прав доступа, а это упрощает обнаружение кода-нарушителя. Также Pageheap позволяет размещать блоки в начале страниц с резервированием предыдущей страницы для выявления опустошения буфера (относительно редкое явление). Pageheap также может защищать свободные страницы от любых обращений, чтобы обнаруживать обращения к блокам кучи после их освобождения.

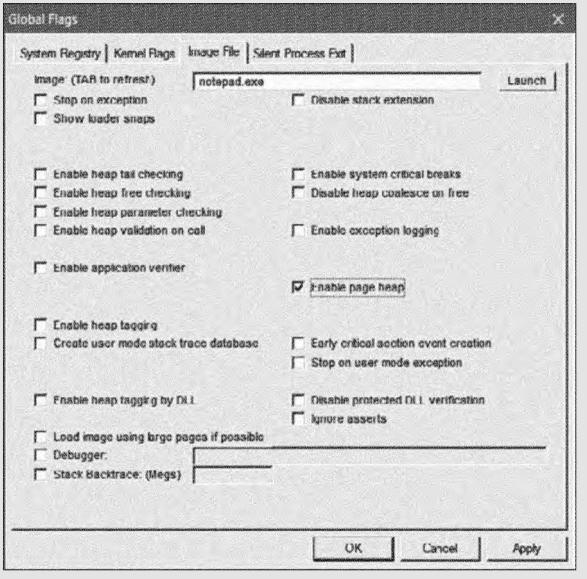
Обратите внимание: Pageheap может привести к исчерпанию адресного пространства (в 32-разрядных процессах) из-за значительных дополнительных затрат ресурсов при выделении малых блоков. Кроме того, может пострадать быстродействие из-за увеличения количества обращений к обнуленным страницам, потери локализации и увеличения дополнительных затрат, обусловленных частыми вызовами для проверки структур данных. Процесс может сократить эти последствия, указав, что функциональность Pageheap должна использоваться только для блоков определенного размера, диапазонов адресов и/или инициирующих DLL-библиотек.

ЭКСПЕРИМЕНТ: ИСПОЛЬЗОВАНИЕ PAGEHEAP

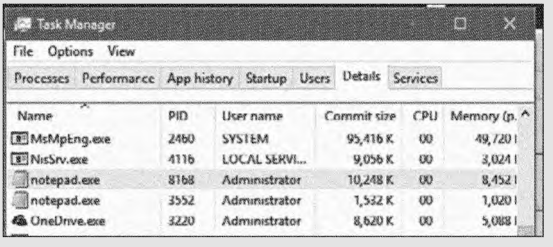
В этом эксперименте мы включим функцию Pageheap для Notepad.exe и понаблюдаем за ее эффектом.

1. Запустите Notepad.exe.
2. Откройте диспетчер задач, перейдите на вкладку Подробности (Details) и включите в выходные данные столбец Commit Size.

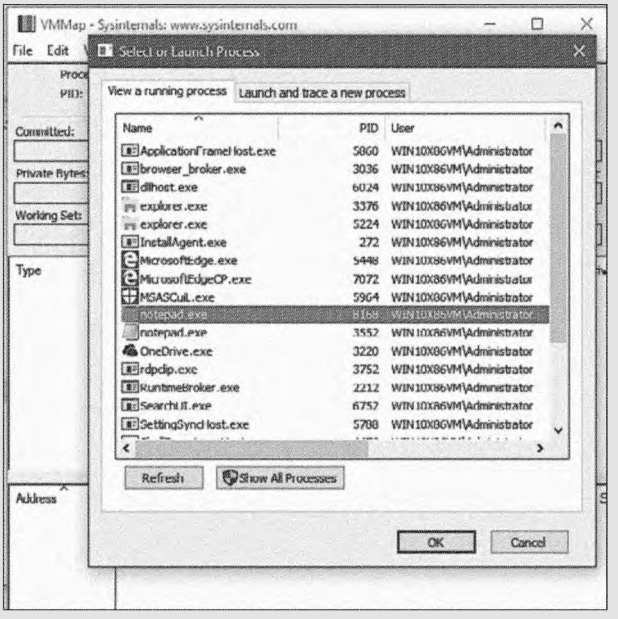
3. Обратите внимание на размер выделенной памяти для только что запущенного экземпляра Notepad.exe.
4. Запустите программу Gflags.exe из папки, в которой были установлены средства отладки для Windows (требует повышения прав).
5. Перейдите на вкладку Image File.
6. В текстовом поле Image введите строку notepad.exe. Нажмите клавишу Tab.
7. Установите флажок Enable Page Heap. Диалоговое окно должно выглядеть так:



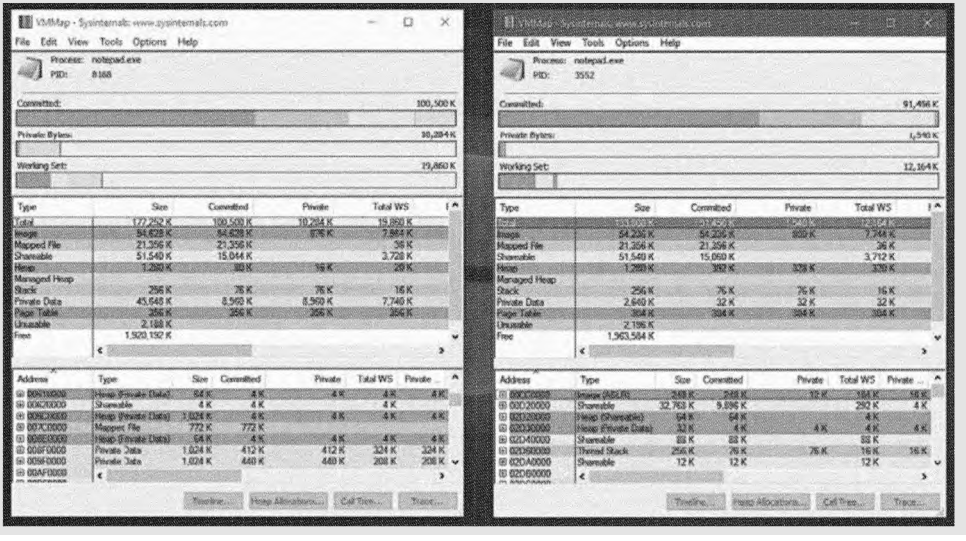
8. Щелкните на кнопке Apply.
9. Запустите другой экземпляр Notepad. (Не закрывайте первый экземпляр!)
10. В диспетчере задач сравните размер выделенной памяти для обоих экземпляров. Заметьте, что второй экземпляр обладает намного большим размером выделенной памяти, хотя оба экземпляра представляют собой пустые процессы; это обусловлено лишним выделением памяти, обеспечиваемым Pageheap. Следующий снимок экрана взят из 32-разрядной версии Windows 10:



11. Чтобы получить лучшее представление о выделении дополнительной памяти, воспользуйтесь программой VMMap из пакета Sysinternal. Пока процессы продолжают работать, откройте VMMap.exe и выберите экземпляр Notepad, использующий Pageheap:

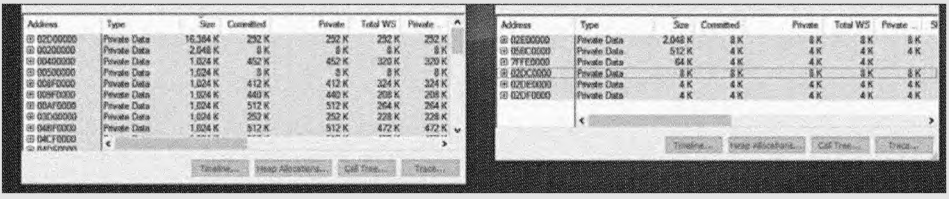


12. Откройте другой экземпляр VMMap и выберите другой экземпляр Notepad. Разместите окна рядом друг с другом, чтобы видеть их одновременно:

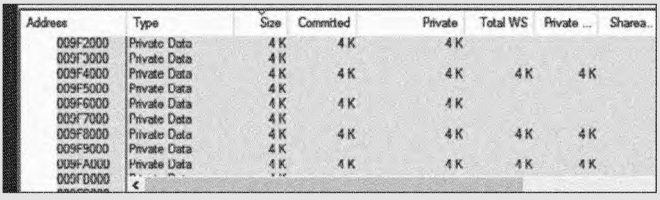


13. Обратите внимание: различия в размере выделенной памяти хорошо видны в разделе Private Data (желтый).

14. Щелкните на строке Private Data в середине обоих экземпляров VMMap, чтобы просмотреть их содержимое в нижней части (на иллюстрации данные отсортированы по размеру):



15. Очевидно, что на левом снимке (с Pagehead) Notepad потребляет больше памяти. Откройте один из 1024-килобайтных фрагментов. Это выглядит примерно так:



Address	Type	Size	Committed	Private	Total WS	Private ...	Sharea...
009F2000	Private Data	4 K	4 K	4 K			
009F3000	Private Data	4 K					
009F4000	Private Data	4 K	4 K	4 K	4 K	4 K	
009F5000	Private Data	4 K					
009F6000	Private Data	4 K	4 K	4 K			
009F7000	Private Data	4 K					
009F8000	Private Data	4 K	4 K	4 K	4 K	4 K	
009F9000	Private Data	4 K					
009FA000	Private Data	4 K	4 K	4 K	4 K	4 K	
009FB000							

16. Между выделенными страницами хорошо видны зарезервированные страницы, которые обеспечивают перехват нарушений границы буферов благодаря Pageheap. Снимите флажок Enable Page Heap в Gflags и щелкните на кнопке Apply, чтобы будущие экземпляры Notepad выполнялись без Pageheap.

За дополнительной информацией о Pageheap обращайтесь к справочному файлу «Debugging Tools for Windows».

Отказоустойчивая куча

Компания Microsoft выделяет повреждение метаданных кучи как одну из самых частых причин сбоев приложений. В Windows включена функция *отказоустойчивой кучи* (FTH, Fault-Tolerant Heap) для компенсации этих проблем и предоставления расширенной информации разработчикам приложений. Реализация FTH содержится в двух основных компонентах:

- Компонент обнаружения (сервер FTH).
- Компонент защитных мер (клиент FTH).

Компонент обнаружения представляет собой DLL-библиотеку с именем **Fthsvc.dll**, загружаемую службой Центра безопасности Windows (**Wscsvc.dll**), которая в свою

очередь выполняется в одном из процессов общих служб под локальной учетной записью. Он получает уведомления о сбоях приложений от службы отчетов об ошибках Windows (WER).

Допустим, у приложения произошел сбой в Ntdll.dll с кодом ошибки, обозначающим либо нарушение прав доступа, либо исключение повреждения кучи. Если приложение еще не входит в список отслеживаемых приложений службы FTH, служба создает для приложения «билет» для хранения данных FTH. Если в дальнейшем в приложении происходят сбои чаще четырех раз в час, служба FTH настраивает приложение для использования клиента FTH в будущем.

Клиент FTH представляет собой оболочку совместимости приложения. Этот механизм использовался со времен Windows XP, чтобы приложения, зависящие от конкретного поведения старых версий Windows, могли выполняться в более поздних системах. В данном случае механизм оболочки совместимости перехватывает вызовы функций кучи и перенаправляет их в собственный код. Код FTH реализует многочисленные средства устранения проблем, благодаря которым приложение может попытаться продолжить работу несмотря на ошибки, связанные с кучей.

Например, чтобы защититься от малых ошибок переполнения буфера, FTH добавляет к каждому выделяемому блоку 8 байт и зарезервированную область FRP. Для решения распространенной проблемы с обращением к блоку кучи после его освобождения вызовы HeapFree реализуются только после задержки. «Освобожденные» блоки помещаются в список и освобождаются только тогда, когда общий размер блоков в списке превысит 4 Мбайт. Попытки освобождения блоков, не являющихся частью кучи — или частью кучи, определяемой аргументом дескриптора кучи при вызове `HeapFree`, — попросту игнорируются. Кроме того, никакие блоки не будут реально освобождаться после вызова `exit` или `RtlExitUserProcess`.

Сервер FTH продолжает отслеживать частоту ошибок приложения после установления защитных мер. Если частота появления ошибок не улучшится, защитные меры снимаются.

За работой отказоустойчивой кучи можно проследить в Event Viewer. Выполните следующие действия:

1. Введите в диалоговом окне Run команду `eventvwr.msc`.
2. На левой панели выберите Просмотр событий (Event Viewer), а затем последовательно выберите узлы Журналы приложений и служб (Applications and Services Logs), Microsoft, Windows и Fault-Tolerant Heap.
3. Щелкните в строке Operational log.
4. FTH можно полностью отключить в реестре: откройте раздел HKLM\Software\Microsoft\FTH и задайте параметру Enabled значение 0.

Этот же раздел содержит различные настройки FTH — например, уже упоминавшуюся выше задержку и список исключений для исполняемых файлов (который

по умолчанию включает такие системные процессы, как smss.exe, csrss.exe, wininit.exe, services.exe, winlogon.exe и taskhost.exe). Также включается список правил (параметр `RuleList`) с перечнем модулей и типов отслеживаемых исключений (а также некоторых флагов) для активизации FTH. По умолчанию включается одно правило для проблем с кучей в Ntdll.dll типа `STATUS_ACCESS_VIOLATION` (0xc0000005).

Обычно FTH не используется со службами и блокируется в серверных системах Windows, чтобы не снижать производительность. Системный администратор может вручную применить оболочку совместимости к исполняемому файлу приложения или службы при помощи Application Compatibility Toolkit.

Структуры виртуального адресного пространства

В данном разделе сначала рассматриваются компоненты пользовательского и системного адресных пространств, а затем — конкретные структуры данных 32-разрядных (x86 и ARM) и 64-разрядных (x64) систем. Эта информация поможет вам понять ограничения, накладываемые на виртуальную память процесса и системы на обеих платформах.

На виртуальное адресное пространство в Windows отображаются три основных типа данных:

- **Закрытый код и данные каждого процесса.** В главе 1 уже объяснялось, что у каждого процесса есть закрытое адресное пространство, недоступное другим процессам. То есть виртуальный адрес всегда вычисляется в контексте текущего процесса и не может ссылаться на адрес, определенный каким-нибудь другим процессом. А значит, потоки внутри процессов не могут получить доступ к виртуальным адресам за пределами этого закрытого адресного пространства. Даже общая память не является исключением из этого правила, поскольку области общей памяти отображаются на каждый из участвующих в ее использовании процессов и поэтому доступны каждому такому процессу по его адресам. Аналогичным образом функции памяти, работающие для нескольких процессов (`ReadProcessMemory` и `WriteProcessMemory`), действуют за счет выполнения кода режима ядра в контексте целевого процесса. Описания виртуального адресного пространства процесса, называемые *таблицами страниц* (page tables), рассматриваются в разделе «Преобразование адресов». Каждый процесс использует собственный набор таблиц страниц. Эти таблицы хранятся в страницах, доступных только в режиме ядра, чтобы программные потоки процесса, выполняемые в пользовательском режиме, не могли изменить структуру собственного адресного пространства.
- **Код и данные уровня сеанса.** Пространство сеанса (session space) содержит информацию, общую для каждого сеанса. (Сеансы рассматриваются в главе 2.) Сеанс состоит из процессов и других системных объектов (таких, как рабочие станции, рабочие столы и окна), относящихся к сеансу входа в систему отдель-

ного пользователя. У каждого сеанса есть относящаяся только к нему область выгружаемого пула, используемая частью подсистемы Windows режима ядра (Win32k.sys) и предназначенная для выделения принадлежащих только данному сеансу структур GUI-данных. Кроме того, у каждого сеанса есть собственная копия процесса подсистемы Windows (Csrss.exe) и процесса входа в систему (Winlogon.exe). Процесс диспетчера сеансов (Smss.exe) отвечает за создание новых сеансов, что включает загрузку принадлежащей сеансу копии Win32k.sys, создание пространства имен диспетчера объектов для конкретного сеанса и создание сеансовых экземпляров процессов Csrss.exe и Winlogon.exe. Для виртуализации сеансов все структуры данных уровня сеанса отображаются на ту область системного пространства, которая называется *пространством сеанса*. При создании процесса этот диапазон адресов отображается на страницы, связанные с сеансом, которому принадлежит процесс.

- **Код и данные общесистемного уровня.** Системное пространство содержит глобальный код операционной системы и структуры данных, видимые коду, выполняемому в режиме ядра, — независимо от того, какой именно процесс выполняется в данный момент. В системное пространство входят следующие компоненты:
  - **Системный код** содержит образ операционной системы, HAL и драйверы устройств, используемые для загрузки системы.
  - **Невыгружаемый пул** — невыгружаемая куча системной памяти.
  - **Выгружаемый пул** — выгружаемая куча системной памяти.
  - **Системный кэш** — это виртуальное адресное пространство, используемое для отображения файлов, открытых в системном кэше. (Подробности можно найти в главе 11.)
  - **Записи системной таблицы страниц.** Пул системных PTE-записей используется для отображения таких системных страниц, как пространство ввода/вывода, стеки ядра и списки дескрипторов памяти. Количество доступных системных PTE-записей можно узнать с помощью счетчика Memory: Free System Page Table Entries (Память: свободных элементов таблицы страниц) монитора производительности.
  - **Списки рабочих наборов системы.** Эти структуры данных описывают три системных рабочих набора (рабочий набор системного кэша, рабочий набор выгружаемого пула и рабочий набор системных PTE-записей).
  - **Системные отображаемые представления** служат для отображения загружаемой части подсистемы Windows, выполняемой в режиме ядра (Win32k.sys), а также для используемых этой подсистемой графических драйверов, работающих в режиме ядра. (Дополнительные сведения о Win32k.sys можно найти в главе 2.)
  - **Гиперпространство** — специальная область для отображения списка рабочего набора процесса и других данных, относящихся к каждому процессу,

к которым не должно быть доступа из произвольного процесса. Гиперпространство также предназначено для временного отображения физических страниц в системном пространстве. Одним из примеров такого отображения может служить аннулирование записей таблицы страниц в таблицах страниц процесса, не являющегося текущим (например, при удалении страницы из резервного списка).

- **Информация аварийного дампа** — пространство, зарезервированное для записи информации о состоянии системы на момент ее отказа.
- **Область, используемая HAL**, — системная память, зарезервированная для структур уровня аппаратных абстракций (HAL).

А теперь, после описания основных компонентов имеющегося в Windows виртуального адресного пространства, давайте перейдем к рассмотрению конкретных структур на платформах x86, ARM и x64.

Структура адресных пространств x86

По умолчанию каждый пользовательский процесс в 32-разрядных версиях Windows имеет закрытое адресное пространство размером 2 Гбайт (а операционная система забирает оставшиеся 2 Гбайт). Однако на платформе x86 система с помощью загрузочного BCD-параметра `increaseuserva` может быть настроена на размер пользовательского адресного пространства вплоть до 3 Гбайт. Два возможных варианта структур адресного пространства показаны на рис. 5.10.

Возможность выхода 32-разрядного процесса за пределы 2 Гбайт была введена для обеспечения потребностей 32-разрядных приложений хранения в памяти большего объема данных, чем в адресном пространстве размером 2 Гбайт. Разумеется, 64-разрядные системы предоставляют намного больший размер адресного пространства.

Чтобы процесс вышел за рамки адресного пространства размером 2 Гбайт, у файла образа в заголовке образа должен быть установлен флаг `IMAGE_FILE_LARGE_ADDRESS_AWARE` (кроме глобального параметра `increaseuserva`). В противном случае Windows зарезервирует дополнительное адресное пространство для этого процесса таким образом, что приложение не увидит виртуального адресного пространства с адресом, большим чем 0x7FFFFFFF. Доступ к дополнительной виртуальной памяти является выборочным, поскольку некоторые приложения предполагают, что им выделяется не более 2 Гбайт адресного пространства. Поскольку старший разряд указателя, ссылающегося на адрес ниже 2 Гбайт, всегда содержит 0 (для обращения к 2-гигабайтному адресному пространству необходим 31 бит), такие приложения будут использовать этот старший разряд в указателях как флаг для своих данных — конечно же, сбрасывая его перед обращением к данным. При работе в адресном пространстве размером 3 Гбайт это приведет к непреднамеренному усечению указателей со значениями, превышающими 2 Гбайт, вызывая тем самым ошибки приложения, включая и возможное повреждение данных. Вы можете установить этот флаг при

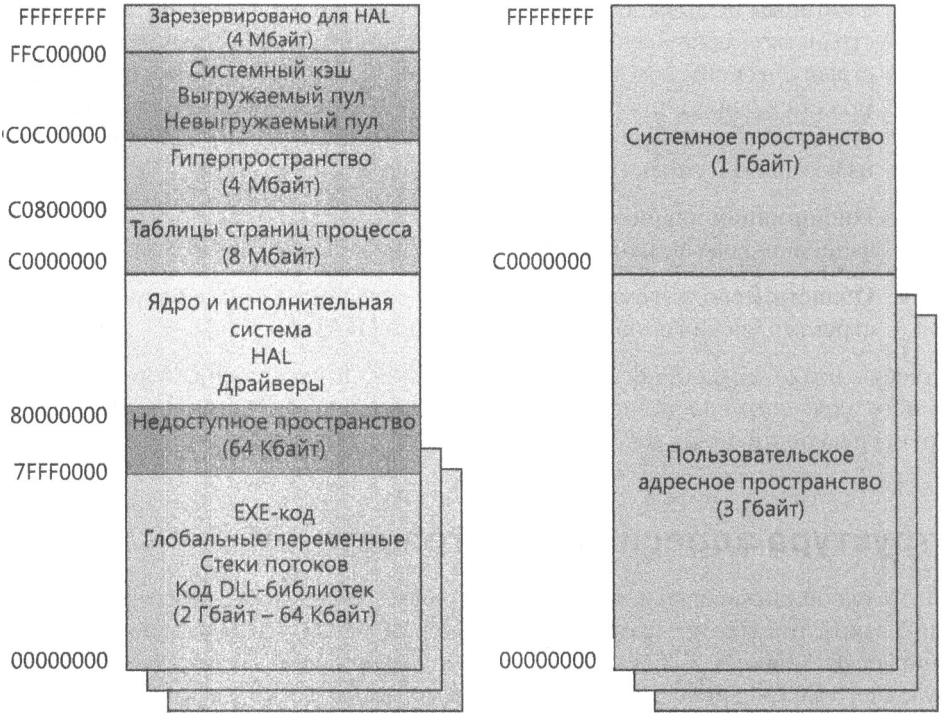


Рис. 5.10. Структура виртуальных адресных пространств на платформе x86

создании исполняемого файла, указав ключ компоновщика `/LARGEADDRESSAWARE`. Также можно воспользоваться страницей свойств в Visual Studio (выберите команду Linker ▸ System ▸ Enable Large Addresses). Флаг также можно добавить в исполняемый образ даже без построения (исходный код при этом не требуется) при помощи такой программы, как Editbin.exe (часть инструментария Windows SDK), — при условии, что файл не подписан. При запуске приложения на системе с пользовательским адресным пространством размером 2 Гбайт этот флаг игнорируется. Некоторые системные образы помечены как поддерживающие большее адресное пространство, чтобы они могли воспользоваться преимуществами систем, запущенных с большим адресным пространством процесса. К их числу относятся:

- **Lsass.exe** — подсистема проверки подлинности локальной системы безопасности;
- **Inetinfo.exe** — Internet Information Server;
- **Chkdsk.exe** — утилита проверки диска;
- **Smss.exe** — диспетчер сеансов;
- **Dllhst3g.exe** — специальная версия Dllhost.exe (для приложений, поддерживающих технологию COM+).

ЭКСПЕРИМЕНТ: ПРОВЕРКА ПОДДЕРЖКИ ПРИЛОЖЕНИЕМ БОЛЬШОГО АДРЕСНОГО ПРОСТРАНСТВА

Утилита Dumpbin из инструментария Visual Studio Tools (и более старых версий Windows SDK) позволяет проверить другие исполняемые файлы на поддержку больших адресных пространств. Для вывода результатов используется флаг/headers. Пример вывода Dumpbin для диспетчера сеансов:

```
dumpbin /headers c:\windows\system32\smss.exe
Microsoft (R) COFF/PE Dumper Version 14.00.24213.1
Copyright (C) Microsoft Corporation.  All rights reserved.
Dump of file c:\windows\system32\smss.exe
PE signature found
File Type: EXECUTABLE IMAGE
FILE HEADER VALUES
             14C machine (x86)
               5 number of sections
        57898F8A time date stamp Sat Jul 16 04:36:10 2016
               0 file pointer to symbol table
               0 number of symbols
              E0 size of optional header
             122 characteristics
                   Executable
                   Application can handle large (>2GB) addresses
                   32 bit word machine
```

Наконец, поскольку выделение памяти с помощью функций `VirtualAlloc`, `VirtualAllocEx` и `VirtualAllocExNuma` начинается с младших виртуальных адресов и по умолчанию продолжается в направлении к старшим адресам, если только процесс не выделит слишком много виртуальной памяти или его виртуальное адресное пространство не будет слишком сильно фрагментировано, он никогда не доберется до самых старших виртуальных адресов. Поэтому с целью тестирования можно заставить начать выделение памяти со старших адресов, воспользовавшись флагом `MEM_TOP_DOWN` функций `VirtualAlloc*` или добавив DWORD-параметр реестра `AllocationPreference` в раздел HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management и установив для него значение 0x100000.

Ниже приведен вывод утилиты TestLimit (рассмотренной в предыдущих экспериментах), реализующей утечку памяти на машине с 32-разрядной версией Windows, загруженной без параметра `increaseuserva`:

```
Testlimit.exe -r

Testlimit v5.24 - test Windows limits
Copyright (C) 2012-2015 Mark Russinovich
Sysinternals - www.sysinternals.com

Process ID: 5500

Reserving private bytes (MB)...
Leaked 1978 MB of reserved memory (1940 MB total leaked). Lasterror: 8
```

Процессу удалось зарезервировать около 2 Гбайт (на самом деле чуть меньше). Адресное пространство процесса содержит отображения EXE-кода и различных DLL-библиотек; естественно, обычный процесс не может зарезервировать все адресное пространство.

В той же системе можно переключиться на 3-гигабайтное адресное пространство, выполнив следующую команду из административного командного окна:

```
C:\WINDOWS\system32>bcdedit /set increaseuserva 3072

The operation completed successfully.
```

Команда позволяет задать любое число (в мегабайтах) от 2048 (2 Гбайт по умолчанию) до 3072 (максимум 3 Гбайт). После перезапуска системы, чтобы изменение вступило в силу, при повторном запуске TestLimit будет получен следующий результат:

```
Testlimit.exe -r

Testlimit v5.24 - test Windows limits
Copyright (C) 2012-2015 Mark Russinovich
Sysinternals - www.sysinternals.com

Process ID: 2308

Reserving private bytes (MB)...
Leaked 2999 MB of reserved memory (2999 MB total leaked). Lasterror: 8
```

Как и ожидалось, утилита TestLimit смогла организовать утечку памяти, равную почти 3 Гбайт. Это стало возможным только потому, что утилита TestLimit была скомпонована с ключом `/LARGEADDRESSAWARE`. Без этого результат был бы точно таким же, как и на системе, загруженной без изменения `increaseuserva`.

ПРИМЕЧАНИЕ Чтобы вернуть систему к обычному лимиту 2 Гбайт адресного пространства на процесс, выполните команду `bcdedit /deletevalue increaseuserva`.

Структура системного адресного пространства на платформе x86

В 32-разрядной версии Windows реализована динамическая структура системного адресного пространства, для чего используется распределитель виртуального адресного пространства (см. далее в этом разделе). Как показано на рис. 5.10, хотя есть еще несколько специально зарезервированных областей, для многих структур режима ядра адресное пространство выделяется динамически. Поэтому эти структуры не обязательно должны быть виртуально непрерывными. Каждая из них может находиться в нескольких несмежных частях системного адресного пространства. В число структур, использующих такой механизм выделения системного адресного пространства, входят:

- невыгружаемый пул;
- выгружаемый пул;

- специальный пул;
- записи системной таблицы страниц (PTE-записи);
- системные отображаемые представления;
- кэш файловой системы;
- база данных PFN;
- пространство сеанса.

Пространство сеанса на платформе x86

Для систем с несколькими сеансами (т. е. практически для всех систем, так как сеанс 0 используется системными процессами и службами, а сеанс 1 — первым пользователем, выполнившим вход) уникальные для каждого сеанса код и данные отображаются на системное адресное пространство, но при этом совместно используются процессами данного сеанса. Общая структура пространства сеанса показана на рис. 5.11. Размеры компонентов пространства сеанса, как и всего остального системного адресного пространства ядра, настраиваются динамически, а их размеры меняются диспетчером памяти по мере необходимости.

Рис. 5.11. Структура пространства сеанса на платформе x86 (без соблюдения пропорций)

ЭКСПЕРИМЕНТ: ПРОСМОТР СЕАНСОВ

О принадлежности процессов тому или иному сеансу можно судить по идентификатору сеанса. Для получения нужной информации используются такие средства, как диспетчер задач, Process Explorer или отладчик ядра. В отладчике ядра можно с помощью команды `!session` вывести список активных сеансов:

```
lkd> !session
Sessions on machine: 3
Valid Sessions: 0 1 2
Current Session 2
```

Затем можно настроиться на активный сеанс, используя команду `!session -s`, и вывести список структур данных сеанса и процессов в сеансе командой `!sprocess`:

```
lkd> !session -s 1
Sessions on machine: 3

Implicit process is now d4921040
Using session 1

lkd> !sprocess
Dumping Session 1

_MM_SESSION_SPACE d9306000
_MMSESSION        d9306c80
PROCESS d4921040  SessionId: 1  Cid: 01d8    Peb: 00668000  ParentCid: 0138
    DirBase: 179c5080  ObjectTable: 00000000  HandleCount:   0.
    Image: smss.exe

PROCESS d186c180  SessionId: 1  Cid: 01ec    Peb: 00401000  ParentCid: 01d8
    DirBase: 179c5040  ObjectTable: d58d48c0  HandleCount: <Data Not Accessible>
    Image: csrss.exe

PROCESS d49acc40  SessionId: 1  Cid: 022c    Peb: 03119000  ParentCid: 01d8
    DirBase: 179c50c0  ObjectTable: d232e5c0  HandleCount: <Data Not Accessible>
    Image: winlogon.exe

PROCESS dc0918c0  SessionId: 1  Cid: 0374    Peb: 003c4000  ParentCid: 022c
    DirBase: 179c5160  ObjectTable: dc28f6c0  HandleCount: <Data Not Accessible>
    Image: LogonUI.exe

PROCESS dc08e900  SessionId: 1  Cid: 037c    Peb: 00d8b000  ParentCid: 022c
    DirBase: 179c5180  ObjectTable: dc249640  HandleCount: <Data Not Accessible>
    Image: dwm.exe
```

Для просмотра подробной информации о сеансе нужно вывести дамп структуры `MM_SESSION_SPACE` с помощью команды `dt`:

```
lkd> dt nt!_mm_session_space d9306000
   +0x000 ReferenceCount   : 0n4
   +0x004 u                : <unnamed-tag>
   +0x008 SessionId        : 1
   +0x00c ProcessReferenceToSession : 0n6
   +0x010 ProcessList      : _LIST_ENTRY [ 0xd4921128 - 0xdc08e9e8 ]
   +0x018 SessionPageDirectoryIndex : 0x1617f
   +0x01c NonPagablePages  : 0x28
   +0x020 CommittedPages   : 0x290
   +0x024 PagedPoolStart   : 0xc0000000 Void
   +0x028 PagedPoolEnd     : 0xffbfffff Void
   +0x02c SessionObject    : 0xd49222b0 Void
   +0x030 SessionObjectHandle : 0x800003ac Void
   +0x034 SessionPoolAllocationFailures : [4] 0
   +0x044 ImageTree        : _RTL_AVL_TREE
   +0x048 LocaleId         : 0x409
   +0x04c AttachCount      : 0
   +0x050 AttachGate       : _KGATE
   +0x060 WsListEntry      : _LIST_ENTRY [ 0xcdcde060 - 0xd6307060 ]
   +0x080 Lookaside        : [24] _GENERAL_LOOKASIDE
   +0xc80 Session          : _MMSESSION
```

ЭКСПЕРИМЕНТ: ПРОСМОТР СТЕПЕНИ ИСПОЛЬЗОВАНИЯ ПРОСТРАНСТВА СЕАНСА

Степень использования пространства памяти сеанса можно просмотреть с помощью команды `!vm 4` отладчика ядра. Например, следующий результат был получен на 32-разрядной клиентской системе Windows с удаленным подключением к рабочему столу. Таким образом, в системе существуют три сеанса — два сеанса по умолчанию и удаленный сеанс (адреса объектов `MM_SESSION_SPACE` были приведены выше):

```
lkd> !vm 4
...
Terminal Server Memory Usage By Session:

Session ID 0 @ d6307000:
Paged Pool Usage:     2012 Kb
NonPaged Usage:        108 Kb
Commit Usage:         2292 Kb

Session ID 1 @ d9306000:
Paged Pool Usage:     2288 Kb
NonPaged Usage:        160 Kb
Commit Usage:         2624 Kb

Session ID 2 @ cdcde000:
Paged Pool Usage:     7740 Kb
NonPaged Usage:        208 Kb
Commit Usage:         8144 Kb

Session Summary
Paged Pool Usage:    12040 Kb
NonPaged Usage:        476 Kb
Commit Usage:        13060 Kb
```

Записи системной таблицы страниц

Записи системной таблицы страниц (PTE-записи) используются для динамического отображения системных страниц пространства ввода/вывода, стеков ядра и списков дескрипторов памяти (см. главу 6). Системные PTE-записи — ресурс не бесконечный. На 32-разрядных версиях Windows количество доступных системных PTE-записей позволяет системе теоретически описать 2 Гбайт последовательного системного виртуального адресного пространства. В 64-разрядной Windows 10 и Server 2016 системные PTE-записи позволяют описать до 16 Тбайт последовательного виртуального адресного пространства.

ЭКСПЕРИМЕНТ: ПРОСМОТР ИНФОРМАЦИИ О СИСТЕМНЫХ PTE-ЗАПИСЯХ

Количество доступных системных PTE-записей можно узнать, изучив значение счетчика монитора производительности Memory: Free System Page Table Entries (Память: свободных элементов таблицы страниц) либо воспользовавшись

командой !sysptes или !vm отладчика. Можно также вывести дамп структуры _MI_SYSTEM_PTE_TYPE как части переменной состояния памяти (MiState) (или глобальной переменной MiSystemPteInfo в Windows 8.x/2012/R2). При этом также выводится количество произошедших в системе отказов, связанных с выделением памяти под PTE-записи, — высокий показатель свидетельствует о проблеме и, возможно, о системной PTE-утечке.

```
kd> !sysptes
System PTE Information
  Total System Ptes 216560
    starting PTE: c0400000
  free blocks: 969   total free: 16334    largest free block: 264

kd> ? MiState
Evaluate expression: -2128443008 = 81228980

kd> dt nt!_MI_SYSTEM_INFORMATION SystemPtes
   +0x3040 SystemPtes : _MI_SYSTEM_PTE_STATE

kd> dt nt!_mi_system_pte_state SystemViewPteInfo 81228980+3040
   +0x10c SystemViewPteInfo : _MI_SYSTEM_PTE_TYPE

kd> dt nt!_mi_system_pte_type 81228980+3040+10c
   +0x000 Bitmap           : _RTL_BITMAP
   +0x008 BasePte          : 0xc0400000 _MMPTE
   +0x00c Flags            : 0xe
   +0x010 VaType           : c ( MiVaDriverImages )
   +0x014 FailureCount     : 0x8122bae4  -> 0
   +0x018 PteFailures      : 0
   +0x01c SpinLock         : 0
   +0x01c GlobalPushLock   : (null)
   +0x020 Vm               : 0x8122c008 _MMSUPPORT_INSTANCE
   +0x024 TotalSystemPtes  : 0x120
   +0x028 Hint             : 0x2576
   +0x02c LowestBitEverAllocated : 0xc80
   +0x030 CachedPtes       : (null)
   +0x034 TotalFreeSystemPtes : 0x73
```

Если вы видите большое количество системных отказов при выделении памяти под PTE-записи, можно включить системное отслеживание PTE-записей, создав новый DWORD-параметр TrackPtes в разделе HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management реестра и присвоив ему значение 1. Для вывода списка распределителей можно воспользоваться командой !sysptes 4.

Структура адресного пространства ARM

Как видно из рис. 5.12, структура адресного пространства ARM практически идентична адресному пространству x86. Диспетчер памяти в отношении «чистого» управления памятью рассматривает системы на базе ARM точно так же, как системы x86. Различия проявляются на уровне преобразования адресов (см. раздел «Преобразование адресов» этой главы).

Рис. 5.12. Структура виртуальных адресных пространств ARM

Структура адресных пространств 64-разрядных систем

Теоретически 64-разрядное виртуальное адресное пространство составляет 16 экзабайтов (18 446 744 073 709 551 616 байтов). Текущие ограничения процессоров допускают до 48 адресных строк, в результате чего возможное адресное пространство ограничивается 256 Тбайт (2 в 48 степени). Адресное пространство делится пополам; младшие 128 Тбайт доступны для пользовательских процессов, а старшие 128 Тбайт образуют системное пространство. Системное пространство делится на несколько областей разного размера (Windows 10 и Server 2016), как показано на рис. 5.13. Конечно, 64 разряда — огромный скачок в размерах адресного пространства по сравнению с 32 разрядами. Фактическое начало различных секций ядра не обязательно совпадает с показанным, так как в последних версиях Windows в пространстве ядра действует механизм рандомизации ASLR.

ПРИМЕЧАНИЕ Windows 8 и Server 2012 ограничиваются 16 Тбайт адресного пространства. Это связано с ограничениями реализации Windows, описанными в главе 10 шестого издания книги. Из них 8 Тбайт используются на уровне процесса, а другие 8 Тбайт используются для системного пространства.

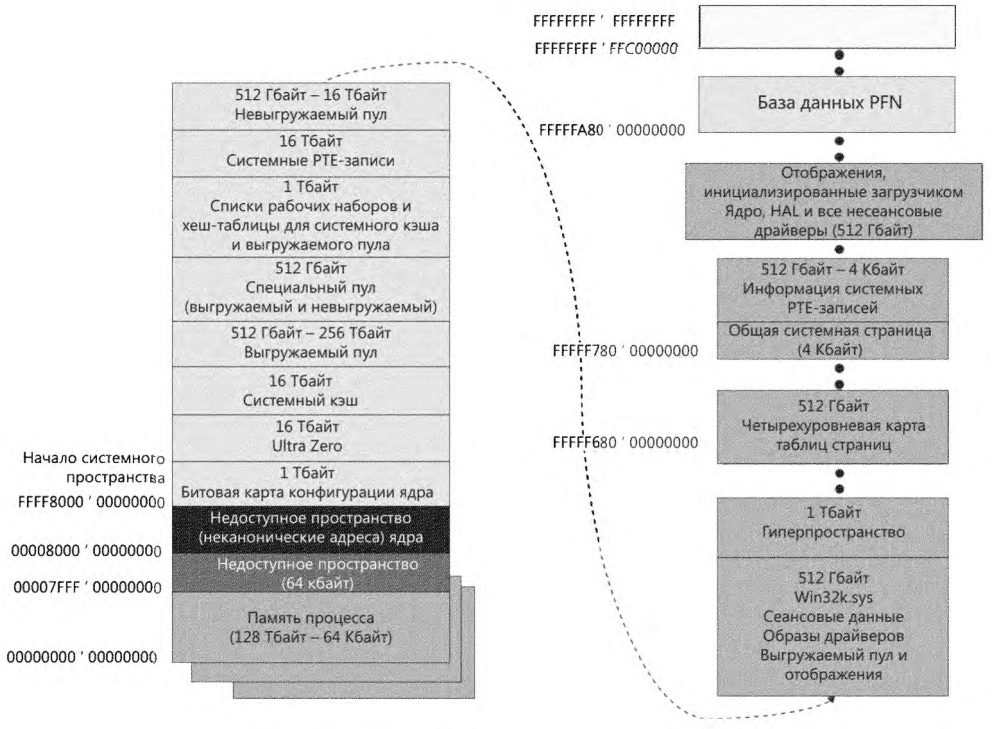


Рис. 5.13. Структура адресного пространства платформы x64

32-разрядные образы с поддержкой большого адресного пространства пользуются дополнительным преимуществом при запуске в 64-разрядной версии Windows (с Wow64). Для такого образа доступны все 4 Гбайт пользовательского адресного пространства. В конце концов, если образ может поддерживать указатели до 3 Гбайт, с указателями для 4 Гбайт проблем быть не должно, потому что для них не потребуются дополнительные биты (в отличие от перехода с 2 Гбайт на 3 Гбайт). В следующем листинге показан результат запуска TestLimit в виде 32-разрядного приложения, резервирующего адресное пространство на машине с 64-разрядной версией Windows.

```
C:\Tools\Sysinternals>Testlimit.exe -r

Testlimit v5.24 - test Windows limits
Copyright (C) 2012-2015 Mark Russinovich
Sysinternals - www.sysinternals.com

Process ID: 264

Reserving private bytes (MB)...
Leaked 4008 MB of reserved memory (4008 MB total leaked). Lasterror: 8
Not enough storage is available to process this command.
```

Для получения этих результатов программа TestLimit должна быть скомпонована с флагом `/LARGEADDRESSAWARE`; иначе в каждом случае результаты составили бы около 2 Гбайт. 64-разрядные приложения, скомпонованные без флага `/LARGEADDRESSAWARE`, ограничиваются первыми 2 Гбайт виртуального адресного пространства процесса, как и 32-разрядные приложения. (Этот флаг устанавливается по умолчанию в Visual Studio для 64-разрядных построений.)

Ограничения виртуальной адресации на платформе x64

Как уже ранее упоминалось, 64-разрядное виртуальное адресное пространство обеспечивает максимально доступный объем виртуальной памяти в 16 экзабайтов (Эбайт) — весьма существенный прирост по сравнению со значением в 4 Гбайт, доступным при 32-разрядной адресации. Понятно, что современным компьютерам, а также тем компьютерам, которые должны появиться в обозримом будущем, поддержка такого огромного объема памяти даже и близко не понадобится.

Соответственно, чтобы упростить архитектуру микросхем и избежать ненужных издержек — в частности, на преобразование адресов (о котором речь пойдет чуть позже), в нынешних процессорах x64 производства AMD и Intel реализовано только 256 Тбайт виртуального адресного пространства. Таким образом, реализованы только младшие 48 разрядов 64-разрядного виртуального адресного пространства. Тем не менее виртуальные адреса по-прежнему имеют размерность 64 разряда, занимая 8 байт в регистрах или при хранении в памяти. Старшие 16 разрядов (от 48 до 63) должны иметь такое же значение, как и у самого старшего реализованного разряда (разряда 47), аналогично расширению знака в арифметических операциях с дополнением до двух. Адрес, отвечающий этому правилу, считается *«каноническим»*.

Согласно этим правилам младшая половина адресного пространства начинается с адреса 0x0000000000000000, как и ожидалось, но заканчивается на адресе 0x00007FFFFFFFFFFF. Старшая половина адресного пространства начинается с адреса 0xFFFF800000000000 и заканчивается на адресе 0xFFFFFFFFFFFFFFFF. Каждая «каноническая» часть составляет 128 Тбайт. Поскольку на самых новых процессорах реализовано больше адресных разрядов, младшая половина памяти будет расширена вверх, к адресу 0x7FFFFFFFFFFFFFFF, а старшая половина памяти — вниз, к адресу 0x8000000000000000.

Динамическое управление системным виртуальным адресным пространством

В 32-разрядных версиях Windows системное адресное пространство управляется через находящийся в ядре внутренний механизм распределения виртуальной памяти, который рассматривается в данном разделе. В настоящее время в 64-разрядных версиях Windows для управления виртуальным адресным пространством

этот распределитель не нужен (благодаря чему на него не тратятся ресурсы), поскольку каждая область определена статически (рис. 5.13).

Основные динамические диапазоны и виртуальные адреса, доступные всему пространству ядра, устанавливаются функцией `MiInitializeDynamicVa` при инициализации системы. Затем функция `MiIntializeSystemVaRange` инициализирует диапазоны адресного пространства для образов начального загрузчика, пространства процесса (гиперпространства) и HAL, а также устанавливает жестко заданные диапазоны адресов (только в 32-разрядных системах). Позже, при инициализации невыгружаемого пула, функция `MiIntializeSystemVaRange` используется еще раз для резервирования виртуального адресного пространства этого пула. И наконец, при каждой загрузке драйвера адресный диапазон переразмечается в диапазон образа драйвера (вместо диапазона начальной загрузки).

С этого момента все остальное системное виртуальное адресное пространство может быть запрошено и освобождено в динамическом режиме через функции `MiObtainSystemVa` (и ее аналог `MiObtainSessionVa`) и `MiReturnSystemVa`. Такие операции, как расширение системного кэша, системных PTE-записей, невыгружаемого пула, выгружаемого пула и/или особого пула, отображение памяти с помощью больших страниц, создание базы данных PFN-номеров и инициирование нового сеанса, приводят к динамическому выделению виртуального адресного пространства для указанного диапазона.

Всякий раз, когда имеющийся в ядре распределитель виртуального адресного пространства получает диапазоны виртуальной памяти для использования конкретным типом виртуальных адресов, он обновляет массив `MiSystemVaType`, в котором содержатся типы виртуальных адресов для только что распределенного диапазона. Значения, которые могут появляться в массиве `MiSystemVaType`, перечислены в табл. 5.8.

Таблица 5.8. Типы системного виртуального адресного пространства

Область	Описание	Возможность ограничения
`MiVaUnused (0)`	Не используется	–
`MiVaSessionSpace (1)`	Адреса для пространства сеанса	Да
`MiVaProcessSpace (2)`	Адреса для адресного пространства процесса	Нет
`MiVaBootLoaded (3)`	Адреса для образов, загруженных с помощью загрузчика операционной системы	Нет
`MiVaPfnDatabase (4)`	Адреса для базы данных PFN-номеров	Нет
`MiVaNonPagedPool (5)`	Адреса для невыгружаемого пула	Да
`MiVaPagedPool (6)`	Адреса для выгружаемого пула	Да
`MiVaSpecialPoolPaged (7)`	Адреса для особого пула	Нет
`MiVaSystemCache (8)`	Адреса для системного кэша	Да
`MiVaSystemPtes (9)`	Адреса для системных PTE-записей	Да

Область	Описание	Возможность ограничения
`MiVaHal (10)`	Адреса для HAL	Нет
`MiVaSessionGlobalSpace (11)`	Адреса для глобального пространства сеанса	Нет
`MiVaDriverImages (12)`	Адреса для загруженных образов драйверов	Нет
`MiVaSpecialPoolNonPaged (13)`	Адреса для особого пула (невыгружаемого)	Да
`MiVaSystemPtesLarge (14)`	Адреса для PTE-записей больших страниц	Да

Хотя возможность динамического резервирования адресного пространства по запросу улучшает управление виртуальной памятью, без возможности освобождения этой памяти пользы от нее не будет. По сути, если может быть урезан выгружаемый пул или системный кэш, или когда освобождаются особый пул и область отображения больших страниц, то освобождаются и связанные с ними виртуальные адреса. (Освобождение реестра загрузки представляет собой отдельный случай.) Тем самым предоставляется возможность динамического управления памятью в зависимости от использования каждого компонента. Кроме того, компоненты могут затребовать память обратно с помощью функции `MiReclaimSystemVa`, которая запрашивает для освобождения виртуальные адреса, связанные с системным кэшем (через поток разыменования сегмента), если доступное адресное пространство становится меньше 128 Мбайт. Восстановление также может выполняться при освобождении исходного невыгружаемого пула.

Кроме улучшенного распределения виртуальных адресов и управления ими для различных потребителей памяти режима ядра, динамический распределитель виртуальных адресов также имеет преимущества, связанные со снижением затрат памяти. Вместо предварительного ручного выделения записей для статических таблиц страниц и самих таблиц страниц, соответствующие структуры выделяются по запросу. Как в 32-разрядных, так и в 64-разрядных системах это приводит к сокращению использования памяти во время загрузки, поскольку для неиспользуемых адресов таблицы страниц не выделяются. Это также означает, что в 64-разрядных системах для зарезервированных областей больших адресных пространств таблицы страниц не обязаны отображаться в памяти. Это позволяет им иметь сколь угодно большие размеры, особенно в системах с небольшим объемом оперативной памяти для хранения создаваемых структур.

ЭКСПЕРИМЕНТ: ПОЛУЧЕНИЕ ИНФОРМАЦИИ ОБ ИСПОЛЬЗОВАНИИ СИСТЕМНЫХ ВИРТУАЛЬНЫХ АДРЕСОВ (WINDOWS 10 И SERVER 2016)

Для просмотра данных текущего и пикового использования каждого типа системных виртуальных адресов можно воспользоваться отладчиком ядра. Глобальная переменная `MiVisibleState` (типа `MI_VISIBLE_STATE`) предоставляет доступную информацию с использованием символических имен. (Пример приведен для Windows 10 на платформе x86.)

1. Чтобы получить представление о данных, предоставляемых `MiVisibleState`, выведите дамп структуры со значениями:

```
lkd> dt nt!_mi_visible_state poi(nt!MiVisibleState)
   +0x000 SpecialPool      : _MI_SPECIAL_POOL
   +0x048 SessionWsList    : _LIST_ENTRY [ 0x91364060 - 0x9a172060 ]
   +0x050 SessionIdBitmap  : 0x8220c3a0 _RTL_BITMAP
   +0x054 PagedPoolInfo    : _MM_PAGED_POOL_INFO
   +0x070 MaximumNonPagedPoolInPages : 0x80000
   +0x074 SizeOfPagedPoolInPages : 0x7fc00
   +0x078 SystemPteInfo    : _MI_SYSTEM_PTE_TYPE
   +0x0b0 NonPagedPoolCommit : 0x3272
   +0x0b4 BootCommit       : 0x186d
   +0x0b8 MdlPagesAllocated : 0x105
   +0x0bc SystemPageTableCommit : 0x1e1
   +0x0c0 SpecialPagesInUse : 0
   +0x0c4 WsOverheadPages  : 0x775
   +0x0c8 VadBitmapPages   : 0x30
   +0x0cc ProcessCommit    : 0xb40
   +0x0d0 SharedCommit     : 0x712a
   +0x0d4 DriverCommit     : 0n7276
   +0x100 SystemWs         : [3] _MMSUPPORT_FULL
   +0x2c0 SystemCacheShared : _MMSUPPORT_SHARED
   +0x2e4 MapCacheFailures : 0
   +0x2e8 PagefileHashPages : 0x30
   +0x2ec PteHeader        : _SYSPTES_HEADER
   +0x378 SessionSpecialPool : 0x95201f48 _MI_SPECIAL_POOL
   +0x37c SystemVaTypeCount : [15] 0
   +0x3b8 SystemVaType     : [1024]  ""
   +0x7b8 SystemVaTypeCountFailures : [15] 0
   +0x7f4 SystemVaTypeCountLimit : [15] 0
   +0x830 SystemVaTypeCountPeak : [15] 0
   +0x86c SystemAvailableVa : 0x38800000
```

2. Обратите внимание на массивы с 15 элементами в конце дампа; они соответствуют типам системных виртуальных адресов из табл. 5.8. Массивы `SystemVaTypeCount` и `SystemVaTypeCountPeak` выглядят так:

```
lkd> dt nt!_mi_visible_state poi(nt!mivisiblestate) -a SystemVaTypeCount
   +0x37c SystemVaTypeCount :
    [00] 0
    [01] 0x1c
    [02] 0xb
    [03] 0x15
    [04] 0xf
    [05] 0x1b
    [06] 0x46
    [07] 0
    [08] 0x125
    [09] 0x38
    [10] 2
    [11] 0xb
    [12] 0x19
    [13] 0
    [14] 0xd
```

```
lkd> dt nt!_mi_visible_state poi(nt!mivisiblestate) -a SystemVaTypeCountPeak
   +0x830 SystemVaTypeCountPeak :
    [00] 0
    [01] 0x1f
    [02] 0
    [03] 0x1f
    [04] 0xf
    [05] 0x1d
    [06] 0x51
    [07] 0
    [08] 0x1e6
    [09] 0x55
    [10] 0
    [11] 0xb
    [12] 0x5d
    [13] 0
    [14] 0xe
```

ЭКСПЕРИМЕНТ: ПОЛУЧЕНИЕ ИНФОРМАЦИИ ОБ ИСПОЛЬЗОВАНИИ СИСТЕМНЫХ ВИРТУАЛЬНЫХ АДРЕСОВ (WINDOWS 8.X И SERVER 2012/R2)

Для просмотра данных текущего и пикового использования каждого типа системных виртуальных адресов можно воспользоваться отладчиком ядра. Для каждого типа системных виртуальных адресов, представленного в табл. 5.8, в глобальных массивах режима ядра `MiSystemVaTypeCount`, `MiSystemVaTypeCountFailures` и `MiSystemVaTypeCountPeak` содержатся размеры, показатели счетчика отказов и пиковые размеры для каждого типа. Размер указывается кратным отображениям PDE (см. раздел «Преобразование адресов» этой главы), т. е. фактически кратным размеру большой страницы (2 Мбайт на платформе x86). Ниже показано, как вывести информацию об использовании памяти и пиковой нагрузке. Аналогичный прием применяется и для вывода значения счетчика отказов. (Пример приведен для 32-разрядной системы Windows 8.1.)

```
lkd> dd /c 1 MiSystemVaTypeCount L f
81c16640  00000000
81c16644  0000001e
81c16648  0000000b
81c1664c  00000018
81c16650  0000000f
81c16654  00000017
81c16658  0000005f
81c1665c  00000000
81c16660  000000c7
81c16664  00000021
81c16668  00000002
81c1666c  00000008
81c16670  0000001c
81c16674  00000000
81c16678  0000000b
lkd> dd /c 1 MiSystemVaTypeCountPeak L f
81c16b60  00000000
81c16b64  00000021
```

```
81c16b68  00000000
81c16b6c  00000022
81c16b70  0000000f
81c16b74  0000001e
81c16b78  0000007e
81c16b7c  00000000
81c16b80  000000e3
81c16b84  00000027
81c16b88  00000000
81c16b8c  00000008
81c16b90  00000059
81c16b94  00000000
81c16b98  0000000b
```

Теоретически различные диапазоны виртуальных адресов, выделенные компонентам, могут произвольно разрастаться, пока для этого имеется достаточно доступного виртуального адресного пространства. Но на практике в 32-разрядных системах распределитель для надежности и стабильности устанавливает ограничение на каждый тип виртуального адреса. (В 64-разрядных системах истощение адресного пространства ядра на данный момент не грозит.) Хотя по умолчанию никакие ограничения не налагаются, системные администраторы могут воспользоваться реестром и изменить ограничения для тех типов виртуальных адресов, для которых на данный момент это возможно (см. табл. 5.8).

Если текущий запрос при вызове функции `MiObtainSystemVa` превышает доступный лимит, делается пометка о сбое (см. предыдущий эксперимент) и независимо от объема доступной памяти запрашивается освобождение памяти. Это должно упростить распределение нагрузки в памяти и может обеспечить работу распределителя виртуальной памяти при следующей попытке. (Но при этом следует помнить, что возврат касается только области системного кэша и невыгружаемого пула.)

ЭКСПЕРИМЕНТ: УСТАНОВКА ОГРАНИЧЕНИЙ СИСТЕМНЫХ ВИРТУАЛЬНЫХ АДРЕСОВ

Массив `MiSystemVaTypeCountLimit` содержит ограничения на использование системных виртуальных адресов, которые могут быть установлены для каждого типа таких адресов. В настоящее время диспетчер памяти позволяет ограничивать только конкретно указанные типы виртуальных адресов и предоставляет возможность использовать недокументированные системные вызовы при динамической установке ограничений на стадии выполнения. (Эти ограничения могут также быть установлены через реестр в соответствии с описаниями, которые можно найти по адресу *http://msdn.microsoft.com/en-us/library/bb870880.aspx*.) Ограничения могут устанавливаться для типов системных виртуальных адресов из табл. 5.8.

Для запроса и установки различных ограничений для этих типов, а также для просмотра текущего и пикового показателей использования виртуального

адресного пространства можно воспользоваться утилитой MemLimit (из архива ресурсов этой книги). Информацию о текущих ограничениях можно запросить с помощью флага -q:

```
C:\Tools>MemLimit.exe -q
MemLimit v1.01 - Query and set hard limits on system VA space consumption
Copyright (C) 2008-2016 by Alex Ionescu
www.alex-ionescu.com

System Va Consumption:

Type                    Current            Peak            Limit
Non Paged Pool            45056 KB        55296 KB            0 KB
Paged Pool               151552 KB       165888 KB            0 KB
System Cache             446464 KB       479232 KB            0 KB
System PTEs               90112 KB       135168 KB            0 KB
Session Space             63488 KB        73728 KB            0 KB
```

В порядке эксперимента установите для выгружаемого пула ограничение в 100 Мбайт:

```
memlimit.exe -p 100M
```

Теперь воспользуйтесь программой TestLimit из пакета Sysinternals для создания максимально возможного количества дескрипторов. Обычно при достаточном объеме выгружаемого пула это количество должно составлять около 16 миллионов. Однако с ограничением в 100 Мбайт оно будет существенно меньше:

```
C:\Tools\Sysinternals>Testlimit.exe -h

Testlimit v5.24 - test Windows limits
Copyright (C) 2012-2015 Mark Russinovich
Sysinternals - www.sysinternals.com
Process ID: 4780

Creating handles...
Created 10727844 handles. Lasterror: 1450
```

За дополнительной информацией об объектах, дескрипторах и расходовании выгружаемого пула обращайтесь к главе 8.

Квоты системного виртуального адресного пространства

Ограничения системного виртуального пространства, рассмотренные в предыдущем разделе, позволяют ограничить использование виртуального адресного пространства некоторыми компонентами ядра. Тем не менее при применении на общесистемном уровне они работают только в 32-разрядных системах. Чтобы реализовать более конкретные требования по выделению квот, которые могут возникнуть у системных администраторов, диспетчер памяти совместно с диспетчером процесса устанавливает для каждого процесса общесистемные или индивидуальные для каждого пользователя квоты.

Объемы памяти каждого типа, доступные для использования заданным процессом, могут устанавливаться при помощи параметров `PagedPoolQuota`, `NonPagedPoolQuota`, `PagingFileQuota` и `WorkingSetPagesQuota` в разделе HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management реестра. Эта информация считывается при инициализации; на ее основании генерируется исходный блок системных квот, который затем назначается всем системным процессам (если, как указано далее, квоты для каждого пользователя не заданы, пользовательские процессы получают копию исходных системных квот).

Чтобы включить квоты уровня пользователя, можно создать в разделе HKLM\SYSTEM\CurrentControlSet\Session Manager\Quota System реестра подразделы с именами, представляющими SID пользователей. В этих конкретных SID-подразделах создаются ранее упомянутые параметры, которые вводят ограничения только для тех процессов, которые созданы заданным пользователем. В табл. 5.9 показано, как настраиваются эти значения (которые также могут задаваться и во время выполнения) и какие для этого нужны привилегии.

Таблица 5.9. Типы квот процесса

Имя параметра	Описание	Тип	Динамический режим	Привилегия
`PagedPoolQuota`	Максимальный размер выгружаемого пула, который может быть выделен данным процессом	Размер (Мбайт)	Только для процессов, запущенных с системным маркером	`SeIncreaseQuotaPrivilege`
`NonPagedPoolQuota`	Максимальный размер невыгружаемого пула, который может быть выделен данным процессом	Размер (Мбайт)	Только для процессов, запущенных с системным маркером	`SeIncreaseQuotaPrivilege`
`PagingFileQuota`	Максимальное количество страниц, поддерживаемых процессом с помощью страничного файла	Страницы	Только для процессов, запущенных с системным маркером	`SeIncreaseQuotaPrivilege`
`WorkingSetPagesQuota`	Максимальное количество страниц, которое может быть у процесса в его рабочем наборе (в физической памяти)	Страницы	Да	`SeIncreaseBasePriority Privilege`, если только операция не запрашивает очистку

Структура пользовательского адресного пространства

По аналогии с динамическим управлением адресным пространством ядра пользовательское адресное пространство также строится динамически. Адреса стеков программных потоков, куч процессов и загруженных образов (например, DLL-библиотек и исполняемых образов приложений) вычисляются динамически (если приложение и его образы поддерживают такой режим) с использованием механизма рандомизации адресного пространства ASLR.

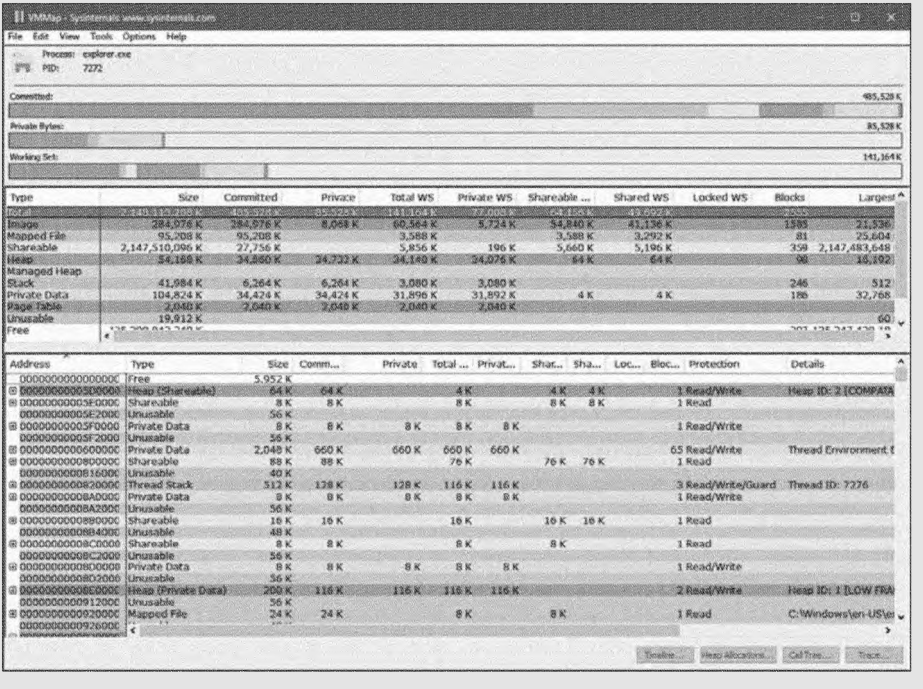
На уровне операционной системы пользовательское адресное пространство разбито на несколько четко определенных областей памяти, показанных на рис. 5.14. Сами исполняемые образы и DLL-библиотеки представлены в виде отображаемых в памяти файлов образов, за которыми следует куча(-и) процесса и стек(-и) его потока (или потоков). Кроме этих областей (и ряда зарезервированных системных структур, таких как TEB и PEB), все остальные операции выделения памяти генерируются и зависят от ситуации времени выполнения. Механизм ASLR привлекается с учетом местоположения всех этих областей, зависящих от ситуации времени выполнения, и в сочетании с механизмом DEP затрудняет удаленное использование системы посредством манипуляций с памятью. Поскольку код и данные в Windows помещаются в динамически определяемые места, атакующий обычно не может жестко задать какое-либо значимое смещение в программе или системной DLL-библиотеке.

Рис. 5.14. Структура пользовательского адресного пространства с включенным механизмом ASLR

ЭКСПЕРИМЕНТ: АНАЛИЗ ПОЛЬЗОВАТЕЛЬСКОГО ВИРТУАЛЬНОГО АДРЕСНОГО ПРОСТРАНСТВА

Утилита VMMap из пакета Sysinternals может выдать подробную информацию о виртуальной памяти, используемой любым процессом на вашей машине. Информация делится на категории по типам выделения:

- Image (Под образ). Показатели выделения памяти, используемой для отображения исполняемых образов и всех их зависимостей (например, динамических библиотек), а также любых других отображаемых на память файлов образов (формат PE).
- Mapped File (Под отображаемый файл). Показатели выделения памяти для отображаемых на память файлов данных.
- Shareable (Под совместно используемую область). Показатели выделения памяти, помеченной как общая. Обычно к ней относится совместно используемая память (но не отображаемые на память файлы, которые относятся либо к категории Image, либо к категории Mapped File).
- Heap (Под кучу). Память, выделенная под кучу (или кучи), которой владеет процесс.
- Managed Heap (Под управляемую кучу). Память, выделенная `.NET CLR` (управляемые объекты). Для процессов, не использующих .NET, не выводится ничего.



- Stack (Под стек). Память, выделенная под стек каждого программного потока данного процесса.
- Private Data (Под закрытые данные). Показатели выделения памяти, помеченной как закрытая (кроме стека и кучи), — например, внутренние структуры данных.

На копии экрана (с. 452) показан обычный вид Проводника (64-разрядного) в программе VMMap.

В зависимости от типа выделения памяти VMMap может показать дополнительную информацию: имена отображаемых файлов, идентификаторы и типы куч (при выделении памяти под кучи) и идентификаторы потоков (при выделении памяти под стеки). Оценка каждого варианта выделения выводится как для подтвержденной памяти, так и для памяти рабочего набора. Также выводятся размер и вариант защиты для каждого варианта выделения.

ASLR начинает действовать на уровне образа с исполняемого кода процесса и его DLL-зависимостей. Любой файл образа, в PE-заголовке которого указана поддержка ASLR (`IMAGE_DLL_CHARACTERISTICS_DYNAMIC_BASE`), обычно задаваемая при компоновке в Microsoft Visual Studio с помощью ключа компоновщика `/DYNAMICBASE`, которая содержит раздел перемещения, будет обрабатываться с помощью ASLR. При обнаружении такого образа система глобально выбирает подходящее смещение для текущей загрузки. Это смещение выбирается из набора 256 значений, выровненных по границе 64 Кбайт.

Рандомизация образа

Для исполняемых образов начальный адрес (смещение) загрузки вычисляется путем нахождения значения дельты при каждой загрузке исполняемого образа. Это значение представляет собой псевдослучайное 8-разрядное число в диапазоне от 0x10000 до 0xFE0000, вычисляемое с помощью счетчика отметок времени (Time Stamp Counter, TSC) текущего процессора, сдвига его на четыре разряда с последующим выполнением деления по модулю на 254 и прибавления 1. Затем это число умножается на показатель гранулярности распределения, равный, как уже упоминалось, 64 Кбайт. Прибавляя 1, диспетчер памяти гарантирует, что значение будет отлично от нуля и исполняемые образы при использовании ASLR никогда не будут загружаться по адресу в PE-заголовке. Эта дельта прибавляется к предпочитаемому адресу загрузки, в результате создается один из 256 возможных вариантов размещения в пределах 16 Мбайт от адреса образа в PE-заголовке.

Для DLL-библиотек вычисление смещения загрузки начинается с создаваемого при каждой начальной загрузке общесистемного значения, которое называется *смещением образа* (image bias), вычисляется с помощью функции `MiInitializeRelocations` и сохраняется в полях `MiState.Sections.ImageBias` глобальной структуры состояния памяти (`MI_SYSTEM_INFORMATION`) или в глобальной переменной `MiImageBias` в Windows 8.x/2012 R2. Это значение совпадает со

значением процессорного счетчика отметок времени (TSC) текущего центрального процессора на момент вызова этой функции в цикле начальной загрузки, получившего смещение и преобразованное в 8-разрядную величину применением маски. В 32-разрядных системах образуются 256 возможных значений; при выполнении аналогичных вычислений в 64-разрядных системах возможных значений будет гораздо больше из-за огромного адресного пространства. В отличие от значения для исполняемых образов, это значение вычисляется только один раз при начальной загрузке и используется во всей системе, чтобы DLL-библиотеки могли оставаться для совместного использования в физической памяти и перемещаться только один раз. Если бы DLL-библиотеки заново отображались по разным адресам внутри различных процессов, их код нельзя было бы использовать совместно. Загрузчику пришлось бы корректировать адресные ссылки для каждого процесса по-разному, превращая таким образом то, что было совместно используемым кодом, предназначенным только для чтения, в закрытые данные процесса. Каждому процессу, применяющему заданную DLL-библиотеку, пришлось бы иметь собственную закрытую копию DLL в физической памяти.

После вычисления смещения диспетчер памяти инициализирует битовую карту `ImageBitMap` (глобальная переменная `MiImageBitMap` в Windows 8.x/2012/R2), являющуюся частью структуры `MI_SECTION_STATE`. Эта битовая карта служит для представления диапазонов от 0x50000000 до 0x78000000 для 32-разрядных систем (диапазоны для 64-разрядных систем приведены ниже), при этом каждый бит представляет одну единицу выделения памяти (64 Кбайт, как уже упоминалось). Когда диспетчер загружает DLL-библиотеку, устанавливается соответствующий разряд, отмечающий ее размещение в системе; когда та же самая DLL-библиотека загружается опять, диспетчер памяти совместно использует ее объект секции с уже перемещенной информацией.

После загрузки каждой DLL-библиотеки система сканирует битовую карту сверху вниз в поиске свободных разрядов. Ранее вычисленное значение `ImageBias` используется в качестве индекса от начала для рандомизации структуры памяти при различных загрузках. Поскольку битовая карта при загрузке первой DLL-библиотеки (которой всегда является Ntdll.dll) полностью пуста, адрес ее загрузки может быть легко вычислен (у 64-разрядных систем используется другое смещение):

- **32-разрядные системы**: *0x78000000* – (`ImageBias` + `NtDllSizein64KBChunks`) * *0x10000*.
- **64-разрядные системы**: *0x7FFFFFF0000* – (`ImageBias64High` + `NtDllSizein64KBChunks`) * *0x10000*

Тогда каждая последующая DLL-библиотека будет загружена в находящийся ниже блок размером 64 Кбайт. Поэтому, если известен адрес Ntdll.dll, адреса других DLL-библиотек могут быть легко вычислены. Чтобы не допустить этого, порядок, в котором отображаются известные DLL-библиотеки диспетчером сеансов, в ходе инициализации также рандомизируется при загрузке Smss.exe.

И наконец, если места в двоичной карте не остается (это означает, что большинство областей, определяемых для ASLR, уже заняты), код перемещения

DLL-библиотек возвращается в исходное исполняемое состояние, загружая DLL-библиотеку в блок размером 64 Кбайт в пределах 16 Мбайт от предпочтительного базового адреса.

ЭКСПЕРИМЕНТ: ВЫЧИСЛЕНИЕ АДРЕСА ЗАГРУЗКИ NTDLL.DLL

После всего, что вы узнали в этом разделе, вы можете вычислить адрес загрузки Ntdll.dll по информации из переменных ядра. Следующие вычисления проводятся в системе Windows 10 для x86:

1. Запустите локальную отладку ядра.
2. Найдите значение ImageBias:

```
lkd> ? nt!mistate
Evaluate expression: -2113373760 = 820879c0
lkd> dt nt!_mi_system_information sections.imagebias 820879c0
   +0x500 Sections           :
      +0x0dc ImageBias          : 0x6e
```

3. Откройте Проводник и определите размер файла Ntdll.dll из каталога System32. В этой системе он равен 1547 Кбайт = 0x182c00, так что размер в 64-килобайтных блоках равен 0x19 (округление всегда выполняется в большую сторону). Результат равен 0x78000000 — (0x6E + 0x19) * 0x10000 = = 0x77790000.
4. Откройте Process Explorer, найдите любой процесс и найдите адрес загрузки (в столбце Base или Image Base) библиотеки Ntdll.dll. Вы увидите то же значение.
5. Попробуйте повторить эксперимент в 64-разрядной системе.

Рандомизация стека

Следующим этапом работы механизма ASLR является рандомизация размещения стека исходного программного потока (и впоследствии каждого нового потока). Рандомизация выполняется, если только для процесса не был установлен флаг `StackRandomizationDisabled`, и реализуется она выбором одного из 32 возможных мест размещения стека по 64 или по 256 Кбайт. Базовый адрес выбирается путем нахождения первой подходящей свободной области памяти с последующим выбором n-й доступной области, где n заново создается на основе значения TSC-счетчика текущего процессора, сдвинутого и замаскированного в 5-разрядном значении (что позволяет иметь 32 возможных варианта размещения).

После выбора базового адреса вычисляется новое значение на базе TSC-счетчика — на этот раз длиной в 9 разрядов. Затем для сохранения выравнивания данное значение умножается на 4, а это означает, что оно может достигнуть 2048 байт (половины страницы). Полученное значение прибавляется к базовому адресу, а результат определяет итоговую базу стека.

Рандомизация кучи

И наконец, механизм ASLR рандомизирует размещение исходной кучи процесса (и последующих куч), создаваемой в пользовательском режиме. Для определения базового адреса кучи функция `RtlCreateHeap` использует еще одно псевдослучайное значение, вычисленное на базе TSC-счетчика. Это значение, на этот раз 5-разрядное, умножается на 64 Кбайт для создания финального базового адреса, начинающегося с 0, что дает для исходной кучи возможный диапазон от 0x00000000 до 0x001F0000. Кроме того, диапазон, находящийся перед базовым адресом кучи, освобождается вручную, чтобы инициировать нарушение прав доступа при выполнении атаки посредством перебора всего диапазона адресов куч методом «грубой силы».

ASLR в адресном пространстве ядра

Механизм ASLR действует также в адресном пространстве ядра. Для 32-разрядных драйверов существует 64 возможных адреса загрузки, для 64-разрядных — 256. Перемещение образов пользовательского пространства требует существенных объемов рабочей области в пространстве ядра, но если пространство ядра стеснено, ASLR может воспользоваться для организации рабочей области адресным пространством пользовательского режима процесса System. В Windows 10 (версия 1607) и Server 2016 ASLR реализуется для большинства областей системной памяти: выгружаемых и невыгружаемых пулов, системного кэша, таблиц страниц и базы данных PFN (инициализация выполняется функцией `MiAssignTopLevelRanges`).

Управление средствами снижения риска безопасности

Мы уже видели, что ASLR и многие другие средства снижения риска безопасности являются в Windows необязательными компонентами из-за своего потенциального влияния на совместимость: ASLR применяется только к образам, имеющим в своих заголовках бит `IMAGE_DLL_CHARACTERISTICS_DYNAMIC_BASE`, механизм аппаратного неисполнения (защита данных от исполнения) может управляться за счет подбора вариантов настройки начальной загрузки и компоновщика и т. д. Чтобы предоставить корпоративным и индивидуальным пользователям более широкие возможности в отношении управления этими средствами, компания Microsoft выпустила инструментарий EMET (Enhanced Mitigation Experience Toolkit). EMET предоставляет централизованное управление средствами снижения риска безопасности, встроенными в Windows, а также добавляет еще несколько подобных средств, которые еще не стали частью Windows. Кроме того, EMET предоставляет средства уведомления, в которых используется журнал событий и которые позволяют администраторам узнавать, когда то или иное программное обеспечение получает отказ в доступе из-за действий средств снижения риска безопасности. И наконец, EMET позволяет вручную отключать конкретные приложения, которые могут создать проблемы совместимости в конкретных средах окружения, даже если эти приложения выбирались разработчиком.

ПРИМЕЧАНИЕ На момент написания книги последней версией EMET была версия 5.51. Срок поддержки был продлен до конца 2018 года. Тем не менее некоторые возможности EMET были интегрированы в текущие версии Windows.

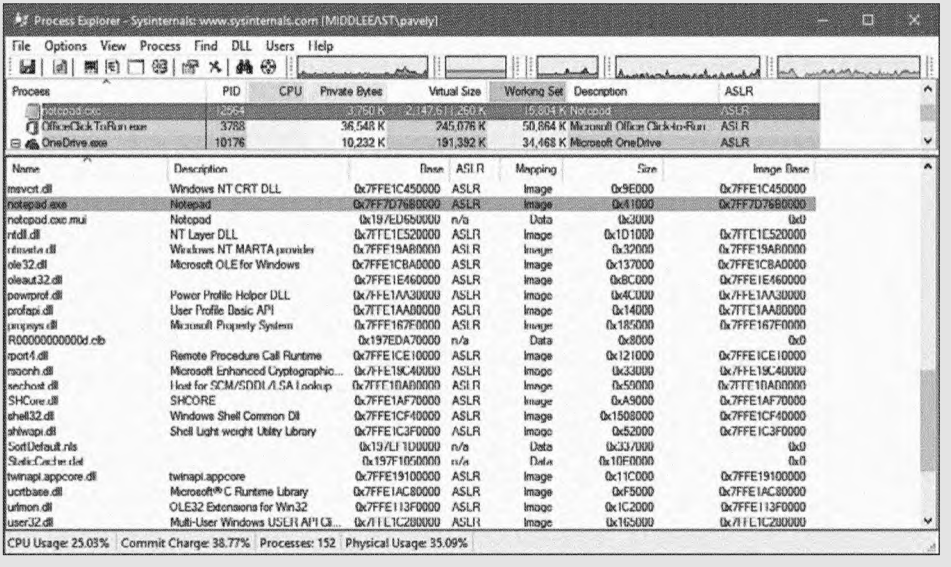
ЭКСПЕРИМЕНТ: ПРОСМОТР ASLR-ЗАЩИТЫ ПРОЦЕССОВ

Для просмотра информации о процессах (а также, что не менее важно, загружаемых ими DLL-библиотеках) можно воспользоваться программой Process Explorer из пакета Sysinternals. Это позволит узнать, поддерживают ли процессы механизм ASLR. Даже если всего одна DLL-библиотека, загруженная процессом, не поддерживает ASLR, она может сделать процесс более уязвимым при атаках.

Чтобы просмотреть информацию ASLR для процесса, выполните следующие действия.

1. Щелкните правой кнопкой мыши на любом столбце в дереве процессов и выберите команду Select Columns.
2. На вкладках Process Image и DLL установите флажок ASLR Enabled.
3. Заметьте, что все входящие в Windows программы и службы запускаются с включенным механизмом ASLR, но у сторонних приложений механизм ASLR может быть отключен.

В нашем примере мы выделили процесс Notepad.exe. В данном случае его адрес загрузки 0x7FF7D76B0000. Если закрыть все экземпляры Notepad, а затем запустить еще один экземпляр, то он окажется на другом адресе загрузки. Если завершить работу и перезагрузить систему, а затем попытаться провести эксперимент еще раз, окажется, что DLL-библиотеки с включенным механизмом ASLR после каждой перезагрузки располагаются на других загрузочных адресах.



Преобразование адресов

Рассмотрев структуру виртуальных адресных пространств в Windows, давайте посмотрим, как операционная система отображает эти адресные пространства на реально существующие физические страницы. Пользовательские приложения и системный код работают именно с виртуальными адресами. А начнем мы с подробного изучения преобразования адресов на 32-разрядных системах x86 в режиме PAE (единственный режим, поддерживаемый в последних версиях Windows) и продолжим кратким описанием разницы между 64-разрядными платформами ARM и x64. В следующем разделе вы узнаете, что происходит, когда в результате преобразования не удается получить адрес физической памяти (ошибка страницы), и узнаете, как Windows управляет физической памятью через рабочие наборы и базу данных страничных блоков.

Преобразование виртуальных адресов на платформе x86

Исходное ядро x86 поддерживало не более 4 Гбайт физической памяти — этот порог определялся процессорами, существовавшими в то время. В процессоре Intel x86 Pentium Pro появился новый режим отображения памяти, называемый *PAE* (Physical Address Extension). С подходящим чипсетом режим PAE позволяет 32-разрядным операционным системам адресовать до 64 Гбайт физической памяти на современных процессорах Intel x86 (начиная с 4 Гбайт без PAE) и до 1024 Гбайт физической памяти при выполнении на процессорах x64 в унаследованном режиме (хотя Windows в настоящее время ограничивает эту величину порогом 64 Гбайт из-за размера базы данных PFN, необходимой для описания такого объема памяти). С тех пор в Windows существовали два разных ядра x86 — с поддержкой PAE и без нее. Начиная с Windows Vista, в процессе установки Windows для x86 всегда устанавливается ядро PAE, даже если объем физической памяти системы не превышает 4 Гбайт. Это позволяет Microsoft ограничиться сопровождением одного ядра x86, так как преимущества ядер без поддержки PAE по производительности и затратам памяти стали пренебрежимо малыми (это требуется для аппаратной поддержки запрета на исполнение). По этой причине мы опишем только преобразование адресов x86 для PAE. Читатели, интересующиеся темой преобразования без поддержки PAE, найдут необходимую информацию в соответствующем разделе шестого издания книги.

Центральный процессор преобразует виртуальные адреса в физические при помощи структур данных, называемых *таблицами страниц*, которые создаются и поддерживаются диспетчером памяти. Каждая страница виртуального адресного пространства связана со структурой системного пространства, называемой *записью таблицы страниц* (Page Table Entry, PTE). PTE-запись содержит физический адрес, на который отображен виртуальный адрес. Например, на рис. 5.15 показано, как на платформе x86 три последовательные виртуальные страницы могут отображаться

на три физические несмежные страницы. PTE-записей может даже не быть для тех областей, которые были помечены как зарезервированные или подтвержденные, но к которым еще не было обращений, поскольку память для самой таблицы страниц выделяется только при первой ошибке страницы. (На рис. 5.15 пунктирная линия, соединяющая виртуальные страницы с PTE-записями, отражает косвенную связь виртуальных страниц и физической памяти.)

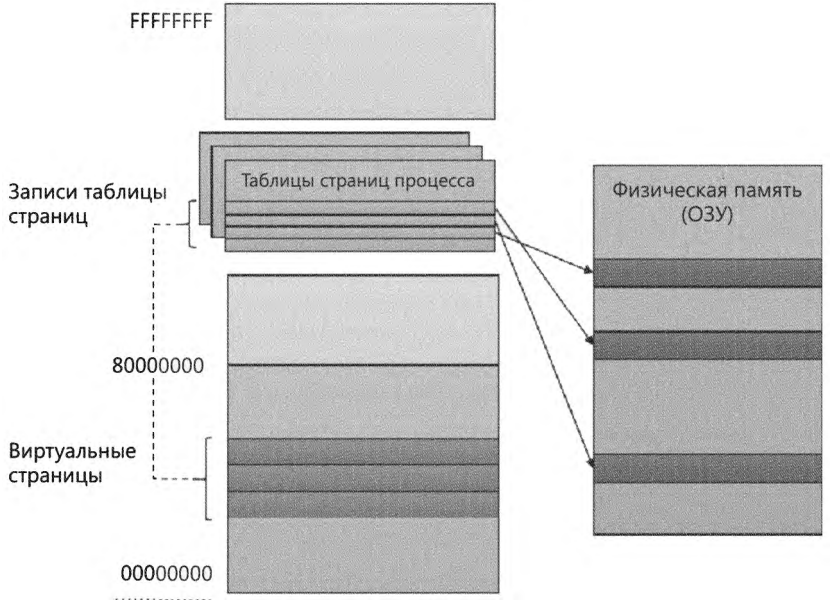


Рис. 5.15. Отображение виртуальных адресов на физическую память (x86)

ПРИМЕЧАНИЕ Даже код, выполняемый в режиме ядра (например, драйверы устройств), не может напрямую обращаться к физическим адресам памяти. Он должен делать это опосредованно, создав сначала отображаемые на них виртуальные адреса. Для получения дополнительных сведений следует просмотреть функции поддержки списков дескрипторов памяти (memory descriptor list, MDL), описания которых имеются в документации WDK.

Процесс преобразования и структура таблиц страниц и каталогов страниц (см. далее) определяются процессором. Операционная система должна «играть по правилам» и построить в памяти правильные структуры, иначе вся концепция работать не будет. На рис. 5.16 изображена общая схема преобразования адресов на платформе x86. Впрочем, тот же общий принцип действует и для других архитектур.

Как видно из рис. 5.16, входные данные для системы преобразования состоят из 32-разрядного виртуального адреса и набора управляющих структур (таблицы страниц, каталоги страниц, таблица указателей на каталоги страниц (PDPT, Page Directory Pointer Table) и буферы быстрого преобразования адресов — см. далее). На выходе должен быть получен 36-разрядный физический адрес в памяти, по ко-

торому фактически размещается байт. Число 36 определяется структурой таблиц страниц; как упоминалось ранее, это требование процессора. При отображении малых страниц (типичный случай, показанный на рис. 5.16) младшие 12 разрядов из виртуального адреса копируются напрямую в итоговый физический адрес. 12 разрядов соответствуют 4 Кбайт — размеру малой страницы.

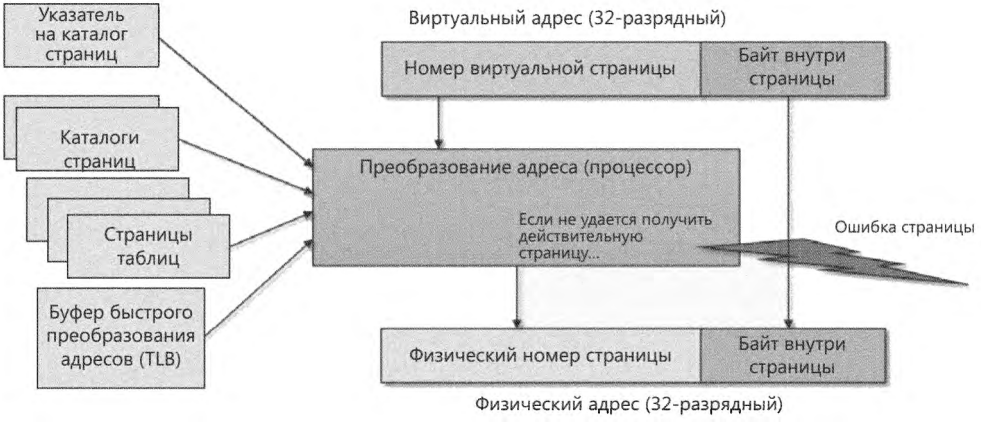


Рис. 5.16. Схема преобразования виртуальных адресов

Если выполнить успешное преобразование адреса не удается (например, страница отсутствует в физической памяти, находясь в страничном файле), процессор выдает исключение, называемое *ошибкой страницы* (page fault). Оно сообщает ОС о том, что найти страницу не удалось. Так как процессор понятия не имеет, где искать страницу (в страничном файле, в отображаемом файле или где-нибудь еще), он поручает ОС извлечь страницу из ее текущего местонахождения (если это возможно), скорректировать таблицы страниц и потребовать, чтобы процессор повторил попытку преобразования. (Ошибки страниц описаны в разделе «Страничные файлы» этой главы.) Весь процесс преобразования виртуальных адресов в физические на платформе x86 представлен на рис. 5.17.

Преобразуемый 32-разрядный виртуальный адрес делится на четыре логические части. Как вы уже видели, младшие 12 разрядов используются в исходном виде для выбора конкретного байта в странице. Процесс преобразования начинает с одной таблицы PDPT на процесс, которая постоянно находится в физической памяти. (Иначе как бы система находила ее?) Ее физический адрес хранится в структуре `KPROCESS` каждого процесса. В специальном регистре CR3 процессоров x86 хранится это значение для текущего процесса (один из программных потоков которого обратился к виртуальному адресу). Это означает, что если при переключении контекста на процессоре окажется, что новый поток принадлежит другому процессу, в регистр CR3 должен быть загружен адрес PDPT нового процесса из его структуры `KPROCESS`. Таблица PDPT должна быть выровнена по 32-разрядной границе, а также должна храниться в первых 4 Гбайт памяти (так как регистр CR3 на платформе x86 остается 32-разрядным).

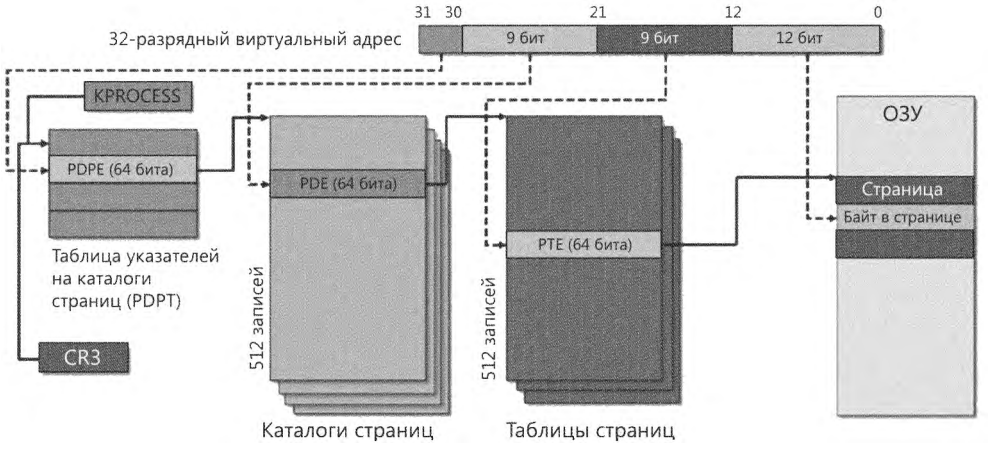


Рис. 5.17. Преобразование виртуальных адресов (x86)

В соответствии со схемой на рис. 5.17, преобразование виртуального адреса в физический происходит следующим образом:

1. Два старших разряда виртуального адреса (биты 32 и 31) содержат индекс в PDPT. Таблица содержит 4 записи. Выбранная запись — запись указателя на каталог страниц (PDPE) — указывает на физический адрес каталога страниц.
2. Каталог страниц содержит 512 записей, одна из которых выбирается битами с 21 по 29 (9 бит) из виртуального адреса. Выбранная *запись каталога страниц* (PDE, Page Directory Entry) содержит указатель на физический адрес таблицы страниц.
3. Таблица страниц также содержит 512 записей, одна из которых выбирается битами с 13 по 28 (9 бит) из виртуального адреса. Выбранная *запись таблицы страниц* (PTE, Page Directory Entry) содержит указатель на физический адрес начала страницы.
4. Смещение виртуального адреса (младшие 12 бит) прибавляется к адресу, на который указывает PTE; результат представляет собой итоговый физический адрес, запрашиваемый при вызове.

Значения записей в различных таблицах также называются *номерами страничных блоков* (PFN, Page Frame Number), потому что они указывают на адрес, выровненный по границе страницы. Размер каждой записи равен 64 битам (следовательно, размер каталога страниц или таблицы страниц не превышает 4 Кбайт), но только 24 бита действительно необходимы для описания 64-гигабайтного физического диапазона (в сочетании с 12 битами смещения для диапазона адресов, что в сумме дает 36 бит). Из этого следует, что реальное количество бит больше, чем необходимо для значений PFN.

Один из дополнительных битов особенно важен для всего механизма: речь идет о *бите достоверности* (valid bit). Этот бит указывает, являются ли данные PFN

действительными, а следовательно, должен ли процессор выполнять только что описанную процедуру. Сброшенный бит является признаком ошибки страницы. Процессор инициирует исключение и ждет, что ОС разумно обработает ошибку страницы. Например, если запрашиваемая страница была ранее записана на диск, диспетчер памяти должен прочитать ее в свободную страницу оперативной памяти, скорректировать PTE-запись и приказать процессору повторить попытку.

Так как Windows выделяет каждому процессу закрытое адресное пространство, каждый процесс имеет собственные структуры PDPT, каталоги страниц и таблицы страниц, отображаемые в закрытое адресное пространство этого процесса. Однако каталоги страниц и таблицы страниц, описывающие системное пространство, совместно используются всеми процессами (а сеансовое пространство совместно используется только процессами, входящими в сеанс). Чтобы избежать создания нескольких таблиц страниц, описывающих одну виртуальную память, записи каталога страниц, описывающие системное пространство, инициализируются указателями на существующие таблицы страниц при создании процесса. Если процесс является частью сеанса, совместное использование таблиц страниц системного пространства также обеспечивается сохранением в записях каталога страниц системного пространства указателей на существующие таблицы страниц сеанса.

Таблицы страниц и их записи

Каждая запись в каталоге страниц указывает на таблицу страниц. Таблица страниц представляет собой простой массив PTE-записей (как и PTDT). Каждая таблица страниц содержит 512 записей, а каждая PTE-запись соответствует одной странице (4 Кбайт). Это означает, что таблица страниц может отображать адресное пространство размером 2 Мбайт (512×4 Кбайт). Это означает, что каталог страниц может отображать 512×2 Мбайт, или 1 Гбайт, адресного пространства. Логично: существуют 4 PDPE-записи, которые вместе позволяют отображать все 32-разрядное адресное пространство объемом 4 Гбайт.

Для больших страниц в PDE-записи 11 начальных бит указывают на начало большой страницы в физической памяти, а смещение в байтах берется из младшего 21 бита исходного виртуального адреса. Это означает, что PDE-запись для большой страницы не указывает ни на какую таблицу страниц.

Структура каталога страниц и таблицы страниц практически одинакова. Для просмотра PTE-записей можно воспользоваться командой отладчика ядра `!pte` (см. далее описание эксперимента «Преобразование адресов»). Достоверные PTE-записи будут рассмотрены здесь, а недостоверные — в разделе «Обработка ошибок страниц». Достоверные PTE-записи содержат два основных поля: PFN-значение физической страницы, содержащей данные, или физического адреса страницы в памяти, и флаги, описывающие состояние и защиту страницы (рис. 5.18).

Биты с пометками «Программируемое поле» и «Зарезервировано» MMU игнорирует независимо от достоверности PTE-записи. Эти биты сохраняются и интерпретируются диспетчером памяти. Краткое описание аппаратных битов достоверной PTE-записи приводится в табл. 5.10.

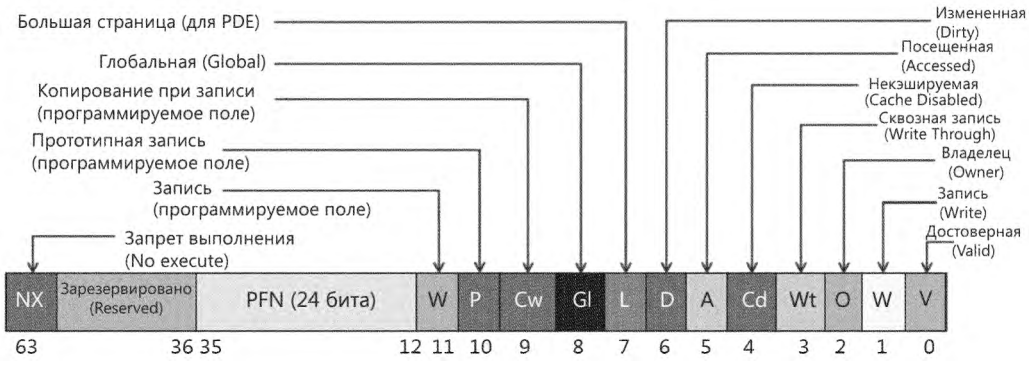


Рис. 5.18. Достоверные PTE-записи для платформы x86

Таблица 5.10. Биты состояния и защиты в PTE-записи

Название	Описание
Посещенная	К странице были обращения
Некэшируемая	Запрет кэширования страницы процессором
Копирование при записи	Для страницы используется копирование при записи (см. ранее)
Измененная	В страницу выполнялась запись
Глобальная	Преобразование распространяется на все процессы (например, сброс буфера преобразования эту PTE-запись не затрагивает)
Большая страница	Показывает, что PDE-запись отображает страницу размером 2 Мбайт. (См. ранее раздел «Большие и малые страницы»)
Запрет выполнения	Означает, что в странице не может содержаться выполняемый код (страница предназначена только для данных)
Владелец	Показывает, может ли код пользовательского режима обращаться к странице или обращение к ней ограничено только кодом режима ядра
Прототипная	PTE-запись является прототипной и используется в качестве шаблона для описания общей памяти, связанной с объектами разделов
Достоверная	Показывает, отображается ли преобразование на страницу в физической памяти
Сквозная запись	Помечает страницу для сквозной записи или (если процессор поддерживает таблицу атрибутов страницы) для объединенной записи (write-combined). Обычно используется для отображения памяти буфера видеокадра
Запись	Сообщает MMU, доступна ли страница для записи

На платформе x86 аппаратная PTE-запись содержит два бита, которые могут быть изменены MMU, — это биты изменения и посещения. MMU устанавливает бит посещения (при условии, что он еще не установлен) при каждом чтении страницы или записи в нее. А бит изменения MMU устанавливается при проведении на странице операции записи. Ответственность за сброс этих битов в нужное время возлагается на операционную систему, MMU этим не занимается.

Для защиты страницы на платформе x86 MMU использует бит записи. Когда этот бит не установлен, страница доступна только для чтения, а когда установлен — для чтения и записи. Если программный поток пытается осуществить запись в страницу с неустановленным битом записи, выдается исключение диспетчера памяти и обработчик ошибок доступа диспетчера памяти (см. далее) должен определить, можно ли потоку разрешить запись в эту страницу (например, если страница на самом деле имела пометку «Копирование при записи») и не нужно ли генерировать ошибку нарушения прав доступа.

Сравнение аппаратного и программного битов записи

Дополнительный программный бит записи, реализованный на программном уровне (см. табл. 5.10), служит в Windows для принудительной синхронизации состояния бита изменения с обновлениями в данных управления памятью. В простой реализации диспетчер памяти устанавливает аппаратный бит записи (бит 1) для любой страницы, в которую может вестись запись. Запись в любую такую страницу заставит MMU установить в записи таблицы страниц бит изменения. Позднее бит изменения сообщит диспетчеру памяти о том, что содержимое физической страницы, прежде чем использоваться для чего-либо другого, должно быть записано во внешнюю память.

На практике в многопроцессорных системах это может породить ситуацию гонки, устранение которой сопряжено с немалыми затратами. MMU-диспетчеры разных процессоров могут в любой момент установить бит изменения для любой PTE-записи, у которой был установлен аппаратный бит записи. Диспетчер памяти может в разные периоды времени обновить рабочий набор процесса, чтобы отразить состояние бита изменения в PTE-записи. Для синхронизации доступа к списку рабочего набора диспетчер памяти использует push-блокировки. Однако на многопроцессорных системах, даже если блокировка удерживается одним процессором, состояние бита изменения может меняться MMU-диспетчерами других центральных процессоров. При этом возникает риск упустить обновление бита изменения.

Чтобы избежать подобного развития событий, диспетчер памяти в Windows инициализирует в PTE-записях аппаратный бит записи (бит 1) значением 0 и сохраняет фактический признак возможности записи в программном бите записи (бит 11) — причем как для страниц, предназначенных только для чтения, так и для страниц, предназначенных для записи. Поскольку аппаратный бит записи сброшен,

при первом же обращении к такой странице по записи процессор выдаст исключение диспетчера памяти, точно так же, как это было бы сделано для страницы, действительно предназначенной только для чтения. Но в таком случае диспетчер памяти знает, что на самом деле страница *допускает* запись (благодаря программному биту записи), запрашивает push-блокировку рабочего набора, устанавливает в PTE-записи бит изменения и аппаратный бит записи, обновляет список рабочего набора, чтобы сделать пометку об изменении страницы, снимает push-блокировку с рабочего набора и аннулирует исключение. Затем аппаратная операция записи проходит в обычном порядке, но установка бита изменения осуществляется при удержании блокировки списка рабочего набора.

При последующих операциях записи в страницу исключения не выдаются, поскольку аппаратный бит записи уже установлен. MMU без особой надобности установит бит изменения, но это ни на что не повлияет, поскольку в списке рабочего набора в состоянии страницы уже будет указано, что в нее велась запись. Необходимость прохода при первой записи через обработку исключения может показаться лишней, но с каждой записываемой страницей, пока она остается достоверной, такое происходит всего лишь раз. Более того, первое обращение почти к каждой странице проходит через обработку исключения диспетчера памяти, поскольку страницы обычно инициализируются в недостоверном состоянии (при сброшенном нулевом бите в PTE-записи). Если первое обращение к странице является также и первой записью в нее, только что рассмотренная процедура с битом изменения будет происходить в ходе обработки ошибки отсутствия страницы при первом обращении к ней, стало быть, дополнительные издержки окажутся небольшими. И наконец, как на однопроцессорных, так и на многопроцессорных системах такая реализация позволяет выполнить сброс буфера быстрого преобразования адресов (см. далее) без удержания блокировки для каждой сбрасываемой страницы.

Буфер быстрого преобразования адресов

Вы уже знаете, что каждое аппаратное преобразование адреса требует проведения трех поисков:

- одного для нахождения нужной записи в PDPT;
- второго для нахождения нужной записи в каталоге страниц (что позволяет узнать местоположение таблицы страниц);
- третьего для нахождения нужной записи в таблице страниц.

Поскольку выполнение двух дополнительных поисковых операций в памяти при каждой ссылке на виртуальный адрес увеличивает требуемое число обращений к памяти вчетверо и негативно сказывается на производительности, все центральные процессоры кэшируют результаты преобразования адресов, что исключает

необходимость повторного преобразования при частом обращении к одним и тем же адресам. Кэш представляет собой массив ассоциативной памяти, которая называется *буфером быстрого преобразования адресов* (TLB, Translation Look-aside Buffer). Ассоциативная память представляет собой вектор, ячейки которого могут читаться одновременно и сравниваться с целевым значением. В случае с TLB вектор содержит, как показано на рис. 5.19, вариант отображения самых последних использованных виртуальных страниц на физические, а также тип защиты страниц, размер, атрибуты и все остальное, относящееся к каждой странице. Каждая запись в TLB похожа на запись в кэше, в теге которого содержатся части виртуального адреса, а в данных — номер физической страницы, поле защиты, бит достоверности и обычно бит, свидетельствующий о внесении изменений. Эти биты показывают состояние страниц, которым соответствует попавшая в кэш PTE-запись. Если в PTE-записи установлен бит глобальности (как делает Windows для страниц системного пространства, видимых всем процессам), TLB-запись не теряет своей достоверности при контекстном переключении процесса.

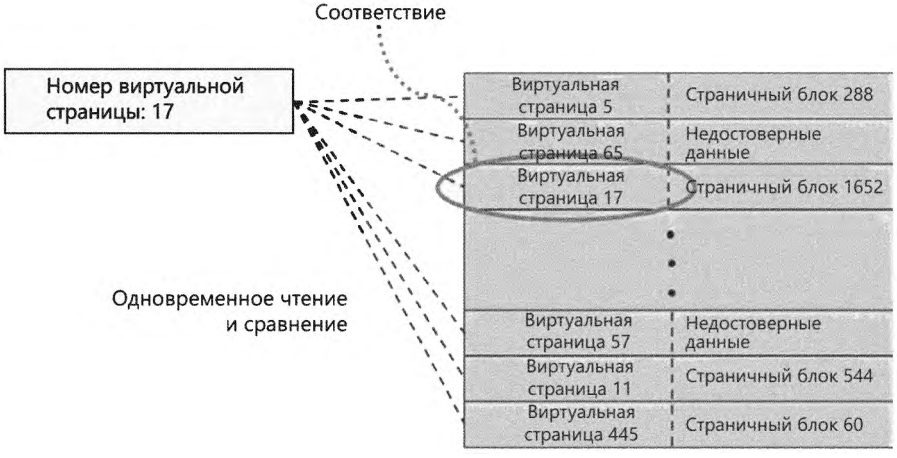


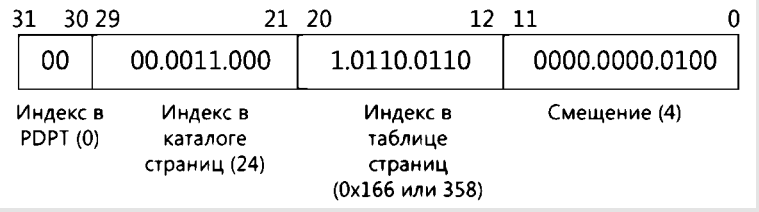
Рис. 5.19. Обращение к TLB

Часто используемые виртуальные адреса с большой вероятностью окажутся в TLB, благодаря чему обеспечивается очень быстрое преобразование виртуального адреса в физический, а следовательно, быстрый доступ к памяти. Если виртуального адреса в TLB нет, он все еще может быть в памяти, но для его нахождения необходимо провести несколько обращений к памяти, что немного увеличивает время доступа. Если виртуальная страница была выгружена из памяти или если диспетчер памяти внес изменение в PTE-запись, ему придется явным образом лишить TLB-запись достоверности. Если процесс обратится к странице снова, произойдет ошибка страницы, и диспетчер памяти вернет страницу в память (если потребуется) и воссоздаст ее PTE-запись (в результате чего в TLB появится запись для этой страницы).

ЭКСПЕРИМЕНТ: ПРЕОБРАЗОВАНИЕ АДРЕСОВ

Чтобы вы лучше поняли, как работает преобразование адресов, следующий пример демонстрирует процесс преобразования виртуального адреса в системе PAE на платформе x86. Мы воспользуемся средствами отладчика ядра для анализа PDPT, каталогов страниц, таблиц страниц и PTE-записей. В этом примере используется процесс с виртуальным адресом 0x3166004, который в настоящее время отображается на действительный физический адрес. В следующих примерах вы увидите, как происходит отслеживание недостоверных адресов с помощью отладчика ядра.

Сначала преобразуем 0x3166004 в двоичную систему счисления и разобьем запись на три поля, используемые для преобразования адреса. В двоичной форме 0x3166004 записывается в виде 11.0001.0110.0110.0000.0000.0100. При разбиении последовательности на поля будет получен следующий результат:



Чтобы начать процесс преобразования, процессор должен знать физический адрес структуры PDPT процесса. Он хранится в регистре CR3 во время выполнения потока этого процесса. Чтобы получить этот адрес, найдите поле DirBase в выходных данных команды `!process`:

```
lkd> !process -1 0
PROCESS 99aa3040  SessionId: 2  Cid: 1690    Peb: 03159000  ParentCid: 0920
    DirBase: 01024800  ObjectTable: b3b386c0  HandleCount:
                       <Data Not Accessible>
    Image: windbg.exe
```

Поле `DirBase` показывает, что PDPT находится по физическому адресу 0x1024800. Как видно из приведенной выше диаграммы, поле индекса в PDPT в виртуальном адресе из нашего примера содержит 0. Следовательно, элемент PDPT с физическим адресом соответствующего каталога страниц хранится в первом элементе PDPT с физическим адресом 0x1024800.

Команда отладчика ядра `!pte` выводит записи PDE и PTE, описывающие виртуальный адрес:

```
lkd> !pte 3166004
                    VA 03166004
PDE at C06000C0            PTE at C0018B30
contains 0000000056238867  contains 800000005DE61867
pfn 56238      ---DA--UWEV  pfn 5de61      ---DA--UW-V
```

Отладчик не выводит PDPT, но содержимое таблицы легко выводится по физическому адресу:

```
lkd> !dq 01024800 L4
# 1024800 00000000'53c88801 00000000'53c89801
# 1024810 00000000'53c8a801 00000000'53c8d801
```

Здесь используется команда расширения отладчика `!dq`. Она напоминает команду `dq` (вывод в формате 64-разрядного значения), но позволяет просматривать содержимое памяти по физическому, а не по виртуальному адресу. Так как мы знаем, что таблица PDPT состоит только из четырех записей, мы добавили аргумент длины L4, чтобы сделать вывод более наглядным.

Как видно на диаграмме, индекс в PDPT (два старших бита) из виртуального адреса равен 0, так что нужная запись PDPT находится в первом четверном слове. Формат записей PDPT сходен с форматом PDE и PTE, и мы видим, что запись с PFN-номером 0x53c88 (всегда выравнивается по границе страницы) для физического адреса 0x53c88000. Это физический адрес каталога страниц.

В выходных данных `!pte` видно, что адрес PDE-записи 0xC06000C0 является виртуальным, а не физическим. В системах x86 первый каталог страниц процесса начинается с виртуального адреса 0xC0600000. В данном случае адрес PDE равен 0xC0 — т. е. 8 байт (размер записи) умноженные на 24 прибавляются к начальному адресу каталога страниц. Следовательно, поле индекса в каталоге страниц виртуального адреса равно 24. Это означает, что мы ищем 25-ю PDE-запись в каталоге страниц.

PDE-запись предоставляет PFN-номер необходимой таблицы страниц. В данном примере PFN-номер равен 0x56238; значит, таблица страниц начинается с физического адреса 0x56238000. К этому значению MMU прибавляет поле индекса в таблице страниц (0x166) из виртуального адреса, умноженное на 8 (размер PTE-записи в байтах). Полученный физический адрес PTE-записи равен 0x56238B30.

Отладчик показывает, что эта PTE-запись размещается по *виртуальному* адресу 0xC0018B30. Обратите внимание: смещение (0xB30) совпадает со смещением от физического адреса, как это всегда бывает при преобразовании адресов. Так как диспетчер памяти отображает таблицы страниц, начиная с адреса 0xC0000000, прибавление 0xB30 к 0xC0018000 (0x18 — запись 24, как вы уже видели) дает виртуальный адрес, показанный в выводе отладчика ядра: 0xC0018B30. Отладчик показывает, что поле PFN в PTE-записи равно 0x5DE61.

Наконец, можно проверить смещение в байтах от исходного адреса. Как упоминалось ранее, MMU присоединяет смещение к PFN из PTE-записи, в результате чего образуется физический адрес 0x5DE61004. Этот физический адрес соответствует исходному виртуальному адресу 0x3166004... пока.

Биты флагов из PTE-записи интерпретируются справа от PFN. Например, PTE-запись, описывающая запрашиваемую страницу, имеет флаги `---DA--UW-V`. Здесь `A` означает наличие обращений (Accessed — страница была прочитана), `U` — доступность для пользовательского режима (в отличие от страниц, доступных только для режима ядра), `W` — возможность записи (в отличие от страниц,

доступных только для чтения) и V — достоверность (Valid — PTE-запись представляет действительную страницу в физической памяти).

Для подтверждения результата вычисления физического адреса просмотрим соответствующую память как по виртуальному, так и по физическому адресу. Сначала в результате выполнения команды отладчика `dd` (вывод в формате двойных слов) с виртуальным адресом будет получен следующий результат:

```
lkd> dd 3166004 L 10
03166004  00000034 00000006 00003020 0000004e
03166014  00000000 00020020 0000a000 00000014
```

Команда `!dd` для только что вычисленного физического адреса выводит то же содержимое:

```
lkd> !dd 5DE61004 L 10
#5DE61004 00000034 00000006 00003020 0000004e
#5DE61014 00000000 00020020 0000a000 00000014
```

Аналогичным образом можно сравнить данные по виртуальным и физическим адресам для PTE и PDE.

Преобразование виртуальных адресов на платформе x64

Преобразование адреса на платформе x64 производится почти так же, как на платформе x86, но при этом появляется четвертый уровень преобразования. У каждого процесса имеется расширенный каталог страниц четвертого уровня, называемый *картой страниц четвертого уровня* (page map level 4 table). В нем содержатся данные о физическом расположении 512 структур третьего уровня, которые называются *родительскими каталогами страниц* (page parent directories). Эти каталоги аналогичны структурам PDPT в системах x86 с поддержкой PAE, но вместо одной таблицы их теперь 512, и каждый родительский каталог страниц занимает целую страницу, содержащую не 4, а 512 записей. Как и PDPT-таблица, записи родительского каталога страниц содержат физическое местоположение каталогов страниц второго уровня. В свою очередь, каждый из этих каталогов содержит 512 записей, предоставляющих местоположение отдельных таблиц страниц. И наконец, таблицы страниц (в каждой из которых 512 записей) содержат физическое местоположение страниц в памяти. Упомянутые варианты «физического местоположения» хранятся в этих структурах в виде номеров страничных блоков (PFN).

В текущих реализациях архитектуры x64 виртуальные адреса ограничиваются 48 разрядами. Компоненты, из которых состоит этот 48-разрядный виртуальный адрес, представлены на рис. 5.20. Связи между этими структурами показаны на рис. 5.21. И наконец, формат аппаратной записи таблицы страниц на платформе x64 представлен на рис. 5.21.

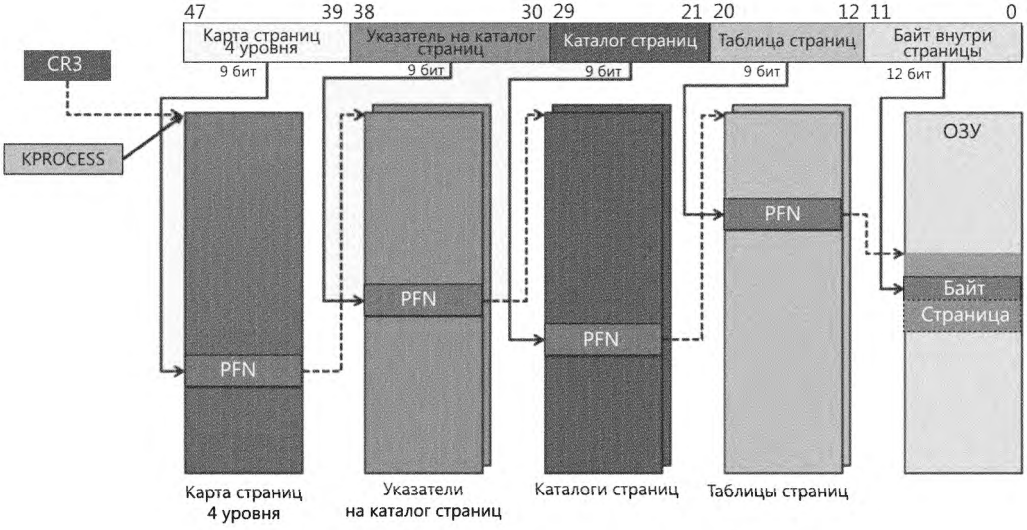


Рис. 5.20. Преобразование адреса на платформе x64

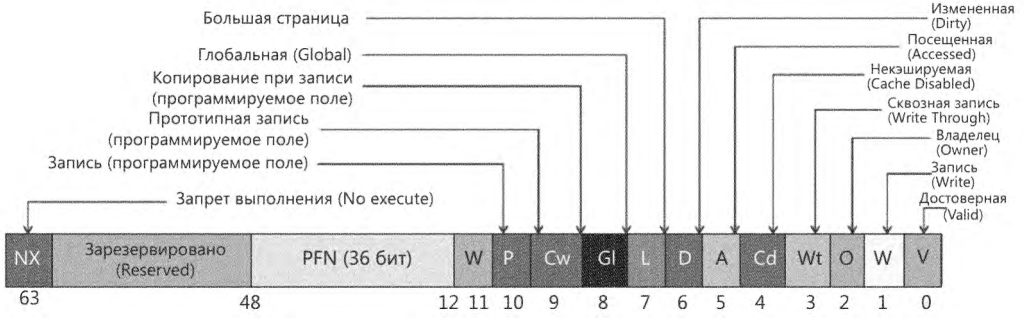


Рис. 5.21. Аппаратная PTE-запись на платформе x64

Преобразование виртуальных адресов на платформе ARM

Механизм преобразования виртуальных адресов на 32-разрядных процессорах ARM использует один каталог страниц с 1024 записями, размер каждой из которых равен 32 битам. Структуры преобразования изображены на рис. 5.22.

Каждый процесс использует один каталог страниц, физический адрес которого хранится в регистре TTBE (аналог регистра CR3 на платформах x86/x64). 10 старших разрядов виртуального адреса выбирают PDE, который может указывать на одну из 1024 таблиц страниц. Конкретная PTE-запись выбирается следующими 10 битами виртуального адреса. Каждая достоверная PTE-запись указывает на начало страницы в физической памяти; смещение задается младшими 12 битами

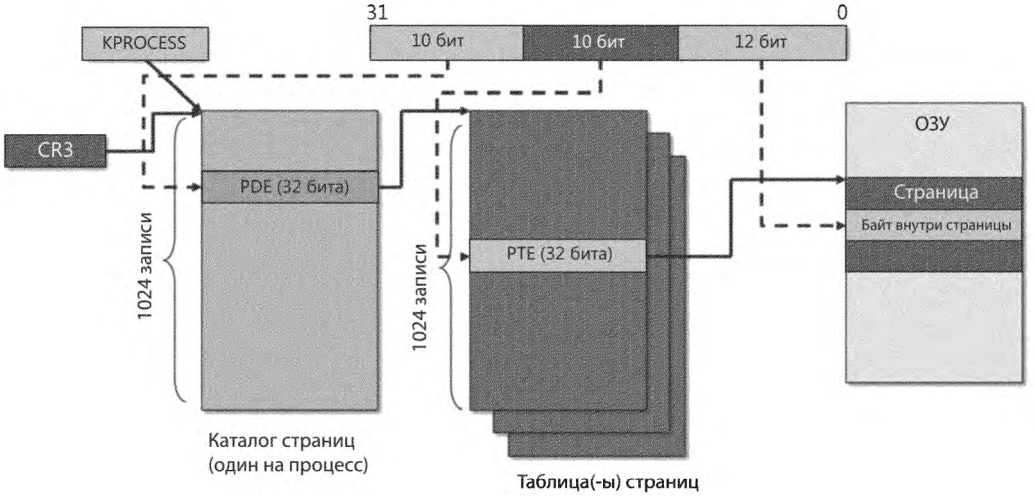


Рис. 5.22. Структуры преобразования виртуальных адресов на платформе ARM

адреса (как и в случаях с x86 и x64). Схема на рис. 5.22 предполагает, что размер адресуемой физической памяти составляет 4 Гбайт, потому что каждая PTE-запись меньше (32 бита), чем для платформы x86/x64 (64 бита); действительно, для PFN используются только 20 бит. Процессоры ARM поддерживают режим PAE (по аналогии с x86), но система Windows эту функциональность не использует. Возможно, будущие версии Windows будут поддерживать 64-разрядную архитектуру ARM, которая снимет ограничения физических адресов и кардинально расширит виртуальное адресное пространство для процессов и системы.

Любопытно, что структуры достоверной PTE-записи, PDE-записи и PDE-записи большой страницы различаются. На рис. 5.23 изображена структура достоверной PTE-записи для ARMv7, используемая текущими версиями Windows. За дополнительной информацией обращайтесь к официальной документации ARM.

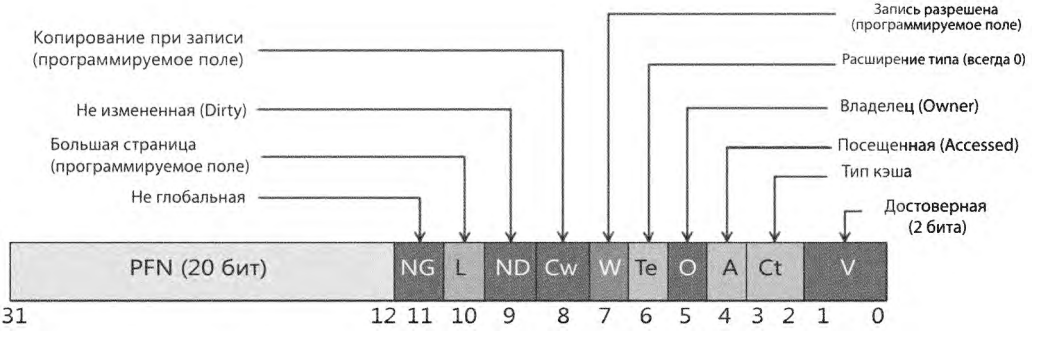


Рис. 5.23. Структура достоверной PTE-записи на платформе ARM

Обработка ошибок страниц

Ранее было показано, как осуществляется преобразование при достоверной PTE-записи. Если в PTE-записи бит достоверности сброшен, это свидетельствует о том, что в данный момент искомая страница по каким-то причинам процессу не доступна. В этом разделе рассматриваются типы недостоверных PTE-записей и порядок разрешения ссылок на такие записи.

ПРИМЕЧАНИЕ В этом разделе подробно описываются форматы PTE-записей, используемые только на 32-разрядной платформе x86. PTE-записи для 64-разрядных платформ и систем ARM содержат аналогичную информацию, но их подробная структура здесь не рассматривается.

Ссылка на недостоверную страницу называется *ошибкой страницы* (page fault). Обработчик системных прерываний ядра (см. раздел «Диспетчеризация системных прерываний» в главе 8) передает ошибку такого рода обработчику ошибок диспетчера памяти (`MmAccessFault`) для ее разрешения. Процедура обработчика запускается в контексте того программного потока, выполнение которого стало причиной ошибки и который отвечает за разрешение этой ошибки (если это возможно) или за выдачу соответствующего исключения. Возникновение подобных ошибок может вызываться различными причинами (табл. 5.11).

Таблица 5.11. Причины ошибок страниц

Причина ошибки	Результат
Поврежденная запись PDE/PTE	Фатальный сбой системы с кодом 0x1A (`MEMORY_MANAGEMENT`)
Обращение к странице, отсутствующей в памяти, но имеющейся на диске в страничном файле или в отображаемом файле	Выделение физической страницы и чтение искомой страницы с диска и в соответствующий рабочий набор
Обращение к странице, которая находится в списке ожидающих использования (standby) или в списке измененных страниц (modified)	Перемещение страницы к соответствующему процессу, сеансу или системному рабочему набору
Обращение к незакрепленной странице (например, в зарезервированном адресном пространстве или в адресном пространстве, которое еще не выделено)	Нарушение прав доступа
Обращение из пользовательского режима к странице, доступной только в режиме ядра	Нарушение прав доступа
Запись в страницу, предназначенную только для чтения	Нарушение прав доступа
Обращение к странице, подлежащей заполнению нулями	Добавление заполненной нулями страницы к соответствующему рабочему набору

Причина ошибки	Результат
Запись в защитную страницу	Нарушение доступа к защитной странице (для обращения к стеку пользовательского режима выполняется автоматическое расширение стека)
Запись в страницу, копируемую при записи	Создание принадлежащей процессу (или сеансу) закрытой копии страницы и замена исходной страницы в процессе, сеансе или системном рабочем наборе
Запись в достоверную страницу, копия которой еще не была записана в текущее резервное хранилище	Установка бита изменения в PTE-записи
Выполнение кода в странице, помеченной как неисполняемая	Нарушение прав доступа
Разрешения PTE не совпадают с разрешениями анклава (см. раздел «Анклавы» этой главы и описание функции `CreateEnclave` в документации SDK)	Пользовательский режим: исключение нарушения прав доступа. Режим ядра: фатальный сбой с кодом 0x50 (`PAGE_FAULT_IN_NONPAGED_AREA`)

В следующем разделе дается описание четырех основных типов недостоверных PTE-записей, с которыми имеет дело обработчик ошибок отсутствия страниц. Далее рассказывается о специальном случае недостоверных PTE-записей — *прототипных PTE-записях*, служащих для реализации страниц, предназначенных для совместного использования.

Недостоверные PTE-записи

Если бит достоверности PTE-записи, встреченный в ходе преобразования адреса, имеет нулевое значение, это означает, что PTE-запись представляет собой недостоверную страницу, одну из тех, при ссылке на которую диспетчер памяти выдает исключение управления памятью или ошибку отсутствия страницы. MMU игнорирует все остальные биты PTE-записи, поэтому операционная система может использовать эти биты для хранения информации о странице, ставшей причиной выдачи ошибки отсутствия страницы.

Ниже перечислены четыре разновидности недостоверных PTE-записей и описана их структура. Зачастую их называют *программными PTE-записями*, поскольку они интерпретируются диспетчером памяти, а не MMU. Некоторые флаги совпадают с флагами аппаратной PTE-записи — их описание приводится в табл. 5.10, другие битовые поля имеют такое же или похожее значение, что и у соответствующих полей аппаратной PTE-записи.

- **Страничный файл.** Искомая страница находится в страничном файле. Как показано на рис. 5.24, 4 бита в PTE-записи показывают, в какой из 16 возможных страниц находится искомая страница, а 32 бита предоставляют номер страницы

в файле. Обработчик ошибок управления памятью инициирует страничную операцию по помещению страницы в память и приданию ей статуса достоверной. Смещение в страничном файле всегда отлично от нуля и никогда не содержит все единичные биты (т. е. самая первая и самая последняя страницы в страничном файле для подкачки не используются), что позволяет существовать другим форматам (см. далее).

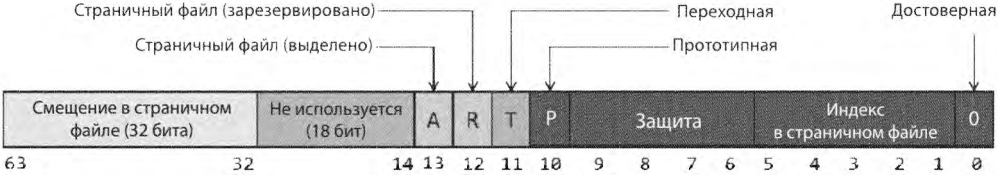


Рис. 5.24. PTE-запись, представляющая страницу в страничном файле

- **Заполнение нулями.** Этот формат PTE-записи аналогичен формату ранее рассмотренной записи для страничного файла, но поле смещения в страничном файле имеет нулевое значение. Искомая страница должна быть заполнена нулями. Обработчик ошибок управления памятью обращается к списку страниц, заполненных нулями. Если этот список пуст, обработчик берет страницу из списка свободных страниц и заполняет ее нулями. Если список свободных страниц также пуст, он берет страницу из списка страниц, ожидающих использования, и заполняет ее нулями.

- **Дескриптор виртуального адреса.** Формат этой PTE-записи такой же, как и у показанной ранее записи для страничного файла, но на этот раз поле смещения в страничном файле заполнено единицами. Это служит признаком страницы, чьи определение и хранящаяся резервная копия, если таковые имеются, могут быть найдены в дереве дескрипторов виртуального адреса (Virtual Address Descriptor, VAD) процесса. Этот формат используется для страниц, резервные копии которых по разделам хранятся в отображаемых файлах. Обработчик ошибок управления памятью находит VAD-дескриптор с описанием диапазона виртуальных адресов, содержащего виртуальную страницу, и запускает страничную операцию для отображаемого файла, на который имелась ссылка в VAD. (Более подробно VAD-дескрипторы рассмотрены в разделе «Дескрипторы виртуальных адресов» этой главы.)

- **Переходное состояние.** Бит переходной записи равен 1. Искомая страница находится в памяти в списке ожидающих использования, измененных или измененных, но не записываемых страниц либо не относится ни к какому списку. Обработчик ошибок управления памятью удаляет страницу из списка (если она состоит в списке) и добавляет ее к рабочему набору процесса. Это называется *ошибкой страницы ОЗУ* (soft page fault), поскольку операции ввода/вывода в ней не задействованы.

- **Неизвестная страница.** PTE-запись содержит нули, или таблица страниц еще не существует (PDE-запись, которая должна предоставить физический адрес

таблицы страниц, содержит нули). В обоих случаях имеющийся в диспетчере памяти обработчик ошибок управления памятью должен проверить VAD-дескрипторы, чтобы определить, был ли зафиксирован в памяти этот виртуальный адрес. Если он был подтвержден, создаются таблицы страниц, представляющие только что зафиксированное адресное пространство. Если адрес не был подтвержден (поскольку страница зарезервирована или вообще не определена), ошибка отсутствия страницы вызывает исключение нарушения доступа.

Прототипные PTE-записи

Если страница может совместно использоваться двумя процессами, то для отображения таких потенциально общих страниц диспетчер памяти применяет программную структуру, которая называется *прототипными записями таблицы страниц* (prototype page table entries), или *прототипными PTE-записями*. Для разделов, поддерживаемых страничным файлом, при создании первого объекта раздела создается массив прототипных PTE-записей; для отображаемых файлов части массива создаются по мере необходимости при отображении каждого представления. Такие прототипные PTE-записи являются частью структуры сегмента, рассматриваемой в разделе «Объекты разделов» этой главы.

Когда процесс впервые ссылается на страницу, отображаемую на представление объекта раздела (тут следует напомнить, что VAD-дескрипторы создаются, только когда представление отображено), диспетчер памяти берет информацию в прототипной PTE-записи для заполнения настоящей PTE-записи, используемой для преобразования адреса в таблице страниц процесса. Когда общая страница сделана достоверной, обе PTE-записи — обычная и прототипная — ссылаются на физическую страницу, содержащую данные. Чтобы отследить количество PTE-записей процесса, ссылающихся на достоверную общую страницу, значение счетчика в его записи базы данных PFN-номеров увеличивается на единицу. Таким образом диспетчер памяти может определить, когда на общую страницу больше не ссылается ни одна таблица страниц, и она должна быть превращена в недостоверную и переведена в список страниц в переходном состоянии или записана на диск.

Как показано на рис. 5.25, при превращении страницы, предназначенной для совместного использования, в недостоверную, PTE-запись в таблице страниц процесса заполняется специальными значениями, указывающими на прототипную PTE-

Рис. 5.25. Структура недостоверной PTE-записи, указывающей на прототипную PTE-запись

запись с описанием страницы. Поэтому при последующем обращении к странице диспетчер памяти по информации, закодированной в исходной PTE-записи, может определить местоположение прототипной PTE-записи, в свою очередь содержащей описание страницы, к которой производится обращение.

Общая страница может находиться в одном из шести различных состояний, описываемых прототипной PTE-записью:

- **Активна/достоверна.** Страница находится в физической памяти в результате обращения к ней другого процесса.
- **Переходное состояние.** Искомая страница находится в памяти и входит в список ожидающих использования или измененных страниц (либо не входит ни в один из этих списков).
- **Изменение без записи.** Искомая страница находится в памяти и входит в список измененных, но не записываемых страниц (см. табл. 5.11).
- **Заполнение нулями.** В качестве искомой должна быть предоставлена страница, заполненная нулями.
- **Страничный файл.** Искомая страница находится в страничном файле.
- **Отображаемый файл.** Искомая страница находится в отображаемом файле.

Хотя формат таких прототипных PTE-записей аналогичен формату ранее рассмотренных настоящих PTE-записей, прототипные PTE-записи для преобразования адресов не используются. Они служат промежуточным уровнем между таблицей страниц и базой данных номеров страничных блоков, никогда не присутствуя в таблицах страниц.

Поскольку все процессы, обращающиеся к потенциально общим страницам, для разрешения ошибок отсутствия страниц указывают на прототипную PTE-запись, диспетчер памяти может управлять общими страницами без обновления таблиц страниц каждого процесса, использующего общую страницу. Например, общий код или данные могут быть в какой-то момент выгружены на диск. Когда диспетчер памяти загружает страницу с диска, то для указания ее нового физического местоположения ему нужно лишь обновить прототипную PTE-запись. При этом PTE-записи в каждом процессе, совместно использующем страницу, остаются прежними (со сброшенным битом достоверности и по-прежнему указывающие на прототипную PTE-запись). Позже, по мере того как процессы обращаются к странице, происходит обновление настоящей PTE-записи.

На рис. 5.26 показаны две виртуальные страницы в отображенном представлении. Одна из них достоверна, вторая не достоверна. Как видите, на первую (достоверную) страницу указывает PTE-запись процесса и прототипная PTE-запись. Вторая страница находится в страничном файле — его точное местоположение содержится в прототипной PTE-записи. PTE-запись процесса (и любых других процессов с этой отображенной страницей) указывает на эту прототипную PTE-запись.

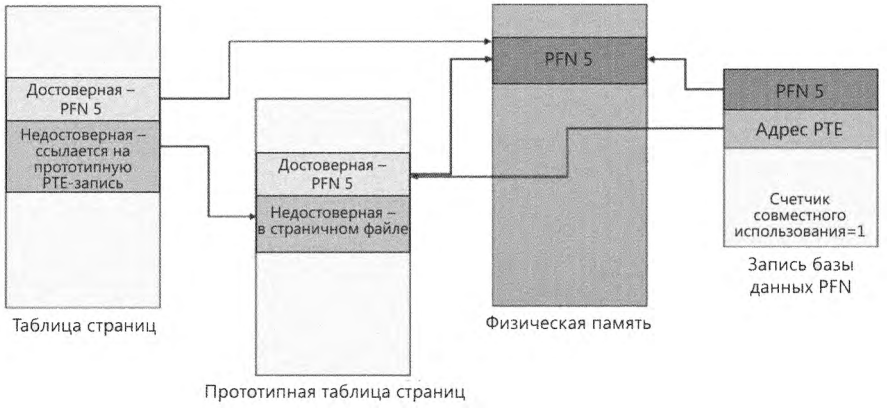


Рис. 5.26. Прототипные PTE-записи

Страничный ввод/вывод

Страничный ввод/вывод (In-paging I/O) осуществляется при выполнении операции чтения файла (страничного или отображаемого) для разрешения ошибки страницы. Кроме того, поскольку таблицы страниц могут выгружаться, обработка ошибки отсутствия страницы может по необходимости инициировать дополнительную операцию ввода/вывода, когда система загружает страницу с таблицей страниц, содержащую PTE-запись или прототипную PTE-запись с описанием исходной страницы, к которой было сделано обращение.

Страничные операции ввода/вывода проводятся в синхронном режиме (т. е. программный поток ожидает события завершения ввода/вывода и не может быть прерван асинхронным вызовом процедуры (Asynchronous Procedure Call, APC)). Обработчик ошибок управления памятью для обозначения страничных операций ввода/вывода использует в функции запроса ввода/вывода специальный модификатор. После завершения страничного ввода/вывода система ввода/вывода запускает событие, которое активирует обработчик ошибок управления памятью и позволяет ему продолжить страничную обработку.

В ходе страничного ввода/вывода программный поток, в котором произошла ошибка отсутствия страницы, не может владеть ни одним из критических объектов синхронизации диспетчера памяти. Другие потоки, принадлежащие тому же процессу, в ходе страничного ввода/вывода могут вызывать функции виртуальной памяти и обрабатывать ошибки страниц. Но при этом при завершении ввода/вывода появляется ряд важных условий, которые должны учитываться обработчиком ошибок управления памятью:

- Ошибка обращения к той же самой странице возникает в другом потоке того же самого процесса или другого процесса (эта ситуация, называемая *конфликтной ошибкой страницы*, рассматривается в следующем разделе).

- Страница может быть удалена из виртуального адресного пространства и отображена заново.
- Защита страницы может быть изменена.
- Ошибка может относиться к прототипной PTE-записи, и страница, отображаемая в прототипной PTE-записи, может отсутствовать в рабочем наборе.

Обработчик ошибок памяти справляется с этими условиями, сохраняя достаточный объем данных состояния в относящемся к потоку стеке ядра, перед тем как запрашивать страничный ввод/вывод. Таким образом, при завершении запроса он может обнаружить возникновение этих условий и, если потребуется, проигнорировать ошибку страницы, не делая страницу достоверной. Если же инструкция, приведшая к ошибке, запускается повторно, обработчик ошибок управления памятью будет вызван еще раз, и PTE-запись заново рассматривается в ее новом состоянии.

Конфликтные ошибки отсутствия страниц

Ситуация, когда у другого потока того же самого процесса или у другого процесса возникает ошибка отсутствия страницы в отношении страницы, уже ставшей причиной страничной операции ввода/вывода, называется *конфликтной ошибкой страницы* (collided page fault). Обнаружение и обработка конфликтных ошибок в обработчике ошибок управления памятью оптимизирована, поскольку на мультипоточных системах подобные ошибки являются распространенным явлением. Если у другого потока или процесса возникает ошибка той же самой страницы, обработчик ошибок управления памятью обнаруживает конфликтную ошибку страницы, отмечая, что страница находится в переходном состоянии и с ней выполняется операция чтения. (Эта информация находится в записи базы данных PFN-номеров.) В таком случае обработчик ошибок управления памятью может запустить операцию ожидания события, указанного в записи базы данных PFN-номеров, или инициировать запуск параллельного ввода/вывода для защиты файловой системы от взаимных блокировок (ввод/вывод, завершающийся первым, «выигрывает», остальные отклоняются). Для разрешения ошибки необходимо событие, инициализированное тем потоком, который первым инициировал операцию ввода/вывода.

Когда операция ввода/вывода завершается, все потоки, ожидающие наступления события, прекращают ожидание. За выполнение завершающих страничных операций отвечает тот поток, который первым установил блокировку базы данных PFN-номеров. К таким операциям относится проверка состояния ввода/вывода, позволяющая убедиться в успешном завершении ввода/вывода, сброс бита выполнения чтения в базе данных PFN-номеров и обновление PTE-записи.

Когда последующие потоки заблокируют базу данных PFN-номеров, чтобы разрешить конфликтную ошибку страницы, обработчик ошибок управления памятью определяет, что начальное обновление произошло, по сброшенному биту выполнения чтения и проверяет в записи базы данных PFN-номеров состояние флага

ошибки страничной операции ввода/вывода, убеждаясь в том, что она была успешно завершена. Если флаг ошибки страничной операции ввода/вывода установлен, PTE-запись не обновляется, и поток, вызвавший ошибку, запускает исключение ошибки страничного ввода/вывода.

Кластерные ошибки страниц

Для разрешения ошибок страниц и заполнения системного кэша диспетчер памяти производит предвыборку больших групп (кластеров) страниц. Операции предвыборки считывают данные непосредственно в системный кэш страниц, а не в рабочий набор в виртуальной памяти, поэтому предварительно извлеченные данные не расходуют виртуальное адресное пространство, а объем данных, занятых в выборке, не ограничен объемом доступного виртуального адресного пространства.

(К тому же, если страница будет использована многократно, не потребуется выполнять затратное межпроцессорное прерывание для очистки TLB.) Предварительно извлеченные страницы помещаются в резервный список и помечаются в PTE-записи как находящиеся в переходном состоянии. Если впоследствии произойдет обращение к предварительно извлеченной странице, диспетчер памяти добавит ее в рабочий набор. Но если обращение к ней так и не состоится, на ее освобождение не нужно будет тратить никаких системных ресурсов. Если какие-либо страницы в предварительно извлеченном кластере уже находятся в памяти, диспетчер памяти заново считывать их не станет. Вместо этого для их представления он воспользуется фиктивной страницей, чтобы можно было все-таки провести эффективную единую масштабную операцию ввода/вывода (рис. 5.27).

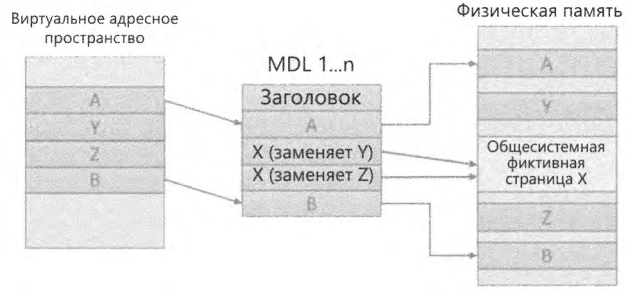


Рис. 5.27. Использование фиктивной страницы при отображении в MDL виртуального адреса на физический

Показанные на рисунке файловые смещения и виртуальные адреса, соответствующие страницам A, Y, Z и B, являются логически смежными, хотя сами по себе физические страницы не обязательно смежные. Страницы A и B не присутствуют в памяти, поэтому диспетчер памяти должен их считать. Страницы Y и Z уже при-

сутствуют в памяти, поэтому считывать их не нужно. (Фактически, со времени их последнего чтения из своего резервного хранилища они уже могли измениться, в таком случае переписывание их содержимого станет серьезной ошибкой.) Но считывание страниц A и B в рамках единой операции более эффективно, чем считывание страницы A с последующим считыванием страницы B. Поэтому диспетчер памяти запускает один запрос на считывание данных из резервного хранилища, охватывающий все четыре страницы (A, Y, Z и B). В этот запрос на считывание вовлекается такое количество страниц, которое есть смысл считывать с учетом объема доступной памяти, текущей нагрузки на систему и т. д.

Когда диспетчер памяти создает список дескрипторов памяти (Memory Descriptor List, MDL), который содержит описание запроса, он предоставляет достоверные указатели на страницы A и B, а записи для страниц Y и Z указывают на единую, общую для всей системы фиктивную страницу X. Диспетчер памяти может заполнить фиктивную страницу X потенциально устаревшими данными из резервного хранилища, поскольку он не делает страницу X видимой. Но если компонент обратится к смещениям Y и Z в MDL, он увидит не страницы Y и Z, а фиктивную страницу X.

Диспетчер памяти может представить в виде единой фиктивной страницы любое количество игнорируемых страниц, и эта фиктивная страница может вставляться несколько раз в один и тот же MDL-список или даже в несколько одновременно задействованных MDL-списков, используемых разными драйверами. Следовательно, данные о местоположении игнорируемых страниц могут измениться в любое время. (За дополнительной информацией о MDL обращайтесь к главе 6.)

Страничные файлы

Страничные файлы служат для хранения измененных страниц, которые все еще нужны некоторым процессам, но должны быть записаны на диск (поскольку они перестали отображаться в память или были выгружены из-за нехватки памяти). Пространство в страничном файле резервируется при начальном подтверждении страниц в памяти, но оптимально выбрать места для размещения страничного файла до выгрузки страниц на диск невозможно.

При начальной загрузке системы процесс диспетчера сеансов (Smss.exe) считывает список открываемых страничных файлов, анализируя параметр реестра HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\PagingFiles. Этот многострочный параметр реестра содержит имя, минимальный и максимальный размеры каждого страничного файла. Windows поддерживает до 16 страничных файлов на платформах x86 и x64 и до 2 файлов на платформе ARM. На платформах x86 и x64 каждый страничный файл может иметь максимальный размер 16 Тбайт, тогда как на платформе ARM максимум составляет 4 Гбайт. Открытые страничные файлы не могут быть удалены при работе системы, потому что открытый дескриптор каждого страничного файла поддерживается процессом System.

Поскольку страничный файл содержит части виртуальной памяти процесса и ядра, в целях безопасности система может быть настроена на очистку страничного файла при завершении своей работы. Чтобы включить этот режим, нужно задать параметру реестра `ClearPageFileAtShutdown` из раздела HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management значение 1. В противном случае после завершения работы системы в страничном файле будут содержаться данные, выгруженные в ходе работы системы. К этим данным затем может получить доступ кто-нибудь, у кого имеется физический доступ к машине.

Если минимальный и максимальный размеры страничного файла равны 0 (или не заданы), это является признаком того, что страничным файлом управляет система. Система выбирает размер страничного файла, исходя из следующих принципов:

- **Минимальный размер файла** — размер оперативной памяти или 1 Гбайт (в зависимости от того, что больше).
- **Максимальный размер файла** — утроенный размер оперативной памяти или 3 Гбайт (в зависимости от того, что больше).

Эти параметры не идеальны. Например, многие современные настольные и портативные компьютеры оснащаются 32 и даже 64 Гбайт оперативной памяти, а на серверах объем памяти может исчисляться сотнями гигабайт. Согласование исходного размера страничного файла с размером оперативной памяти может привести к существенным потерям дискового пространства, особенно если диск имеет относительно небольшой объем, что характерно для SSD-дисков. Кроме того, объем памяти в системе не всегда определяет типичную нагрузку на память в рабочем режиме данной системы.

Текущая реализация вычисляет «хороший» минимальный размер страничного файла по более сложной схеме, которая учитывает не только размер оперативной памяти, но и данные об истории использования страничного файла и другие факторы. В процессе создания и инициализации страничного файла Smss.exe вычисляет минимальные размеры страничного файла с учетом четырех факторов, хранимых в глобальных переменных:

- **Размер оперативной памяти** (`SmpDesiredPfSizeBasedOnRAM`) — рекомендуемый размер страничного файла, основанный на размере оперативной памяти.
- **Аварийный дамп** (`SmpDesiredPfSizeForCrashDump`) — рекомендуемый размер страничного файла, необходимый для хранения аварийного дампа.
- **История** (`SmpDesiredPfSizeBasedOnHistory`) — рекомендуемый размер страничного файла, определяемый на основании истории использования. Smss.exe использует таймер, который срабатывает один раз в час и сохраняет данные об использовании страничного файла.
- **Приложения** (`SmpDesiredPfSizeForApps`) — рекомендуемый размер страничного файла для приложений Windows.

Эти значения вычисляются по правилам, представленным в табл. 5.12.

Таблица 5.12. Базовые вычисления рекомендуемых размеров страничного файла

Фактор	Рекомендуемый размер страничного файла
Оперативная память	Если ОЗУ ≤ 1 Гбайт, то размер файла = 1 Гбайт. Если ОЗУ>1 Гбайт, прибавить 1/8 Гбайт для каждого дополнительного гигабайта
Аварийный дамп	Если для аварийного дампа настроен специальный файл, то для его хранения страничный файл не используется и размер файла=0. (Чтобы настроить специальный файл аварийного дампа, добавьте параметр DedicatedDumpFile в раздел HKLM\System\CurrentControlSet\Control\CrashControl.) Если же аварийный дамп настроен в режиме Automatic (используется по умолчанию), то: если ОЗУ < 4 Гбайт, то размер файла = ОЗУ/6; в противном случае размер = 2/3 Гбайт + 1/8 Гбайт для каждого дополнительного гигабайта сверх 4 (максимум 32 Гбайт); если недавно произошел фатальный сбой, при котором размер страничного файла оказался недостаточным, рекомендуемый размер увеличивается до размера ОЗУ или 32 Гбайт (меньшего из двух значений); если в системе настроена выдача полного дампа, то возвращаемый размер = размер ОЗУ + размер дополнительной информации, хранящейся в файле дампа; если настроена выдача дампа ядра, то размер = ОЗУ
История	Если сохранено достаточно данных, возвращается 90-й процентиль в качестве размера. В противном случае возвращается размер, вычисленный на основе размера ОЗУ (см. выше)
Приложения	Для сервера возвращается 0. Рекомендуемый размер вычисляется на основании фактора, который используется диспетчером жизненного цикла процессов (PLM, Process Lifecycle Manager) для определения того, когда следует завершать приложение. Текущий фактор равен 2,3*ОЗУ; он был выбран для ОЗУ=1 Гбайт (приблизительный минимум для мобильных устройств). Рекомендуемый размер (на основании упомянутого фактора) составляет около 2,5 Гбайт. Если эта величина больше размера ОЗУ, то из нее вычитается размер ОЗУ. В противном случае возвращается 0

Максимальный размер страничного файла, управляемого системой, устанавливается равным утроенному размеру ОЗУ или 4 Гбайт (в зависимости от того, что больше). Минимальный (исходный) размер страничного файла определяется следующим образом:

- Если это первый страничный файл, управляемый системой, базовый размер задается на основании истории использования страничного файла (см. табл. 5.12). В противном случае базовый размер вычисляется на основании размера ОЗУ.
- Для первого страничного файла, управляемого системой:
  - Если базовый размер меньше вычисленного размера страничного файла для приложений (`SmpDesiredPfSizeForApps`), то в качестве базового задается размер, вычисленный для приложений (см. табл. 5.12).
  - Если (новый) базовый размер меньше вычисленного размера страничного файла для аварийных дампов (`SmpDesiredPfSizeForCrashDump`), то в качестве базового задается размер, вычисленный для аварийных дампов.

ЭКСПЕРИМЕНТ: ПРОСМОТР ПАРАМЕТРОВ СТРАНИЧНЫХ ФАЙЛОВ

Для просмотра списка страничных файлов нужно заглянуть в параметр `PagingFiles` из раздела HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management реестра. В нем содержатся варианты настройки конфигурации страничного файла, изменяемые через диалоговое окно Advanced System Settings (Дополнительные параметры системы). Чтобы получить доступ к этим средствам конфигурации, выполните следующие действия:

1. Откройте Панель управления.
2. Щелкните на апплете Система и безопасность (System And Security), а затем — на пункте Система (System). Диалоговое окно Свойства системы (System Properties) также можно открыть, щелкнув правой кнопкой мыши на значке Компьютер (Computer) в Проводнике и выбрав команду Свойства (Properties).
3. Щелкните на пункте Дополнительные параметры системы (Advanced System Settings).
4. В области Быстродействие (Performance) щелкните на кнопке Параметры (Settings). На экране появится диалоговое окно Параметры быстродействия (Performance Options).
5. Перейдите на вкладку Дополнительно (Advanced).
6. В области Виртуальная память (Virtual Memory) щелкните на кнопке Изменить (Change).

ЭКСПЕРИМЕНТ: ПРОСМОТР РЕКОМЕНДУЕМЫХ РАЗМЕРОВ СТРАНИЧНОГО ФАЙЛА

Чтобы просмотреть фактические значения переменных, вычисленные по табл. 5.12, выполните следующие действия (эксперимент проводился в системе Windows 10 для платформы x86).

1. Запустите локальный отладчик ядра.
2. Найдите процессы Smss.exe:

```
lkd> !process 0 0 smss.exe
PROCESS 8e54bc40  SessionId: none  Cid: 0130    Peb: 02bab000  ParentCid:
0004
    DirBase: bffe0020  ObjectTable: 8a767640  HandleCount: <Data Not
Accessible>
    Image: smss.exe

PROCESS 9985bc40  SessionId: 1  Cid: 01d4    Peb: 02f9c000  ParentCid: 0130
    DirBase: bffe0080  ObjectTable: 00000000  HandleCount:   0.
    Image: smss.exe

PROCESS a122dc40  SessionId: 2  Cid: 02a8    Peb: 02fcd000  ParentCid: 0130
    DirBase: bffe0320  ObjectTable: 00000000  HandleCount:   0.
    Image: smss.exe
```

3. Найдите первый экземпляр (с идентификатором сеанса None); это главный экземпляр Smss.exe. (За подробностями обращайтесь к главе 2.)

4. Переключите контекст отладчика на этот процесс:

```
lkd> .process /r /p 8e54bc40
Implicit process is now 8e54bc40
Loading User Symbols
```

5. Выведите четыре переменные из предыдущего раздела. (Размер каждой переменной составляет 64 бита.)

```
lkd> dq smss!SmpDesiredPfSizeBasedOnRAM L1
00974cd0  00000000'4fff1a00
lkd> dq smss!SmpDesiredPfSizeBasedOnHistory L1
00974cd8  00000000'05a24700
lkd> dq smss!SmpDesiredPfSizeForCrashDump L1
00974cc8  00000000'1ffecd55
lkd> dq smss!SmpDesiredPfSizeForApps L1
00974ce0  00000000'00000000
```

6. Так как на машине существует всего один том (C:\), будет создан один страничный файл. При отсутствии специальной настройки он будет находиться под управлением системы. Просмотрите фактический размер файла C:\PageFile.sys на диске или воспользуйтесь командой отладчика `!vm`:

```
lkd> !vm 1
Page File: \??\C:\pagefile.sys
  Current:    524288 Kb  Free Space:    524280 Kb
  Minimum:    524288 Kb  Maximum:      8324476 Kb
Page File: \??\C:\swapfile.sys
  Current:    262144 Kb  Free Space:    262136 Kb
  Minimum:    262144 Kb  Maximum:      4717900 Kb
No Name for Paging File
  Current:  11469744 Kb  Free Space:  11443108 Kb
  Minimum:  11469744 Kb  Maximum:     11469744 Kb
...
```

Обратите внимание на минимальный размер C:\PageFile.sys (524 288 Кбайт). (Другие характеристики страничного файла будут рассмотрены в следующем разделе.) Из переменных следует, что переменная `SmpDesiredPfSizeForCrashDump` имеет наибольшее значение, поэтому она является определяющим фактором (0x1FFECD55 = 524 211 Кбайт), что очень близко к приведенному лимиту. (Размеры страничных файлов округляются до значений, кратных 64 Мбайт.)

Для добавления нового страничного файла панель управления использует системную сервисную функцию `NtCreatePagingFile`, которая определена в файле Ntdll.dll. (Для этого необходима привилегия `SeCreatePagefilePrivilege`.) Страничные файлы всегда создаются несжимаемыми (даже если каталог, в котором они находятся, подвергается сжатию), и им присваивается имя *PageFile.sys* (кроме некоторых особых случаев, упоминаемых в следующем разделе). Они создаются в корневом каталоге разделов с установленным атрибутом `Hidden`, чтобы они не

были непосредственно видны пользователю. Чтобы уберечь новые страничные файлы от удаления, дескриптор дублируется в процессе `System`; даже после того, как создававший файлы процесс закроет дескриптор нового страничного файла, у него всегда будет оставаться открытый дескриптор.

Файл подкачки

В мире приложений UWP при переходе приложения на задний план (например, при сворачивании) потоки процесса приостанавливаются, чтобы процесс не потреблял ресурсы процессора. Закрытая физическая память, используемая процессом, теоретически может быть использована для других целей. Если нагрузка на память достаточно велика, закрытый рабочий набор (физическая память, используемая процессом) может выгружаться на диск, чтобы физическая память могла использоваться другими процессами.

В Windows 8 появилась новая разновидность страничных файлов, называемая *файлом подкачки* (swap file). По сути это то же самое, что и обычный страничный файл, но файл подкачки используется исключительно для приложений UWP. Он создается в клиентских SKU только в том случае, если был создан по крайней мере один обычный страничный файл (нормальная ситуация). Этот файл с именем SwapFile.sys находится в системном корневом разделе — например, C:\SwapFile.Sys.

После того как будут созданы обычные страничные файлы, система обращается к разделу реестра `HKLM\System\CurrentControlSet\Control\Session Manager\Memory Management`. Если в нем существует параметр с именем `SwapFileControl` типа DWORD и его значение равно 0, создание файла подкачки отменяется. Если существует параметр с именем `SwapFile`, его значение читается как строка с именем файла, исходным и максимальным размером, как и для обычного страничного файла. Различие заключается в том, что нулевые значения размеров интерпретируются как отмена создания файла. Два параметра реестра не существуют по умолчанию, в результате чего файл SwapFile.sys создается в системном корневом разделе с минимальным размером 16 Мбайт для быстрых дисков малого размера (например, SSD-дисков) или 256 Мбайт для медленных дисков (или больших SSD-дисков). Максимальный размер файла подкачки выбирается равным 1,5*ОЗУ или 10 % от размера системного корневого раздела (в зависимости от того, какое значение меньше). За дополнительной информацией о приложениях UWP обращайтесь к главам 7 и 8 части 2.

ПРИМЕЧАНИЕ Файл подкачки не учитывается при определении максимального размера страничных файлов.

Виртуальный страничный файл

Команда отладчика `!vm` также выводит информацию о страничном файле с именем «No Name for Paging File». Это *виртуальный страничный файл*. Как подсказывает само название, этот файл не существует реально, а используется косвенно как вспомогательное хранилище данных для сжатия памяти (см. далее раздел «Сжатие

памяти» этой главы). Файл достаточно велик, но его размер выбирается произвольно так, чтобы избежать нехватки памяти. Недостоверные PTE-записи страниц, которые подверглись сжатию, указывают на этот виртуальный страничный файл и позволяют подсистеме сжатия памяти получить доступ к сжатиям данных, интерпретируя биты недостоверной PTE-записи для получения правильного хранилища, области и индекса.

ЭКСПЕРИМЕНТ: ПРОСМОТР ИНФОРМАЦИИ О ФАЙЛЕ ПОДКАЧКИ И ВИРТУАЛЬНОМ СТРАНИЧНОМ ФАЙЛЕ

Команда отладчика `!vm` выводит информацию обо всех страничных файлах, включая файл подкачки и виртуальный страничный файл:

```
lkd> !vm 1
Page File: \??\C:\pagefile.sys
  Current:    524288 Kb  Free Space:    524280 Kb
  Minimum:    524288 Kb  Maximum:      8324476 Kb
Page File: \??\C:\swapfile.sys
  Current:    262144 Kb  Free Space:    262136 Kb
  Minimum:    262144 Kb  Maximum:      4717900 Kb
No Name for Paging File
  Current:  11469744 Kb  Free Space:  11443108 Kb
  Minimum:  11469744 Kb  Maximum:     11469744 Kb
```

В этой системе минимальный размер файла подкачки составляет 256 Мбайт, так как команда выполняется в виртуальной машине с Windows 10. (Виртуальный жесткий диск считается медленным.) Максимальный размер файла подкачки составляет около 4,5 Гбайт, так как система оснащена 3 Гбайт ОЗУ, а размер дискового раздела составляет 64 Гбайт (меньшее из значений 4,5 Гбайт и 6,4 Гбайт).

Показатель подтверждения и системный лимит подтверждения

Теперь настало время более подробно рассмотреть такие понятия, как *показатель подтверждения* (commit charge) и *системный лимит подтверждения* (system commit limit).

Как только создается виртуальное адресное пространство, например с помощью функции `VirtualAlloc` (для подтвержденной памяти) или `MapViewOfFile`, до того как успешно завершить запрос на создание, система должна предоставить место для его хранения либо в оперативной памяти, либо в резервном хранилище. Для отображаемой памяти (кроме разделов, отображаемых на страничный файл) файлу, связанному с отображаемым объектом, указанному при вызове функции `MapViewOfFile`, предоставляется запрашиваемое резервное хранилище. Все остальные варианты выделения виртуальной памяти полагаются на хранилище, относящееся к общим ресурсам, управляемым системой: оперативную память или страничный файл (или файлы). Назначение системного лимита подтверждения

и показателя подтверждения состоит в отслеживании всевозможных вариантов использования этих ресурсов, чтобы ни при каких условиях не случилось избыточного подтверждения — т. е. чтобы определяемое виртуальное адресное пространство никогда не превысило размеров пространства, имеющегося для хранения его содержимого либо в оперативной памяти, либо в резервном хранилище (на диске).

ПРИМЕЧАНИЕ В этом разделе часто упоминаются страничные файлы. Но в принципе можно запустить Windows вообще без использования страничных файлов (хотя делать так не рекомендуется). Фактически это означает, что при исчерпании памяти ее дальнейшее выделение становится невозможным и происходит фатальный сбой с выдачей «синего экрана». Тем не менее формально при любых упоминаниях страничных файлов в этом разделе всегда следует мысленно добавлять «если такой файл (или файлы) существует».

Концептуально системный *лимит подтверждения* (commit limit) представляет собой все виртуальное адресное пространство, которое может быть создано в дополнение к вариантам выделения виртуальной памяти, связанным с собственными механизмами хранения, — т. е. в дополнение к разделам, отображаемым на файлы. Его числовое значение просто равно объему оперативной памяти, доступному для Windows, в сумме с текущими размерами всех страничных файлов. Если страничный файл увеличивается в размере или создается новый страничный файл, то соответственно увеличивается и лимит подтверждения. Если страничные файлы отсутствуют, то системный лимит подтверждения просто равен объему оперативной памяти, доступному для Windows.

Показатель подтверждения (commit charge) — это общесистемный совокупный объем всех *подтвержденных* (committed) фрагментов выделенной памяти, которые должны находиться либо в оперативной памяти, либо в страничном файле. Из названия должно быть понятно, что одной из составляющих показателя подтверждения является закрытое подтвержденное виртуальное адресное пространство процесса. Однако есть множество других факторов, вносящих свой вклад в это значение, и некоторые из них не столь очевидны.

Windows также ведет для каждого процесса счетчик, который называется *квотой процесса на страничный файл* (process page file quota). Многие варианты выделения, вносящие свой вклад в показатель подтверждения, имеют также отношение и к квоте процесса на страничный файл. Все они отражают закрытый вклад процесса в системный показатель подтверждения. Но следует заметить, что все это не отражает текущий показатель использования страничного файла, а представляет собой лишь потенциальный, или максимальный, показатель использования страничного файла в том случае, если бы все эти варианты выделения хранились в нем.

Свой вклад в системный показатель подтверждения (и во многих случаях — в квоту процесса на страничный файл) вносят следующие варианты выделения памяти (некоторые из них подробно рассматриваются в следующих разделах данной главы):

- **Закрытая подтвержденная память** (private committed memory) выделяется вызовом функции `VirtualAlloc` с параметром `MEM_COMMIT`. Это наиболее рас-

пространенный тип вклада в показатель подтверждения. Подобные варианты выделения также вносят свой вклад и в квоту процесса на страничные файлы.

- **Отображаемая память, поддерживаемая страничным файлом** (page-file-backed mapped memory), выделяется при вызове функции `MapViewOfFile`, ссылающейся на объект раздела, который, в свою очередь, не связан с файлом. Система использует в качестве резервного хранилища часть страничного файла. Такие варианты выделения не вносят вклад в квоту процесса на страничный файл.
- **Разделы отображаемой памяти, копируемые при записи** (copy-on-write regions of mapped memory), даже если они связаны с обычными отображаемыми файлами. Отображаемый файл предоставляет резервное хранилище для своего собственного неизмененного содержимого. Но как только страница в разделе, копируемом при записи, изменится, она больше не сможет использовать исходный отображаемый файл в качестве резервного хранилища. Она должна храниться в оперативной памяти или в страничном файле. Такие варианты выделения не вносят свой вклад в квоту процесса на страничный файл.
- **Невыгружаемый и выгружаемый пулы** (nonpaged and paged pool), а также другие варианты выделения памяти в системном пространстве, которые не поддерживаются явно связанными с ними файлами. Следует учесть, что свой вклад в показатель подтверждения вносят даже свободные на данный момент разделы пулов системной памяти. Невыгружаемые разделы учитываются в показателе подтверждения даже при том, что они никогда не будут записаны в страничный файл, поскольку они постоянно снижают объем оперативной памяти, доступный для закрытых страничных данных. Такие варианты выделения не вносят свой вклад в квоту процесса на страничный файл.
- **Стеки ядра** (kernel stacks) при выполнении в режиме ядра.
- **Таблицы страниц** (page tables), большинство из которых сами по себе доступны для выгрузки, не поддерживаются отображаемыми файлами. Но даже если они не доступны для выгрузки, они занимают оперативную память. Поэтому пространство, требуемое для них, вносит свой вклад в показатель подтверждения.
- **Пространство под таблицы страниц, которое еще фактически не выделено.** Как вы вскоре увидите, существуют большие области виртуального пространства, которые были определены, но на которые еще не было обращений (например, закрытое подтвержденное виртуальное пространство), и для их описания системе пока еще не нужно создавать таблицы страниц. Тем не менее пространство для этих таблиц страниц (еще не существующих) учитывается в показателе подтверждения, чтобы гарантировать возможность создания таблиц страниц по мере надобности.
- **Варианты выделения физической памяти через API-функции оконного расширения адресов** (Address Windowing Extension, AWE). Как упоминалось ранее, в этом варианте физическая память потребляется напрямую.

Для многих из этих позиций показатель подтверждения может быть показателем потенциального, но не фактического использования хранилища. Например, стра-

ница закрытой подтвержденной памяти фактически не занимает ни физическую страницу оперативной памяти, ни эквивалентное ей пространство страничного файла до тех пор, пока к ней не будет сделано хотя бы одно обращение. До этого страница останется *страницей, подлежащей заполнению нулями* (см. далее). Тем не менее такие страницы учитываются в показателе подтверждения при первоначальном создании виртуального пространства. Этим обеспечивается доступность для таких страниц физического пространства хранилища на тот случай, если позже к ним будет сделано обращение.

К разделу отображаемого файла, копируемого при записи, предъявляются аналогичные требования. Пока процесс ведет запись в раздел, все страницы в нем поддерживаются отображаемым файлом. Но процесс может вести запись в любую из страниц раздела в любое время, и когда это происходит, такие страницы рассматриваются как закрытые страницы процесса. После этого поддерживающим их хранилищем становится страничный файл. Учет таких разделов в показателе подтверждения при первоначальном создании раздела гарантирует для них наличие закрытого хранилища в дальнейшем при обращениях, связанных с записью данных.

Особый интерес представляет собой случай резервирования закрытой памяти с последующим ее подтверждением. При создании зарезервированного раздела с помощью функции `VirtualAlloc` фактический виртуальный раздел в показателе подтверждения не учитывается. В Windows 8, Server 2012 и более ранних версиях он учитывается для любых новых страниц с таблицами страниц, которые потребуются для хранения описания раздела, хотя этого раздела пока еще может и не быть. Начиная с Windows 8.1 и Server 2012 R2, иерархии таблиц страниц для зарезервированных областей не учитываются немедленно; это означает, что огромные области зарезервированной памяти могут выделяться без создания излишней нагрузки на таблицы страниц. Этот аспект играет особенно важную роль в некоторых механизмах безопасности — таких, как CFG (Control Flow Guard — см. главу 7). Если позже раздел или его часть подтверждается, размер раздела учитывается в системном показателе подтверждения (а также в квоте процесса на страничный файл).

Иначе говоря, когда система успешно завершает (к примеру) вызов функции `VirtualAlloc` или `MapViewOfFile`, она берет на себя обязательства, что нужное хранилище будет доступно при возникновении необходимости, даже если на данный момент такой необходимости нет. Таким образом, последующие обращения к памяти в выделенном разделе никогда не получат отказа из-за нехватки пространства хранилища. (Конечно, отказ может произойти по другим причинам: из-за защиты страницы, освобождения раздела и т. д.) Механизм подсчета показателя подтверждения позволяет системе выполнять свои обязательства.

Показатель подтверждения фигурирует среди счетчиков монитора производительности под названием **Память: Байт выделенной виртуальной памяти** (Memory: Committed Bytes). Он также представлен первым из двух чисел на вкладке **Быстродействие** (Performance) диспетчера задач рядом с надписью **Выделено** (Commit) — второе число показывает лимит подтверждения. В программе Process Explorer

показатель подтверждения выводится на вкладке Memory окна System Information в области Commit Charge рядом с надписью Current.

Квота процесса на страничный файл фигурирует в составе счетчиков монитора производительности под названием Процесс: Байт файла подкачки (Process: Page File Bytes). Те же самые данные выводятся в счетчике производительности Процесс: Байт исключительного пользования (Process: Private Bytes). (Ни одно из этих названий не дает точного описания истинного назначения счетчика.)

Если показатель подтверждения когда-либо достигнет лимита подтверждения, диспетчер памяти попытается увеличить лимит подтверждения, увеличив размер одного или нескольких страничных файлов. Если это невозможно, последующие попытки выделения виртуальной памяти на базе показателя подтверждения окажутся неудачными, пока не освободится какая-то часть подтвержденной памяти. Счетчики производительности, перечисленные в табл. 5.13, позволяют проанализировать показатель использования закрытой подтвержденной памяти, относящийся ко всей системе, к отдельному процессу или к отдельному страничному файлу.

Таблица 5.13. Счетчики производительности, относящиеся к подтвержденной памяти и страничным файлам

Счетчик производительности	Описание
Память: Байт выделенной виртуальной памяти (Memory: Committed Bytes)	Количество подтвержденных байтов виртуальной (не зарезервированной) памяти. Это количество может не отражать показатель использования страничного файла, поскольку в него включены закрытые подтвержденные страницы в физической памяти, которые никогда не выгружаются. Скорее оно отражает общий объем, который должен поддерживаться пространством страничного файла и/или оперативной памяти
Память: Предел выделенной виртуальной памяти (Memory: Commit Limit)	Количество байтов виртуальной памяти, которое может быть подтверждено без увеличения размера страничных файлов; если страничные файлы могут увеличиваться в размерах, этот лимит может меняться
Процесс: Байт файла подкачки (Process: Page File Quota)	Вклад процесса в показание счетчика Память: Байт выделенной виртуальной памяти (Memory: Committed Bytes)
Процесс: Байт исключительного пользования (Process: Private Bytes)	То же самое, что счетчик Процесс: Байт файла подкачки (Process: Page File Quota)
Процесс: Рабочий набор (частный) (Process: Working Set—Private)	Составляющая показания счетчика Процесс: Байт файла подкачки (Process: Page File Quota), которая в данный момент находится в оперативной памяти и к которой можно обратиться без получения ошибки страницы. Также является составляющей показания счетчика Процесс: Рабочий набор (Process: Working Set)
Процесс: Рабочий набор (Process: Working Set)	Составляющая показания счетчика Процесс: Байт виртуальной памяти (Process: Virtual Bytes), которая в данный момент находится в оперативной памяти и к которой можно обратиться без получения ошибки страницы

Счетчик производительности	Описание
Процесс: Байт виртуальной памяти (Process: Virtual Bytes)	Общий объем виртуальной памяти, выделенной процессу, включая отображаемые области, закрытые подтвержденные области и закрытые зарезервированные области
Файл подкачки: % использования (Paging File: % Usage)	Процент текущего используемого пространства страничного файла
Файл подкачки: % использования (пик) (Paging File: % Usage Peak)	Наивысшее зафиксированное значение счетчика Файл подкачки: % использования (Paging File: % Usage)

Показатель подтверждения и размер страничного файла

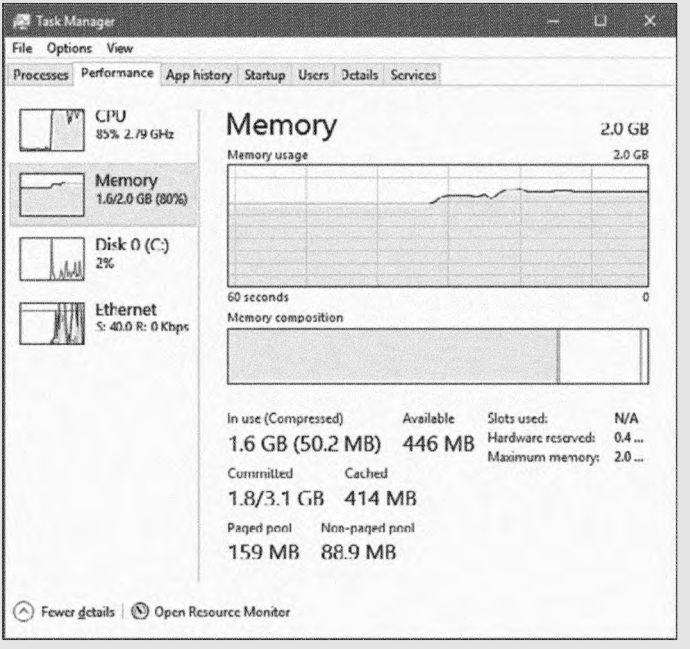
Счетчики из табл. 5.13 могут помочь в выборе нестандартного размера страничного файла. Политика по умолчанию основывается на объеме рабочей оперативной памяти, доступной большинству машин, но зависит от рабочей нагрузки при работе со страничным файлом, размер которого в результате может оказаться завышенным или заниженным.

Чтобы узнать, какой размер страничного файла действительно необходим вашей системе, нужно основываться на усредненном значении для приложений, которые запускаются после начальной загрузки системы. Для этого можно изучить пиковый показатель подтверждения в программе Process Explorer на вкладке Memory окна System Information. Это число представляет пиковый объем пространства страничного файла с момента завершения начальной загрузки системы, который понадобится, если системе придется выгрузить в него основную часть закрытой подтвержденной виртуальной памяти (что случается довольно редко).

Если страничный файл в вашей системе слишком велик, система вряд ли воспользуется всем его объемом. Иными словами, увеличение размера страничного файла не повлияет на производительность системы, это будет просто означать, что она сможет иметь больше подтвержденной виртуальной памяти. Если страничный файл слишком мал для усредненного значения числа выполняемых приложений, можно получить предупреждение об ошибке исчерпания объема виртуальной памяти (system running low on virtual memory). В таком случае сначала нужно проверить, нет ли у процесса утечки памяти, изучив показания счетчика Процесс: Байт исключительного пользования (Process: Private Bytes). Если процессы с утечкой найдены не будут, следует проверить размер системного выгружаемого пула — если драйвер устройства допускает утечку пространства выгружаемого пула, это также может объяснить причину ошибки. (Приемы поиска и устранения утечек пула приведены в эксперименте «Поиск и устранение утечек пула» раздела «Кучи режима ядра».)

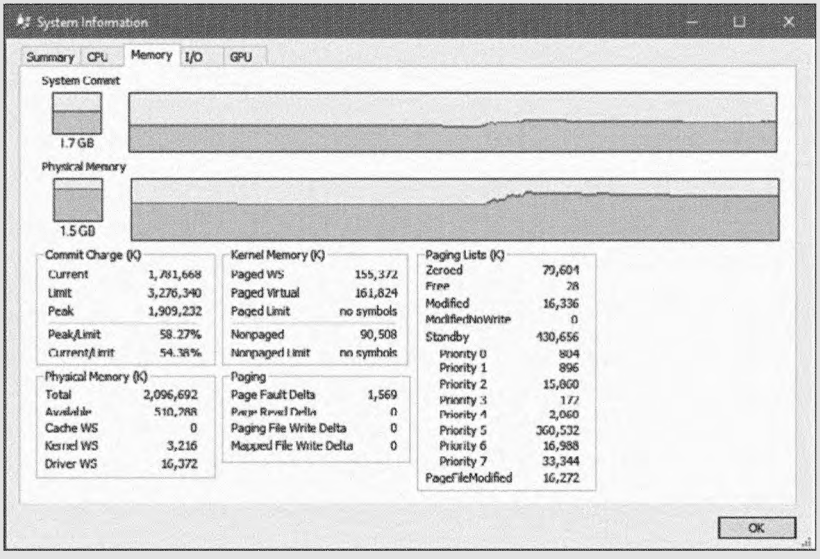
ЭКСПЕРИМЕНТ: ПРОСМОТР ПОКАЗАТЕЛЯ ИСПОЛЬЗОВАНИЯ СТРАНИЧНОГО ФАЙЛА С ПОМОЩЬЮ ДИСПЕТЧЕРА ЗАДАЧ

Данные о расходовании подтвержденной памяти можно также увидеть в диспетчере задач на вкладке Быстродействие (Performance). Там расположены счетчики, имеющие отношение к страничным файлам.



Общий объем подтвержденной системной памяти выводится с надписью Committed в виде двух чисел. Первое число показывает *потенциально возможный*, а не текущий показатель использования страничного файла. То есть оно показывает, какой объем пространства страничного файла будет использован, если понадобится выгрузить на него сразу всю закрытую подтвержденную виртуальную память, имеющуюся в системе. Второе число показывает *лимит подтверждения* — максимальный объем виртуальной памяти, который может поддерживаться системой (сюда относится виртуальная память, поддерживаемая как физической памятью, так и страничными файлами). По сути, лимит подтверждения — это объем оперативной памяти плюс текущий размер страничных файлов. Таким образом, в нем не учитывается возможное увеличение размеров страничных файлов.

В окне System Information программы Process Explorer можно найти дополнительную информацию об использовании системной подтвержденной памяти, а именно — пиковый процентный показатель в сравнении с лимитом и текущий показатель использования в сравнении с лимитом.



Стеки

Программный поток должен иметь доступ к временной области памяти для хранения параметров функций, локальных переменных и адреса возврата после вызова функции. Эта часть памяти называется *стеком* (stack). Диспетчер памяти в Windows предоставляет каждому потоку два стека, пользовательский стек (user stack) и стек ядра (kernel stack), есть также стеки для каждого процессора, которые называются DPC-стеками (DPC stacks). В главе 2 уже было описано, как системные функции заставляют программные потоки переключаться с пользовательского стека на стек ядра. А теперь мы рассмотрим ряд дополнительных функций, предоставляемых диспетчером памяти для эффективного использования памяти стека.

Пользовательские стеки

При создании программного потока диспетчер памяти автоматически резервирует предопределенный объем виртуальной памяти, который по умолчанию составляет 1 Мбайт. Этот объем может быть задан при вызове функции `CreateThread` или `CreateRemoteThread(Ex)` либо в процессе компиляции приложения с помощью ключа `/STACK:`*`зарезервированный_объем`* компилятора Microsoft C/C++, который сохраняет эту информацию в заголовке образа. Несмотря на резервирование пространства объемом 1 Мбайт, подтверждается только первая страница стека (если только в PE-заголовке образа не указано иное) наряду со сторожевой страницей. Когда стек потока разрастется и достигнет сторожевой страницы, выдается исключение, которое приводит к попытке выделения еще одной сторожевой страницы.

Благодаря этому механизму пользовательский стек не занимает одномоментно всю подтвержденную память объемом 1 Мбайт, а растет по мере надобности. (Но обратно стек никогда не сжимается.)

ЭКСПЕРИМЕНТ: СОЗДАНИЕ МАКСИМАЛЬНОГО КОЛИЧЕСТВА ПОТОКОВ

Поскольку для каждого 32-разрядного процесса объем доступного пользовательского адресного пространства составляет всего 2 Гбайт, относительно большой объем, резервируемый для стека каждого программного потока, позволяет легко вычислить максимальное количество программных потоков, поддерживаемых процессом, — их немногим менее 2048 для общего объема памяти, составляющего примерно 2 Гбайт (если только не используется BCD-параметр increaseuserva, а образ не поддерживает большое адресное пространство). Если заставить каждый новый поток довольствоваться минимально возможным резервируемым значением стека в 64 Кбайт, этот предел может возрасти приблизительно до 30 000 потоков, что можно самостоятельно проверить при помощи утилиты TestLimit, разработанной в Sysinternals. Пример выводимых этой утилитой данных:

```
C:\Tools\Sysinternals>Testlimit.exe -t -n 64

Testlimit v5.24 - test Windows limits
Copyright (C) 2012-2015 Mark Russinovich
Sysinternals - www.sysinternals.com

Process ID: 17260

Creating threads with 64 KB stacks...
Created 29900 threads. Lasterror: 8
```

Казалось бы, при проведении этого эксперимента для 64-разрядной версии Windows (где доступно 128 Тбайт пользовательского адресного пространства) будет выводиться информация о потенциально возможных сотнях тысяч создаваемых потоков (если, конечно, для этого хватит памяти). Но, как ни странно, утилита TestLimit создаст еще меньше потоков, чем на 32-разрядной машине, поскольку Testlimit.exe является 32-разрядным приложением и запускается в среде Wow64. (Дополнительные сведения о Wow64 можно найти в главе 8 части 2.) В результате у каждого потока будет создан не только его 32-разрядный стек в среде Wow64, но и 64-разрядный стек, и оба будут расходовать в два с лишним раза больше памяти, хотя потоку по-прежнему будет выделяться адресное пространство объемом всего лишь 2 Гбайт. Чтобы получить правильные результаты тестирования возможностей создания потоков на 64-разрядной версии Windows, нужно вместо прежней утилиты воспользоваться ее 64-разрядной версией Testlimit64.exe.

Учтите, что прервать работу утилиты TestLimit удастся только из Process Explorer или диспетчера задач. Комбинация клавиш Ctrl+C для этого не сработает, поскольку вызываемая этой комбинацией операция сама по себе создает новый программный поток, что будет невозможно сделать по причине исчерпания памяти процесса.

Стеки ядра

Хотя размер пользовательского стека обычно составляет 1 Мбайт, объем памяти, предназначенной стеку ядра, значительно меньше: 12 Кбайт для платформы x86 и 16 Кбайт для платформы x64, далее следует сторожевая страница (что в целом составляет 16 или 20 Кбайт виртуального адресного пространства). Считается, что код, запускаемый в режиме ядра, имеет меньше рекурсий, чем пользовательский код, а также более эффективно задействует переменные и сохраняет размеры буфера стека на низком уровне. Поскольку стеки ядра находятся в системном адресном пространстве (которое совместно используется всеми процессами), расходование ими памяти имеет более существенное влияние на систему.

Хотя код ядра обычно не рекурсивен, взаимодействия между графическими системными вызовами обрабатываются драйвером Win32k.sys, и его последующие обратные вызовы кода, выполняемого в пользовательском режиме, могут стать причиной рекурсивных повторных вхождений в ядро в том же самом стеке ядра. По этой причине Windows предоставляет механизм динамического расширения и сокращения стека ядра от его начального размера. Поскольку каждый дополнительный вызов графики выполняется из одного и того же потока, выделяется еще один стек ядра размером 16 Кбайт (где-нибудь в системном адресном пространстве; диспетчер памяти позволяет переходить между стеками при приближении к сторожевой странице). Как показано на рис. 5.28, когда каждый вызов возвращает управление вызывавшему коду (раскрутка стека вызовов), диспетчер памяти освобождает дополнительно выделенный стек ядра. Этот механизм позволяет надежно поддерживать рекурсивные системные вызовы и эффективно использовать системное адресное пространство, а также, если необходимо, его могут применять разработчики драйверов при выполнении рекурсивных внешних вызовов с помощью API-функции `KeExpandKernelStackAndCallout(Ex)`.

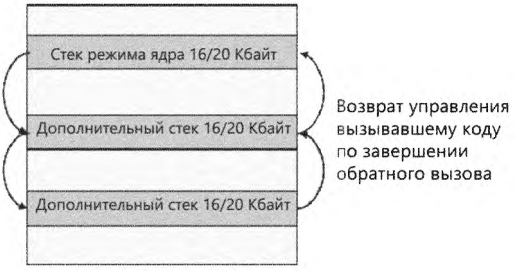


Рис. 5.28. Переходы между стеками ядра

ЭКСПЕРИМЕНТ: ПРОСМОТР ПОКАЗАТЕЛЯ ИСПОЛЬЗОВАНИЯ СТЕКА ЯДРА

Для вывода на экран сведений о текущей занятости физической памяти стеками ядра можно воспользоваться программой RamMap из пакета Sysinternals. Следующий снимок экрана сделан на вкладке Use Counts:

Driver Locked	3,419,852 K	3,419,852 K	
Kernel Stack	47,496 K	41,380 K	6,116 K
Unused	10,773,740 K	77,504 K	

Чтобы просмотреть данные об использовании стека режима ядра, выполните следующие действия.

1. Повторите предыдущий эксперимент с TestLimit, но пока не завершайте TestLimit.
2. Переключитесь на RamMap.
3. Откройте меню File и выберите команду Refresh (или нажмите F5). Размер стека ядра должен увеличиться:

Driver Locked	3,419,852 K	3,419,852 K	
Kernel Stack	346,076 K	339,916 K	6,160 K
Unused	10,318,232 K	77,680 K	

Запуск утилиты TestLimit еще несколько раз (без закрытия предыдущих экземпляров) приведет на 32-разрядной системе к быстрому истощению физической памяти. Это ограничение ведет к одному из основных общесистемных ограничений на количество 32-разрядных потоков.

DPC-стек

Для каждого процесса Windows поддерживает DPC-стек, доступный для использования системой при выполнении отложенных вызовов процедур (DPC). При таком подходе DPC-код изолируется от стека ядра текущего программного потока (не имеющего отношения к фактической операции DPC-вызова, поскольку DPC-вызовы выполняются в контексте произвольного потока; см. главу 6). DPC-стек настраивается в качестве исходного стека для обслуживания в ходе системного вызова инструкции `sysenter`(x86), `svc` (ARM) или `syscall`(x64). За переключение стека при выполнении этих инструкций отвечает центральный процессор, который действует на основании состояния одного из зависящих от модели регистров (Model-Specific Register, MSR на платформах x86/x64). Тем не менее Windows не в состоянии перепрограммировать MSR для каждого контекстного переключения, поскольку это очень затратная операция. Поэтому Windows задает в MSR указатель на DPC-стек каждого процессора.

Дескрипторы виртуальных адресов

Чтобы узнать, когда загружать страницы в память, диспетчер памяти использует алгоритм *подкачки страниц по требованию* (demand-paging algorithm), ожидая перед извлечением страницы с диска ссылки потока на адрес и получение им ошибки отсутствия страницы. По аналогии с копированием при записи, подкачка страниц по требованию является одной из форм *отложенного вычисления* (lazy

evaluation), при котором решение задачи откладывается до момента, когда в нем возникнет прямая необходимость.

Диспетчер памяти выполняет отложенное вычисление не только при внесении страниц в память, но и при создании таблиц страниц, необходимых для описания новых страниц. Например, когда поток подтверждает большую область виртуальной памяти с помощью функции `VirtualAlloc` или `VirtualAllocExNuma`, диспетчер памяти может тут же создать таблицы страниц, отвечающие за доступ ко всему диапазону выделенной памяти. А что, если к части диапазона вообще не будет обращений? Создание таблиц страниц для всего диапазона обернется напрасной тратой ресурсов. Вместо этого диспетчер памяти откладывает создание таблицы страниц до тех пор, пока поток не получит ошибку страницы, и только потом создает таблицу страниц для этой страницы. Такой метод существенно повышает производительность процессов, выполняющих резервирование и/или подтверждение больших объемов памяти, к которой редко обращаются.

Виртуальное адресное пространство, которое будет занято такими еще не существующими таблицами страниц, входит в квоту процесса на страничный файл и в системный показатель подтверждения. Тем самым гарантируется, что пространство будет доступно для этих таблиц при их фактическом создании. Благодаря алгоритму отложенного вычисления выделение даже больших блоков памяти является довольно быстрой операцией. Когда поток выделяет память, диспетчер памяти должен нести ответственность за диапазон адресов, используемых потоком. Для этого диспетчер памяти содержит еще один набор структур данных, призванных отслеживать, какие виртуальные адреса зарезервированы в адресном пространстве процесса, а какие нет. Эти структуры данных известны как *дескрипторы виртуальных адресов* (VAD, Virtual Address Descriptors). Память для VAD-дескрипторов выделяется в невыгружаемом пуле.

Дескрипторы виртуальных адресов процесса

Для каждого процесса диспетчер памяти поддерживает набор VAD-дескрипторов, описывающий состояние адресного пространства процесса. VAD-дескрипторы сведены в самобалансирующееся АВЛ-дерево (названное по первым буквам фамилий его создателей — Адельсона-Вельского и Ландиса), которое обладает оптимальной сбалансированностью. Это приводит к меньшему среднему количеству сравнений при поиске VAD-дескриптора, соответствующего виртуальному адресу. Один дескриптор виртуального адреса выделяется каждому виртуально непрерывному диапазону несвободных виртуальных адресов, имеющих одинаковые характеристики (при этом зарезервированные адреса отличаются и от подтвержденных, и от отображаемых, учитывается также защита памяти и т. д.). Схематически VAD-дерево показано на рис. 5.29.

Когда процесс резервирует адресное пространство или отображает представление раздела, диспетчер памяти создает VAD-дескриптор для хранения любой информа-

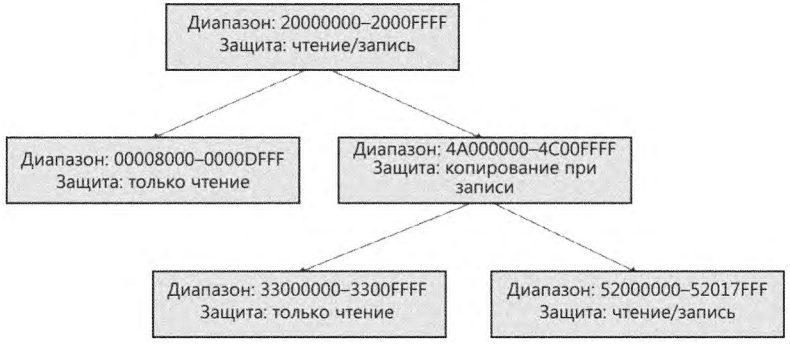


Рис. 5.29. Дескрипторы виртуальных адресов

ции, предоставленной запросом на выделение памяти: диапазона зарезервированных адресов, признаков того, является ли этот диапазон совместно используемым или закрытым, может ли дочерний процесс унаследовать содержимое диапазона, применима ли к страницам диапазона защита страниц.

При первом обращении потока к адресу диспетчер памяти должен создать PTE-запись для страницы, содержащейся по этому адресу. Для этого он находит VAD-дескриптор, в адресный диапазон которого входит запрашиваемый адрес, и использует найденную информацию для заполнения PTE-записи. Если адрес не входит в диапазон, перекрываемый VAD-дескриптором, или в диапазон зарезервированных, но не подтвержденных адресов, диспетчер памяти узнает о том, что поток не выделил память перед попыткой ее использования, и поэтому генерирует нарушение прав доступа.

ЭКСПЕРИМЕНТ: ПРОСМОТР ДЕСКРИПТОРОВ ВИРТУАЛЬНЫХ АДРЕСОВ

Для просмотра VAD-дескрипторов заданного процесса можно воспользоваться командой `!vad` отладчика ядра. Но сначала с помощью команды `!process` нужно найти адрес корневого элемента VAD-дерева, а затем указать этот адрес в команде `!vad`, как в следующем примере просмотра VAD-дерева для процесса, в котором выполняется Explorer.exe:

```
lkd> !process 0 1 explorer.exe
PROCESS ffffc8069382e080
    SessionId: 1  Cid: 43e0    Peb: 00bc5000  ParentCid: 0338
    DirBase: 554ab7000  ObjectTable: ffffda8f62811d80  HandleCount: 823.
    Image: explorer.exe
    VadRoot ffffc806912337f0 Vads 505 Clone 0 Private 5088. Modified 2146.
            Locked 0.
...
lkd> !vad ffffc8068ae1e470
VAD           Level     Start       End Commit
ffffc80689bc52b0  9       640       64f      0 Mapped       READWRITE
Pagefile section, shared commit 0x10
ffffc80689be6900  8       650       651      0 Mapped       READONLY
Pagefile section, shared commit 0x2
```

```
ffffc80689bc4290  9       660       675      0 Mapped       READONLY
Pagefile section, shared commit 0x16
ffffc8068ae1f320  7       680       6ff     32 Private      READWRITE
ffffc80689b290b0  9       700       701      2 Private      READWRITE
ffffc80688da04f0  8       710       711      2 Private      READWRITE
ffffc80682795760  6       720       723      0 Mapped       READONLY
Pagefile section, shared commit 0x4
ffffc80688d85670 10       730       731      0 Mapped       READONLY
Pagefile section, shared commit 0x2
ffffc80689bdd9e0  9       740       741      2 Private      READWRITE
ffffc80688da57b0  8       750       755      0 Mapped       READONLY
\Windows\en-US\explorer.exe.mui
...
Total VADs: 574, average level: 8, maximum depth: 10
Total private commit: 0x3420 pages (53376 KB)
Total shared commit:  0x478 pages (4576 KB)
```

Чередование дескрипторов виртуальных адресов

Как правило, драйверу видеокарты нужно копировать данные из графического приложения пользовательского режима в другую системную память разного происхождения, включая память видеокарты и память AGP-порта, причем обе имеют разные атрибуты кэширования и разные адреса. Чтобы дать этим разным представлениям памяти возможность быстро отображаться на процесс и поддерживать разные атрибуты кэширования, диспетчер памяти реализует чередующиеся дескрипторы виртуальных адресов (rotate virtual address descriptors), позволяющие видеодрайверам осуществлять прямую передачу данных с использованием графического процессора (Graphical Processing Unit, GPU) и по мере надобности менять для страниц представления процесса ненужную память на нужную. Пример того, как один и тот же виртуальный адрес может поочередно переходить между видеопамятью и виртуальной памятью, показан на рис. 5.30.

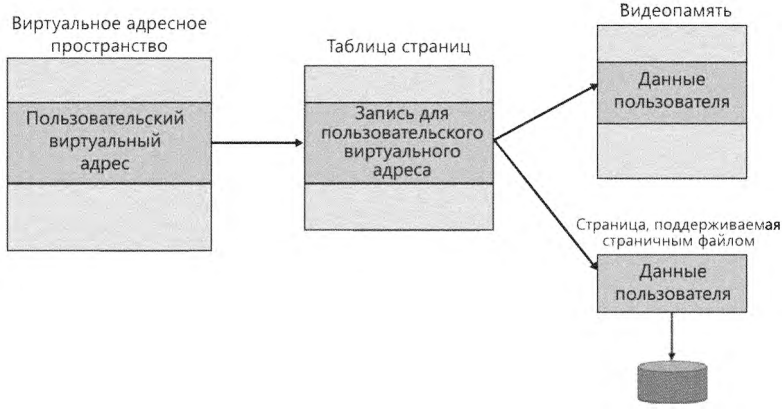


Рис. 5.30. Чередующиеся дескрипторы виртуальных адресов

NUMA

Каждый новый выпуск Windows предлагает новые варианты совершенствования диспетчера памяти для наилучшего использования машин с архитектурой неоднородного доступа к памяти (Non Uniform Memory Architecture, NUMA), например больших серверных систем (а также рабочих SMP-станций на процессорах Intel i7 и AMD Opteron). Поддержка технологии NUMA в диспетчере памяти наделяет его средствами получения информации об узлах, их местоположении, топологии и издержках доступа, что позволяет приложениям и драйверам пользоваться возможностями технологии NUMA, абстрагируясь от базовых деталей оборудования.

При инициализации диспетчера памяти он вызывает функцию `MiComputeNumaCosts` для выполнения различных операций, касающихся страниц и кэша на различных узлах с последующим вычислением затрачиваемого на завершение этих операций времени. На основании этой информации диспетчер памяти создает граф издержек доступа к узлу (расстояния между узлом и другими узлами системы). Когда системе потребуются страницы для заданной операции, она обращается к графу, чтобы выбрать наиболее оптимальный узел (т. е. ближайший). Если на этом узле нет доступной памяти, выбирается следующий ближайший узел и т. д.

Хотя диспетчер памяти гарантирует, что память по возможности будет выделяться процессорным узлом того программного потока, который реализует выделение, — так называемым *идеальным узлом* (ideal node), он также предоставляет API-функции `VirtualAllocExNuma`, `CreateFileMappingNuma`, `MapViewOfFileExNuma` и `AllocateUserPhysicalPagesNuma`, позволяющие приложениям выбирать собственный узел.

Идеальный узел используется не только при выделении памяти приложением, но и в ходе операций ядра и ошибок страницы. Например, когда поток выполняется на неидеальном процессоре и сталкивается с ошибкой отсутствия страницы, диспетчер памяти не станет трогать текущий узел, а вместо этого выделит память из идеального узла потока. Хотя это может привести к увеличению времени доступа, пока поток по-прежнему выполняется на данном центральном процессоре, в общем и целом, как только поток вернется на свой идеальный узел, доступ к памяти станет оптимальным. В любом случае, если идеальный узел не располагает свободными ресурсами, в качестве идеального будет выбран не какой-то случайный узел, а ближайший узел. Впрочем, драйверы, как и пользовательские приложения, могут указать собственный узел при помощи API-функции `MmAllocatePagesforMdlEx` или `MmAllocateContiguousMemorySpecifyCacheNode`.

Различные пулы и структуры данных диспетчера памяти также оптимизируются с целью получения преимуществ NUMA-узлов. Для поддержки невыгружаемого пула диспетчер памяти старается равномерно использовать физическую память всех узлов системы. После выделения памяти для невыгружаемого пула диспетчер памяти использует идеальный узел в качестве индекса при выборе диапазона адресов виртуальной памяти внутри невыгружаемого пула, который соответствует физической памяти, принадлежащей этому узлу. Кроме того, для эффективного

использования этих вариантов конфигурации памяти для каждого NUMA-узла создаются списки свободных пулов узлов. Кроме невыгружаемого пула точно так же между всеми узлами распределяются системный кэш и системные PTE-записи, а также ассоциативные списки диспетчера памяти.

И наконец, когда система удовлетворяет свои потребности в страницах, заполненных нулями, она делает это параллельно на разных NUMA-узлах, создавая потоки на родственных по типу оборудования NUMA-узлах, соответствующих узлам, на которых находится физическая память. Механизмы предварительной выборки и супервыборки (см. далее) также используют идеальный узел целевого процесса, а разрешимые ошибки страницы заставляют страницы мигрировать на идеальный узел, на котором выполняется поток, столкнувшийся с такой ошибкой.

Объекты разделов

Ранее при описании общей памяти уже отмечалось, что *объект раздела*, который в подсистеме Windows называется *объектом отображения файла* (file mapping object), представляет собой блок памяти, который может совместно использоваться двумя и более процессами. Объект раздела может быть отображен на страничный файл или на какой-нибудь другой файл на диске.

Исполнительная система применяет разделы для загрузки в память исполняемых образов, а диспетчер кэша использует их для доступа к данным в кэшированном файле. (Дополнительные сведения о том, как диспетчер кэша задействует объекты разделов, можно найти в главе 14 части 2.) Объекты разделов можно также использовать для отображения файла внутри адресного пространства процесса. После этого к файлу можно будет обращаться как к большому массиву, отображая различные представления объекта раздела и читая их из памяти и записывая в память, а не из файла и в файл (такие действия называются *вводом/выводом отображаемого файла*). При обращении программы к недостоверной странице (к странице, отсутствующей в физической памяти) происходит ошибка страницы, и диспетчер памяти автоматически помещает страницу в память из отображаемого (или страничного) файла. Если приложение вносит в страницу изменения, диспетчер памяти записывает изменения в файл в ходе обычных операций подкачки (или же приложение может сбросить представление, воспользовавшись Windows-функцией `FlushViewOfFile`).

Объекты разделов, так же как и другие объекты, выделяются и освобождаются диспетчером объектов. Диспетчер объектов создает и инициализирует заголовок объекта, который используется им для управления объектами, а диспетчер памяти определяет тело объекта раздела. Диспетчер памяти также реализует службы, которые могут вызываться потоками пользовательского режима для извлечения и изменения атрибутов, хранящихся в теле объектов разделов. Структура объекта раздела показана на рис. 5.31, а уникальные атрибуты, хранящиеся в объектах разделов, перечислены в табл. 5.14.

Рис. 5.31. Объект раздела

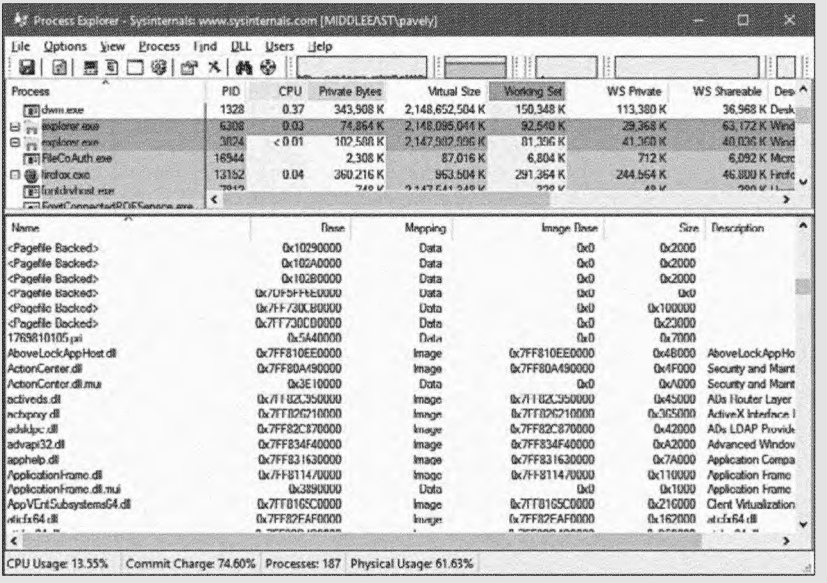
Таблица 5.14. Атрибуты тела объекта раздела

Атрибут	Назначение
Максимальный размер	Наибольший размер, до которого может дорасти раздел, указанный в байтах; если объект используется для отображаемого файла, то максимальный размер является размером этого файла
Защита страницы	Страничная защита памяти, назначаемая всем страницам раздела при его создании
Страничный/отображаемый файл	Показывает, создан ли раздел пустым (с поддержкой страничного файла — как упоминалось ранее, разделы, поддерживаемые страничными файлами, используют ресурсы этого файла, только когда страницы нужно записать на диск) или загружен с файлом (с поддержкой отображаемого файла)
Базовый или нет	Показывает, является ли раздел базовым, который должен размещаться в одном виртуальном адресном пространстве для всех совместно использующих его процессов, или же небазовым, который может размещаться по разным виртуальным адресам для разных процессов

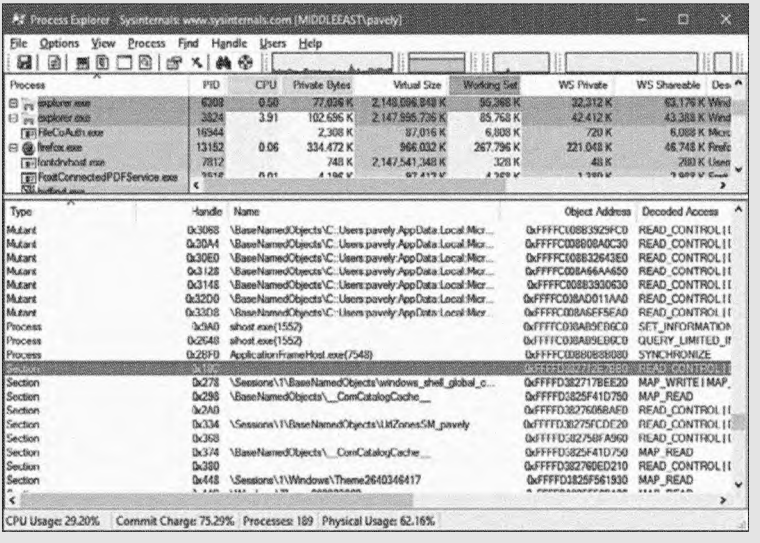
ЭКСПЕРИМЕНТ: ПРОСМОТР ОБЪЕКТОВ РАЗДЕЛОВ

Для просмотра файлов, отображаемых процессом, можно воспользоваться программой Process Explorer из пакета Sysinternals. Выполните следующие действия.

1. Откройте меню View, выберите команду Lower Pane View и выберите вариант DLLs.
2. Откройте меню View, выберите команду Select Columns, выберите вариант DLL и включите столбец Mapping Type.
3. Обратите внимание: в столбце Mapping файлы помечаются как данные (Data). Это отображаемые файлы, а не DLL-библиотеки или другие файлы, которые загрузчик образов загружает как модули. Объекты разделов, поддерживаемые страничным файлом, обозначаются в столбце Name меткой <Pagefile Backed>. В противном случае выводится имя файла.



Существует и другой способ просмотра объектов разделов: переключитесь на представление дескрипторов (откройте меню View, выберите команду Lower Pane View и выберите вариант Handles) и поищите объекты типа Section. На следующем снимке экрана выводится имя объекта (если оно существует). Это имя не совпадает с именем поддерживающего файла (если он есть); это имя присваивается разделу в пространстве имен диспетчера объектов. (За дополнительной информацией о диспетчере объектов обращайтесь к главе 8 части 2.) Двойной щелчок в строке выводит дополнительную информацию об объекте (например, количество открытых дескрипторов и его дескриптор безопасности).



Структуры данных, поддерживаемые диспетчером памяти и описывающие отображаемые разделы, показаны на рис. 5.32. Эти структуры гарантируют согласованность данных, считываемых из отображаемых файлов, независимо от типа доступа (открытый файл, отображаемый файл и т. д.). Для каждого открытого файла (представленного объектом файла) существует отдельная структура указателей объекта раздела (section object pointers). Эта структура играет ключевую роль в поддержании целостности данных для всех типов доступа к файлам, а также для реализации кэширования файлов. Структура указателей объекта раздела указывает на одну или две области управления (control areas). Одна область управления служит для отображения файла, когда к нему обращаются как к файлу данных, другая — для отображения файла, когда он запускается как исполняемый образ. Область управления, в свою очередь, указывает на структуры подразделов (subsections), которые описывают информацию отображения для каждого раздела файла (только для чтения, для чтения и записи, с копированием при записи и т. д.). Область управления также указывает на структуру сегмента (segment), созданную в выгружаемом пуле, которая, в свою очередь, указывает на прототипные PTE-записи, предназначенные для отображения реальных страниц, отображаемых объектом раздела. Как уже упоминалось, на эти прототипные PTE-записи указывают таблицы страниц процесса, а те, в свою очередь, отображают страницы, к которым были сделаны обращения.

Рис. 5.32. Внутренние структуры разделов

Хотя Windows гарантирует, что любой процесс, обращающийся к файлу (по чтению или записи), всегда будет видеть одни и те же согласованные данные, есть одна ситуация, при которой в физической памяти могут оказаться две копии страниц файла (но даже в этом случае все обратившиеся к файлу процессы получат самую последнюю копию, и согласованность данных будет соблюдена). Такое копирование может произойти, когда к файлу образа происходит обращение как к файлу данных (по чтению или по записи), а затем он запускается как исполняемый образ (например, когда образ подвергся компоновке с последующим запуском — у компоновщика будет открытый файл для доступа к нему как к файлу данных, а когда

образ будет запущен, загрузчик отобразит его в качестве исполняемого файла). Внутри системы происходят следующие действия:

1. Если исполняемый файл был создан с помощью API-функций файлового отображения (или диспетчером кэша), создается область управления данными, чтобы представлять страницы данных в файле образа, подвергшегося чтению или записи.
2. Когда образ запущен и для отображения образа как исполняемого создан объект раздела, диспетчер памяти обнаруживает, что указатели объекта раздела для файла образа указывают на область управления данными, и сбрасывает раздел. Этот шаг необходим, чтобы гарантировать, что любые измененные страницы будут записаны на диск перед обращением к образу через область управления образами.
3. Диспетчер памяти создает область управления для файла образа.
4. После того как образ начинает выполняться, доступ к его страницам (предназначенным только для чтения) вызывает ошибку обращения к файлу образа (или же страницы копируются непосредственно из файла данных, если соответствующая страница данных является резидентной).

Поскольку страницы, отображаемые областью управления данными, должны по-прежнему оставаться резидентными (в списке ожидающих использования), возникает именно этот единственный случай, при котором две копии одних и тех же данных находятся в памяти в двух разных страницах. Но это дублирование не вызывает проблем несогласованности данных, поскольку, как уже упоминалось, область управления данными уже осуществила сброс на диск, в результате страницы, считываемые из образа, соответствуют самым последним изменениям (и эти страницы никогда не записываются на диск).

ЭКСПЕРИМЕНТ: ПРОСМОТР ОБЛАСТЕЙ УПРАВЛЕНИЯ

Чтобы найти адрес структур области управления для файла, сначала нужно получить адрес интересующего вас объекта файла. Этот адрес можно получить с помощью отладчика ядра путем вывода дампа таблицы дескрипторов процессов, запустив команду `!handle` и обозначив адрес объекта файла. Еще одна команда, `!file`, отладчика ядра выводит основную информацию об объекте файла, но не показывает указатель на структуру указателей объекта раздела. Затем, используя команду `dt`, нужно отформатировать объект файла, чтобы получить адрес структуры указателей объекта раздела. Эта структура состоит из трех указателей: указателя на область управления данными, указателя на совместно используемое отображение кэша (см. главу 14 части 2) и указателя на область управления образом. Из структуры указателей объекта раздела можно получить адрес области управления для файла (если таковой имеется) и предоставить этот адрес команде `!ca`.

Например, если открыть файл приложения PowerPoint и с помощью команды `!handle` вывести таблицу дескрипторов для этого процесса, можно будет, как

здесь показано, найти открытый дескриптор, относящийся к PowerPoint-файлу (в этом можно убедиться при помощи текстового поиска). (Использование команды !handle рассматривается в разделе «Диспетчер объектов» главы 8.)

```
lkd> !process 0 0 powerpnt.exe
PROCESS ffffc8068913e080
    SessionId: 1  Cid: 2b64    Peb: 01249000  ParentCid: 1d38
    DirBase: 252e25000  ObjectTable: ffffda8f49269c40  HandleCount: 1915.
    Image: POWERPNT.EXE
lkd> .process /p ffffc8068913e080
Implicit process is now ffffc806'8913e080
lkd> !handle
...
0c08: Object: ffffc8068f56a630  GrantedAccess: 00120089 Entry:
            ffffda8f491d0020
Object: ffffc8068f56a630  Type: (ffffc8068256cb00) File
    ObjectHeader: ffffc8068f56a600 (new version)
        HandleCount: 1  PointerCount: 30839
        Directory Object: 00000000  Name: \WindowsInternals\7thEdition\
                          Chapter05\
diagrams.pptx {HarddiskVolume2}
...
```

Возьмите адрес объекта файла (865d2768) и отформатируйте его командой dt. Результат выглядит так:

```
lkd> dt nt!_file_object ffffc8068f56a630
   +0x000 Type             : 0n5
   +0x002 Size             : 0n216
   +0x008 DeviceObject     : 0xffffc806'8408cb40 _DEVICE_OBJECT
   +0x010 Vpb              : 0xffffc806'82feba00 _VPB
   +0x018 FsContext        : 0xffffda8f'5137cbd0 Void
   +0x020 FsContext2       : 0xffffda8f'4366d590 Void
   +0x028 SectionObjectPointer : 0xffffc806'8ec0c558 _SECTION_OBJECT_POINTERS
...
```

Возьмите адрес структуры указателей объекта разделов и отформатируйте его командой dt. Результат выглядит так:

```
lkd> dt nt!_section_object_pointers 0xffffc806'8ec0c558
   +0x000 DataSectionObject : 0xffffc806'8e838c10 Void
   +0x008 SharedCacheMap    : 0xffffc806'8d967bd0 Void
   +0x010 ImageSectionObject : (null)
```

Наконец, команда !ca используется для вывода управляющей области по ее адресу:

```
lkd> !ca 0xffffc806'8e838c10
ControlArea  @ ffffc8068e838c10
  Segment      ffffda8f4d97fdc0  Flink      ffffc8068ecf97b8  Blink
ffffc8068ecf97b8
  Section Ref               1  Pfn Ref               58  Mapped Views        2
  User Ref                  0  WaitForDel             0  Flush Count         0
  File Object  ffffc8068e5d3d50  ModWriteCount         0  System Views        2
  WritableRefs                0
```

```
  Flags (8080) File WasPurged   \WindowsInternalsBook\7thEdition\Chapter05\
                                 diagrams.pptx

Segment @ ffffda8f4d97fdc0
  ControlArea     ffffc8068e838c10  ExtendInfo    0000000000000000
  Total Ptes                   80
  Segment Size               80000  Committed                    0
  Flags (c0000) ProtectionMask

Subsection 1 @ ffffc8068e838c90
  ControlArea  ffffc8068e838c10  Starting Sector      0  Number Of Sectors  58
  Base Pte     ffffda8f48eb6d40  Ptes In Subsect     58  Unused Ptes         0
  Flags                     d  Sector Offset        0  Protection          6
  Accessed
  Flink        ffffc8068bb7fcf0  Blink   ffffc8068bb7fcf0  MappedViews       2

Subsection 2 @ ffffc8068c2e05b0
  ControlArea  ffffc8068e838c10  Starting Sector     58  Number Of Sectors  28
  Base Pte     ffffda8f3cc45000  Ptes In Subsect     28  Unused Ptes       1d8
  Flags                     d  Sector Offset        0  Protection          6
  Accessed
  Flink        ffffc8068c2e0600  Blink   ffffc8068c2e0600  MappedViews       1
```

Еще один способ вывода списка всех областей управления основан на использовании команды `!memusage`. Следующий фрагмент взят из выводимых этой командой данных (в системе с большим объемом памяти выполнение команды может занять много времени):

```
lkd> !memusage
 loading PFN database
loading (100% complete)

Compiling memory usage data (99% Complete).

             Zeroed:    98533 (  394132 kb)
               Free:     1405 (    5620 kb)
            Standby:   331221 ( 1324884 kb)
           Modified:    83806 (  335224 kb)
    ModifiedNoWrite:      116 (     464 kb)
       Active/Valid:  1556154 ( 6224616 kb)
         Transition:        5 (      20 kb)
          SLIST/Bad:     1614 (    6456 kb)
            Unknown:        0 (       0 kb)
              TOTAL:  2072854 ( 8291416 kb)

Dangling Yes Commit:      130 (     520 kb)
 Dangling No Commit:   514812 ( 2059248 kb)
  Building kernel map
  Finished building kernel map

 (Master1 0 for 1c0)

 (Master1 0 for e80)

 (Master1 0 for ec0)
```

```
 (Master1 0 for f00)
Scanning PFN database - (02% complete)

 (Master1 0 for de80)
Scanning PFN database - (100% complete)

  Usage Summary (in Kb):
Control       Valid Standby Dirty Shared Locked PageTables  name
ffffffffd 1684540      0    0    0 1684540    0    AWE
ffff8c0b7e4797d0    64     0    0    0   0   0  mapped_file( Microsoft-
Windows-Kernel-PnP%4Configuration.evtx )
ffff8c0b7e481650     0     4    0    0   0   0  mapped_file( No name for file )
ffff8c0b7e493c00     0    40    0    0   0   0  mapped_file( FSD-{ED5680AF-
0543-4367-A331-850F30190B44}.FSD )
ffff8c0b7e4a1b30     8    12    0    0   0   0  mapped_file( msidle.dll )
ffff8c0b7e4a7c40   128     0    0    0   0   0  mapped_file( Microsoft-
Windows-Diagnosis-PCW%4Operational.evtx )
ffff8c0b7e4a9010    16     8    0   16   0   0  mapped_file( netjoin.dll
)8a04db00     ...
ffff8c0b7f8cc360  8212     0    0    0   0   0  mapped_file( OUTLOOK.EXE )
ffff8c0b7f8cd1a0    52    28    0    0   0   0  mapped_file( verdanab.ttf )
ffff8c0b7f8ce910    0     4    0    0   0   0  mapped_file( No name for file )
ffff8c0b7f8d3590    0     4    0    0   0   0  mapped_file( No name for file )
```

В столбце `Control` содержится указатель на структуру области управления, описывающую отображаемый файл. Команда `!ca` отладчика ядра выводит на экран области управления, сегменты и подразделы. Например, для вывода дампа области управления отображаемого файла Outlook.exe в этом примере введите команду `!ca` с числом, указанным в столбце `Control`:

```
lkd> !ca ffff8c0b7f8cc360

ControlArea  @ ffff8c0b7f8cc360
  Segment      ffffdf08d8a55670  Flink      ffff8c0b834f1fd0  Blink
ffff8c0b834f1fd0
  Section Ref                1  Pfn Ref             806  Mapped Views        1
  User Ref                   2  WaitForDel            0  Flush Count       c5a0
  File Object  ffff8c0b7f0e94e0  ModWriteCount         0  System Views      ffff
  WritableRefs         80000161
  Flags (a0) Image File

      \Program Files (x86)\Microsoft Office\root\Office16\OUTLOOK.EXE

Segment @ ffffdf08d8a55670
  ControlArea     ffff8c0b7f8cc360  BasedAddress  0000000000be0000
  Total Ptes                 1609
  Segment Size              1609000  Committed                    0
  Image Commit                 f4  Image Info    ffffdf08d8a556b8
  ProtoPtes       ffffdf08dab6b000
  Flags (c20000) ProtectionMask

Subsection 1 @ ffff8c0b7f8cc3e0
  ControlArea  ffff8c0b7f8cc360  Starting Sector      0  Number Of Sectors    2
  Base Pte     ffffdf08dab6b000  Ptes In Subsect      1  Unused Ptes          0
```

```
  Flags                       2  Sector Offset        0  Protection          1

Subsection 2 @ ffff8c0b7f8cc418
  ControlArea  ffff8c0b7f8cc360  Starting Sector      2  Number Of Sectors 7b17
Base Pte     ffffdf08dab6b008  Ptes In Subsect      f63  Unused Ptes         0
  Flags                       6  Sector Offset        0  Protection          3

Subsection 3 @ ffff8c0b7f8cc450
  ControlArea  ffff8c0b7f8cc360  Starting Sector   7b19  Number Of Sectors 19a4
  Base Pte     ffffdf08dab72b20  Ptes In Subsect    335  Unused Ptes         0
  Flags                       2  Sector Offset        0  Protection          1

Subsection 4 @ ffff8c0b7f8cc488
  ControlArea  ffff8c0b7f8cc360  Starting Sector   94bd  Number Of Sectors  764
  Base Pte     ffffdf08dab744c8  Ptes In Subsect     f2  Unused Ptes         0
  Flags                       a  Sector Offset        0  Protection          5

Subsection 5 @ ffff8c0b7f8cc4c0
  ControlArea  ffff8c0b7f8cc360  Starting Sector   9c21  Number Of Sectors    1
  Base Pte     ffffdf08dab74c58  Ptes In Subsect      1  Unused Ptes         0
  Flags                       a  Sector Offset        0  Protection          5

Subsection 6 @ ffff8c0b7f8cc4f8
  ControlArea  ffff8c0b7f8cc360  Starting Sector   9c22  Number Of Sectors    1
  Base Pte     ffffdf08dab74c60  Ptes In Subsect      1  Unused Ptes         0
  Flags                       a  Sector Offset        0  Protection          5

Subsection 7 @ ffff8c0b7f8cc530
  ControlArea  ffff8c0b7f8cc360  Starting Sector   9c23  Number Of Sectors  c62
  Base Pte     ffffdf08dab74c68  Ptes In Subsect    18d  Unused Ptes         0
  Flags                       2  Sector Offset        0  Protection          1

Subsection 8 @ ffff8c0b7f8cc568
  ControlArea  ffff8c0b7f8cc360  Starting Sector   a885  Number Of Sectors  771
  Base Pte     ffffdf08dab758d0  Ptes In Subsect     ef  Unused Ptes         0
  Flags
```

Рабочие наборы

Итак, вы знаете, как Windows отслеживает физическую память и какой объем этой памяти она может поддерживать. Теперь разберемся, как Windows поддерживает поднабор виртуальных адресов в физической памяти.

Вспомним, что для описания поднабора виртуальных страниц, присутствующих в физической памяти, используется понятие *рабочего набора* (working set). Существуют три типа рабочих наборов:

- **Рабочие наборы процессов** содержат страницы, на которые ссылаются программные потоки отдельно взятого процесса.
- **Системные рабочие наборы** содержат резидентный поднабор системного кода со страничной организацией (например, Ntoskrnl.exe и драйверы), выгружаемый пул и системный кэш.

- **Рабочий набор каждого сеанса** содержит резидентный поднабор характерных для того или иного сеанса структур данных режима ядра, память для которых выделена частью подсистемы Windows, работающей в режиме ядра (Win32k.sys), выгружаемый пул сеанса, отображаемые представления сеанса и прочие драйверы устройств пространства сеанса.

Прежде чем подробно рассматривать все эти типы рабочих наборов, рассмотрим общую политику принятия решений относительно того, какие страницы должны попадать в физическую память и как долго они должны в этой памяти оставаться.

Подкачка по требованию

Для загрузки страниц в память диспетчер памяти Windows использует алгоритм подкачки по требованию с кластеризацией. Когда поток получает ошибку страницы, диспетчер памяти загружает в память отсутствующую страницу плюс небольшое количество страниц до и после нее. Эта стратегия стремится минимизировать объем страничного ввода/вывода потока. Поскольку программы (особенно большие) в любой момент времени выполняются обычно в небольших областях своего адресного пространства, загрузка кластеров виртуальных страниц сокращает количество чтений с диска. Для устранения ошибок страниц при ссылке на страницы данных в образах размер кластера составляет три страницы. Для всех остальных ошибок страниц размер кластера составляет семь страниц.

Однако политика подкачки по требованию может привести к тому, что процессу придется сталкиваться с множеством ошибок страниц, когда его потоки только начинают выполняться или когда их выполнение возобновляется после паузы. Для оптимизации запуска процесса (и системы) в Windows имеется интеллектуальный механизм предвыборки, называемый *компонентом логической предвыборки* (logical prefetcher) и рассматриваемый в следующем разделе. Дальнейшая оптимизация и предвыборка осуществляются еще одним компонентом, реализующим *супервыборку*, — он будет рассмотрен в этой главе позднее.

Компонент логической предвыборки

В ходе типичной начальной загрузки системы или запуска приложения порядок возникающих ошибок отсутствия страниц таков, что некоторые страницы берутся из одной части файла, затем, возможно, из отдаленной части того же самого файла, потом из другого файла, возможно, из каталога, а дальше снова из первого файла. Все эти переходы существенно замедляют обращения, причем анализ показывает, что затраты времени на поиск страниц на диске являются доминирующим фактором в замедлении начальной загрузки и запуска приложения. Предвыборка целых пакетов страниц позволяет повысить эффективность обращения к диску без лишних возвратов; тем самым сокращается общее время запуска системы и приложений. Требуемые страницы могут быть известны заранее благодаря высокому

показателю совпадений при обращениях в случае разных вариантов начальной загрузки и запуске разных приложений.

Компонент предвыборки пытается ускорить процессы начальной загрузки и запуска приложений путем отслеживания данных и кода, к которым ведется обращение при начальной загрузке и запуске приложений, а затем использовать эту информацию в процессе чтения кода и данных при последующих начальных загрузках или запусках приложений. Когда компонент предвыборки активен, диспетчер памяти извещает код компонента предвыборки в ядре об ошибках страниц — причем как о тех, которые требуют чтения данных с диска (более серьезный сбой), так и о тех, которые просто требуют, чтобы данные, уже имеющиеся в памяти, были добавлены к рабочему набору процесса (менее серьезный сбой). Компонент предвыборки следит за запуском приложений первые 10 секунд. А при начальной загрузке компонент предвыборки по умолчанию следит за развитием событий, начиная от пуска системы и до истечения 30 секунд после запуска пользовательской оболочки (обычно это Проводник), а в случае неудачи — через 60 секунд после инициализации служб Windows или через 120 секунд (в зависимости от того, что произойдет ранее).

Результаты отслеживания, собранные в ядре, отмечают ошибки, повлекшие за собой обращения к файлу метаданных главной таблицы файлов (Master File Table, MFT) файловой системы NTFS (если приложение обращается к файлам или каталогам на NTFS-томах), а также к файлам и каталогам, к которым были обращения. Имея собранные результаты отслеживания, код компонента предвыборки ядра ожидает запросов от компонента супервыборки (%SystemRoot%\System32\Sysmain.dll), запущенного в копии Svchost. Служба супервыборки отвечает как за компонент логической предвыборки в ядре, так и за компонент супервыборки, речь о котором пойдет чуть позже. Компонент предвыборки выдает событие `\KernelObjects\PrefetchTracesReady`, сообщая службе супервыборки, что теперь она может запросить отслеженные данные.

ПРИМЕЧАНИЕ Для включения и отключения предвыборки в ходе начальной загрузки или запуска используется DWORD-параметр реестра в разделе HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\PrefetchParameters\EnablePrefetcher. Присваивание этому параметру 0 выключает всю предвыборку, присваивание 1 включает предвыборку только для приложений, присваивание 2 включает предвыборку только для начальной загрузки, а присваивание 3 включает предвыборку как для начальной загрузки, так и для запуска приложений.

Служба супервыборки (которая обслуживает логическую предвыборку, хотя последняя является совершенно отдельным компонентом, не имеющим отношения к непосредственному функционированию службы супервыборки), запрашивая отслеженные данные, обращается к внутреннему системному вызову `NtQuerySystemInformation`. Компонент логической предвыборки проводит обработку собранных данных после их сбора, комбинируя их с ранее собранными данными, и записывает данные в файл в папке %SystemRoot%\Prefetch (рис. 5.33). В качестве имени файла используется имя приложения, к которому применяются

отслеженные данные, далее следует дефис и шестнадцатеричный хеш-код пути к файлу. Этот файл имеет расширение .pf (например, NOTEPAD.EXE-9FB27C0E.PF).

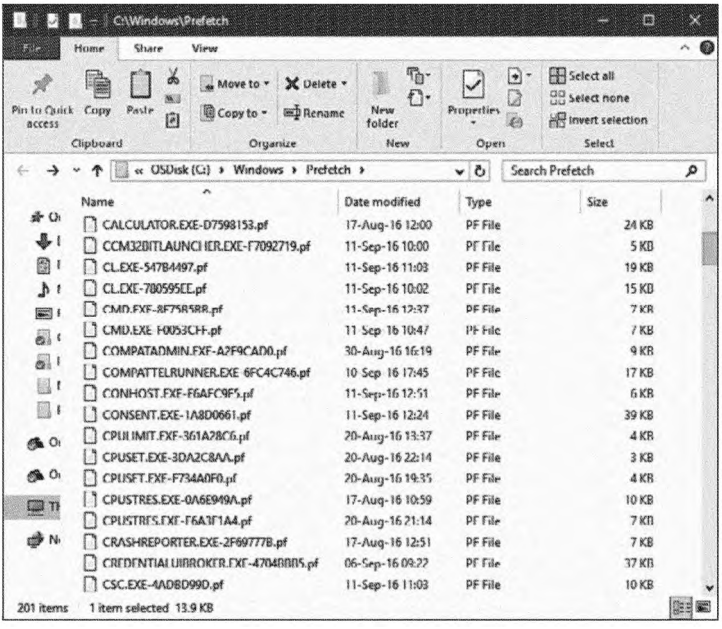


Рис. 5.33. Папка Prefetch

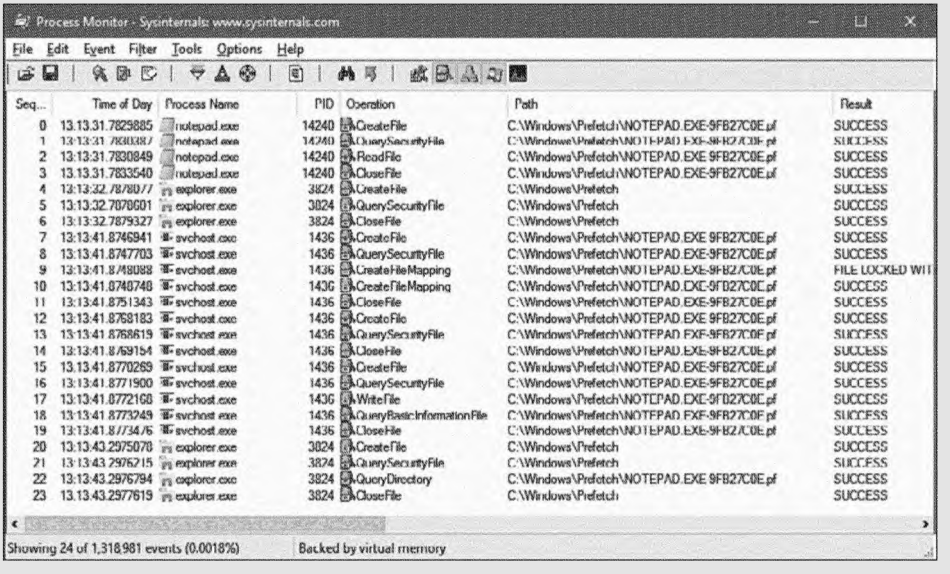
У правила формирования имени файла есть исключение для образов, содержащих другие компоненты, включая образ консоли управления Microsoft (%SystemRoot%\System32\Mmc.exe), образ для запуска хост-процесса служб Windows (%SystemRoot%\System32\Svchost.exe), образ для запуска хост-процесса компонентов DLL-библиотек (%SystemRoot%\System32\Rundll32.exe) и образ Dllhost (%SystemRoot%\System32\Dllhost.exe). Поскольку дополнительные компоненты для этих приложений указываются в командной строке, компонент предвыборки включает в создаваемый хеш командную строку. Таким образом, запуск этих приложений с различными компонентами в командной строке позволяет получать различные варианты данных отслеживания.

Для системной загрузки используется другой механизм, называемый *ReadyBoot*. Он пытается оптимизировать операции ввода/вывода за счет формирования крупных и эффективных операций ввода/вывода с хранением данных в памяти. Когда системным компонентам требуются данные, они предоставляются из оперативной памяти. Такой подход особенно эффективно работает для механических дисков, но также может быть полезен для SSD-дисков. Информация о файлах предвыборки сохраняется после загрузки в подкаталоге ReadyBoot каталога Prefetch на рис. 5.33. После завершения загрузки кэшированные данные в памяти удаляются. Для очень быстрых SSD-дисков механизм ReadyBoot по умолчанию отключается, потому что выигрыш от его применения будет незначительным.

При начальной загрузке системы или при запуске приложения компонент предвыборки вызывается для выполнения предварительной выборки. Компонент предвыборки заглядывает в каталог предвыборки, чтобы посмотреть, существует ли файл трассировки для требуемого сценария предвыборки. Если такой файл существует, компонент предвыборки вызывает NTFS для предварительной выборки любых ссылок файла метаданных MFT, считывает содержимое всех упоминаемых каталогов и, наконец, открывает каждый файл, на который есть ссылка. Затем он вызывает функцию `MmPrefetchPages` диспетчера памяти для считывания любых данных и кода, которые указаны в трассировке, но которых еще нет в памяти. Диспетчер памяти инициирует все считывания в асинхронном режиме, а затем ожидает их завершения перед тем, как позволить приложению продолжить работу.

ЭКСПЕРИМЕНТ: НАБЛЮДЕНИЕ ЗА ЧТЕНИЕМ И ЗАПИСЬЮ ФАЙЛА ПРЕДВЫБОРКИ

Если в клиентской версии Windows (в версиях Windows Server предвыборка изначально отключена) записать трассировку запуска приложения, используя программу ProcessMonitor, созданную в Sysinternals, можно увидеть, как компонент предвыборки проверяет наличие файла предвыборки приложения и, если таковой имеется, считывает его содержимое. Примерно через 10 секунд после запуска приложения можно увидеть, как компонент предвыборки записывает новую копию файла. На следующем рисунке показана зафиксированная трассировка запуска приложения Блокнот с включающим фильтром (`Include`), настроенным на предвыборку, чтобы программа Process Monitor показывала только обращения к каталогу %SystemRoot%\Prefetch.



Из строк 0–3 видно, что файл предвыборки для приложения Notepad был считан в контексте процесса Notepad в ходе запуска. В строках 7–19, имеющих метку времени на 10 секунд позже, чем у первых четырех строк, показана служба супервыборки, запущенная в контексте процесса Svchost и записывающая на диск обновленный файл предвыборки.

Чтобы еще больше сократить время поиска на диске, примерно каждые три дня в моменты простоя системы служба супервыборки составляет список файлов и каталогов в том порядке, в котором на них делаются ссылки в ходе начальной загрузки или запуска приложений, и сохраняет список в файле %SystemRoot%\Prefetch\Layout.ini (рис. 5.34). В этот список также включаются отслеженные службой супервыборки файлы, к которым происходят частые обращения.

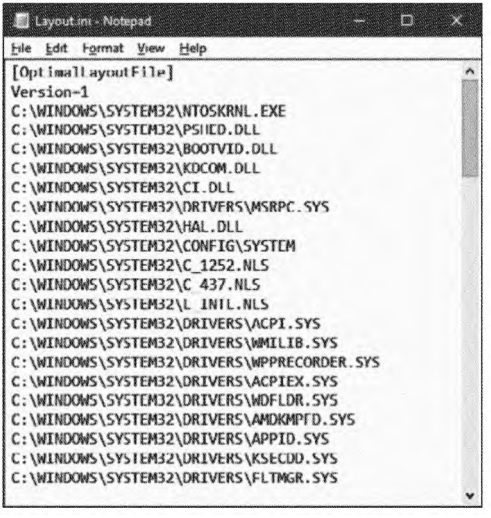


Рис. 5.34. Структура файла дефрагментации предвыборки

Затем служба супервыборки запускает системную программу дефрагментации диска с ключом командной строки, который приказывает провести не полную дефрагментацию, а дефрагментацию на основе содержимого созданного файла. Программа дефрагментации находит непрерывные области на каждом томе, которые имеют достаточный размер для размещения всех перечисленных файлов и каталогов этого тома, а затем перемещает их целиком в эту область, чтобы они хранились последовательно, один за другим. Таким образом, операции будущих предвыборок станут еще эффективнее, поскольку все считываемые данные физически сохраняются на диске в порядке их считывания. Поскольку счет дефрагментируемых для предвыборки файлов обычно ограничивается сотнями, такая дефрагментация проводится намного быстрее, чем дефрагментация всего тома.

Политика размещения

Когда поток сталкивается с ошибкой страницы, диспетчер памяти должен определить, где в физической памяти разместить виртуальную страницу. Для определения наилучшего места используется ряд правил, называемых *политикой размещения* (placement policy). Для сведения к минимуму ненужной пробуксовки кэша, при выборе страничных блоков Windows учитывает размер кэш-памяти центрального процессора.

Если при возникновении ошибки отсутствия страницы физическая память полностью заполнена, то определить, какая виртуальная страница должна быть удалена из памяти, чтобы освободить место для новой страницы, позволяет политика замещения (replacement policy). В число наиболее распространенных политик замещения входят алгоритмы LRU (Least Recently Used — дольше всех не используемый) и FIFO (First In, First Out — первым пришел, первым вышел). Алгоритм LRU (также известный как *алгоритм таймера* и реализованный на большинстве версий UNIX) требует от виртуальной памяти отслеживать востребованность страницы в памяти. Когда нужен новый страничный блок, из рабочего набора удаляется та страница, которая не была востребована дольше других. Алгоритм FIFO работает несколько проще — он требует удалить страницу, которая дольше других находится в физической памяти, независимо от того, как часто она была востребована.

Далее политики замещения делятся на глобальные и локальные. Глобальная политика замещения позволяет разрешать ошибки страниц за счет любого страничного блока независимо от того, принадлежит он другому процессу или нет. Например, политика глобального замещения, использующая алгоритм FIFO, позволит найти страницу, которая была в памяти дольше других, и освободить ее для разрешения ошибки отсутствия страницы. Локальная же политика замещения ограничивает поиск наиболее старой из страниц того процесса, при выполнении которого произошла ошибка страницы. Глобальные политики замещения делают процесс зависимым от поведения других процессов — приложение с ненадлежащим поведением может подорвать работу всей операционной системы, вызвав чрезмерную активность подкачки во всех процессах.

В Windows сочетается локальная и глобальная политики замещения. Когда рабочий набор достигает своего лимита и/или нуждается в усечении из-за потребностей в физической памяти, диспетчер памяти удаляет страницы из рабочих наборов до тех пор, пока не определит, что в системе достаточно свободных страниц.

Управление рабочими наборами

Каждый процесс запускается с исходным рабочим набором минимум в 50 страниц, а максимум рабочего набора составляет 345 страниц. Хотя это мало на что влияет, лимиты рабочего набора процесса можно изменить с помощью Windows-функции `SetProcessWorkingSetSize`, но для этого у вас должна быть пользовательская при-

вилегия на повышение приоритета при планировании (increase scheduling priority). Однако пока процесс не будет настроен на жесткие лимиты рабочих наборов, эти лимиты будут игнорироваться по причине того, что диспетчер памяти разрешит процессу расширение за установленный максимум, если он слишком интенсивно использует подкачку при изобилии памяти (и наоборот, диспетчер памяти сократит рабочий набор процесса ниже минимума, если процесс не использует подкачку и у системы имеется высокая потребность в физической памяти). Жесткие лимиты рабочих наборов можно установить с помощью функции `SetProcessWorkingSetSizeEx` с флагом `QUOTA_LIMITS_HARDWS_MIN_ENABLE`, но практически всегда лучше позволить системе самой управлять вашим рабочим набором, а не устанавливать для него собственные жесткие ограничения.

Максимальный размер рабочего набора не может превышать общесистемного максимума, вычисляемого в ходе инициализации системы и сохраняемого в переменной `MiMaximumWorkingSet` ядра. На платформе x64 практической верхней границей из-за огромного виртуального адресного пространства является размер физической памяти. Максимумы рабочих наборов перечислены в табл. 5.15.

Таблица 5.15. Верхние пределы для максимумов рабочих наборов

Версия Windows	Максимум рабочих наборов
x86, ARM	2 Гбайт — 64 Кбайт (0x7FFF0000)
x86-версии Windows, загруженные с параметром `increaseuserva`	2 Гбайт — 64 Кбайт плюс расширение пользовательского виртуального адресного пространства
x64 (Windows 8, Server 2012)	8192 Гбайт (8 Тбайт)
x64 (Windows 8.1, 10, Server 2012, 2016)	128 Тбайт

При возникновении ошибки страницы анализируются лимиты рабочего набора процесса и объем имеющейся в системе свободной памяти. Если условия позволяют, диспетчер памяти разрешает процессу расширить его рабочий набор до максимума (или даже выше, если у процесса нет жестких лимитов рабочего набора, а в системе достаточно доступных свободных страниц). Однако в случае дефицита памяти при возникновении ошибки страницы Windows замещает страницы в рабочем наборе, а не добавляет их.

Хотя Windows пытается сохранять доступность памяти путем записи измененных файлов на диск, когда измененные страницы создаются очень высокими темпами, чтобы восполнить потребности в памяти, требуется дополнительная память. Поэтому, если физической памяти становится мало, диспетчер рабочих наборов (working set manager) — процедура, запускаемая в контексте системного программного потока диспетчера настройки баланса (см. далее в этой главе), — инициирует автоматическое усечение рабочего набора для увеличения объема свободной памяти в системе. (С помощью ранее упомянутой функции Windows `SetProcessWorkingSetSizeEx` также можно инициировать усечение рабочего набора собственного процесса, например после инициализации процесса.)

Диспетчер рабочих наборов исследует доступную память и решает, какой из имеющихся рабочих наборов должен быть усечен. При наличии достаточной памяти диспетчер рабочих наборов вычисляет, сколько страниц может быть удалено из рабочих наборов, если понадобится. Если требуется усечение, диспетчер оценивает рабочие наборы, близкие к своим минимальным установкам. Он также динамически регулирует темп исследования рабочих наборов и составляет список процессов, являющихся оптимальными кандидатами на усечение. Например, процессы, имеющие множество страниц, к которым в последнее время не было обращений, проверяются в первую очередь; более масштабные процессы, пребывающие в длительном простое, рассматриваются раньше менее масштабных, но чаще запускаемых; процесс, в рамках которого запущено приложение, работающее в фоновом режиме, рассматривается последним и т. д.

Как только обнаруживаются процессы, расходующие больше своих минимумов, диспетчер рабочих наборов ищет страницы для удаления из рабочих наборов, делая эти страницы доступными для других пользователей. Если объем свободной памяти все еще слишком низок, диспетчер рабочих наборов продолжает удалять страницы из рабочих наборов процессов до тех пор, пока не будет достигнуто минимальное количество свободных страниц в системе.

Диспетчер рабочих наборов пытается удалить страницы, к которым в последнее время не было обращений. Он делает это путем проверки бита посещения в аппаратной PTE-записи, чтобы увидеть, осуществлялся ли доступ к странице. Если бит посещения сброшен, страница считается *возрастной* (aged), т. е. показание счетчика возрастает, указывая на то, что к странице не было обращений со времени последнего сканирования рабочих наборов с целью их усечения. Позже возраст страниц позволяет обнаруживать среди страниц кандидатов на удаление из рабочего набора.

Если бит посещения в аппаратной PTE-записи установлен, диспетчер рабочих наборов сбрасывает этот бит и продолжает исследовать следующую страницу в рабочем наборе. Таким образом, если бит посещения при следующем исследовании страницы диспетчером рабочих наборов будет сброшен, станет понятно, что со времени последнего исследования этой страницы к ней не было обращений. Такое сканирование страниц с целью их последующего удаления продолжается по списку рабочих наборов до тех пор, пока не будет удалено нужное количество страниц или пока процедура сканирования не вернется в исходную точку. (При следующем усечении рабочих наборов процедура сканирования начинается с того места, на котором она остановилась в прошлый раз.)

ЭКСПЕРИМЕНТ: ПРОСМОТР РАЗМЕРОВ РАБОЧИХ НАБОРОВ ПРОЦЕССОВ

Для анализа размеров рабочих наборов процессов можно воспользоваться монитором производительности, представленным в следующей таблице. Другие утилиты просмотра информации о процессах (например, диспетчер задач и Process Explorer) также выводят информацию о рабочих наборах процессов.

Счетчик	Описание
Процесс: Рабочий набор (Process: Working Set)	Текущий размер рабочих наборов выбранных процессов в байтах
Процесс: Рабочий набор (пик) (Process: Working Set Peak)	Пиковый размер рабочих наборов выбранных процессов в байтах
Процесс: Ошибок страницы/с (Process: Page Faults/sec)	Количество ошибок отсутствия страниц в секунду для процесса

Кроме того, можно получить общий размер всех рабочих наборов процессов, выбрав в списке экземпляров в мониторе производительности вариант `_Total`. Этот вариант не является реальным процессом; он просто позволяет вывести на экран суммарные показания счетчиков, относящихся к процессам, для всех процессов, запущенных на данный момент в системе. Выводимая сумма больше фактического объема используемой оперативной памяти, потому что размер рабочего набора каждого процесса включает в себя страницы, используемые совместно с другими процессами. Таким образом, если два и более процесса имеют общую страницу, эта страница учитывается в рабочем наборе каждого процесса.

ЭКСПЕРИМЕНТ: СРАВНЕНИЕ РАБОЧЕГО НАБОРА С РАЗМЕРОМ ВИРТУАЛЬНОЙ ПАМЯТИ

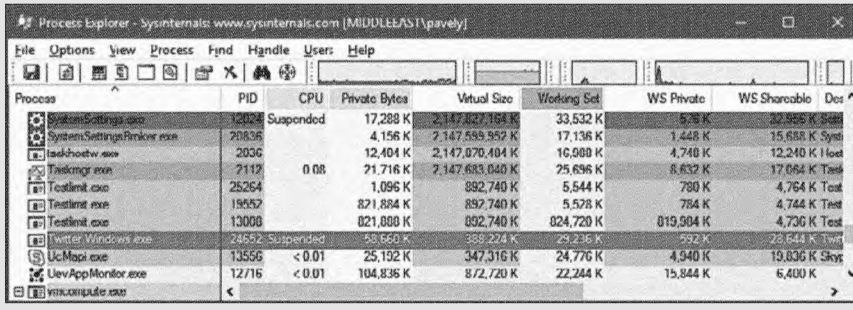
Ранее в данной главе утилита TestLimit использовалась для создания двух процессов: одного с большим объемом зарезервированной памяти и второго с закрытой и подтвержденной памятью, — и с помощью программы Process Explorer мы изучали разницу между ними. Теперь мы создадим третий процесс TestLimit, который не только подтверждает память, но и обращается к ней, вводя ее тем самым в свой рабочий набор:

1. Создайте новый процесс TestLimit.

```
C:\Users\pavely>testlimit -d 1 -c 800

Testlimit v5.24 - test Windows limits
Copyright (C) 2012-2015 Mark Russinovich
Sysinternals - www.sysinternals.com

Process ID: 13008

Leaking private bytes with touch 1 MB at a time...
Leaked 800 MB of private memory (800 MB total leaked). Lasterror: 0
The operation completed successfully.
```

2. Запустите программу Process Explorer.
3. Откройте меню View, выберите команду Select Columns и перейдите на вкладку Process Memory.

4. Включите счетчики Private Bytes, Virtual Size, Working Set Size, WS Shareable Bytes и WS Private Bytes.
5. Найдите три экземпляра TestLimit, как показано на следующем рисунке:



Новый процесс TestLimit (показан третьим по счету) имеет PID-идентификатор 13008. Он является единственным из трех процессов, который действительно ссылается на выделенную память, следовательно, он единственный, имеющий рабочий набор, который отражает размер выделенной в тесте памяти.

Следует учесть, что этот результат возможен только на системе с достаточным объемом оперативной памяти, позволяющим процессу разрастись до такого размера. Даже на этой системе не все байты закрытой памяти (821 888 Кбайт) находятся в составе той части рабочего набора, которая показана как количество байтов закрытой памяти (столбец WS Private). Некоторое небольшое количество закрытых страниц либо исчезло из рабочего набора процесса в результате замещения, либо еще не было подкачано с диска.

ЭКСПЕРИМЕНТ: ПРОСМОТР СПИСКА РАБОЧЕГО НАБОРА В ОТЛАДЧИКЕ

Отдельные записи в рабочем наборе можно просмотреть с помощью команды `!wsle` отладчика ядра. В следующем примере приведен фрагмент выведенного списка рабочего набора процесса WinDbg (32-разрядная система):

```
lkd> !wsle 7

Working Set Instance @ c0802d50
Working Set Shared @ c0802e30

    FirstFree      f7d  FirstDynamic        6
    LastEntry     203d  NextSlot            6  LastInitialized     2063
    NonDirect        0  HashTable           0  HashTableSize          0

Reading the WSLE data ..................................................
.......

Virtual Address           Age  Locked  ReferenceCount
        c0603009          0       0       1
        c0602009          0       0       1
        c0601009          0       0       1
```

```
        c0600009        0       0       1
        c0802d59        6       0       1
        c0604019        0       0       1
        c0800409        2       0       1
        c0006209        1       0       1
        77290a05        5       0       1
        7739aa05        5       0       1
        c0014209        1       0       1
        c0004209        1       0       1
        72a37805        4       0       1
          b50409        2       0       1
          b52809        4       0       1
        7731dc05        6       0       1
          bbec09        6       0       1
          bbfc09        6       0       1
        6c801805        4       0       1
        772a1405        2       0       1
          944209        1       0       1
        77316a05        5       0       1
        773a4209        1       0       1
        77317405        2       0       1
        772d6605        3       0       1
          a71409        2       0       1
          c1d409        2       0       1
        772d4a05        5       0       1
        77342c05        6       0       1
        6c80f605        3       0       1
        77320405        2       0       1
        77323205        1       0       1
        77321405        2       0       1
        7ffe0215        1       0       2
          a5fc09        6       0       1
        7735cc05        6       0       1
...
```

Следует учесть, что некоторые записи в списке рабочего набора относятся к страницам таблиц страниц (те, чьи адреса больше чем 0xC0000000), другие страницы берутся из системных DLL-библиотек (из диапазона 0x7nnnnnnn), третьи относятся к страницам с кодом самого файла Windbg.exe.

Диспетчер рабочего баланса и потока подкачки

Расширение и усечение рабочего набора происходят в контексте системного программного потока, который называется *диспетчером настройки баланса* (balance set manager). Этот диспетчер (процедура `KeBalanceSetManager`) создается в ходе инициализации системы. Хотя технически диспетчер настройки баланса является частью ядра, для анализа и настройки рабочего набора он вызывает диспетчер рабочих наборов, принадлежащий диспетчеру памяти (`MmWorkingSetManager`).

Диспетчер настройки баланса находится в ожидании двух объектов событий: события от таймера, настроенного на срабатывание один раз в секунду, и события

от внутреннего диспетчера рабочих наборов, которым диспетчер памяти подает сигнал в различные моменты, когда обнаруживает, что рабочие наборы нуждаются в настройке. Например, если система часто сталкивается с ошибками страниц или список свободных страниц слишком мал, диспетчер памяти активирует диспетчер настройки баланса, чтобы тот вызвал диспетчер рабочих наборов и приступил к усечению этих наборов. При избытке памяти диспетчер рабочих наборов разрешает процессам, сталкивающимся с ошибками отсутствия страниц, постепенно увеличивать размер своих рабочих наборов, возвращая такие страницы обратно в память (при этом рабочие наборы будут расти только по мере надобности).

Когда диспетчер настройки баланса активируется по тайм-ауту своего односекундного таймера, он выполняет следующие действия:

1. Если система поддерживает режим VSM (Virtual Secure Mode) (Windows 10 и Server 2016), диспетчер обращается с вызовом к защищенному ядру для выполнения периодических учетных операций (`VslSecureKernelPeriodicTick`).
2. Диспетчер вызывает функцию корректировки данных IRP для оптимизации использования резервных списков уровня отдельных процессоров, задействованных при завершении IRP (`IoAdjustIrpCredits`). Тем самым улучшается масштабируемость при интенсивной нагрузке ввода/вывода на некоторых процессорах (за дополнительной информацией о IRP обращайтесь к главе 6).
3. Диспетчер настройки баланса проверяет ассоциативные списки и настраивает в случае необходимости их глубину. Это делается для ускорения доступа, а также для снижения нагрузки на пул и уменьшения его фрагментации (`ExAdjustLookasideDepth`).
4. Диспетчер выполняет вызов для корректировки размера пула буфера ETW (Event Tracing for Windows) для повышения эффективности буферов памяти ETW (`EtwAdjustTraceBuffers`). (За дополнительной информацией о ETW обращайтесь к главе 8 части 2.)
5. Диспетчер настройки баланса вызывает диспетчер рабочих наборов диспетчера памяти. (У диспетчера рабочих наборов имеются собственные внутренние счетчики, которые регулируют момент усечения рабочего набора и определяют, насколько агрессивно это нужно делать.)
6. Контролирует время выполнения заданий (`PsEnforceExecutionLimits`).
7. При каждой восьмой активации диспетчера настройки баланса в связи с тайм-аутом односекундного таймера он выдает событие, которое активизирует другой системный поток, называемый *потоком подкачки* (процедура `KeSwapProcessOrStack`). Он пытается выгрузить стеки ядра потоков, которые не выполнялись в течение долгого времени. Поток подкачки (выполняемый с приоритетом 23) ищет потоки, которые находятся в состоянии ожидания пользовательского режима более 15 секунд. Если такой поток будет найден, стек ядра потока переводится в переходное состояние (страницы перемещаются в измененные или резервные списки) для освобождения их физической памяти.

Предполагается, что если поток ожидал так долго, у него не будет проблем с тем, чтобы подождать еще немного. Когда последний поток процесса будет выгружен из памяти, процесс помечается как полностью выгруженный. Именно по этой причине у процессов, бездействующих в течение долгого времени (таких, как Wininit или Winlogon), размер рабочего набора может быть равен 0.

Системные рабочие наборы

По аналогии с тем, как рабочие наборы процессов используются для управления выгружаемыми частями их адресных пространств, управление выгружаемым кодом и данными в системном адресном пространстве осуществляется с помощью трех глобальных рабочих наборов с общим названием *системные рабочие наборы* (system working sets):

- **Рабочий набор системного кэша** (`MmSystemCacheWs`) содержит страницы, находящиеся резидентно в системном кэше.
- **Рабочий набор выгружаемого пула** (`MmPagedPoolWs`) содержит страницы, находящиеся резидентно в выгружаемом пуле.
- **Рабочий набор системных PTE-записей** (`MmSystemPtesWs`) содержит выгружаемый код и данные загруженных драйверов и образа ядра, а также страницы из разделов, отображаемых на системное пространство.

В табл. 5.16 показано, как хранятся разные типы системных рабочих наборов.

Таблица 5.16. Системные рабочие наборы

Тип системного рабочего набора	Способ хранения (Windows 8.x, Server 2012/R2)	Способ хранения (Windows 10, Server 2016)
Системный кэш	`MmSystemCacheWs`	`MiState.SystemVa.SystemWs[0]`
Выгружаемый пул	`MmPagedPoolWs`	`MiState.SystemVa.SystemWs[1]`
Системные PTE-записи	`MmSystemPtesWs`	`MiState.SystemVa.SystemWs[2]`

Размеры этих рабочих наборов, как и размеры составляющих их компонентов, можно узнать с помощью счетчиков производительности или системных переменных (табл. 5.17). Следует иметь в виду, что счетчики производительности показывают значения в байтах, а системные переменные — в страницах.

Страничную активность в рабочем наборе системного кэша можно также проанализировать по показаниям счетчика Память: Ошибки кэша/с (Memory: Cache Faults/sec), в котором регистрируются ошибки страниц, возникающие в рабочем наборе системного кэша (как ошибки страниц ОЗУ, так и ошибки страниц физической памяти). Значение этого счетчика содержится в системной переменной `PageFaultCount` структуры рабочего набора системного кэша.

Таблица 5.17. Счетчики производительности системных рабочих наборов

Счетчик производительности (в байтах)	Системная переменная (в страницах)	Описание
Память: Байт кэш-памяти (Memory: Cache Bytes), также Память: Резидентных байт системного кэша (Memory: System Cache Resident Bytes)	Поле `WorkingSetSize`	Физическая память, потребляемая кэшем файловой системы
Память: Байт кэш-памяти (пик) (Memory: Cache Bytes Peak)	Поле `PeakWorkingSetSize` (Windows 10 и 2016) Поле Peak (Windows 8.x и 2012/R2)	Пиковый размер системного рабочего набора
Память: Резидентных байт системных драйверов (Memory: System Driver Resident Bytes)	`SystemPageCounts.SystemDriverPage` (глобальная, Windows 10 и Server 2016) `MmSystemDriverPage` (глобальная, Windows 8.x и Server 2012/R2)	Физическая память, потребляемая выгружаемым кодом драйверов устройств
Memory: Pool Paged Resident Bytes (Память: Байт в резидентном страничном пуле)	Поле `WorkingSetSize`	Физическая память, потребляемая выгружаемым пулом

События уведомлений в памяти

Windows предоставляет средства для уведомления процессов пользовательского режима и драйверов режима ядра о том, что объемы физической памяти, выгружаемого и невыгружаемого пулов, а также показатель подтверждения слишком малы и/или велики. При необходимости эта информация может пригодиться для определения порядка использования памяти. Например, при небольшом объеме доступной памяти приложение может сократить потребление памяти. Если объем доступного выгружаемого пула велик, драйвер может выделить больше памяти. И наконец, диспетчер памяти также предоставляет событие уведомления при обнаружении поврежденной страницы.

Процессы пользовательского режима могут уведомляться только о малом или большом объеме памяти. Приложение может вызвать функцию `CreateMemoryResourceNotification`, указав, какое ему нужно уведомление — о малом или большом объеме памяти. Возвращаемый дескриптор может быть передан любой функции ожидания. Когда памяти слишком мало (или слишком много), ожидание завершается; таким образом поток уведомляется о наступлении ожидаемого условия. Кроме того, с помощью функции `QueryMemoryResourceNotification` можно выполнить запрос о ситуации с системной памятью в любое время без блокировки вызывающего потока.

В то же время драйверы используют особые имена событий, которые диспетчер памяти создает в каталоге \KernelObjects, поскольку уведомление, реализованное диспетчером памяти, сигнализирует об одном из определенных им глобальных

именованных объектов событий (табл. 5.18). При обнаружении данных условий в состоянии памяти выдается сигнал о наступлении соответствующего события, в результате ожидающие программные потоки активируются.

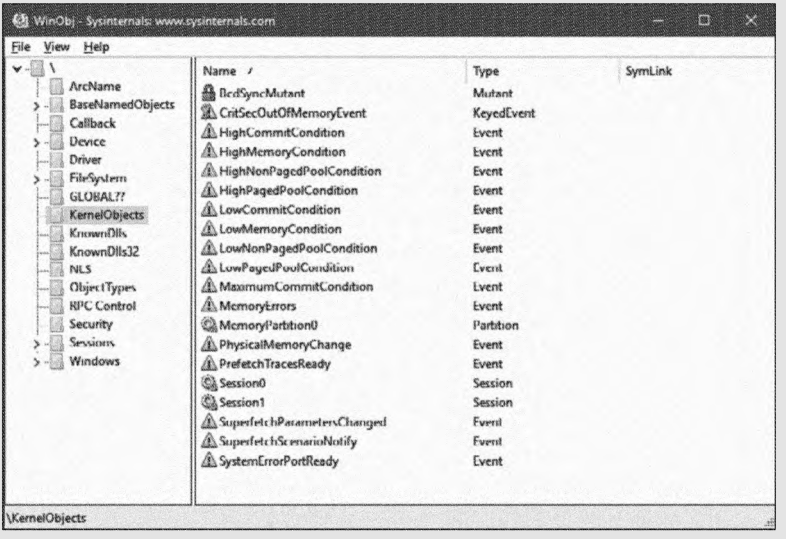
Таблица 5.18. События уведомлений диспетчера памяти

Имя события	Описание
`HighCommitCondition`	Общий объем подтвержденной памяти приближается к максимальному значению лимита подтверждения. Иными словами, потребление памяти очень велико, а в физической памяти или в страничных файлах очень мало места, и операционная система не может увеличить размер своих страничных файлов
`HighMemoryCondition`	Объем свободной физической памяти превысил определенное значение
`HighNonPagedPoolCondition`	Объем невыгружаемого пула превысил определенное значение
`HighPagedPoolCondition`	Объем выгружаемого пула превысил определенное значение
`LowCommitCondition`	Показатель подтверждения ниже текущего лимита подтверждения. Иными словами, потребление памяти невелико, и в физической памяти или в страничных файлах имеется много доступного пространства
`LowMemoryCondition`	Объем свободной физической памяти стал меньше определенного значения
`LowNonPagedPoolCondition`	Объем свободного невыгружаемого пула стал меньше определенного значения
`LowPagedPoolCondition`	Объем свободного выгружаемого пула стал меньше определенного значения
`MaximumCommitCondition`	Показатель подтверждения приблизился к максимальному значению лимита подтверждения. Иными словами, потребление памяти очень велико, а в физической памяти или в страничных файлах очень мало места, и операционная система не может увеличить размер или количество страничных файлов
`MemoryErrors`	Обнаружена плохая страница (необнуленная страница, подлежавшая обнулению)

ПРИМЕЧАНИЕ Чтобы переопределить пороговые значения, добавьте в раздел реестра HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management параметры `LowMemoryThreshold` или `HighMemoryThreshold` типа DWORD. Параметры определяют количество мегабайтов для использования в качестве малых и больших показателей потребления памяти. Система может быть настроена так, чтобы при обнаружении плохой страницы она аварийно завершала свою работу, а не сигнализировала о событии ошибки памяти. Для этого нужно в том же самом разделе реестра создать DWORD-параметр `PageValidationAction`.

ЭКСПЕРИМЕНТ: ПРОСМОТР УВЕДОМЛЕНИЙ О СОСТОЯНИИ РЕСУРСОВ ПАМЯТИ

Чтобы увидеть события, уведомляющие о состоянии ресурсов памяти, запустите программу Winobj из пакета Sysinternals и щелкните на папке KernelObjects. На правой панели появятся события, касающиеся как малых, так и больших показателей потребления памяти.



Если дважды щелкнуть кнопкой мыши на строке любого события, можно увидеть количество использованных дескрипторов и/или обращений к объектам. Чтобы убедиться, что любые процессы в системе запрашивают уведомления о ресурсах памяти, нужно поискать в таблице дескрипторов ссылки вида «LowMemoryCondition» или «HighMemoryCondition». Для этого откройте в Process Explorer меню Find и выберите вариант Find Handle or DLL или воспользуйтесь программой WinDbg. (Описание таблицы дескрипторов можно найти в разделе «Диспетчер объектов» главы 8.)

База данных номеров страничных блоков

В нескольких предыдущих разделах основное внимание было уделено виртуальному представлению Windows-процесса — таблицам страниц, PTE-записям, VAD-дескрипторам. В оставшейся части этой главы рассказывается, как Windows управляет физической памятью, начиная с того, как Windows следит за физической памятью. Тогда как рабочие наборы описывают резидентные страницы, которыми владеют процесс или система, база данных номеров страничных блоков (Page Frame Number, PFN) описывает состояние каждой страницы в физической памяти. Варианты состояния страниц перечислены в табл. 5.19.

Таблица 5.19. Состояния страниц

Состояние	Описание
Активная страница (active), также называемая достоверной (valid)	Страница является частью рабочего набора (либо рабочего набора процесса, либо рабочего набора сеанса, либо системного рабочего набора) или она не входит ни в какой рабочий набор (например, невыгружаемая страница ядра), и на нее обычно указывает достоверная PTE-запись
Переходная страница (transition)	Временное состояние страницы, которая не принадлежит рабочему набору и не присутствует ни в одном списке подкачки. Страница оказывается в этом состоянии, когда в отношении нее осуществляются операции ввода/вывода. PTE-запись закодирована таким образом, чтобы могли быть распознаны и правильно обработаны конфликтные ошибки отсутствия страницы. (Следует заметить, что трактовка понятия «переходная» отличается от таковой в разделе о недостоверных PTE-записях; недостоверная переходная PTE-запись относится к странице, находящейся в списке ожидающих или измененных страниц)
Ожидающая страница (standby)	Страница ранее принадлежала рабочему набору, но была удалена из него (или же была предварительно извлечена или считана в составе кластера и попала непосредственно в список ожидающих страниц). Со времени своей последней записи на диск страница не изменялась. PTE-запись по-прежнему ссылается на физическую страницу, но помечена как недостоверная и переходная
Измененная страница (modified)	Ранее страница принадлежала рабочему набору, но была из него удалена. Однако пока страница использовалась, она была изменена, а ее текущее содержимое еще не было записано на диск или в удаленное хранилище. PTE-запись по-прежнему ссылается на физическую страницу, но помечена как недостоверная и переходная. Перед повторным использованием физической страницы эта страница должна быть записана в резервное хранилище
Измененная, но не подлежащая записи страница (modified no-write)	То же самое, что и измененная страница, но эта страница помечается таким образом, что процедура записи диспетчера памяти не может записывать ее на диск. Диспетчер кэша помечает страницы как измененные, но не записываемые, по запросу драйверов файловой системы. Например, NTFS использует это состояние для страниц, содержащих метаданные файловой системы, поэтому сначала нужно обеспечить сброс на диск записей журнала транзакций, а потом только записывать на диск защищаемые ими страницы. (О протоколировании транзакций в NTFS рассказывается в главе 13 «Файловые системы» части 2)
Свободная страница (free)	Страница свободна, но содержит некие измененные данные. (По соображениям безопасности такие страницы не могут предоставляться в качестве пользовательских страниц пользовательскому процессу без их заполнения нулями)
Заполненная нулями страница (zeroed)	Страница свободна и была заполнена нулями программным потоком обнуления страниц (или была определена как уже содержащая нули)
Страница, предназначенная только для чтения (rom)	Страница представляет память, предназначенную только для чтения
Нерабочая страница (bad)	В отношении страницы была сгенерирована ошибка четности или другие аппаратные ошибки, поэтому страница не может быть использована (сама по себе или как часть анклава)

База данных PFN-номеров состоит из массива структур, представляющих каждую физическую страницу памяти системы. Эта база данных и ее взаимосвязи с таблицами страниц показаны на рис. 5.35. Как следует из рисунка, достоверные PTE-записи обычно указывают на записи в базе данных PFN-номеров, а записи базы данных PFN-номеров (для непрототипных PFN-записей) — обратно на таблицу страниц, которая их использует (если они используются таблицей страниц). Что касается прототипных PFN-записей, то они указывают обратно на прототипную PTE-запись.

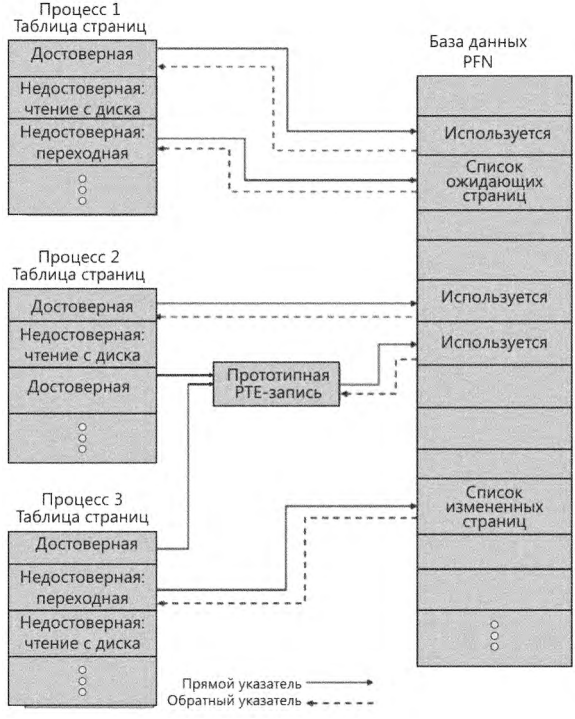


Рис. 5.35. Таблицы страниц и база данных PFN

Из тех состояний страниц, которые перечислены в табл. 5.19, шесть состояний организованы в связанные списки, чтобы диспетчер памяти мог быстро находить страницы определенного типа. (Активные, или достоверные, страницы, страницы в переходном состоянии и перезагруженные «нерабочие» страницы не присутствуют ни в одном общесистемном списке страниц.) Кроме того, состояние резерва (standby) фактически ассоциируется с восемью различными списками, выстроенными по приоритетам (о приоритетах страниц речь пойдет в этом разделе чуть позже). На рис. 5.36 показан пример структуры связей между этими записями.

Из следующего раздела вы узнаете, как эти связанные списки используются для разрешения ошибок страниц, как страницы заносятся в различные списки и удаляются из них.

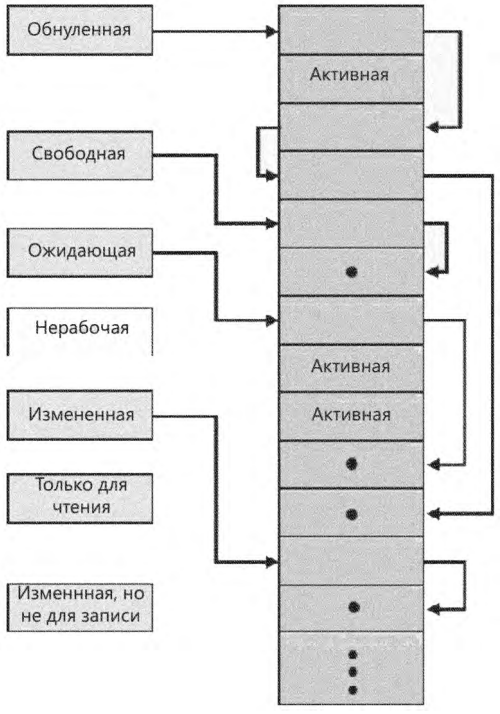


Рис. 5.36. Списки страниц в базе данных PFN

ЭКСПЕРИМЕНТ: ПРОСМОТР БАЗЫ ДАННЫХ PFN-НОМЕРОВ

Для вывода размеров различных страничных списков можно воспользоваться программой MemInfo с сайта книги (программа запускается с ключом `-s`). Выводимые данные выглядят следующим образом:

```
C:\Tools>MemInfo.exe -s
MemInfo v3.00 - Show PFN database information
Copyright (C) 2007-2016 Alex Ionescu
www.alex-ionescu.com

Initializing PFN Database... Done

PFN Database List Statistics
              Zeroed:    4867 (   19468 kb)
                Free:    3076 (   12304 kb)
             Standby: 4669104 (18676416 kb)
            Modified:    7845 (   31380 kb)
     ModifiedNoWrite:     117 (     468 kb)
        Active/Valid: 3677990 (14711960 kb)
          Transition:       5 (      20 kb)
                 Bad:       0 (       0 kb)
             Unknown:    1277 (    5108 kb)
               TOTAL: 8364281 (33457124 kb)
```

Команда `!memusage` отладчика ядра выдает такую же информацию, хотя этот вариант займет больше времени и потребует начальной загрузки в режиме отладки.

Динамика списков страниц

На рис. 5.37 показана диаграмма состояний переходов страничных блоков. Для простоты на ней не представлен список измененных, но не подлежащих записи страниц.

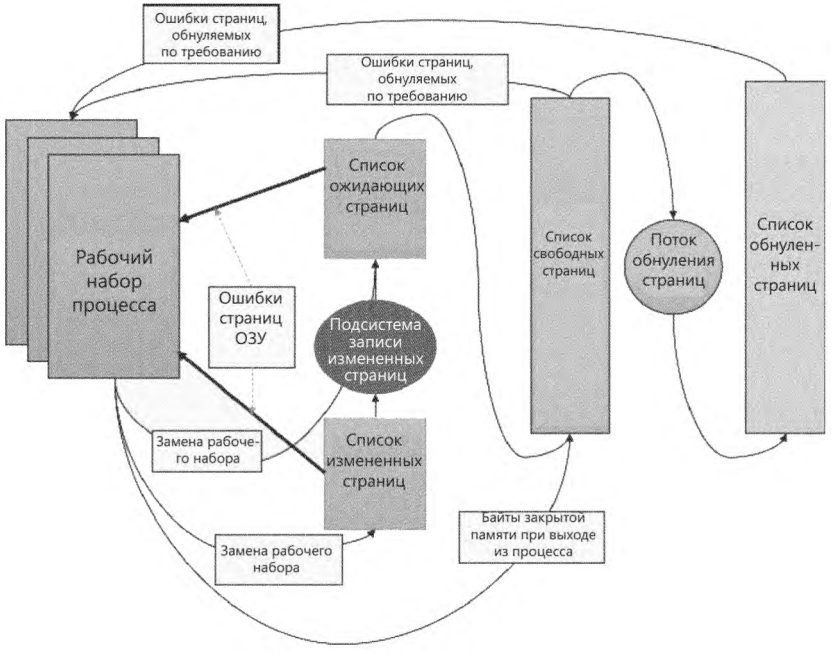


Рис. 5.37. Диаграмма состояний страничных блоков

Страничные блоки перемещаются между страничными списками следующими способами.

- Когда диспетчеру памяти для обслуживания ошибки страницы, связанной с требованием обнуленной страницы (обращение к странице, которая определена как полностью заполненная нулями, или к закрытой подтвержденной странице пользовательского режима, к которой еще не было обращений), нужна страница, заполненная нулями, сначала предпринимается попытка получить эту страницу из списка таких страниц. Если список пуст, страница берется из списка свободных страниц и заполняется нулями. Если пуст и список свободных страниц, происходит обращение к списку ожидающих страниц, и заполняется нулями страница из этого списка.

- Одна из причин востребованности страниц, заполненных нулями, заключается в выполнении различных требований безопасности — например, общих критериев (Common Criteria). Большинство положений общих критериев указывает на то, что процессы пользовательского режима должны получать обнуленные страничные блоки, чтобы они не могли прочитать содержимое памяти предыдущих процессов. Поэтому диспетчер памяти предоставляет процессам пользовательского режима обнуленные страничные блоки, если только страница не была считана из резервного хранилища. В таком случае диспетчер памяти использует необнуленные страничные блоки, инициализируя их данными с диска или с удаленного хранилища. Список страниц, заполненных нулями, пополняется из списка свободных страниц системным программным потоком, который называется потоком обнуления страниц (zero page thread), — это поток 0 в процессе System. Поток обнуления страниц ждет сигнала на работу от объекта шлюза. Когда в списке свободных имеется восемь и более страниц, шлюз подает сигнал. Но поток обнуления страниц запускается, только если хотя бы у одного процессора нет других выполняемых потоков, поскольку поток обнуления страниц запускается с приоритетом 0, а наименьший приоритет, который может быть установлен для пользовательского потока, равен 1.

ПРИМЕЧАНИЕ Когда память в результате выделения физической страницы драйвером, вызывающим функцию `MmAllocatePagesForMdl` или `MmAllocatePagesForMdlEx`, должна быть заполнена нулями Windows-приложением, вызывающим функцию `AllocateUserPhysicalPages` или `AllocateUserPhysicalPagesNuma`, или когда приложение выделяет большие страницы, диспетчер памяти обнуляет память, используя высокопроизводительную функцию под названием `MiZeroInParallel`, которая отображает более крупные области, чем поток обнуления страниц, обнуляющий только одну страницу за раз. Кроме того, на мультипроцессорных системах диспетчер памяти создает дополнительный системный поток для обнуления в параллельном режиме (а на NUMA-платформах это делается в стиле, оптимизированном под технологию NUMA).

- Когда диспетчеру памяти не нужна страница, заполненная нулями, он сначала обращается к списку свободных страниц. Если этот список пуст, он переходит к списку обнуленных страниц. Если и список обнуленных страниц пуст, он переходит к списку ожидающих страниц. Перед тем как диспетчер памяти сможет воспользоваться страничным блоком из списка ожидающих страниц, он должен сначала вернуться и удалить ссылку из недостоверной PTE-записи (или из прототипной PTE-записи), которая все еще указывает на страничный блок. Поскольку в записях базы данных PFN-номеров содержатся обратные указатели на предыдущую страницу пользовательской таблицы страниц (или на страницу пула прототипной PTE-записи для общих страниц), диспетчер памяти может быстро найти PTE-запись и внести в нее соответствующее изменение.
- Когда процесс должен отказаться от страницы из своего рабочего набора (либо потому, что он ссылается на новую страницу и его рабочий набор заполнен, либо потому, что диспетчер памяти урезал его рабочий набор), страница переходит в список ожидающих, если она оставалась нетронутой (неизмененной),

или в список измененных страниц, если страница была изменена, находясь в физической памяти.

- Когда процесс завершает работу, все закрытые страницы переходят в список свободных страниц. Кроме того, если при закрытии последней ссылки на раздел, поддерживаемый страничным файлом, в разделе не осталось отображенных представлений, страницы этого раздела также попадают в список свободных страниц.

ЭКСПЕРИМЕНТ: ПРОСМОТР СПИСКОВ СВОБОДНЫХ И ОБНУЛЕННЫХ СТРАНИЦ

За освобождением закрытых страниц при завершении работы процесса можно наблюдать в окне System Information (Системная информация) программы Process Explorer. Сначала нужно создать процесс с большим количеством закрытых страниц в его рабочем наборе. Мы уже делали это в одном из предыдущих экспериментов с помощью утилиты TestLimit:

```
C:\Tools\Sysinternals>Testlimit.exe -d 1 -c 1500

Testlimit v5.24 - test Windows limits
Copyright (C) 2012-2015 Mark Russinovich
Sysinternals - www.sysinternals.com

Process ID: 13928

Leaking private bytes with touch 1 MB at a time...
Leaked 1500 MB of private memory (1500 MB total leaked). Lasterror: 0
The operation completed successfully.
```

Ключ `-d` заставляет TestLimit не только выделить память в качестве закрытой и подтвержденной, но и «прикоснуться» к ней, т. е. обратиться к этой памяти. Это приводит к выделению физической памяти и присвоению ее процессу, чтобы освободить область закрытой подтвержденной виртуальной памяти. Если в системе имеется достаточный объем доступной оперативной памяти, для процесса в оперативной памяти будет выделено целых 1500 Мбайт. Теперь этот процесс будет ждать, пока вы не заставите его завершить или прервать работу (возможно, с комбинацией клавиш Ctrl+C в его командном окне). Выполните следующие действия.

1. Откройте Process Explorer.
2. Выберите команду View ▸ System Information и перейдите на вкладку Memory.
3. Понаблюдайте за размерами списков свободных (Free) и обнуленных (Zeroed) страниц.
4. Завершите или прервите процесс TestLimit.

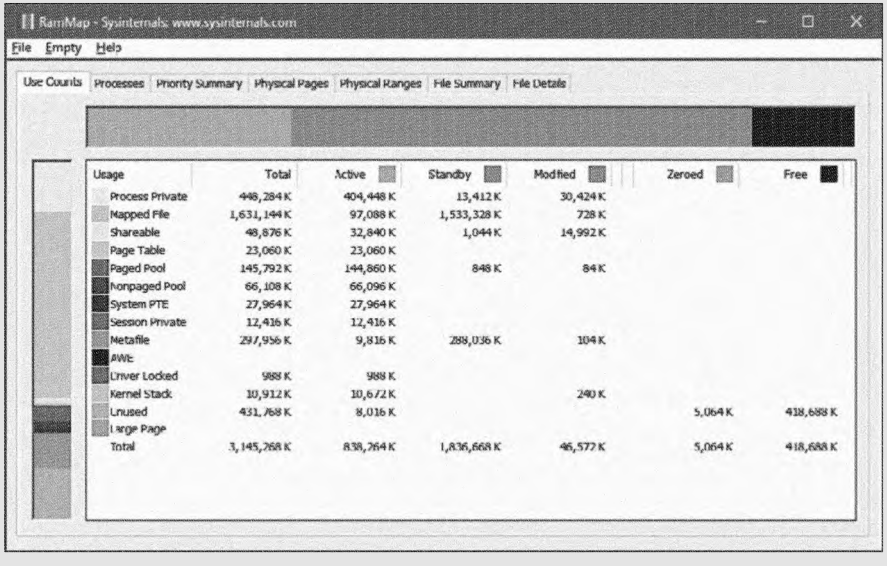
Возможно, вам удастся увидеть, что список свободных страниц кратковременно увеличился в размере. Мы говорим «возможно», потому что поток обнуления страниц «проснется», как только в списке обнуленных страниц останется всего восемь записей, и отработает очень быстро. Process Explorer обновляет это окно только раз в секунду, и похоже, что остальные страницы уже успе-

вают обнулиться и попасть в список обнуленных страниц, пока нам удалось «поймать» это состояние. Если вам удалось заметить временное увеличение списка свободных страниц, то вслед за этим вы увидите, что его размер упадет до нуля, а в списке обнуленных страниц произойдет соответствующее увеличение. Если же момент будет упущен, вы просто увидите увеличение списка обнуленных страниц.

ЭКСПЕРИМЕНТ: ПРОСМОТР СПИСКОВ ИЗМЕНЕННЫХ И ОЖИДАЮЩИХ СТРАНИЦ

За перемещением страниц из рабочего набора процесса в список измененных страниц и затем в список ожидающих страниц можно понаблюдать при помощи программ VMMap и RAMMap из пакета Sysinternals или в отладчике ядра. Выполните следующие действия.

1. Запустите программу RAMMap и понаблюдайте за спокойным состоянием системы. В данном случае это система x86 с 3 Гбайт оперативной памяти. Столбцы в окне отражают различные состояния страниц (см. рис. 5.37). Некоторые столбцы, которые не имеют значения для данного эксперимента, для удобства были сужены.



Usage	Total	Active	Standby	Modified	Zeroed	Free
Process Private	448,284 K	404,448 K	13,412 K	30,424 K		
Mapped File	1,631,144 K	97,088 K	1,533,328 K	728 K		
Shareable	48,876 K	32,840 K	1,044 K	14,992 K		
Page Table	23,060 K	23,060 K				
Paged Pool	145,792 K	144,860 K	848 K	84 K		
Nonpaged Pool	66,108 K	66,096 K				
System PTE	27,964 K	27,964 K				
Session Private	12,416 K	12,416 K				
Metafile	297,956 K	9,816 K	288,036 K	104 K		
AWE						
Driver Locked	988 K	988 K				
Kernel Stack	10,912 K	10,672 K		240 K		
Unused	431,768 K	8,016 K			5,064 K	418,688 K
Large Page						
Total	3,145,768 K	838,264 K	1,836,668 K	46,572 K	5,064 K	418,688 K

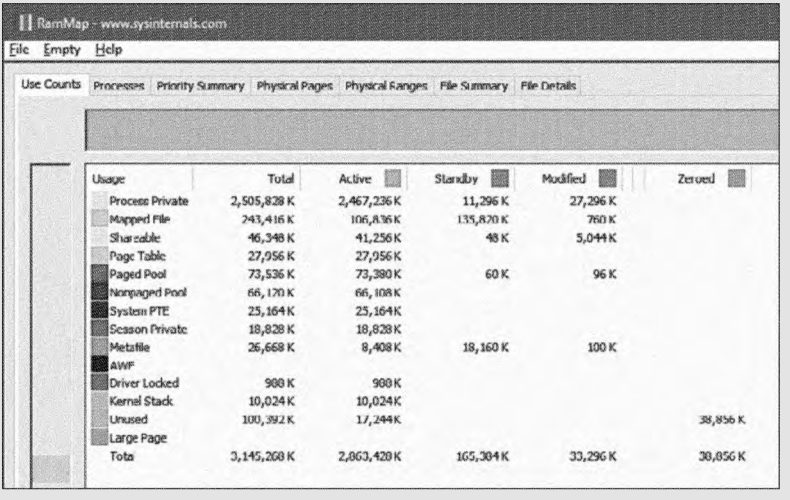
2. У системы имеется около 420 Мбайт свободной оперативной памяти (складывающейся из свободных и обнуленных страниц). Около 580 Мбайт фигурирует в списке ожидающих страниц (следовательно, часть из них «доступна», но, скорее всего, содержит данные, ранее утраченные процессами или используемые при супервыборке). Около 830 Мбайт активны, будучи отображенными непосредственно на виртуальные адреса через достоверные записи таблицы страниц.

3. Каждая строка далее разбивается в соответствии с состояниями страниц по использованию или происхождению (закрытые страницы процесса, отображаемый файл и т. д.). Например, на данный момент из активных 830 Мбайт около 400 Мбайт обусловлено выделением закрытых страниц процесса.

4. Теперь, как и в предыдущем эксперименте, воспользуйтесь утилитой TestLimit, чтобы создать процесс с большим количеством страниц в рабочем наборе. Здесь опять мы используем ключ –d, чтобы заставить TestLimit сделать запись в каждую страницу, но на этот раз без ограничения, чтобы создать как можно больше закрытых измененных страниц:

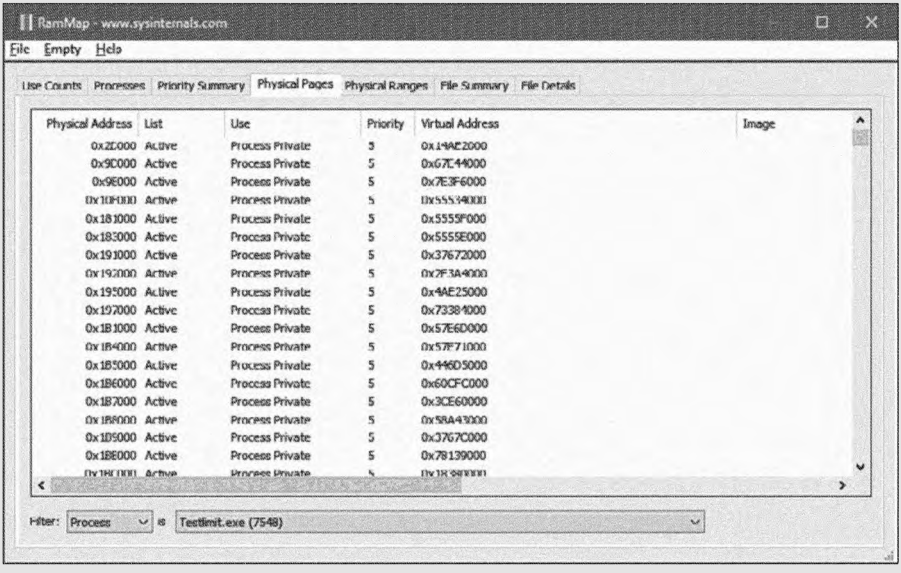
```
C:\Tools\Sysinternals>Testlimit.exe -d

Testlimit v5.24 - test Windows limits
Copyright (C) 2012-2015 Mark Russinovich
Sysinternals - www.sysinternals.com

Process ID: 7548

Leaking private bytes with touch (MB)...
Leaked 1975 MB of private memory (1975 MB total leaked). Lasterror: 8
```

5. Теперь программа TestLimit создала 1975 областей выделения по 1 Мбайт каждая. Для обновления экрана в программе RAMMap нужно воспользоваться командой File ▸ Refresh для обновления экрана, поскольку самостоятельно программа RAMMap этого не делает (из-за больших затрат на выполнение этой операции).



6. Как видите, активно свыше 2,8 Гбайт, из которых 2,4 Гбайт находятся в строке закрытых страниц процесса (строка Process Private). Это результат выделения памяти и доступа к ней со стороны процесса TestLimit. Также обратите внимание на то, что списки ожидающих (Standby), обнуленных (Zeroed) и свободных (Free) страниц теперь стали намного меньше. Большая часть памяти, выделенной программе TestLimit, взята из страниц, фигурировавших в этих списках.

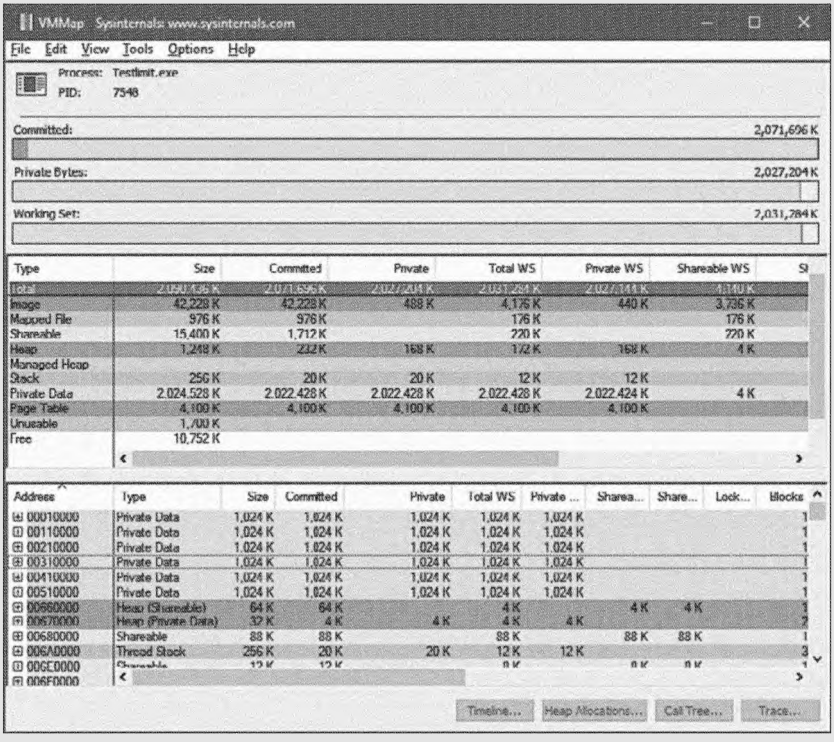
7. Далее с помощью RAMMap нужно оценить выделение физических страниц процесса. Перейдите на вкладку Physical Pages и установите фильтр, находящийся в нижней части столбца Process, присвоив ему значение Testlimit.exe. В следующем окне показаны все физические страницы, являющиеся частью рабочего набора процесса.

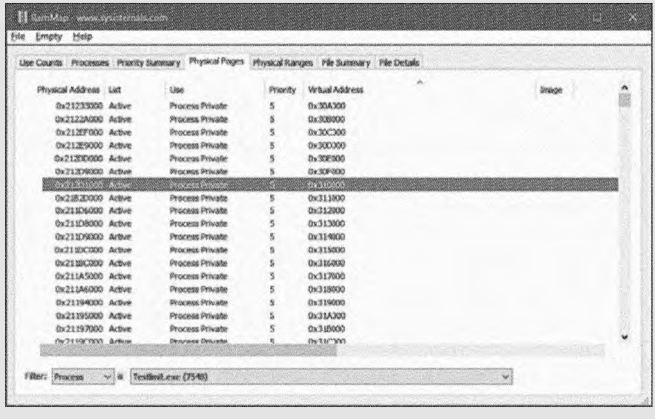


8. Нам нужно идентифицировать физическую страницу, задействованную в выделении физического адресного пространства, которое было выполнено с помощью ключа `-d` при запуске программы TestLimit. RAMMap не дает никаких указаний на то, какие виртуальные области были выделены благодаря вызову из RAMMap функции `VirtualAlloc`. Но мы можем получить ценную подсказку на этот счет с помощью программы VMMap. Вызвав VMMap для того же процесса, мы получим следующий результат (см. рис. на с. 535 вверху).

9. В нижней части выводимой информации находятся сотни выделенных областей для закрытых данных процесса, каждая из которых имеет размер 1 Мбайт при 1 Мбайт подтвержденной памяти. Это соответствует размеру памяти, выделенной программой TestLimit. В предыдущей копии экрана подсвечен первый из таких вариантов распределения. Заметьте, что его начальный виртуальный адрес равен 0x310000.

10. Теперь вернемся к информации о физической памяти, выводимой на экран программой RAMMap. Перестройте столбцы так, чтобы хорошо был виден столбец Virtual Address. Щелкните на нем, чтобы отсортировать строки по этому значению, и вы сможете найти нужный виртуальный адрес (см. рисунок на с. 535 внизу).





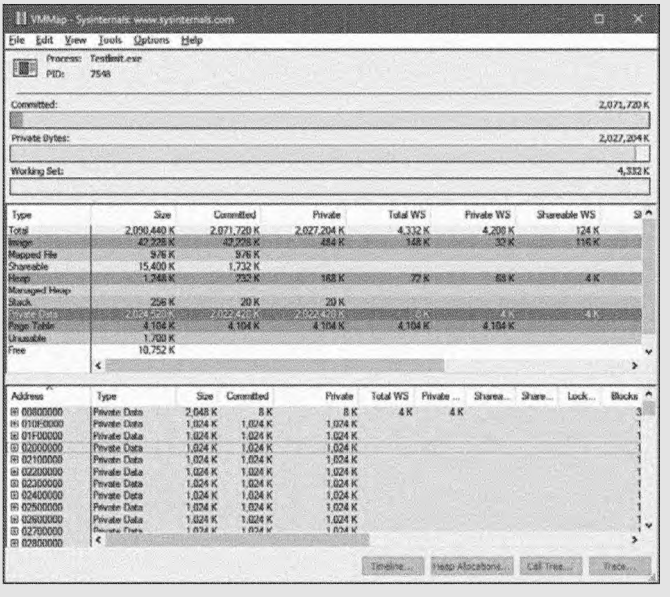
11. Здесь показано, что виртуальная страница, начинающаяся с адреса 0x310000, в данный момент отображена на физический адрес 0x212D1000. С ключом `-d` программа TestLimit записывает в первые байты каждой выделенной области свое имя. Это можно продемонстрировать с помощью команды `!dc` локаль-

ного отладчика ядра (dc — сокращение от «display characters», т. е. вывод символов по физическому адресу):

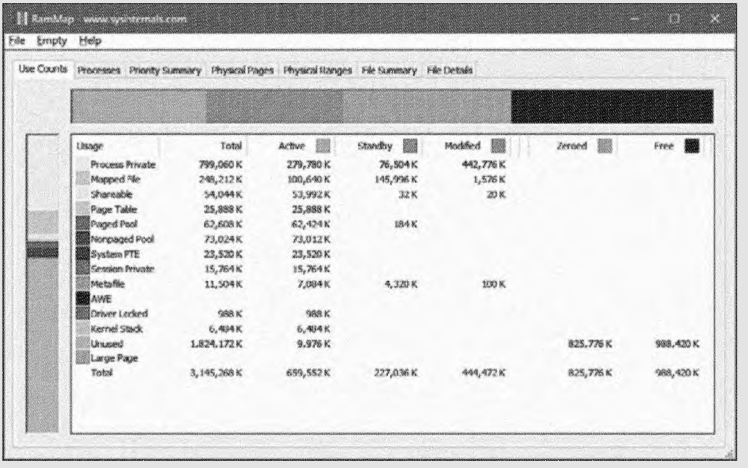
```
lkd> !dc 0x212d1000
#212d1000 74736554 696d694c 00000074 00000000 TestLimit.......
#212d1010 00000000 00000000 00000000 00000000 ................
...
```

12. Если промедлить, попытка может завершиться неудачей — страница может быть уже удалена из рабочего набора. В последней фазе эксперимента мы покажем, что данные остаются неизменными (по крайней мере, на какое-то время) после того, как рабочий набор процесса сократится, а страница переместится сначала в список измененных, а затем — ожидающих страниц.

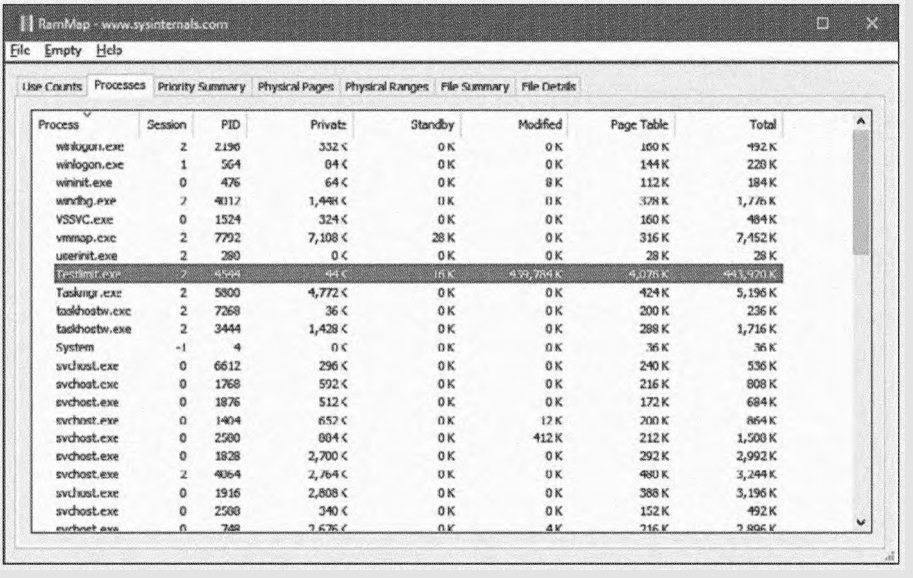
13. Выбрав в программе VMMap процесс TestLimit, откройте меню View и выберите команду Empty Working Set, чтобы сократить рабочий набор процесса до минимума. Теперь в окне VMMap должна выводиться следующая информация:



14. Обратите внимание на то, что линейка Working Set (Рабочий набор) практически пуста. В средней части для процесса показано, что общий размер рабочего набора равен всего лишь 4 Кбайт, причем почти все его пространство занято таблицами страниц. Теперь вернитесь в RAMMap. На вкладке Use Counts видно, что количество активных страниц существенно сократилось, а большое количество страниц находится в списке измененных и существенное количество страниц — в списке ожидающих.



15. Данные на вкладке Processes программы RAMMap подтверждают, что большинство этих страниц появилось в данных списках из-за процесса TestLimit.



Приоритеты страниц

Диспетчер памяти назначает приоритет каждой физической странице, присутствующей в системе. Приоритет страницы представляет собой число в диапазоне от 0 до 7. Его главное назначение — определение порядка расходования страниц в списке ожидающих страниц. Диспетчер памяти делит список ожидающих страниц на восемь подсписков, в каждом из которых хранятся записи для страниц определенного при-

оритета. Когда диспетчер памяти собирается взять страницу из списка ожидающих, сначала он берет страницу из подсписков страниц с наименьшими приоритетами.

Каждый поток и каждый процесс в системе также присваивает странице приоритет. Приоритет страницы обычно отражает приоритет страницы того программного потока, который стал причиной первого выделения памяти. (Если страница общая, он отражает наивысший приоритет страницы среди совместно использующих ее потоков.) Поток наследует свое значение страничного приоритета у процесса, которому принадлежит. Для страниц, которые диспетчер памяти считывает с диска, выстраивая предположение о порядке обращения к памяти со стороны процесса, он использует низкие уровни приоритета.

По умолчанию процессы задействуют приоритет страниц, равный 5, но функции пользовательского режима `SetProcessInformation` и `SetThreadInformation` позволяют приложениям и системе изменять приоритет страниц, используемый процессами и потоками. Приоритет, применяемый потоком, можно увидеть с помощью программы Process Explorer (приоритет каждой страницы можно выяснить путем анализа PFN-записей, что показано в одном из следующих экспериментов данной главы). На рис. 5.38 показана вкладка Threads программы Process Explorer с информацией об основном потоке программы Winlogon. Хотя приоритет самого потока довольно высок, приоритет памяти имеет стандартное значение 5.

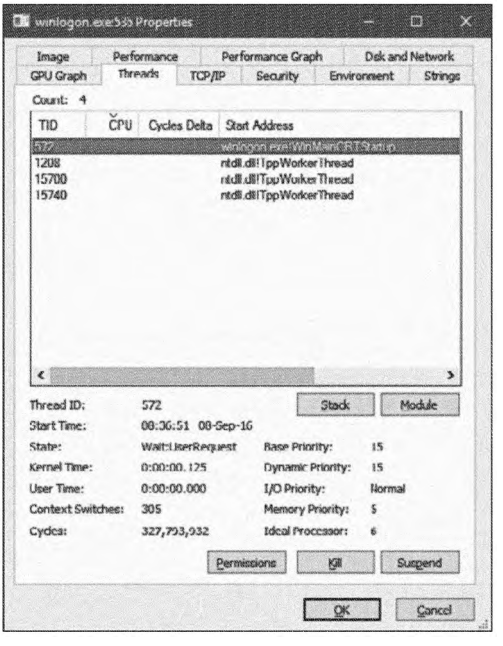
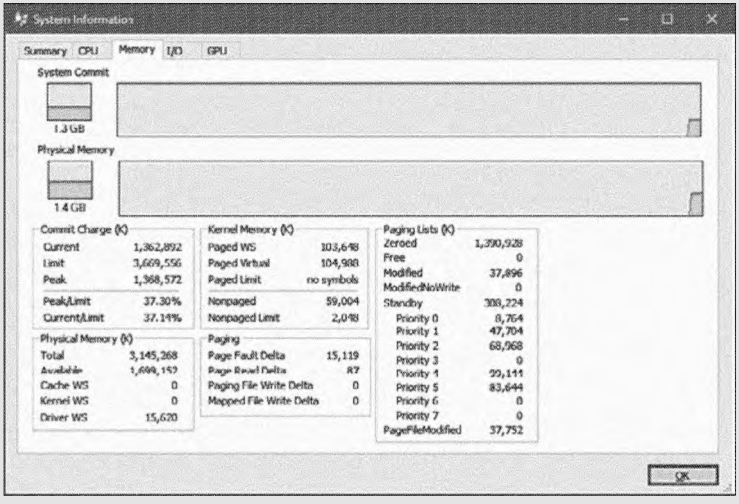


Рис. 5.38. Вкладка Threads в программе Process Explorer

Реальное влияние приоритетов памяти проявляется только при восприятии приоритетов страниц на высоком уровне. В этом и состоит роль супервыборки, рассматриваемой в конце данной главы.

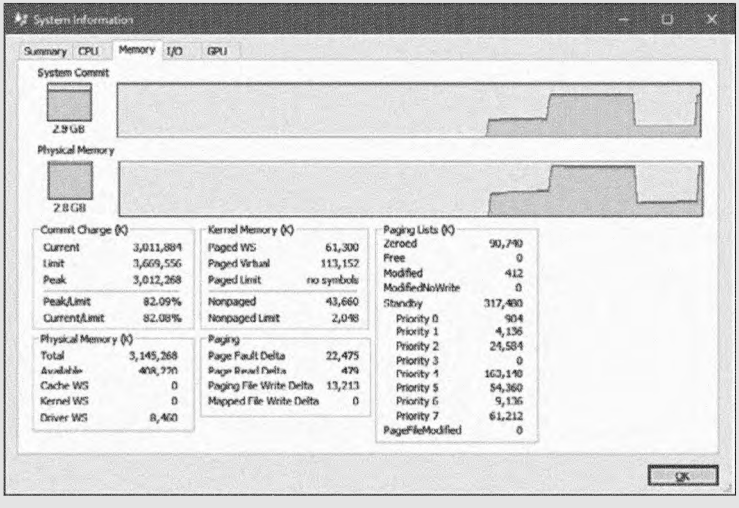
ЭКСПЕРИМЕНТ:ПРОСМОТР СПИСКОВ ОЖИДАЮЩИХ СТРАНИЦ, УПОРЯДОЧЕННЫХ ПО ПРИОРИТЕТАМ

Чтобы просмотреть все списки ожидающих страниц, воспользуйтесь программой Process Explorer — откройте диалоговое окно System Information и перейдите на вкладку Memory:



На недавно запущенной системе x86, использованной в эксперименте, присутствует около 9 Мбайт данных, кэшированных с приоритетом 0, данных с приоритетом 1–47 Мбайт, данных с приоритетом 2–68 Мбайт и т. д. На следующем снимке показано, что произойдет, если воспользоваться программой TestLimit из пакета TestLimit для подтверждения и фиксации как можно большего объема памяти:

```
C:\Tools\Sysinternals>Testlimit.exe -d
```



Обратите внимание: сначала были использованы страницы из списков ожидающих страниц с низшими приоритетами (о чем свидетельствуют показания подсчета повторно использованных страниц), которые теперь исчерпаны, а в списках страниц с более высокими приоритетами по-прежнему содержатся ценные кэшированные данные.

Подсистема записи измененных страниц

Диспетчер памяти для записи страниц обратно на диск и для перемещения таких страниц в списки ожидающих страниц (на основе их приоритетов) задействует два системных потока. Один системный поток (`MiModifiedPageWriter`) записывает измененные страницы в страничный файл, другой (`MiMappedPageWriter`) записывает измененные страницы в отображаемые файлы. Два потока необходимы для того, чтобы избежать взаимных блокировок, возникающих, если запись страниц отображаемого файла становится причиной ошибки страницы, которая, в свою очередь, требует свободной страницы при отсутствии доступных свободных страниц (что требует от потока записи измененных страниц создания дополнительных свободных страниц). За счет того, что подсистема записи измененных страниц выполняет страничный ввод/вывод отображаемого файла из второго системного потока, этот поток может ожидать без блокировки обычного страничного ввода/вывода.

Оба потока выполняются с приоритетом 18, и после инициализации они ждут для запуска своих операций разных объектов событий. Поток, записывающий отображаемые страницы, ждет наступления 18 объектов событий.

- Событие выхода, сигнализирующее о выходе из потока (к данному обсуждению не относится).
- Событие подсистемы записи отображенных страниц, хранящееся в глобальной переменной `MiSystemPartition.Modwriter.MappedPageWriterEvent` (`MmMappedPageWriterEvent` в Windows 8.x и Server 2012/R2).
- Это событие может выдаваться в следующих случаях:
  - В ходе операции со списками страниц (`MiInsertPageInList`); эта функция вставляет страницу в один из списков (ожидающих, измененных и т. д.) в зависимости от аргументов. Функция выдает событие, если количество страниц, предназначенных для файловой системы и занесенных в список измененных страниц, превысило 16, а количество доступных страниц упало ниже 1024.
  - При попытке получить свободные страницы (`MiObtainFreePages`).
  - Диспетчером рабочих наборов в диспетчере памяти (`MmWorkingSetManager`), который запускается как часть имеющегося в ядре диспетчера настройки баланса (один раз в секунду). Диспетчер рабочих наборов выдает сигнал в виде этого события, если количество страниц, предназначенных для файловой системы и занесенных в список измененных страниц, превысило 800.

- По запросу на сброс всех измененных страниц (`MmFlushAllPages`).
- По запросу на сброс всех предназначенных для файловой системы измененных страниц (`MmFlushAllFilesystemPages`). Следует учесть, что в большинстве случаев запись измененных отображаемых страниц в файлы их резервного хранилища не происходит, если количество отображаемых страниц в списке измененных страниц меньше, чем максимальный размер «кластера записи», который составляет 16 страниц. В `MmFlushAllFilesystemPages` или `MmFlushAllPages` такая проверка не проводится.

✦ Массив из 16 событий, связанных 16 списками отображаемых страниц; хранится в `MiSystemPartition.PageLists.MappedPageListHeadEvent` (`MiMappedPageListHeadEvent` в Windows 8.*x* и Server 2012/R2). При каждом внесении изменений в отображаемую страницу она вставляется в один из 16 списков отображаемых страниц по номеру гнезда (`MiCurrentMappedPageBucket`). Этот номер гнезда обновляется диспетчером рабочих наборов, когда система посчитает, что отображаемые страницы до определенной степени устарели и сейчас эта степень составляет 100 секунд (контролируется значением переменной `MiWriteGapCounter` в Windows 8.*x* и Server 2012/R2), которое увеличивается на 1 при каждом запуске диспетчера рабочих наборов). Все эти дополнительные события используются для сокращения риска потери данных в случае фатального сбоя системы или отказа электропитания; для этого происходит периодическая запись измененных отображаемых страниц, даже если список измененных страниц не достиг своего порога в 800 страниц.

Подсистема записи измененных страниц ожидает записи двух событий: первое — событие Exit, а второе хранится в `MiSystemPartition.Modwriter.ModifiedPageWriterEvent` (в Windows 8.*x* и Server 2012/R2 ожидание ведется по событию объекта шлюза (`MmModifiedPageWriterGate`), от которого может поступить сигнал при развитии следующих сценариев:

✦ Поступление запроса на сброс всех страниц.

✦ Падение количества доступных страниц, хранимого в `MiSystemPartition.Vp.AvailablePages` (`MmAvailablePages` в Windows 8.*x* и Server 2012/R2), ниже уровня 128 единиц.

✦ Падение общего количества страниц, заполненных нулевыми байтами, и свободных страниц ниже уровня 20 000, а также превышение количества измененных страниц, предназначенных для страничных файлов, до уровня, который превышает наименьшее из двух значений: либо одной шестнадцатой от доступных страниц, либо 64 Мбайт (16 384 страниц).

✦ Когда рабочий набор усечен и не может вместить дополнительные страницы, если количество доступных страниц стало менее 15 000.

✦ В ходе операции со списком страниц (`MiInsertPageInList`). Эти процедуры сигнализируют шлюзу, если количество страниц, предназначенных для страничных

файлов в списке измененных страниц, стало больше 16, а количество доступных страниц упало ниже 1024.

Кроме того, подсистема записи измененных страниц ждет наступления двух событий после предыдущего события. Одно указывает на необходимость повторного сканирования страничного файла (например, создания нового страничного файла); оно хранится в `MiSystemPartition.Modwriter.RescanPageFilesEvent` (`MiRescanPageFilesEvent` в Windows 8.*x* и Server 2012/R2). Второе событие, внутреннее для заголовка страничного файла — `MiSystemPartition.Modwriter.RescanPageFilesEvent` (`MiRescanPageFilesEvent` в Windows 8.*x* и Server 2012/R2). Оба события позволяют системе самостоятельно запрашивать сброс данных в страничный файл по мере надобности.

При активации подсистема записи отображаемых файлов делает попытку записать на диск как можно больше страниц за один запрос ввода/вывода. Для этого изучается исходное поле PTE, имеющееся в записи базы данных PFN-номеров для страниц, входящих в список измененных страниц, с целью размещения страниц в непрерывных областях на диске. Как только список будет создан, страницы удаляются из списка измененных страниц, выдается запрос ввода/вывода, и при успешном завершении запроса ввода/вывода страницы помещаются в конец списка ожидающих страниц в соответствии со своими приоритетами.

К страницам, находящимся в стадии записи, могут быть обращения со стороны других потоков. В таком случае показания счетчика ссылок и счетчика количества пользователей в PFN-записи, представляющей физическую страницу, увеличивается на единицу, показывая тем самым, что страница используется другим процессом. Когда операция ввода/вывода завершается, подсистема записи измененных страниц замечает, что счетчик ссылок больше не является нулевым, и не помещает страницу в список ожидающих страниц.

Структуры данных PFN-записи

Хотя записи базы данных PFN-номеров имеют фиксированную длину, они могут находиться в нескольких различных состояниях в зависимости от состояния страницы. Таким образом, в зависимости от этого состояния отдельные поля имеют разный смысл. Формат PFN-записей для разных состояний показан на рис. 5.39.

Для некоторых типов PFN-записей поля имеют одинаковое предназначение, другие поля характерны только для определенных типов PFN-записей. В нескольких типах PFN-записей есть следующие поля:

- **Адрес PTE-записи** (PTE address). Виртуальный адрес PTE-записи, указывающей на данную страницу. Кроме того, поскольку адреса PTE-записей всегда выравниваются по 4-байтовой границе (8-байтовой в 64-разрядных системах), два младших разряда используются в качестве блокировочного механизма для последовательного доступа к PFN-записи.

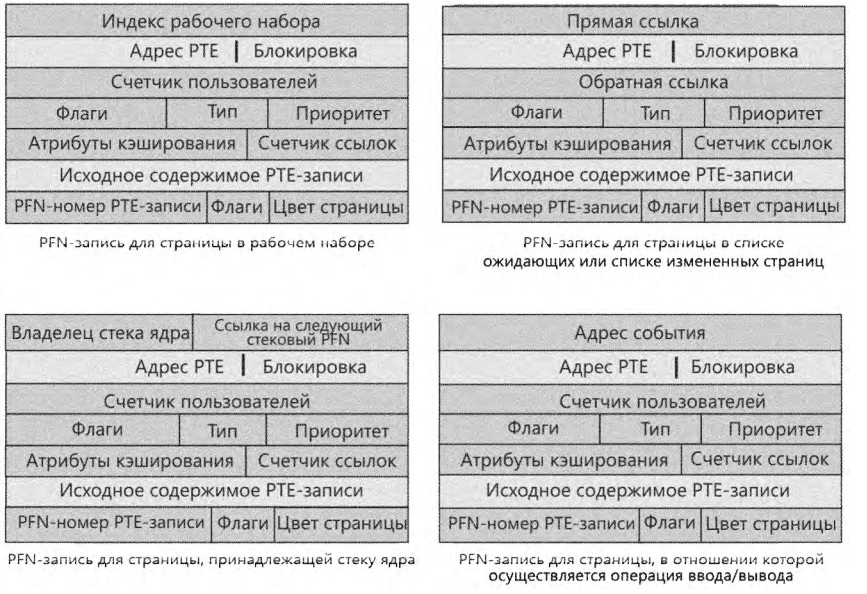


Рис. 5.39. Состояния записей базы данных PFN-номеров (концептуальные, а не конкретные)

- **Счетчик ссылок** (reference count). Количество ссылок на данную страницу. Показание счетчика ссылок увеличивается на 1, когда страница впервые добавляется к рабочему набору и/или когда страница заблокирована в памяти для выполнения операции ввода/вывода (например, драйвером устройства). Счетчик ссылок уменьшается на 1, когда счетчик пользователей обнуляется или когда страница деблокируется в памяти. Когда счетчик пользователей обнуляется, страница больше не принадлежит рабочему набору. Затем, если счетчик ссылок также равен 0, запись базы данных PFN-номеров, которая описывает страницу, обновляется для добавления страницы в список свободных, ожидающих или измененных страниц.
- **Тип** (type). Тип страницы, представленной данной PFN-записью. (Имеются следующие типы: активная/достоверная; ожидающая; измененная; измененная, но не предназначенная для записи; свободная; заполненная нулевыми байтами; нерабочая; в переходном состоянии.)
- **Флаги** (flags). Информация, содержащаяся в поле флагов, показана в табл. 5.20.
- **Приоритет** (priority). Приоритет, связанный с этой PFN-записью, определяет, в какой из списков ожидающих страниц попадет данная страница.
- **Исходное содержимое PTE-записи** (original PTE contents). Во всех записях базы данных PFN-номеров содержится исходное содержимое той PTE-записи, которая указывает на страницу (и которая может быть прототипной PTE-записью). Сохранение содержимого PTE-записи позволяет ее восстановить,

когда физическая страница уже не находится в резидентной памяти. Исключением являются PFN-записи для AWE-вариантов размещения; вместо этого в данном поле в них хранится счетчик AWE-ссылок.

- **PFN-номер для PTE-записи** (PFN of PTE). Номер той физической страницы, в которой находится таблица страниц с PTE-записью, указывающей на страницу с PFN-записью.
- **Цвет** (Color). Кроме того что записи базы данных PFN-номеров связаны одним списком, в них используется дополнительное поле, связывающее физические страницы по «цвету», представляющему собой номер NUMA-узла страницы.
- **Флаги** (Flags). Второе поле флагов используется для кодировки дополнительной информации, касающейся PTE-записи. Эти флаги описаны в табл. 5.21.

Все остальные поля характерны для того или иного типа PFN-записей. Например, первая PFN-запись на рис. 5.39 представляет страницу, находящуюся в активном состоянии и входящую в рабочий набор. Поле счетчика пользователей представляет количество PTE-записей, ссылающихся на данную страницу. (Страницы, помеченные как предназначенные только для чтения, для копирования при

Таблица 5.20. Флаги в записях базы данных PFN-номеров

Флаг	Значение
Write in progress (Выполняется запись)	Выполняется операция записи страницы. Первый DWORD-параметр содержит адрес объекта события, который подаст сигнал, когда операция ввода/вывода будет завершена
Modified state (Измененное состояние)	Страница была изменена. (Если страница была изменена, ее содержимое должно быть сохранено на диске перед удалением этой страницы из памяти.)
Read in progress (Выполняется чтение)	В отношении данной страницы выполняется страничная операция. Первый DWORD-параметр содержит адрес объекта события, который подаст сигнал, когда операция ввода/вывода будет завершена
ROM (Получена из постоянной памяти)	Страница поступила из встроенного компьютерного программного обеспечения или из другого места памяти, предназначенного только для чтения, например из регистра устройства
In-page error (Ошибка при проведении операции над страницей)	В ходе страничной операции в отношении данной страницы произошла ошибка ввода/вывода. (В этом случае в первом поле PFN-записи содержится код ошибки.)
Kernel stack (Стек ядра)	Страница использована как содержимое стека ядра. В этом случае в PFN-записи содержатся сведения о владельце стека и о следующей стековой PFN-записи данного потока
Removal requested (Запрошено удаление)	Страница подлежит удалению (из-за ECC-кода или очистки либо из-за удаления блока памяти из работающей машины)
Parity error (Ошибка четности)	В физической странице содержится ошибка четности или ошибка контроля коррекции ошибок

Таблица 5.21. Вторичные флаги в записях базы данных PFN

Флаг	Значение
PFN image verified (Образ PFN проверен)	Код проверенной сигнатуры для данной PFN-записи (содержащийся в каталоге криптографической сигнатуры для образа, поддержанного этой PFN-записью)
AWE allocation (AWE-выделение)	Эта PFN-запись поддерживает AWE-выделение памяти
Prototype PTE (Прототипная PTE-запись)	Показывает, что PTE-запись, на которую ссылается данная PFN-запись, является прототипной. (Например, страница предназначена для совместного использования.)

записи или для совместного чтения/записи, могут совместно использоваться сразу несколькими процессами.) Для страниц, содержащих таблицы страниц, это поле содержит количество достоверных или переходных PTE-записей в таблице страниц. Пока значение счетчика пользователей больше нуля, страница не подлежит удалению из памяти.

Поле индекса рабочего набора является индексом в списке рабочего набора процесса (или в списках рабочих наборов системы или сеанса, при отсутствии каких-либо рабочих наборов содержит нуль), в котором находится виртуальный адрес, отображаемый на данную физическую страницу. Если страница является закрытой, поле индекса рабочего набора ссылается непосредственно на запись в списке рабочего набора, потому что страница отображена только на один виртуальный адрес. В случае с совместно используемыми страницами индекс рабочего набора является рекомендацией, правильность которой гарантируется только для первого процесса, сделавшего страницу достоверной. (Другие процессы будут пытаться по возможности использовать тот же индекс.) Процессу, изначально установившему значение для данного поля, гарантируется ссылка на правильный индекс, и он не нуждается в добавлении записи хеша списка рабочего набора, ссылающейся по виртуальному адресу в его дереве хешей рабочего набора. Тем самым обеспечивается уменьшение размера дерева хешей рабочего набора и ускоряется поиск таких записей.

Вторая PFN-запись на рис. 5.39 предназначена для страницы, занесенной либо в список ожидающих, либо в список измененных страниц. В этом случае поля прямой и обратной ссылок связывают элементы списка друг с другом. Эта связь упрощает работу со страницами при устранении ошибок страниц. Если страница принадлежит одному из списков, счетчик пользователей по определению равен нулю (поскольку страница не используется ни одним рабочим набором), и поэтому страница может быть перекрыта обратной ссылкой. Если страница входит в один из списков, счетчик ссылок также равен нулю. Если он не равен нулю (из-за того, что со страницей выполняется операция ввода/вывода, например когда страница записывается на диск), то сначала страница удаляется из списка.

Третья PFN-запись на рис. 5.39 предназначена для страницы, принадлежащей стеку ядра. Как уже упоминалось, стеки ядра в Windows выделяются в динамическом режиме, расширяясь и освобождаясь при обратном вызове кода пользовательского

режима и/или при возвращении управления, либо когда драйвер выполняет обратный вызов и запрашивает расширение стека. Для таких PFN-записей диспетчер памяти должен следить за потоком, который в данный момент связан со стеком ядра, или, если стек свободен, следить за связью со следующим свободным стеком.

Четвертая PFN-запись на рис. 5.39 предназначена для страницы, с которой выполняется операция ввода/вывода (например, чтение страницы). Пока идет операция ввода/вывода, значение первого поля указывает на объект события, который подаст сигнал о завершении ввода/вывода. Если происходит ошибка, связанная со страничной операцией, это поле содержит код статуса ошибки, отражающий ошибку ввода/вывода. Такой тип PFN-записи используется для разрешения конфликтных ошибок страниц.

В дополнение к базе данных PFN, общее состояние физической памяти описывается системными переменными, перечисленными в табл. 5.22.

Таблица 5.22. Системные переменные, описывающие физическую память

Переменная (Windows 10 и Server 2016)	Переменная (Windows 8.x и Server 2012/R2)	Описание
`MiSystemPartition.Vp.NumberOfPhysicalPages`	`MmNumberOfPhysicalPages`	Общее количество физических страниц, доступных в системе
`MiSystemPartition.Vp.AvailablePages`	`MmAvailablePages`	Общее количество доступных страниц в системе (сумма страниц в списках обнуленных, свободных и ожидающих страниц)
`MiSystemPartition.Vp.ResidentAvailablePages`	`MmResidentAvailablePages`	Общее количество физических страниц, которые станут доступными, если каждый процесс задействует усеченный до минимума размер рабочего набора и все измененные страницы будут сброшены на диск

ЭКСПЕРИМЕНТ: ПРОСМОТР PFN-ЗАПИСЕЙ

Отдельные PFN-записи можно изучать с помощью команды `!pfn` отладчика ядра. В аргументе этой команде передается номер PFN-записи. (Например, команда `!pfn 0` покажет первую запись, `!pfn 1` — вторую и т. д.) В следующем примере показана PTE-запись для виртуального адреса 0xD20000, за которой следует PFN-запись, содержащая каталог страниц, затем следует фактическая страница:

```
lkd> !pte d20000
                   VA 00d20000
PDE at C0600030            PTE at C0006900
contains 000000003E989867  contains 8000000093257847
pfn 3e989     ---DA--UWEV  pfn 93257     ---D---UW-V
```

```
lkd> !pfn 3e989
    PFN 0003E989 at address 868D8AFC
    flink       00000071  blink / share count 00000144  pteaddress C0600030
    reference count 0001   Cached     color 0   Priority 5
    restore pte 00000080  containing page 0696B3  Active     M
    Modified

lkd> !pfn 93257
    PFN 00093257 at address 87218184
    flink       000003F9  blink / share count 00000001  pteaddress C0006900
    reference count 0001   Cached     color 0   Priority 5
    restore pte 00000080  containing page 03E989  Active     M
    Modified
```

Для получения информации о PFN-записи можно воспользоваться программой MemInfo. Иногда она может дать даже больше информации, чем отладчик, к тому же она не требует перезагрузки в режиме отладки. Вот как выглядят выводимые программой MemInfo данные для тех же самых двух PFN-записей:

```
C:\Tools>MemInfo.exe -p 3e989
0x3E989000 Active      Page Table       5  N/A          0xC0006000 0x8E499480

C:\Tools>MemInfo.exe -p 93257

0x93257000 Active      Process Private  5  windbg.exe   0x00D20000 N/A
```

Слева направо указываются: физический адрес, тип, приоритет страницы, имя процесса, виртуальный адрес и необязательная дополнительная информация. MemInfo правильно распознает тот факт, что первая PFN-запись относится к таблице страниц, а вторая PFN-запись принадлежит процессу WinDbg, который был активным, когда в отладчике использовалась команда `!pte 20000`.

Резервирование страничных файлов

Мы уже видели некоторые механизмы, используемые диспетчером памяти для сокращения физического потребления памяти, а следовательно, сокращения количества обращений к страничным файлам. Один из таких механизмов — списки ожидающих и измененных страниц, как и сжатие памяти (см. раздел «Сжатие данных» этой главы). Другой механизм оптимизации напрямую связан с обращением к самим страничным файлам.

У механических жестких дисков имеется подвижная головка, которая перед выполнением фактического чтения или записи должна переместиться к целевому сектору. Время позиционирования относительно велико (порядка миллисекунд), и фактическое время чтения/записи увеличивается на время физических операций с диском. Если система работатет с достаточно большим блоком смежных данных от установленной позиции, время позиционирования может быть пренебрежимо малым. Но если головка должна много перемещаться для обращения к данным, разбросанным по диску, суммарное время позиционирования может стать серьезной проблемой.

При создании страничного файла диспетчер сеансов (Smss.exe) запрашивает информацию у диска с файловым разделом и определяет, является ли диск механическим и твердотельным (SSD). Для механического диска активизируется механизм *резервирования страничных файлов*, который старается хранить страницы, смежные в физической памяти, смежными в страничном файле. Для SSD-диска (или гибридного диска, который в контексте резервирования страниц рассматривается как SSD-диск) резервирование страничных файлов особой пользы не приносит (из-за отсутствия механического привода), и для этого конкретного страничного файла данный механизм не применяется.

Резервирование страничных файлов производится диспетчером памяти в трех местах: в диспетчере рабочих наборов, подсистеме измененных страниц и обработчике ошибок страниц. Диспетчер рабочих наборов выполняет усечение рабочих наборов вызовом функции `MiFreeWsleList`. Функция получает список страниц от рабочего набора и для каждой страницы уменьшает счетчик пользователей. Если значение счетчика уменьшается до 0, страница может быть помещена в список измененных страниц, превращая актуальную PTE-запись в переходную. Старая достоверная PTE-запись сохраняется в PFN.

Недостоверная PTE-запись содержит два бита, связанные с резервированием страничных файлов: страничный файл зарезервирован и страничный файл выделен (см. рис. 5.24). Когда при возникновении необходимости физическая страница берется из одного из списков «незадействованных» страниц (свободных, заполненных нулями или ожидающих), чтобы стать активной (достоверной), недостоверная PTE-запись сохраняется в поле исходной PTE-записи в PFN. Это поле играет ключевую роль в отслеживании резервирования страничных файлов.

Функция `MiCheckReservePageFileSpace` пытается создать кластер резервирования страничных файлов, начиная с заданной страницы. Она проверяет, было ли отключено резевирование для целевого страничного файла и существует ли резервирование страничного файла для этой страницы (на основании исходной PTE-записи); если хотя бы одно из этих условий истинно, функция отменяет дальнейшую обработку этой страницы. Функция также проверяет, относится ли страница к типу пользовательских страниц, и если нет — прерывает выполнение. Для других типов страниц (например, страничного пула) попытки резервирования не делаются, потому что они не приносят заметной пользы (скажем, из-за непредсказуемых закономерностей использования) из-за малых размеров кластеров. Наконец, `MiCheckReservePageFileSpace` вызывает функцию `MiReservePageFileSpace` для выполнения непосредственной работы.

Поиск резервирования страничных файлов ведется в обратном направлении от исходной PTE-записи. Его целью является поиск подходящих смежных страниц, для которых возможно резервироваие. Если PTE-запись для соседней страницы представляет неподтвержденную страницу, страницу из невыгружаемого пула или если она уже зарезервирована, то эта страница не может использоваться; текущая страница становится нижней границей кластера резервирования. В противном случае поиск продолжается в обратном направлении. Затем поиск продолжается

от исходной страницы в прямом направлении, чтобы собрать как можно больше подходящих страниц. Для выполнения резервирования размер кластера должен составлять не менее 16 страниц (максимальный размер кластера 512 страниц). На рис. 5.40 изображен пример кластера, ограниченного недостоверной страницей с одной стороны и существующим кластером — с другой (обратите внимание: кластер может охватывать несколько таблиц страниц в одном каталоге страниц).

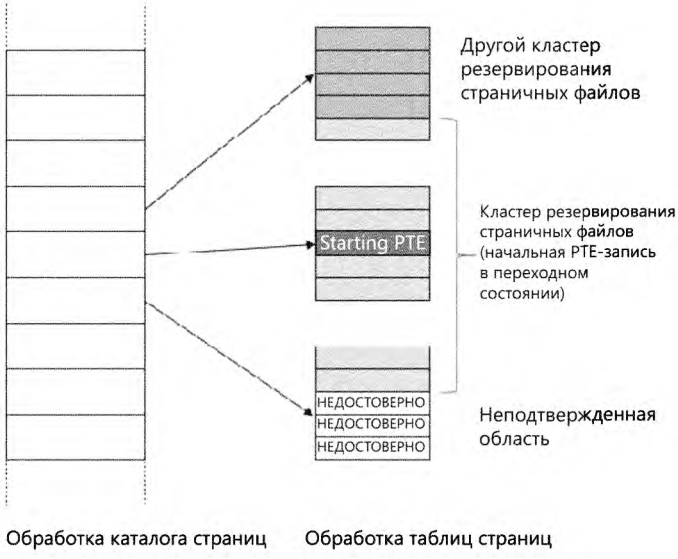


Рис. 5.40. Кластер резервирования страничных файлов

После того как страничный кластер будет определен, необходимо найти свободное пространство в страничном файле, которое будет зарезервировано для этого кластера. Выделением пространства в страничном файле управляет битовая карта (в которой каждый установленный бит обозначает используемую страницу в файле). Для резервирования страничного файла используется вторая битовая карта, которая указывает, какие страницы были зарезервированы (но в которые еще не обязательно осуществлялась запись — за это отвечает битовая карта выделения памяти в страничном файле). При появлении в страничном файле пространства, которое еще не было зарезервировано и не было выделено (на основании этих битовых карт), соответствующие биты устанавливаются только в битовой карте резервирования. Подсистема записи измененных страниц должна устанавливать эти биты в битовой карте выделения при записи содержимого таких страниц на диск. Если найти достаточное пространство в страничном файле для заданного размера кластера не удается, делается попытка расширения страничных файлов, и, если это уже происходило ранее (или максимальный размер страничного файла равен расширенному размеру), размер кластера сокращается до того размера резервирования, который удалось обнаружить.

ПРИМЕЧАНИЕ Страницы из кластера (кроме исходной начальной PTE-записи) не связываются ни с какими списками физических страниц. Информация резервирования помещается в поле исходной PTE-записи PFN.

Подсистема записи измененных страниц должна обрабатывать запись страниц с резервированием как особый случай. Он использует всю собранную информацию, описанную выше, для построения MDL-списка с правильными PFN кластера, который будет использоваться в процессе записи в страничный файл. При построении кластера производится поиск смежных страниц, которые могут охватывать несколько кластеров резервирования. Если между кластерами есть «дыры», они заполняются фиктивными страницами (все байты таких страниц содержат 0xFF). Если счетчик фиктивных страниц превышает 32, то кластер разбивается. Для построения итогового кластера для записи производится «обход» в прямом и обратном направлении. На рис. 5.41 показан пример состояния страниц после построения такого кластера подсистемой записи измененных страниц.

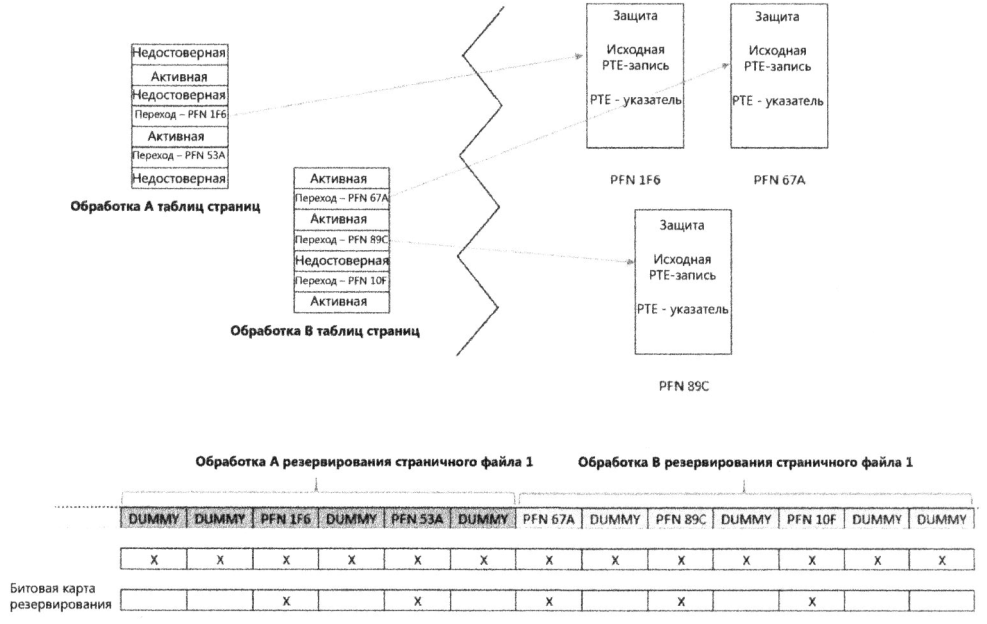


Рис. 5.41. Кластер, построенный перед записью

Наконец, обработчик ошибок страниц использует информацию из битовой карты резервирования и PTE-записей для определения начальной и конечной точки кластеров, чтобы эффективно загрузить необходимые страницы с минимальным позиционированием головки механического диска.

Лимиты физической памяти

Теперь, когда вы узнали, как в Windows организовано отслеживание физической памяти, поговорим о том, насколько большой объем этой памяти Windows фактически может поддерживать. Поскольку большинство систем в процессе своей работы обращаются к коду и данным, объем которых превышает емкость физической памяти, эта память по сути представляет собой окно для обращения к используемым коду и данным. Объем памяти может тем самым влиять на производительность; когда данные (или код), в которых нуждается процесс или операционная система, отсутствуют, диспетчер памяти должен доставить их в память с диска или из удаленного хранилища.

Кроме влияния на производительность объем физической памяти определяет лимиты других ресурсов. Например, вполне очевидно, что объемы невыгружаемого пула, буферов операционной системы, поддерживаемых физической памятью, объемом этой памяти и ограничиваются. Физическая память учитывается также и при определении лимита виртуальной памяти системы, который приблизительно определяется суммой размера физической памяти и текущего настроенного размера любых страничных файлов. Физическая память также может косвенно ограничивать максимальное количество процессов.

Поддержка физической памяти со стороны Windows диктуется аппаратными ограничениями, условиями лицензирования, структурами данных операционной системы и совместимостью драйверов. Текущие поддерживаемые объемы физической памяти в различных редакциях Windows наряду с ограничивающими факторами перечислены по адресу *https://msdn.microsoft.com/en-us/library/windows/desktop/aa366778.aspx*. В табл. 5.23 приведена сводка лимитов для Windows 8 и более новых версий.

Таблица 5.23. Ограничения на поддержку физической памяти в Windows

Версия/редакция операционной системы	32-разрядный лимит	64-разрядный лимит
Windows 8.x Professional и Enterprise	4 Гбайт	512 Гбайт
Windows 8.x (все остальные редакции)	4 Гбайт	128 Гбайт
Windows Server 2012/R2 Standard и Datacenter	–	4 Тбайт
Windows Server 2012/R2 Essentials	–	64 Гбайт
Windows Server 2012/R2 Foundation	–	32 Гбайт
Windows Storage Server 2012 Workgroup	–	32 Гбайт
Windows Storage Server 2012 Standard Hyper-V Server 2012	–	4 Тбайт
Windows 10 Home	4 Гбайт	128 Гбайт
Windows 10 Pro, Education и Enterprise	4 Гбайт	2 Тбайт
Windows Server 2016 Standard и Datacenter	–	24 Тбайт

На момент написания книги максимальный лимит физической памяти в 4 Тбайт в некоторых редакциях Server 2012/R2 и 24 Тбайт в версиях Server 2016. Эти ограничения не обусловлены какими-либо реализационными или аппаратными ограничениями, а являются причиной того, что Microsoft будет поддерживать только те варианты настройки, которые сможет протестировать. На момент написания данной книги они были наибольшими протестированными и поддерживаемыми конфигурациями памяти.

Лимиты памяти клиентских версий Windows

64-разрядные клиентские версии Windows отличаются тем, что поддерживают различные объемы памяти, с нижней границы 4 Гбайт в начальных версиях, и до 2 Тбайт в корпоративной (Enterprise) и профессиональной (Professional) редакциях. Но все 32-разрядные клиентские версии Windows поддерживают максимум в 4 Гбайт физической памяти, что дает максимальный физический адрес, доступный в стандартном режиме диспетчера памяти на платформе x86.

Хотя клиентские версии на платформе x86 поддерживают режимы PAE-адресации с целью предоставления аппаратной защиты от выполнения данных (что также обеспечивает доступ к более чем 4 Гбайт физической памяти), тестирование показало, что в системе происходит фатальный сбой, она зависает или перестает загружаться, поскольку некоторые драйверы устройств (обычно это касается видео- и аудиоустройств, характерных для клиентских, но не серверных машин) не были запрограммированы с возможностью использования физических адресов, превышающих 4 Гбайт. В результате драйверы усекают такие адреса, что приводит к порче содержимого памяти и побочным разрушительным эффектам. Серверные системы обычно обладают более универсальными устройствами с более простыми и стабильными драйверами, поэтому, как правило, для них эти проблемы не характерны. Из-за проблем с драйверами в клиентских системах было решено игнорировать физическую память, выходящую за границы 4 Гбайт, хотя они теоретически могут ее адресовать. Разработчики драйверов поощряются тестировать свои системы с BCD-параметром `nolowmem`, который заставляет ядро использовать физические адреса, превышающие 4 Гбайт, только если в системе есть достаточный для этого объем памяти. Это тут же приведет к выявлению подобных проблем на недоработанных драйверах.

Хотя значение 4 Гбайт является лицензированным лимитом для 32-разрядных клиентских версий, реальный лимит ниже и зависит от системного чипсета и подключенных устройств. Причина в том, что карта адресов физической памяти включает не только оперативную память, но и память устройств, и системы на платформах x86 и x64 обычно отображают всю память устройств, не выходя за пределы адресной границы в 4 Гбайт; это делается для сохранения совместимости с 32-разрядными операционными системами, которые не умеют работать с адресами, превышающими 4 Гбайт. Современные наборы микропроцессоров поддерживают переотображение устройств на базе PAE, но клиентские версии Windows его

не поддерживают из-за упоминавшихся ранее проблем совместимости драйверов (в противном случае драйверы получали бы 64-разрядные указатели на память их устройств).

Если у системы имеется 4 Гбайт оперативной памяти, а видео- и аудиоустройства, а также сетевые адаптеры реализуют в своей памяти окна, размер которых доходит в сумме до 500 Мбайт, то, как показано на рис. 5.42, 500 Мбайт из 4 Гбайт оперативной памяти системы выйдет за адресную границу в 4 Гбайт.

Рис. 5.42. Структура физической памяти в системе с 4 Гбайт памяти

В результате, если имеется система с памятью размером 3 Гбайт и более и на ней запущена 32-разрядная клиентская версия Windows, пользы от доступа ко всей оперативной памяти можно и не получить. Какой объем установленной оперативной памяти «видит» операционная система, можно посмотреть в диалоговом окне Свойства системы (System Properties), но чтобы узнать, какой объем памяти *фактически* доступен Windows, нужно открыть вкладку Быстродействие (Performance) диспетчера задач или использовать утилиты Msinfo32 и Winver. Как показала утилита Msinfo32, на одном конкретном ноутбуке с оперативной памятью в 4 Гбайт и установленной 32-разрядной версией Windows 10 объем доступной физической памяти составил 3,87 Гбайт:

```
Установленная оперативная память (RAM)   4,00 Гбайт;
Полный объем физической памяти           3,87 Гбайт.
```

Структуру физической памяти можно посмотреть с помощью программы MemInfo. Ниже показан результат запуска утилиты MemInfo на 32-разрядной системе с ключом `-r`, предназначенным для вывода дампа диапазонов физической памяти:

```
C:\Tools>MemInfo.exe -r
MemInfo v3.00 - Show PFN database information
```

```
Copyright (C) 2007-2016 Alex Ionescu
www.alex-ionescu.com

Physical Memory Range: 00001000 to 0009F000 (158 pages, 632 KB)
Physical Memory Range: 00100000 to 00102000 (2 pages, 8 KB)
Physical Memory Range: 00103000 to F7FF0000 (1015533 pages, 4062132 KB)
MmHighestPhysicalPage: 1015792
```

Обратите внимание на пропуск в диапазоне адресов памяти от A0000 до 100 000 (384 Кбайт) и еще один пропуск от F8000000 до FFFFFFFF (128 Мбайт).

С помощью диспетчера устройств можно посмотреть, чем на вашей машине заняты различные диапазоны зарезервированной памяти, которые могут использоваться системой Windows (в выводимых утилитой MemInfo данных эти диапазоны выглядят как дыры). Выполните следующие действия:

1. Запустите файл оснастки Devmgmt.msc.
2. Выберите в меню команду Вид ▸ Ресурсы по подключению (View ▸ Resources By Connection).
3. Раскройте узел Память (Memory). На ноутбуке, использованном для вывода информации, основным потребителем отображаемой памяти устройств вполне ожидаемо является видеокарта (Hyper-V S3 Cap), на которую расходуется 128 Мбайт в диапазоне F8000000-FBFFFFFF (рис. 5.43).

Рис. 5.43. Зарезервированные оборудованием диапазоны памяти в 32-разрядной версии Windows

Большая часть остальной памяти приходится на разные другие устройства, шина PCI резервирует дополнительные диапазоны для устройств в рамках консервативной оценки встроенного программного обеспечения в ходе начальной загрузки.

Сжатие памяти

Диспетчер памяти Windows 10 реализует механизм, который сжимает закрытые и поддерживаемые страничными файлами страницы из списка измененных страниц. Основными кандидатами на сжатие становятся закрытые страницы,

принадлежащие UWP-приложениям, потому что сжатие очень хорошо сочетается с выгрузкой и очисткой рабочих наборов, уже происходящих в таких приложениях при нехватке памяти. После того как приложение будет приостановлено, а его рабочий набор выгружен, рабочий набор может быть очищен в любой момент, а измененные страницы могут быть сжаты. Создаваемой при этом дополнительной доступной памяти может оказаться достаточно для хранения в памяти другого приложения, при этом страницам первого приложения даже не придется покидать память.

ПРИМЕЧАНИЕ Эксперименты показали: алгоритм Microsoft Xpress сжимает страницы приблизительно до 30–50 % от их исходного размера. Этот алгоритм сочетает скорость с эффективностью, что позволяет добиться значительной экономии памяти.

Архитектура сжатия памяти должна соответствовать следующим требованиям.

- Страница не может находиться в памяти в сжатой и несжатой форме, потому что дублирование приведет к непроизводительным затратам памяти. Это означает, что каждая страница после успешного сжатия должна становиться свободной страницей.
- Хранилище сжатия должно хранить свои основные структуры данных и сжатые данные таким образом, чтобы всегда обеспечивать экономию памяти для системы в целом. Это означает, что если страница сжимается недостаточно эффективно, она не будет добавлена в хранилище.
- Сжатые страницы должны отображаться как доступная память (потому что при необходимости их действительно можно использовать для других целей), чтобы не создавать у пользователя ложного ощущения того, будто сжатие каким-то образом повышает потребление памяти.

Сжатие памяти включается по умолчанию в клиентских версиях (телефон, PC, Xbox и т. д.). Серверные версии в настоящее время не поддерживают сжатие памяти, но, скорее всего, в будущем ситуация изменится.

ПРИМЕЧАНИЕ В Windows Server 2016 диспетчер задач выводит в круглых скобках информацию о сжатии памяти, но это число всегда равно нулю. Кроме того, процесс сжатия памяти в системе не существует.

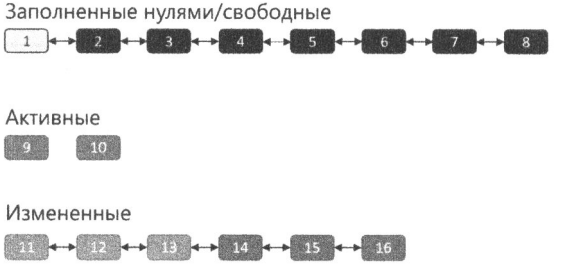
При запуске системы служба супервыборки (sysmain.dll, содержится в экземпляре svchost.exe — см. далее раздел «Упреждающее управление памятью (супервыборка)») вызовом функции `NtSetSystemInformation` приказывает диспетчеру хранения (store manager) из исполнительной подсистемы создать единое системное хранилище для не-UWP-приложений. При запуске приложения каждое UWP-приложение связывается со службой супервыборки и запрашивает создание хранилища для себя.

Пример сжатия

Чтобы получить представление о том, как работает сжатие памяти, рассмотрим содержательный пример. Предположим, в какой-то момент времени существуют следующие физические страницы:

Списки обнуленных и свободных страниц, содержащие нули и «мусорные» данные соответственно, могут использоваться для удовлетворения запросов на выделение памяти; в контексте нашего обсуждения они будут рассматриваться как один список. Активные страницы принадлежат разным процессам, тогда как измененные страницы содержат модифицированные данные, которые еще не были записаны в страничный файл, но могут быть перемещены в рабочий набор процесса по ошибке страницы ОЗУ без операции ввода/вывода, если этот процесс обратится к измененной странице.

Теперь предположим, что диспетчер памяти пытается провести усечение списка измененных страниц — например, потому что он стал слишком большим или список нулевых/свободных страниц стал слишком мал. Допустим, из списка измененных страниц необходимо удалить три страницы. Диспетчер памяти сжимает их содержимое в одну страницу (из списка обнуленных/свободных страниц):

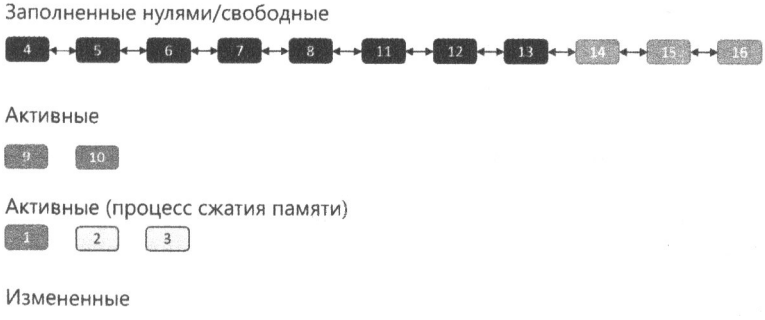


Страницы 11, 12 и 13 сжимаются в страницу 1. После того как это будет сделано, страница 1 уже не является свободной; она становится активной страницей, частью рабочего набора процесса сжатия памяти (см. следующий раздел). Страницы 11,

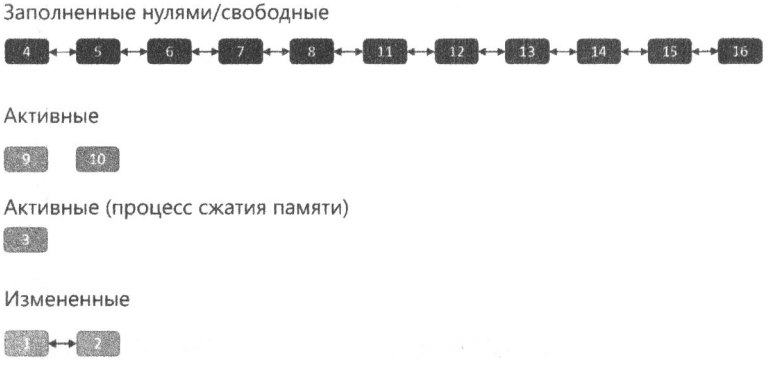
12 и 13 становятся лишними и перемещаются в список свободных; таким образом, сжатие позволило сэкономить две страницы:

Допустим, процесс повторяется. На этот раз страницы 14, 15 и 16 сжимаются, скажем, в две страницы (2 и 3):

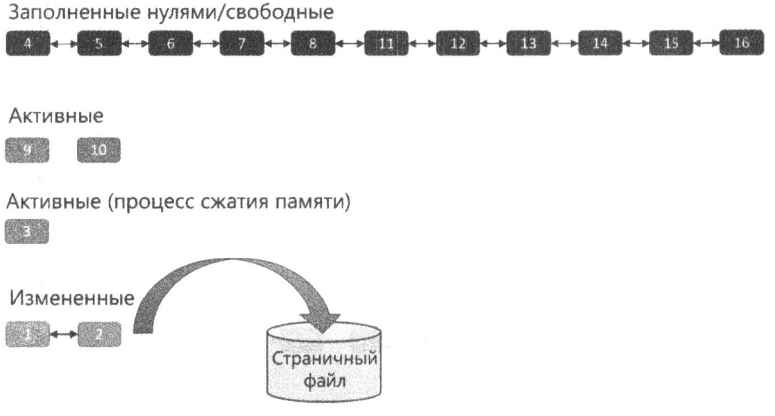
В результате страницы 2 и 3 присоединяются к рабочему набору процесса сжатия памяти, тогда как страницы 14, 15 и 16 становятся свободными:



Предположим, позднее диспетчер памяти решает провести усечение рабочего набора процесса сжатия памяти. В таком случае страницы перемещаются в список измененных, потому что они содержат данные, еще не записанные в страничный файл. Конечно, они могут в любой момент быть перемещены по ошибке страницы ОЗУ в свой исходный процесс (при этом происходит их распаковка с использованием свободных страниц). На следующей схеме страницы 1 и 2 исключаются из активных страниц процесса сжатия памяти и перемещаются в список измененных страниц:

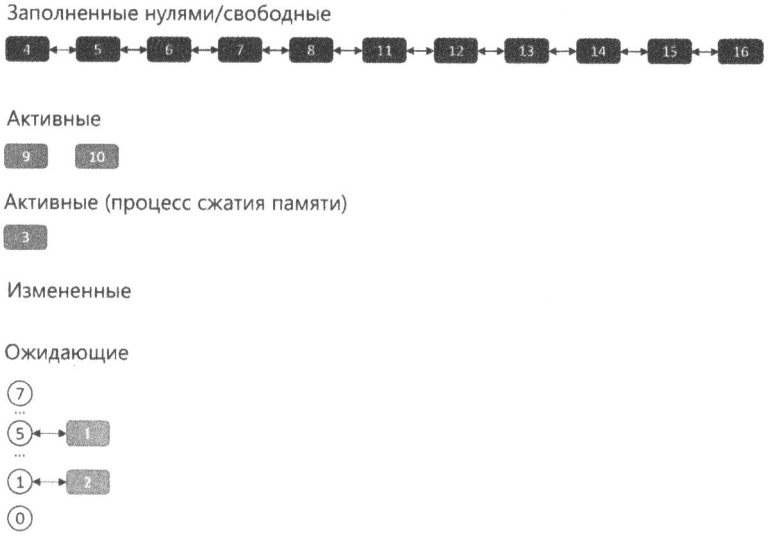


Если в системе возникнет нехватка памяти, диспетчер памяти может принять решение о записи сжатых измененных страниц в страничный файл:



Наконец, после того как такие страницы будут записаны в страничный файл, они перемещаются в список ожидающих страниц, потому что их содержимое было сохранено, и при необходимости они могут использоваться для других целей. Также к ним может применяться ошибка страницы ОЗУ (когда они входят в список изме-

ненных страниц); страницы распаковываются и перемещаются в активное состояние в рабочем наборе соответствующего процесса. Когда страницы находятся в списке ожидающих, они присоединяются к соответствующему подсписку в зависимости от приоритета (см. «Приоритеты страниц и перебалансировка» этой главы):



Архитектура сжатия

Механизму сжатия необходима «рабочая область» памяти для хранения сжатых страниц и структур данных, управляющих ими. В версиях Windows 10 до 1607 использовалось пользовательское адресное пространство процесса System. Начиная с Windows 10 версии 1607, для этой цели используется новый специализированный процесс *Memory Compression*. Одна из причин для создания нового процесса заключалась в том, что стороннему наблюдателю потребление памяти процессом System могло показаться слишком высоким; создавалось впечатление, что система расходует слишком много памяти. В действительности это было не так, потому что сжатая память не учитывается в лимите подтверждения. Тем не менее иногда даже субъективные впечатления слишком важны.

Процесс Memory Compression является минимальным процессом, т. е. он не загружает никаких DLL-библиотек. Вместо этого он просто предоставляет адресное пространство для работы. Никакие исполняемые образы в нем тоже не выполняются — ядро просто использует его адресное пространство пользовательского режима (за дополнительной информацией о минимальных процессах обращайтесь к главе 3 «Процессы и задания»).

ПРИМЕЧАНИЕ Диспетчер задач не показывает процесс сжатия памяти в своем детализированном представлении, но в Process Explorer эта информация доступна. При использовании отладчика ядра образ процесса сжатия отображается под именем MemCompression.

Для каждого хранилища диспетчер хранения выделяет память блоками настраиваемого размера. В настоящее время этот размер составляет 128 Кбайт. Память выделяется обычными вызовами `VirtualAlloc` по мере необходимости. Фактические сжатые страницы хранятся в областях 16-байтовыми фрагментами. Естественно, сжатая страница (4 Кбайт) может охватывать много фрагментов. На рис. 5.44 показано хранилище с массивом областей и некоторые структуры данных, обеспечивающие управление таким хранилищем.

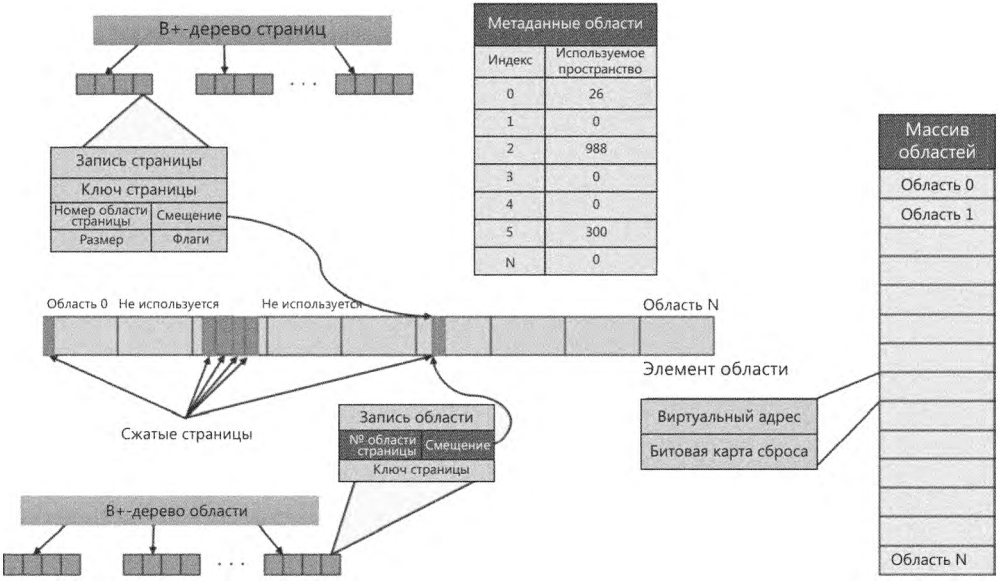


Рис. 5.44. Структуры данных хранилища

Как видно из рис. 5.44, для управления страницами используется B^+-*дерево* (по сути это дерево, в котором у каждого узла может быть любое количество дочерних узлов), в котором каждая запись страницы указывает на ее сжатое содержимое в одной из областей. Хранилище начинается с нуля областей, после чего области выделяются и освобождаются по мере необходимости.

Области также связываются с приоритетами, как описано в разделе «Приоритеты страниц и перебалансировка» этой главы.

Процедура добавления страницы состоит из следующих этапов:

1. Если в настоящее время не существует области с приоритетом страницы, выделяется новая область, фиксируется в физической памяти, и ей назначается приоритет добавляемой страницы. В качестве текущей области для данного приоритета назначается выделенная область.
2. Страница сжимается и сохраняется в области с округлением до единицы гранулярности (16 байт). Например, если страница сжимается до 687 байт, она будет занимать 43 16-байтовые единицы (округление всегда производится вверх). Сжатие выполняется в текущем потоке и с низким приоритетом процессора (7) для снижения влияния на работу системы. Когда потребуется выполнить распаковку сжатых данных, она выполняется параллельно с использованием всех доступных процессоров.
3. Информация страниц и областей обновляется в B+-деревьях страниц и областей.
4. Если оставшегося пространства в текущей области недостаточно для хранения сжатой страницы, выделяется новая область (с тем же приоритетом страницы), которая назначается текущей областью для данного приоритета.

Процедура удаления страницы из хранилища состоит из следующих этапов:

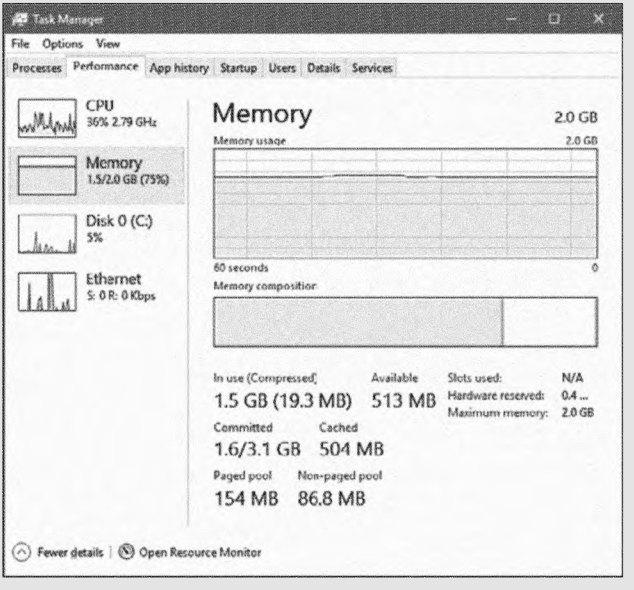
1. Поиск записи страницы в B^+-дереве страниц и записи области в B^+-дереве областей.
2. Записи удаляются, а информация об использовании пространства в области обновляется.
3. Если область становится пустой, она уничтожается.

Со временем при добавлении и удалении страниц области постепенно фрагментируются. Память области не освобождается до тех пор, пока область не опустеет. А значит, нужен некий механизм уплотнения для сокращения непроизводительных потерь памяти. Операция уплотнения планируется неформально, а степень ее воздействия зависит от степени фрагментации. Объединение областей осуществляется с учетом их приоритетов.

ЭКСПЕРИМЕНТ: СЖАТИЕ ПАМЯТИ

Сжатие памяти, происходящее в системе, остается практически незаметным. Для наблюдения за ним можно воспользоваться Process Explorer или отладчиком ядра. На следующей иллюстрации изображена вкладка Performance в свойствах процесса `Memory Compression` программы Process Explorer (которая должна выполняться с административными привилегиями).

Обратите внимание: время процесса в пользовательском режиме равно 0 (так как в этом процессе «работают» только потоки ядра), а его рабочий набор закрыт (т. е. недоступен для совместного использования). Это связано с тем, что сжатая память не может совместно использоваться ни в каком отношении. Сравните с представлением Память (Memory) в диспетчере задач:



Сжатая память в круглых скобках должна коррелировать с рабочим набором процесса `Memory Compression` (этот снимок экрана был сделан примерно через минуту после предыдущего), потому что это пространство, потребляемое сжатой памятью.

Секции памяти

Традиционно виртуальные машины (VM) использовались для изоляции приложений, чтобы в разных VM могли выполняться полностью изолированные приложения (или группы приложений) — по крайней мере, с точки зрения безопасности. VM не могут не взаимодействовать друг с другом; таким образом формируются сильные границы безопасности и ресурсов. Хотя этот механизм работает, VM достаточно дорого обходятся в отношении ресурсов оборудования, на котором они работают, и затрат на управление. Данное обстоятельство способствовало появлению контейнерных технологий — таких, как Docker. Эти технологии пытаются снизить затраты ресурсов на изоляцию и управление ресурсами за счет создания контейнеров, управляющих работой приложений — на одной физической или виртуальной машине.

Создавать такие контейнеры непросто. Для них нужны драйверы ядра, реализующие некую разновидность виртуализации поверх обычных механизмов Windows. Несколько примеров таких драйверов (один драйвер может охватывать все эти функциональные аспекты):

- (Мини) фильтр файловой системы, создающий иллюзию изолированной файловой системы.
- Драйвер виртуализации реестра, создающий иллюзию отдельного реестра (`CmRegisterCallbacksEx`).
- Закрытое пространство имен диспетчера объектов, использующее участки (за подробностями обращайтесь к главе 3).
- Механизм управления процессами, связывающий процессы с правильными контейнерами (с использованием `PsSetCreateNotifyRoutineEx`).

Даже при наличии этой функциональности некоторые аспекты создают немало трудностей при виртуализации, а именно управление памятью. Желательно, чтобы каждый контейнер использовал собственную базу данных PFN, собственный страничный файл и т. д. Windows 10 (только 64-разрядные версии) и Windows Server 2016 предоставляют такие средства управления памятью через механизм *секционирования памяти* (memory partitions).

Секция памяти имеет собственные управляющие структуры, относящиеся к управлению памятью (списки страниц — ожидающие, измененные, свободные и т. д.), собственный показатель подтверждения, рабочий набор, подсистему записи измененных страниц, поток обнуления страниц и т. д. Все это изоли-

ровано от других секций. В системе секции памяти представлены объектами `Partition`, которым могут назначаться атрибуты защиты и имена (как и для других объектов исполнительной среды). Одна секция, называемая *системной*, существует всегда; она представляет систему в целом и является конечным родителем любой явно созданной секции. Адрес системной секции хранится в глобальной переменной (`MiSystemPartition`); к ней можно обратиться по имени `KernelObjects\MemoryPartition0` из таких программ, как WinObj из пакета Sysinternals (рис. 5.45).

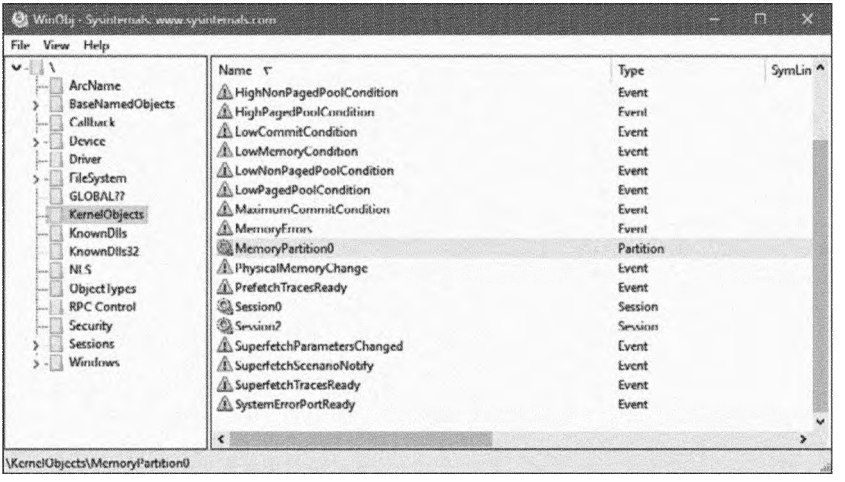


Рис. 5.45. Системная секция System Partition в WinObj

Все объекты секций хранятся в глобальном списке. Максимальное количество секций на данный момент равно 24 (10 бит), потому что индекс секции должен быть закодирован в PTE для быстрого доступа к данным секции. Один из индексов относится к системной секции, еще два используются в качестве сторожевых значений; доступными остаются еще 1021 секция.

Секции памяти могут создаваться из пользовательского режима или режима ядра внутренней (и недокументированной) функцией `NtCreatePartition`; чтобы вызов завершился успешно, сторона, вызывающая эту функцию из пользовательского режима, должна обладать привилегией SeLockMemory. Функция может получать родительскую секцию, от которой будут браться исходные страницы и в которую они вернутся при уничтожении секции. Если родитель не указан, по умолчанию используется системная секция. `NtCreatePartition` делегирует непосредственную работу внутренней функции диспетчера памяти `MiCreatePartition`.

Существующую секцию можно открыть по имени функцией `NtOpenPartition` (никакие особые привилегии для этого не нужны, так как объект защищается обычными средствами ACL). Непосредственными операциями с разделом занимается функция `NtManagePartition`. Эта функция может использоваться для добавления

памяти в секцию, добавления страничного файла, копирования памяти из одной секции в другую, а также для получения информации о секции.

ЭКСПЕРИМЕНТ: ПРОСМОТР СЕКЦИЙ ПАМЯТИ

В этом эксперименте мы воспользуемся отладчиком ядра для просмотра информации объектов секций.

1. Запустите локальный сеанс отладчика ядра и введите команду `!partition`. Команда выводит все объекты секций в системе.

```
lkd> !partition
Partition0 fffff803eb5b2480 MemoryPartition0
```

2. По умолчанию выводится системная секция, всегда присутствующая в системе. Команда `!partition` может получить адрес объекта секции и вывести более подробную информацию:

```
lkd> !partition fffff803eb5b2480
PartitionObject @ ffffc808f5355920 (MemoryPartition0)
_MI_PARTITION 0 @ fffff803eb5b2480
  MemoryRuns: 0000000000000000
  MemoryNodeRuns: ffffc808f521ade0
  AvailablePages:         0n4198472 ( 16 Gb 16 Mb 288 Kb)
  ResidentAvailablePages: 0n6677702 ( 25 Gb 484 Mb 792 Kb)
     0 _MI_NODE_INFORMATION @ fffff10000003800
           TotalPagesEntireNode:   0x7f8885
                                Zeroed                          Free
           1GB                    0 ( 0)                            0 (
0)
           2MB                   41 ( 82 Mb)                        0 (
0)
           64KB                3933 ( 245 Mb 832 Kb)                0 (
0)
           4KB                82745 ( 323 Mb 228 Kb)                0 (
0)
           Node Free Memory:     ( 651 Mb 36 Kb )
           InUse Memory:         ( 31 Gb 253 Mb 496 Kb )
           TotalNodeMemory:      ( 31 Gb 904 Mb 532 Kb )
```

В результатах присутствует информация о нижележащей структуре `MI_PARTITION` (адрес которой также приводится). Обратите внимание: команда выводит информацию памяти на уровне узлов NUMA (в данном примере узел только один). Так как этот раздел является системным, объемы используемой, свободной и общей памяти должны соответствовать данным, выводимым такими программами, как диспетчер задач и Process Explorer. Для получения информации о структуре `MI_PARTITION` также можно воспользоваться обычной командой `dt`.

Вероятно, в будущем появится возможность использования секционирования памяти для связывания конкретных процессов (через объекты заданий) с секциями — например, если монопольный контроль над физической памятью может принести

пользу. Одним из таких сценариев, запланированных для издания Creators Update, является игровой режим (за дополнительной информацией об игровом режиме обращайтесь к главе 8 части 2).

Комбинирование памяти

Диспетчер памяти использует ряд механизмов, обеспечивающих экономию памяти, — например, совместное использование страниц для графики, копирование при записи для страниц данных и сжатие. В этом разделе мы рассмотрим еще один механизм такого рода, называемый *комбинированием памяти*.

Общий принцип прост: найти в оперативной памяти дубликаты страниц и объединить их в одну страницу, чтобы убрать из памяти лишние дубликаты. Очевидно, при этом необходимо решить ряд проблем:

- Какие страницы станут «лучшими» кандидатами для комбинирования?
- Когда лучше всего запускать комбинирование памяти?
- Должно ли комбинирование ориентироваться на конкретный процесс, секцию памяти или всю систему?
- Как ускорить процесс комбинирования, чтобы он не оказывал нежелательного влияния на нормальное выполнение кода?
- Если комбинированная страница, доступная для записи, позднее будет изменена одним из своих клиентов, как она получит отдельную закрытую копию?

В этом разделе мы ответим на эти вопросы, начиная с последнего. В этом случае используется механизм копирования при записи: если в комбинированную страницу не осуществляется запись, то не делается ничего. Если процесс пытается записывать данные в страницу, то для записывающего процесса создается закрытая копия страницы, а для только что выделенной закрытой страницы снимается флаг копирования при записи.

ПРИМЕЧАНИЕ Чтобы отключить комбинирование страниц, присвойте значение 1 DWORD-параметру DisablePageCombining в разделе реестра HKLM\System\CurrentControlSet\Control\Session Manager\Memory Management.

ПРИМЕЧАНИЕ В этом разделе термины *CRC* и *хеш* используются как синонимы. Они обозначают статистически уникальное (с очень высокой вероятностью) 64-разрядное число, идентифицирующее содержимое страницы.

Функция инициализации диспетчера памяти `MmInitSystem` создает системную секцию (см. предыдущий раздел «Секции памяти»). В структуре `MI_PARTITION`, описывающей секцию, хранится массив из 16 АВЛ-деревьев, описывающих дублируемые

страницы. Массив сортируется по последним 4 битам CRC-кода комбинированной страницы. Вскоре вы увидите, какое место эта сортировка занимает в алгоритме.

Существует два специальных типа страниц — так называемые *общие страницы* (common pages). Страницы первого типа содержат только нулевые байты, страницы второго типа заполняются единичными битами (т. е. содержат байты 0xFF); их CRC-код вычисляется только один раз и хранится в памяти. Такие страницы легко опознаются при сканировании содержимого.

Для запуска комбинирования памяти вызывается платформенная API-функция `NtSetSystemInformation` с системным информационным классом `SystemCombinePhysicalMemoryInformation`. В маркере вызывающей стороны должна присутствовать привилегия `SeProfileSingleProcessPrivilege`, обычно предоставляемая группе локальных администраторов. Аргумент функции содержит набор параметров, определяемый сочетанием флагов:

- Выполнение комбинирования памяти в системной секции (т. е. в рамках всей системы) или только в текущем процессе.
- Поиск общих страниц (заполненных только нулями или единицами) для комбинирования или любых страниц-дубликатов с любым содержимым.

Входная структура также содержит необязательный дескриптор события, который может передаваться при вызове и при активизации (другим потоком) отменяет комбинирование страниц. В настоящее время служба супервыборки (см. раздел «Супервыборка» в конце главы) использует специальный поток с низким приоритетом (4), который инициирует комбинирование памяти для общесистемной секции, когда пользователь отошел от компьютера, или, если пользователь занят, — через каждые 15 минут.

В обновлении Creators Update, если объем физической памяти превышает 3,5 Гбайт (3584 Мбайт), для большинства встроенных служб под управлением Svchost в каждом процессе Svchost размещается одна служба. При этом в системе появляются десятки процессов, но снижается вероятность того, что одна служба повлияет на работу другой (из-за нестабильности или дефекта безопасности). В этом сценарии SCM (Service Control Manager) использует новую возможность API комбинирования памяти и запускает комбинирование страниц в каждом из процессов Svchost каждые три минуты. Для этого используется таймер пула потоков с базовым приоритетом 6 (функция `ScPerformPageCombineOnServiceImages`). Это делается для снижения затрат памяти, которые могут быть выше, чем при меньшем количестве экземпляров Svchost. Обратите внимание: службы, не находящиеся под управлением Svchost, комбинированию страниц не подвергаются — как и службы, работающие с правами пользователей или закрытых пользовательских учетных записей.

Функция `MiCombineIdenticalPages` является фактической точкой входа для процесса комбинирования страниц. Для каждого NUMA-узла в секции памяти она выделяет и сохраняет список страниц с их CRC-кодами в структуре PCS (Page Combining Support), обеспечивающей управление всей необходимой информацией

для операции комбинирования страниц. (Именно в этой структуре содержится массив АВЛ-деревьев, о котором упоминалось ранее.) Работа выполняется запрашивающим потоком; он должен выполняться на процессорах, принадлежащих текущему NUMA-узлу, с соответствующим изменением их родственности в случае необходимости. Для простоты объяснения мы разделим алгоритм комбинирования памяти на три стадии: поиск, классификация и совместное использование страниц. Дальнейшее объяснение предполагает, что запрашивается полное комбинирование страниц (в отличие от комбинирования для текущего процесса) и для всех страниц (не только для общих страниц); остальные случаи принципиально не отличаются от этого, но реализуются немного проще.

Фаза поиска

Цель этой исходной фазы — вычисление CRC-кодов всех физических страниц. Алгоритм анализирует каждую физическую страницу, принадлежащую списку активных, измененных или ожидающих страниц, пропуская обнуленные и свободные страницы (поскольку они фактически не используются).

Хорошим кандидатом на комбинирование памяти должна быть неактивная страница, не находящаяся в совместном использовании, которая принадлежит рабочему набору и не отображается на страничную структуру. Кандидат даже может находиться в ожидающем или измененном состоянии, но его счетчик ссылок должен быть равен 0. По сути, система выявляет для комбинирования три типа страниц: страницы пользовательских процессов, страничного пула и пространства сеанса. Другие типы страниц пропускаются.

Чтобы правильно вычислить CRC-код страницы, система должна отобразить физическую страницу на системный адрес (потому что контекст процесса отличен от контекста вызывающего потока, вследствие чего страница недоступна в низких адресах пользовательского режима) с использованием новой системной PTE-записи. Затем CRC-код страницы вычисляется с применением специализированного алгоритма (функция `MiComputeHash64`), а системная PTE-запись освобождается (отображение страницы в системное адресное пространство отменяется).

АЛГОРИТМ ХЕШИРОВАНИЯ СТРАНИЦ

Для вычисления 8-байтового хеша страницы (CRC-кода) система использует следующий алгоритм: два 64-разрядных больших простых числа перемножаются, а результат используется в качестве исходного хеша. Страница сканируется от начала к концу; алгоритм хеширует 64 байта на цикл. Каждое прочитанное 8-байтовое значение из страницы прибавляется к исходному хешу, после чего выполняется циклический сдвиг хеша на простое число битов (начиная с 2, затем 3, 5, 7, 11, 13, 17 и 19). Полное хеширование страницы требует 512 обращений к памяти (4096/8).

Фаза классификации

После того как все хеши страниц, принадлежащих узлу NUMA, будут успешно вычислены, запускается вторая часть алгоритма. Цель этой фазы — обработка каждой записи CRC/PFN из списка и их стратегическое упорядочение. Алгоритм совместного использования страниц должен минимизировать переключение контекстов процессов и работать как можно быстрее.

Функция `MiProcessCrcList` начинает работу с сортировки списка CRC/PFN по хешу (с использованием алгоритма быстрой сортировки). Другая ключевая структура данных — *комбинационный блок* — используется для отслеживания всех страниц с одним хешем и, что еще важнее, для хранения новой прототипной PTE-записи, которая будет отображать новую комбинированную страницу. Все CRC/PFN отсортированного списка обрабатываются по очереди. Система должна проверить, является ли текущий хеш общим (т. е. относится ли он к странице, заполненной нулями или единицами) и равен ли он предыдущему или следующему хешу (напомним, что список отсортирован). Если это не так, система проверяет, существует ли комбинационный блок в структуре PCS. Если он существует, это означает, что комбинированная страница уже была идентифицирована при предыдущем выполнении алгоритма или в другом узле системы. В противном случае это означает, что CRC-код уникален и страница не может комбинироваться, а алгоритм переходит к следующей странице в списке.

Если обнаруженный общий хеш никогда не встречался ранее, алгоритм выделяет новый пустой комбинационный блок (для главного PFN) и вставляет его в список, используемый кодом совместного использования страниц (следующая фаза). В противном случае, если хеш уже существует (страница не является основной копией), ссылка на комбинационный блок добавляется в текущую запись CRC/PFN.

К этому моменту алгоритм подготовил все данные, необходимые для алгоритма совместного использования страниц: список комбинационных блоков, используемых для хранения основных физических страниц и прототипных PTE-записей, список записей CRC/PFN, упорядоченных по рабочему набору-владельцу, и физическая память, необходимая для хранения содержимого новых общих страниц.

Затем алгоритм получает адрес физической страницы (который должен существовать из-за исходной проверки, выполняемой ранее функцией `MiCombineIdenticalPages`) и ищет структуру данных, используемую для хранения всех страниц, принадлежащих конкретному рабочему набору (в дальнейшем мы будем называть эту структуру *WS CRC-узлом*). Если структура не существует, то она создается и вставляется в другое АВЛ-дерево. CRC/PFN и виртуальный адрес страницы объединяются в WS CRC-узле.

После того как все идентифицированные страницы будут обработаны, система выделяет физическую память для новых основных общих страниц (с использованием MDL) и обрабатывает каждый WS CRC-узел; для повышения быстродействия страницы-кандидаты, найденные в узле, сортируются по исходному виртуальному адресу. Система готова к непосредственному выполнению комбинирования страниц.

Фаза комбинирования страниц

Фаза комбинирования страниц начинается со структуры WS CRC-узла, содержащей все страницы, принадлежащие конкретному набору и являющиеся кандидатами на комбинирование, и списка свободных комбинационных блоков, которые используются для хранения прототипных PTE-записей и общей страницы. Алгоритм присоединяется к целевому процессу и блокирует его рабочий набор (повышая IRQL до уровня диспетчеризации). Таким образом он может напрямую читать и записывать все страницы без необходимости отображать их заново.

Алгоритм обрабатывает каждую CRC/PFN-запись в списке, но, поскольку он работает на уровне диспетчеризации IRQL и выполнение может потребовать некоторого времени, перед анализом следующей позиции алгоритм проверяет, имеются ли у процессора в очереди DPC или запланированные объекты (вызовом `KeShouldYieldProcessor`). Если проверка дает положительный результат, то алгоритм принимает соответствующие меры к сохранению состояния.

Стратегия совместного использования страниц предполагает три возможных сценария:

- Страница активна и достоверна, но заполнена нулями, поэтому вместо комбинирования PTE-запись заменяется PTE-записью страницы, обнуляемой по требованию. Напомним, что это состояние является исходным для обычных операций выделения памяти в стиле `VirtualAlloc`.
- Страница активна и достоверна, но не заполнена нулями; это означает, что она предназначена для совместного использования. Алгоритм проверяет, нужно ли повысить страницу до основной. Если запись CRC/PFN содержит указатель на действительный комбинационный блок, это означает, что страница не является основной (в противном случае страница является основной). Хеш основной страницы проверяется заново, и для совместного использования выделяется новая физическая страница. В противном случае используется уже существующий комбинационный блок (с увеличением его счетчика ссылок). Теперь система готова преобразовать закрытую страницу в общую и вызывает функцию `MiConvertPrivateToProto` для выполнения непосредственной работы.
- Страница находится в списке измененных или ожидающих страниц. В этом случае она связывается с системным адресом как достоверная, и ее хеш пересчитывается. Алгоритм выполняет те же действия, что и в приведенном выше сценарии; единственное различие заключается в том, что PTE-запись преобразуется из общей в прототипную функцией `MiConvertStandbyToProto`.

Когда совместное использование текущей страницы завершается, система вставляет комбинационный блок основной копии в структуру PCS. Это важно, потому что комбинационный блок образует связь между закрытой PTE-записью и комбинированной страницей.

От закрытой PTE-записи к общей

Цель функции `MiConvertPrivateToProto` — преобразование PTE-записи активной и достоверной страницы. Если функция обнаруживает, что прототипная PTE-запись в комбинационном блоке равна 0, это означает, что основную страницу необходимо создать (вместе с основной общей прототипной PTE-записью). Затем свободная физическая страница связывается с системным адресом, а содержимое закрытой страницы копируется в новую общую страницу. Прежде чем фактически создавать общую PTE-запись, система должна освободить все резервирование страничных файлов (см. раздел «Резервирование страничных файлов» этой главы) и заполнить PFN-дескриптор общей страницы. У общего PFN устанавливается прототипный бит, а указателю на блок PTE присваивается PFN физической страницы, содержащей прототипную PTE-запись. Что еще важнее, указатель на PTE указывает на PTE внутри комбинационного блока, но 63-й бит равен 0. Этот признак сообщает системе, что PFN принадлежит комбинированной странице.

Затем система должна изменить PTE-запись закрытой страницы, чтобы в качестве целевого PFN была задана общая физическая страница, маска защиты была изменена на копирование при записи, а PTE-запись закрытой страницы была помечена как достоверная. Прототипная PTE-запись в комбинационном блоке также помечается как достоверная аппаратная PTE-запись: содержимое идентично новой PTE-записи закрытой страницы. Наконец, пространство страничного файла, выделенное для закрытой страницы, освобождается, а исходный PFN закрытой страницы помечается как удаленный. Кэш TLB сбрасывается, а размер рабочего набора закрытого процесса уменьшается на 1 страницу.

В противном случае (прототипная PTE-запись в комбинационном блоке отлична от нуля) это означает, что закрытая страница должна быть копией основной страницы. Преобразование необходимо только для активной PTE-записи закрытой страницы. PFN общей страницы связывается с системным адресом, и содержимое двух страниц сравнивается. Это важный момент, поскольку алгоритм CRC не гарантирует уникальности значений в общем случае. Если две страницы не совпадают, функция прекращает обработку и возвращает управление. В противном случае отображение общей страницы уничтожается, а приоритет страницы общего PFN задается равным большему из двух значений. На рис. 5.46 изображено состояние, в котором существует только основная страница.

Теперь алгоритм вычисляет новую недостоверную программную прототипную PTE-запись, которая должна быть вставлена в таблицу закрытых страниц процесса. Для этого он читает адрес аппаратной PTE-записи, которая отображает общую страницу (находящуюся в комбинационном блоке), сдвигает его и устанавливает биты прототипной и комбинированной записи. Алгоритм проверяет, что счетчик пользователей закрытого PFN равен 1. В противном случае обработка отменяется. Алгоритм записывает новую программную PTE-запись в закрытую таблицу страниц процесса и уменьшает счетчик пользователей таблицы страниц старого PFN (не забудьте, что активные PFN всегда содержат указатель

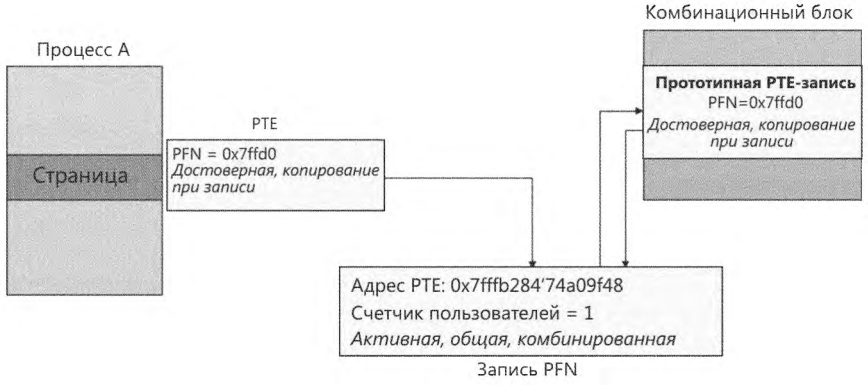


Рис. 5.46. Комбинированная основная страница

на таблицу страниц). Размер рабочего набора целевого процесса уменьшается на 1 страницу, а кэш TLB сбрасывается. Старая закрытая страница переводится в переходное состояние, а его PFN помечается для удаления. На рис. 5.47 изображены две страницы, причем новая страница указывает на прототипную PTE-запись, но еще не является достоверной.

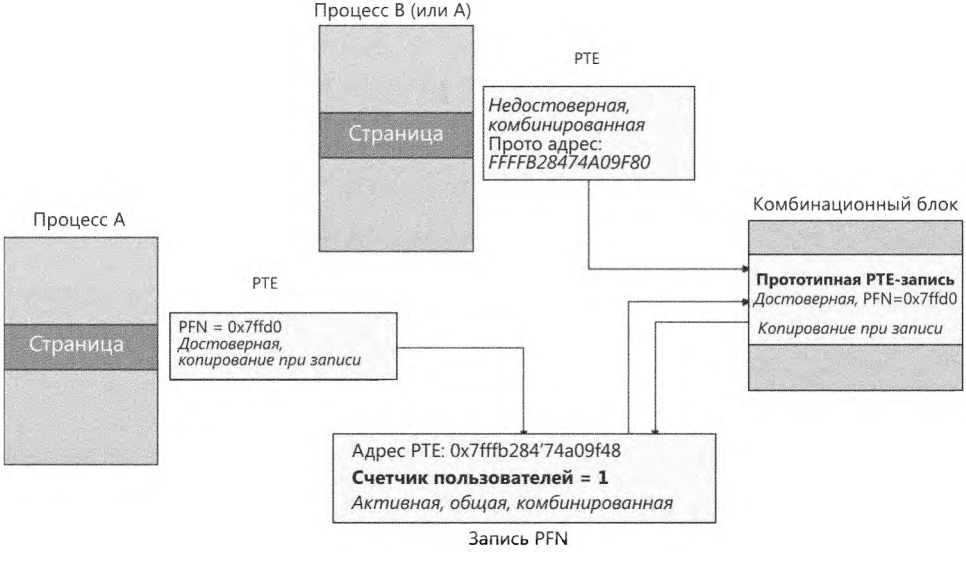


Рис. 5.47. Комбинированная основная страница

Наконец, система использует еще один прием для предотвращения усечения рабочего набора общих страниц, имитируя ошибку страницы. При этом счетчик пользователей общего PFN снова увеличивается, и в тот момент, когда процесс пытается прочитать общую страницу, ошибка страницы не происходит. В результа-

те закрытая PTE-запись снова становится достоверной аппаратной PTE-записью. На рис. 5.48 показана схема обработки программной ошибки страницы для второй страницы, в результате которой она становится достоверной, а счетчик пользователей увеличивается.

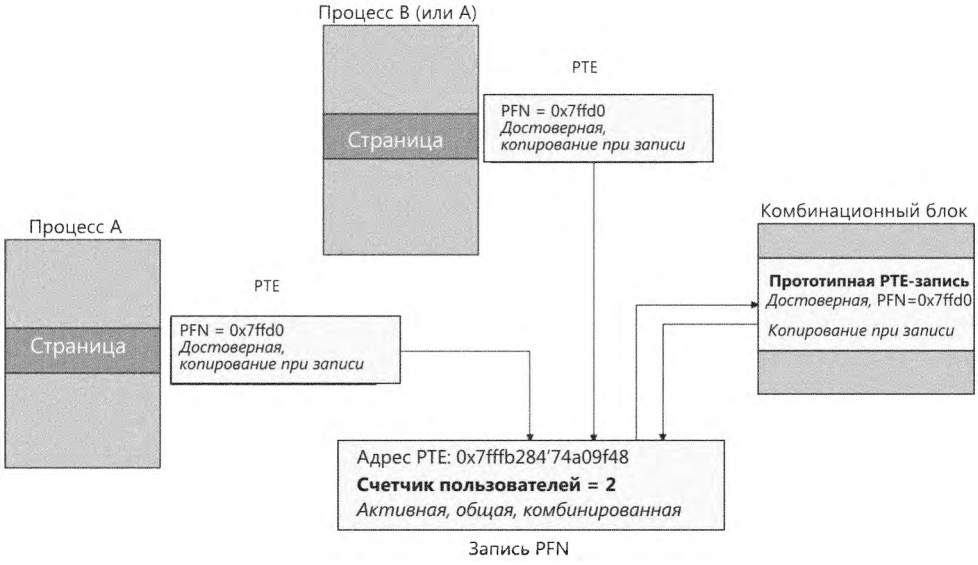


Рис. 5.48. Комбинированные страницы после имитации ошибки страницы

Освобождение комбинированных страниц

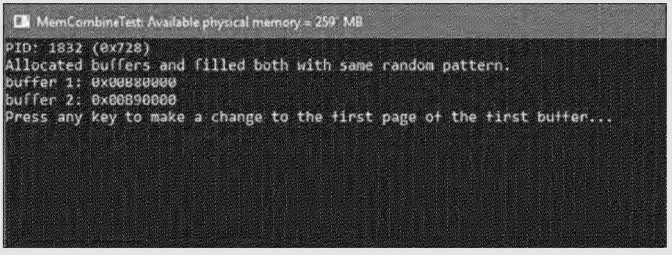
Когда системе потребуется освободить конкретный виртуальный адрес, она сначала находит адрес PTE-записи, которая отображает эту страницу. В PFN комбинированной страницы установлены биты прототипной и комбинированной записи. Запрос на освобождение комбинированной PFN-записи обрабатывается точно так же, как запросы для прототипных PFN. Единственное различие заключается в том, что система (если установлен бит комбинированной страницы) вызывает `MiDecrementCombinedPte` после обработки прототипной PFN.

`MiDecrementCombinedPte` — простая функция для уменьшения счетчика ссылок комбинационного блока прототипной PTE-записи. (Помните, что на этой стадии PTE находится в переходном состоянии, потому что диспетчер памяти уже разорвал связь с отображаемой страницей. Счетчик пользователей физической страницы уже уменьшился до 0, поэтому система перевела PTE в переходное состояние.) Если счетчик ссылок уменьшается до 0, прототипная PTE-запись будет освобождена, физическая страница помещается в список свободных, а комбинационный блок возвращается в список свободных комбинированных страниц структуры PCS секции памяти.

ЭКСПЕРИМЕНТ: КОМБИНИРОВАНИЕ ПАМЯТИ

В этом эксперименте мы понаблюдаем за работой комбинирования памяти. Выполните следующие действия.

1. Запустите сеанс отладчика ядра с целевой виртуальной машиной (см. главу 4).
2. Скопируйте исполняемые файлы MemCombine32.exe (для 32-разрядных систем) или MemCombine64.exe (для 64-разрядных систем) и MemCombineTest.exe из архива загружаемых ресурсов книги на целевую машину.
3. Запустите MemCombineTest.exe на целевой машине. Результат выглядит примерно так:



4. Обратите внимание на два приведенных адреса. По ним располагаются два буфера, заполненные случайными последовательностями байтов, повторяющихся таким образом, что обе страницы имеют одинаковое содержимое.
5. Передайте управление отладчику. Найдите процесс MemCombineTest:

```
0: kd> !process 0 0 memcombinetest.exe
PROCESS ffffe70a3cb29080
    SessionId: 2  Cid: 0728    Peb: 00d08000  ParentCid: 11c4
    DirBase: c7c95000  ObjectTable: ffff918ede582640  HandleCount: <Data Not
Accessible>
    Image: MemCombineTest.exe
```

6. Переключитесь на найденный процесс:

```
0: kd> .process /i /r ffffe70a3cb29080
You need to continue execution (press 'g' <enter>) for the context
to be switched. When the debugger breaks in again, you will be in
the new process context.
0: kd> g
Break instruction exception - code 80000003 (first chance)
nt!DbgBreakPointWithStatus:
fffff801'94b691c0 cc              int     3
```

7. При помощи команды !pte найдите блок PFN, в котором хранятся два буфера:

```
0: kd> !pte b80000
                                           VA 0000000000b80000
PXE at FFFFA25128944000    PPE at FFFFA25128800000    PDE at
FFFFA25100000028    PTE at FFFFA20000005C00
contains 00C0000025BAC867  contains 0FF00000CAA2D867  contains
```

```
00F000003B22F867  contains B9200000DEDFB867
pfn 25bac     ---DA--UWEV  pfn caa2d     ---DA--UWEV  pfn 3b22f     ---DA--
UWEV  pfn dedfb     ---DA--UW-V

0: kd> !pte b90000
                                           VA 0000000000b90000
PXE at FFFFA25128944000    PPE at FFFFA25128800000    PDE at
FFFFA25100000028    PTE at FFFFA20000005C80
contains 00C0000025BAC867  contains 0FF00000CAA2D867  contains
00F000003B22F867  contains B9300000F59FD867
pfn 25bac     ---DA--UWEV  pfn caa2d     ---DA--UWEV  pfn 3b22f     ---DA--
UWEV  pfn f59fd     ---DA--UW-V
```

8. Обратите внимание на различающиеся значения PFN; это означает, что страницы отображаются на разные физические адреса. Возобновите выполнение целевого процесса.

9. В целевом процессе откройте привилегированное окно командной строки, перейдите в каталог, в который был скопирован файл MemCombine(32/64), и запустите его. Программа инициирует полное комбинирование памяти, которое может занять несколько секунд.

10. Когда комбинирование будет завершено, снова передайте управление отладчику. Повторите шаги 6 и 7. Вы увидите, что значения PFN изменились:

```
1: kd> !pte b80000
                                           VA 0000000000b80000
PXE at FFFFA25128944000    PPE at FFFFA25128800000    PDE at
FFFFA25100000028    PTE at FFFFA20000005C00
contains 00C0000025BAC867  contains 0FF00000CAA2D867  contains
00F000003B22F867  contains B9300000EA886225
pfn 25bac     ---DA--UWEV  pfn caa2d     ---DA--UWEV  pfn 3b22f     ---DA--
UWEV  pfn ea886     C---A--UR-V

1: kd> !pte b90000
                                           VA 0000000000b90000
PXE at FFFFA25128944000    PPE at FFFFA25128800000    PDE at
FFFFA25100000028    PTE at FFFFA20000005C80
contains 00C0000025BAC867  contains 0FF00000CAA2D867  contains
00F000003B22F867  contains BA600000EA886225
pfn 25bac     ---DA--UWEV  pfn caa2d     ---DA--UWEV  pfn 3b22f     ---DA--
UWEV  pfn ea886     C---A--UR-V
```

11. На этот раз PFN совпадают; это означает, что страницы отображаются на один адрес оперативной памяти. Также обратите внимание на флаг C в PFN (от Copy-on-Write).

12. Возобновите выполнение целевого процесса и нажмите любую клавишу в окне MemCombineTest. Это приводит к изменению одного байта в первом буфере.

13. Снова передайте управление целевому процессу и повторите шаги 6 и 7.

```
1: kd> !pte b80000
                                           VA 0000000000b80000
```

```
PXE at FFFFA25128944000    PPE at FFFFA25128800000    PDE at
FFFFA25100000028    PTE at FFFFA20000005C00
contains 00C0000025BAC867  contains 0FF00000CAA2D867  contains
00F000003B22F867  contains B9300000813C4867
pfn 25bac     ---DA--UWEV  pfn caa2d     ---DA--UWEV  pfn 3b22f     ---DA--
UWEV  pfn 813c4     ---DA--UW-V

1: kd> !pte b90000
                                           VA 0000000000b90000
PXE at FFFFA25128944000    PPE at FFFFA25128800000    PDE at
FFFFA25100000028    PTE at FFFFA20000005C80
contains 00C0000025BAC867  contains 0FF00000CAA2D867  contains
00F000003B22F867  contains BA600000EA886225
pfn 25bac     ---DA--UWEV  pfn caa2d     ---DA--UWEV  pfn 3b22f     ---DA--
UWEV  pfn ea886     C---A--UR-V
```

14. Значение PFN первого буфера изменилось, а флаг копирования при записи был снят. Страница изменилась и была перемещена в другой адрес оперативной памяти.

Анклавы в памяти

Потокам, выполняющимся в процессе, доступно все адресное пространство процесса (доступ определяется защитой страницы, которая может быть изменена). В большинстве случаев этот факт нас устраивает; но если вредоносному коду удастся внедриться в процесс, он будет обладать точно такими же правами. Он сможет читать данные, содержащие конфиденциальную информацию, и даже изменять их.

Компания Intel создала технологию Intel Software Guard Extensions (SGX), которая позволяет создавать в памяти защищенные *анклавы* (enclaves) — защищенные зоны в адресном пространстве процесса. Код и данные в таком адресном пространстве защищаются процессором от кода, выполняемого за пределами анклава. И наоборот, код, выполняющийся в анклаве, обладает полным (нормальным) доступом к адресному пространству процесса вне анклава. Естественно, защита распространяется на доступ из других процессов и даже кода, выполняемого в режиме ядра. Упрощенная диаграмма анклавов представлена на рис. 5.49.

Рис. 5.49. Анклавы

Intel SGX поддерживаются процессорами Core шестого поколения («Skylake») и последующих поколений. Intel поддерживает собственный инструментарий SDK для разработчиков приложений, который может использоваться в Windows 7 и последующих версиях (только 64-разрядных). Начиная с Windows 10 версии 1511 и Server 2016, Windows предоставляет абстракцию на базе функций Windows API, которая снимает необходимость в Intel SDK. Возможно, другие фирмы-разработчики процессоров создадут в будущем аналогичные решения, которые будут инкапсулироваться теми же

API-функциями, образуя относительно портируемую прослойку для создания и заполнения анклавов.

ПРИМЕЧАНИЕ Не все процессоры Core шестого поколения поддерживают SGX. Также для работы SGX в системе должно быть установлено соответствующее обновление BIOS. За дополнительной информацией обращайтесь к документации Intel SGX. Сайт Intel SGX находится по адресу *https://software.intel.com/en-us/sgx*.

ПРИМЕЧАНИЕ На момент написания книги компания Intel выпустила две версии SGX (версии 1.0 и 2.0). Windows в настоящее время поддерживает только версию 1.0. Различия выходят за рамки книги; за подробностями обращайтесь к документации SGX.

ПРИМЕЧАНИЕ Текущие версии SGX не поддерживают анклавы в кольце 0 (режим ядра). Поддерживаются только анклавы в кольце 3 (пользовательский режим).

Программный интерфейс

С точки зрения прикладного программиста создание и работа с анклавами состоят из следующих действий (дополнительная информация приводится в следующих разделах):

1. Сначала программа должна проверить, поддерживаются ли анклавы, вызывая функцию `IsEnclaveTypeSupported` и передавая значение, определяющее технологию анклавов (в настоящее время поддерживается только значение `ENCLAVE_TYPE_SGX`).

2. Новый анклав создается вызовом функции `CreateEnclave`, которая по набору аргументов напоминает функцию `VirtualAllocEx`. Например, можно создать анклав в процессе, отличном от вызывающего процесса. Использование этой функции усложняется необходимостью предоставления конфигурационной структуры, зависимой от фирмы-разработчика, что для Intel означает 4-килобайтную структуру данных *SECS* (*SGX Enclave Control Structure*), которую Microsoft не определяет явно. Предполагается, что разработчики создадут собственную структуру для конкретной используемой технологии, определяемую в документации.

3. После того как будет создан пустой анклав, на следующем этапе анклав необходимо заполнить кодом и данными из-за границ анклава. Эта задача решается вызовом функции `LoadEnclaveData`, которая использует внешнюю (по отношению к анклаву) память для копирования данных в анклав. Для заполнения анклава могут использоваться многократные вызовы `LoadEnclaveData`.

4. Последняя необходимая операция — «активизация» анклава — выполняется вызовом функции `InitializeEnclave`. К этому моменту код, настроенный для выполнения в анклаве, может начать выполнение.

5. К сожалению, выполнение кода в анклаве не инкапсулируется в API. Разработчик должен использовать язык ассемблера. Инструкция EENTER передает управление в анклав, а инструкция EEXIT возвращает его в вызывающую функцию. Также возможно аномальное завершение кода в анклаве инструкцией AEX (Asynchronous Enclave Exit) вследствие сбоя. Подробности выходят за рамки этой книги, так как они не связаны с Windows. За подробностями обращайтесь к документации SGX.
6. Наконец, для уничтожения анклава обычная функция `VirtualFree(Ex)` вызывается для указателя на анклав, полученного при вызове `CreateEnclave`.

Инициализация анклавов

Во время загрузки Winload (начальный загрузчик Windows) вызывает функцию `OslEnumerateEnclavePageRegions`, которая сначала проверяет поддержку SGX при помощи инструкции CPUID. Если SGX поддерживается, загрузчик выполняет инструкции для перебора дескрипторов EPC (Enclave Page Cache). EPC — защищенная память, предоставляемая процессором для создания и использования анклавов. Для каждого EPC `OslEnumerateEnclavePageRegions` вызывает функцию `BlMmAddEnclavePageRange`, которая добавляет информацию диапазона страниц в отсортированный список дескрипторов памяти с типом `LoaderEnclaveMemory`. Список в конечном итоге сохраняется в поле `MemoryDescriptorListHead` структуры `LOADER_PARAMETER_BLOCK`, используемой для передачи информации от загрузчика ядру.

В фазе 1 процесса инициализации вызывается функция диспетчера памяти `MiCreateEnclaveRegions` для создания АВЛ-дерева обнаруженных анклавов (для проведения быстрого поиска в случае необходимости); дерево хранится в поле `MiState.Hardware.EnclaveRegions`. Ядро добавляет новый список страниц анклава, а специальный флаг, передаваемый `MiInsertPageInFreeOrZeroedList`, включает функциональность использования нового списка. Но поскольку у диспетчера памяти кончаются идентификаторы списков (в конце концов, 3 бита в любом случае позволяют представить не более 8 значений), ядро идентифицирует эти страницы как «нерабочие», в которых в настоящее время происходит ошибка. Диспетчер памяти знает, что использовать нерабочие страницы нельзя, поэтому «нерабочесть» страниц анклава предотвращает их использование в ходе обычных операций управления памятью. Таким образом, в итоге эти страницы оказываются в списке нерабочих страниц.

Построение анклава

API-функция `CreateEnclave` в конечном итоге вызывает функцию ядра `NtCreateEnclave`. Как упоминалось ранее, при вызове должна передаваться структура SECS, документированная в спецификации Intel SGX (табл. 5.24).

`NtCreateEnclave` сначала проверяет, поддерживаются ли анклавы памяти; для этого она проверяет корень АВЛ-дерева (не с помощью более медленной инструкции CPUID). Затем она создает копию переданной структуры (как это обычно делается функциями ядра, получающими данные из пользовательского режима) и присоединяется к целевому процессу (`KeStackAttachProcess`), если анклав создается в другом процессе, отличном от вызывающего процесса. Затем управление передается `MiCreateEnclave` для выполнения непосредственной работы.

Таблица 5.24. Структура SECS

Поле	Смещение (в байтах)	Размер (в байтах)	Описание
`SIZE`	0	8	Размер анклава в байтах; должен быть равен степени 2
`BASEADDR`	8	8	Базовый линейный адрес анклава должен естественным образом выравниваться по размеру
`SSAFRAMESIZE`	16	4	Размер одного блока SSA в страницах (включая XSAVE, дополнение, GPR и, возможно, MISC)
`MRSIGNER`	128	32	Регистр метрики, расширенный открытым ключом, используемым для проверки анклава. За информацией о формате обращайтесь к описанию SIGSTRUCT
`RESERVED`	160	96	
`ISVPRODID`	256	2	Идентификатор продукта для анклава
`ISVSVN`	258	2	Номер версии безопасности (SVN) для анклава
`EID`	Зависит от реализации	8	Идентификатор анклава
`PADDING`	Зависит от реализации	352	Дополнение из сигнатуры (используется для построения производных строк ключей)
`RESERVED`	260	3836	Включает EID, другое ненулевое зарезервированное поле, а также поля, которые должны быть заполнены нулями

Функция `MiCreateEnclave` прежде всего создает в памяти информационную структуру AWE (Address Windowing Extension), которая также должна использоваться AWE API. Дело в том, что по аналогии с функциональностью AWE анклавы позволяют приложениям пользовательского режима напрямую обращаться к физическим страницам (т. е. к физическим страницам, которые являются EPC-страницами на основании процедуры, описанной ранее). Каждый раз, когда приложение пользовательского режима может напрямую управлять физическими страницами, должны использоваться блокировки и структуры данных AWE. Эта структура данных сохраняется в поле AweInfo структуры EPROCESS.

Затем `MiCreateEnclave` вызывает `MiAllocateEnclaveVad` для создания VAD-дескриптора, описывающего диапазон виртуальной памяти анклава. У этого

VAD-дескриптора устанавливается флаг `VadAwd` (как и у всех VAD-дескрипторов AWE), но также имеется дополнительный флаг `Enclave`, который отличает его от «настоящих» AWE VAD-дескрипторов. Наконец, в процессе создания VAD здесь должен быть выбран адрес пользовательского режима для памяти анклава (если он не задан явно при исходном вызове `CreateEnclave`).

На следующем шаге `MiCreateEnclave` получает страницу анклава — независимо от размера анклава или исходного подтверждения. Дело в том, что согласно документации SGX от компании Intel с каждым анклавом должна быть связана управляющая страница, содержащая хотя бы одну страницу. Функция `MiGetEnclavePage` используется для получения необходимого выделенного блока. Функция просто сканирует список страниц анклава, упоминавшийся выше, и извлекает одну страницу. Возвращенная страница отображается с использованием системной PTE-записи, хранящейся как части VAD-дескриптора анклава; функция `MiInitializeEnclavePfn` создает соответствующую структуру данных PFN и помечает ее как измененную (Modified), активную и достоверную (ActiveAndValid).

Нет никаких специальных битов, по которым можно было бы отличить PFN анклава от любой другой активной области памяти (например, невыгружаемого пула). Здесь в игру вступает АВЛ-дерево областей анклавов; `MI_PFN_IS_ENCLAVE` — функция, которая используется ядром каждый раз, когда требуется проверить, действительно ли PFN описывает EPC-область.

При инициализации PFN системная PTE-запись преобразуется в итоговую глобальную PTE-запись, и вычисляется итоговый виртуальный адрес. На последнем шаге `MiCreateEnclave` вызывается функция `KeCreateEnclave`, которая выполняет низкоуровневые операции ядра по созданию анклава, включая взаимодействие с аппаратной реализацией SGX. В частности, `KeCreateEnclave` отвечает за заполнение базового адреса, необходимого для структуры SECS, если этот адрес не был указан вызывающей стороной, так как этот адрес должен быть занесен в структуру SECS перед взаимодействием с оборудованием SGX для создания анклава.

Загрузка данных в анклав

После того как анклав будет создан, в него необходимо загрузить информацию. Для этой цели предоставляется функция `LoadEnclaveData`. Она всего лишь передает запрос нижележащей функции исполнительной подсистемы `NtLoadEnclaveData`. Эта функция напоминает комбинацию операции копирования памяти с некоторыми атрибутами `VirtualAlloc` (такими, как защита страниц).

Если анклав, созданный `CreateEnclave`, еще не содержит подтвержденных страниц анклава, сначала эти страницы необходимо получить; это приведет к заполнению нулями памяти, добавляемой в анклав, которая может быть заполнена данными из необнуленной памяти за пределами анклава. В противном случае, если был передан исходный размер инициализации до подтверждения, страницы анклава могут быть напрямую заполнены необнуленной памятью за пределами анклава.

Поскольку анклавная память описывается VAD-дескриптором, многие традиционные API-функции управления памятью будут работать и с этой памятью (по крайней мере, частично). Например, вызов `VirtualAlloc` (приводящий к вызову `NtAllocateVirtualMemory`) для такого адреса с флагом `MEM_COMMIT` приведет к вызову `MiCommitEnclavePages`, который проверит совместимость маски защиты для новых страниц (т. е. комбинацию атрибутов чтения, защиты и/или выполнения без специальных флагов кэширования или комбинированной записи), и последующему вызову `MiAddPagesToEnclave` с передачей указателя на VAD-дескриптор анклава, связанный с диапазоном адресов, маской защиты, переданной `VirtualAlloc`, и PTE-адресами, соответствующими подтверждаемому диапазону виртуальных адресов.

Функция `MiAddPagesToEnclave` сначала проверяет, связаны ли с анклавным VAD-дескриптором какие-либо существующие EPC-страницы и достаточно ли их для подтверждения. Если их недостаточно, вызывается функция `MiReserveEnclavePages` для получения достаточного объема памяти. Функция `MiReserveEnclavePages` обращается к текущему списку страниц анклава и подсчитывает сумму. Если физических EPC-страниц, предоставляемых процессором, оказывается недостаточно (на основании информации, полученной при загрузке), вызов функции завершается неудачей. В противном случае функция `MiGetEnclavePage` вызывается в цикле для нужного количества страниц.

Каждая полученная PFN-запись связывается с массивом PFN в VAD-дескрипторе анклава. Фактически это означает, что после того, как PFN-блок анклава извлекается из списка страниц анклава и переводится в активное состояние, VAD-дескриптор анклава действует как список активных PFN анклава.

После того как необходимые подтвержденные страницы будут получены, `MiAddPagesToEnclave` преобразует атрибуты защиты страниц, переданные `LoadEnclaveData`, в их SGX-эквиваленты. Затем функция резервирует соответствующее количество системных PTE-записей для хранения информации обо всех EPC-страницах, которые могут потребоваться. При наличии такой информации будет вызвана функция `KeAddEnclavePage`, которая обращается к оборудованию SGX для непосредственного добавления страниц.

Специальный атрибут защиты страниц `PAGE_ENCLAVE_THREAD_CONTROL` обозначает, что память выделяется для структуры TCS (Thread Control Structure), определяемой SGX. Каждая структура TCS представляет отдельный программный поток, который может независимо выполняться в анклаве.

Функция `NtLoadEnclaveData` проверяет параметры, а затем вызывает `MiCopyPagesIntoEnclave` для выполнения непосредственной работы, которая может потребовать получения подтвержденных страниц, упоминавшихся выше.

Инициализация анклава

После того как анклав был создан и в него были переданы данные, перед фактическим выполнением кода в анклаве остается сделать один последний шаг. Не-

обходимо вызвать функцию `InitializeEnclave`, которая сообщила бы SGX, что анклав перешел в финальное состояние перед началом выполнения. Функции `InitializeEnclave` должны передаваться две структуры, специфические для SGX (`SIGSTRUCT` и `EINITTOKEN`; за подробностями обращайтесь к документации SGX).

Функция `NtInitializeEnclave`, вызываемая `InitializeEnclave`, проводит проверку параметров, убеждается в том, что полученный VAD-дескриптор анклава обладает правильными атрибутами, после чего передает структуры оборудованию SGX. Учтите, что инициализация анклава может быть выполнена только один раз.

И теперь остается последний шаг: выполнение инструкции ассемблера Intel EENTER для запуска кода (за подробностями также обращайтесь к документации SGX).

Упреждающее управление памятью (супервыборка)

Традиционная схема управления памятью в операционных системах строится на модели подкачки по требованию, которую мы и рассматривали до сих пор с небольшими, связанными с кластеризацией и предвыборкой усовершенствованиями, позволяющими оптимизировать дисковый ввод/вывод к моменту возникновения ошибки отсутствия запрашиваемой страницы. Однако в клиентской версии Windows произошло еще одно важное улучшение схемы управления физической памятью, и связано оно с реализацией *супервыборки* (superfetch). В этой схеме усовершенствован подход, основанный на самых последних обращениях к памяти и реализуемый с помощью истории обращений к файлам и упреждающего управления памятью.

В схеме управления списком ожидающих страниц прежних версий Windows было два ограничения. Во-первых, присваивание приоритета тем или иным страницам основывалось только на недавнем поведении процессов без прогнозирования их будущих требований к памяти. Во-вторых, данные, используемые для присваивания приоритетов, ограничивались списком страниц, которые принадлежали процессу в тот или иной период времени. Эти ограничения способны привести к ситуациям, когда компьютер, действуя по своему усмотрению, запускает системные приложения, которые интенсивно расходуют память (например, выполняющие антивирусное сканирование или дефрагментацию диска) и сильно тормозят работу запущенных (или запускаемых) интерактивных приложений. Сходная ситуация может наблюдаться, когда пользователь намеренно запускает приложение, интенсивно работающее с данными и/или памятью, а затем возвращается к другим программам, реакция которых на его действия существенно замедляется.

Снижение производительности происходит из-за того, что код и данные, которые работающие приложения кэшировали в памяти, начинают переписываться данными приложений, активно потребляющих память. В результате выполнение других

приложений замедляется, потому что они вынуждены получать свои данные и код не из кэша, а с диска. В клиентских версиях Windows эти ограничения в значительной мере преодолеваются благодаря технологии супервыборки.

Компоненты

Технологию супервыборки реализуют в системе несколько компонентов, которые совместно осуществляют упреждающее управление памятью и сводят к минимуму влияние на пользовательскую активность механизма супервыборки, когда он делает свою работу.

- **Трассировщик** (Tracer). Трассировщик является частью компонента ядра (Pf), который позволяет механизму супервыборки в любое время запрашивать подробную информацию о страницах, сеансах и процессах. Механизм супервыборки для отслеживания использования файлов задействует также драйвер `FileInfo` (%SystemRoot%\System32\Drivers\Fileinfo.sys).
- **Сборщик и обработчик данных трассировки** (Trace collector and processor). Этот сборщик работает совместно с компонентами трассировки, предоставляя простой журнал на основе полученных трассировочных данных. Трассировочные данные хранятся в памяти и передаются обработчику. Затем обработчик передает журнальные записи трассировки агентам, которые обслуживают файлы истории (см. далее) в памяти и сохраняют их на диске, когда служба останавливается (например, в процессе перезагрузки).
- **Агенты** (Agents). Механизм супервыборки хранит информацию об обращении к файлам в файлах истории, которые отслеживают виртуальные смещения. Агенты группируют страницы по следующим признакам:
  - обращения к страницам при активности пользователя;
  - обращения к страницам со стороны фонового процесса;
  - серьезный сбой при активности пользователя;
  - обращения к страницам в ходе запуска приложений;
  - обращения к страницам пользователя после длительного простоя.
- **Диспетчер сценариев** (Scenario manager). Этот компонент, также называемый *агентом контекста*, обслуживает три сценарных плана супервыборки: гибернацию, ожидание (сон) и быстрое переключение между пользователями. Та часть диспетчера сценариев, которая работает в режиме ядра, предоставляет API-функции для инициирования и остановки сценариев, управления текущим состоянием сценария и привязки трассировочной информации к этим сценариям.
- **Перенастройщик баланса** (Rebalancer). Работа перенастройщика баланса основана на информации, предоставляемой агентами супервыборки, а также

на текущем состоянии системы (например, на состоянии списков страниц, составленных по приоритетам). Перенастройщик баланса — специализированный агент в службе супервыборки из пользовательского режима — запрашивает базу данных PFN-номеров и меняет в ней приоритеты, основываясь на оценке каждой страницы и создавая таким образом списки ожидающих страниц, расставленных по приоритетам. Перенастройщик баланса может также выдавать команды диспетчеру памяти на изменение рабочих наборов процессов системы; он является единственным агентом, который предпринимает реальные действия в отношении системы, все остальные агенты просто фильтруют информацию для перенастройщика баланса, используемую в его решениях. Помимо изменения приоритетов, перенастройщик баланса также инициирует предвыборку через поток предвыборки, который использует драйвер FileInfo и службы ядра для предварительной загрузки памяти полезными страницами.

И наконец, все эти компоненты используют средства диспетчера памяти, позволяющие запрашивать подробную информацию о состоянии каждой страницы в базе данных PFN, текущем счетчике страниц для каждого списка страниц и приоритетного списка, а также о многом другом. На рис. 5.50 представлена диаграмма работы нескольких компонентов супервыборки. Компоненты супервыборки применяют также приоритетный ввод/вывод, чтобы свести к минимуму влияние на пользователя (приоритеты ввода/вывода рассматриваются в главе 8 части 2).

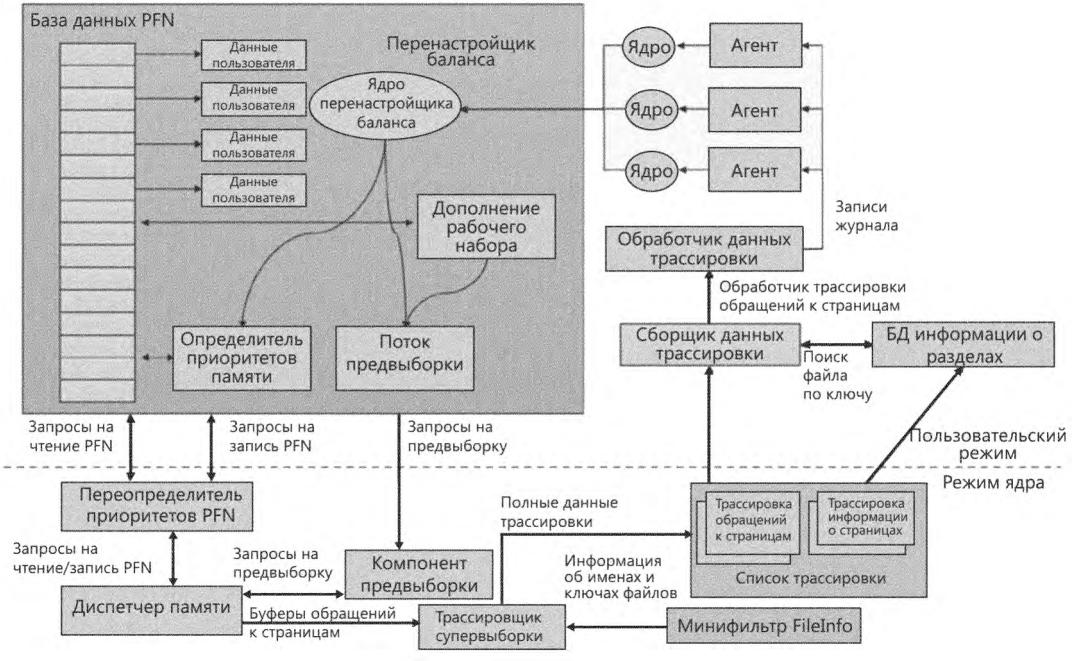


Рис. 5.50. Архитектура супервыборки

Трассировка и протоколирование

Большинство своих решений механизм супервыборки принимает на основе той информации, которая была сведена воедино, проанализирована и окончательно обработана после получения из необработанных трасс и журналов, что делает компоненты трассировки и протоколирования одними из наиболее важных. Трассировка в механизме супервыборки чем-то похожа на трассировку событий для Windows (Event Tracing for Windows, ETW), поскольку в ней используются инициаторы в коде, через которые система генерирует события, но она работает и с существующими системными механизмами, например с уведомлениями диспетчера электропитания, обратными вызовами процессов и фильтрацией в файловой системе. Кроме того, трассировщик использует традиционные механизмы старения страниц, находящиеся в диспетчере памяти, а также недавно созданные для супервыборки механизмы старения рабочих наборов и отслеживания обращений.

Механизм супервыборки никогда не отключает трассировку и постоянно запрашивает трассировочные данные у системы, которая отслеживает использование страниц и обращения к ним. Это делается с помощью диспетчера памяти, который проверяет бит обращения (access) и следит за старением рабочего набора. Информация, относящаяся к файлам, не менее важна, чем информация об использовании страниц, поскольку она позволяет расставлять приоритеты файлов данных в кэше. Механизм супервыборки задействует существующие функциональные возможности фильтрации и в дополнение к ним драйвер FileInfo. (Дополнительные сведения о фильтрующих драйверах есть в главе 6.) Этот драйвер входит в стек устройства файловой системы и отслеживает обращения к файлам и их изменения на уровне потоков данных (дополнительные сведения о потоках данных в файловой системе NTFS можно найти в главе 13), что дает ему возможность детально контролировать обращения к файлам. Основная работа драйвера FileInfo — привязка потоков данных (идентифицируемым по уникальному ключу, реализуемому в настоящее время в виде поля `FsContext` соответствующего файлового объекта) к именам файлов, чтобы служба супервыборки пользовательского режима могла идентифицировать конкретный файловый поток данных и смещение, с которым связана страница в списке ожидающих страниц, принадлежащем отображаемому на память разделу. Он также предоставляет интерфейс для прозрачной предвыборки файловых данных без помех из-за заблокированных файлов и других состояний файловой системы. Остальная часть кода драйвера обеспечивает целостность информации, отслеживая с помощью порядковых номеров операции удаления, переименования, усечения и повторного использования ключей файлов.

В любой момент в ходе трассировки перенастройщик баланса может активироваться для изменения заполнения страницы. Решения принимаются путем анализа такой информации, как выделение памяти внутри рабочих наборов, содержимое списков обнуленных, измененных и ожидающих страниц, количество ошибок страниц, состояние битов обращения в PTE, результаты постраничного отслеживания использования страниц, текущий уровень потребления виртуальных адресов и размер рабочего набора.

Данными трассировки могут быть либо данные трассировки обращений к страницам, когда трассировщик отслеживает (с помощью бита обращения), к каким страницам обращался процесс (как к файловым страницам, так и к закрытой памяти), либо данные трассировки имен, когда отслеживаются обновления ассоциаций между именами файлов и ключами файлов применительно к реальному файлу на диске. Это позволяет механизму супервыборки отобразить страницу, связанную с файловым объектом.

Хотя в трассировочных данных супервыборки отслеживаются только обращения к страницам, служба супервыборки обрабатывает эти данные в пользовательском режиме, но копает значительно глубже, добавляя собственную более насыщенную информацию, например, указывая, каковы обстоятельства загрузки страницы (загружена из резидентной памяти или вследствие серьезной ошибки отсутствия страницы), было ли это обращение к странице начальным, каков в данный момент фактический показатель обращений к странице. Сохраняется и дополнительная информация, например о состоянии системы, а также о том, по какому из недавних сценариев была сделана последняя ссылка на каждую отслеживаемую страницу. Создаваемая трассировочная информация сохраняется в памяти, попадая через регистратор в структуры данных, которые идентифицируют в случае трассировки обращений к странице пару «виртуальный адрес–рабочий набор», а в случае регистрации имен — пару «файл–смещение». Таким образом, механизм супервыборки может отслеживать, какие диапазоны виртуальных адресов заданного процесса имеют связанные со страницами события и какие диапазоны смещений для заданного файла имеют аналогичные события.

Сценарии

Одним из аспектов супервыборки, отличающим супервыборку от ее главных механизмов, реализующих изменения приоритетов страниц и предвыборки (более подробно рассматриваемых в следующем разделе), является поддержка сценариев, представляющих собой конкретные действия на машине, которые механизм супервыборки пытается сделать более удобными для пользователя.

- **Гибернация** — основная цель заключается в разумном выборе страниц, которые нужно сохранять в файле гибернации помимо существующих страниц рабочих наборов. Задача заключается в минимизации времени, необходимого, чтобы система вновь начала реагировать на действия пользователя после возобновления работы машины.
- **Ожидание** — целью является полное устранение ошибок страниц физической памяти после возобновления работы машины. Поскольку обычная система способна возобновить работу менее чем за 2 секунды, а на раскручивание жесткого диска после длительного простоя может уйти до 5 секунд, одна-единственная ошибка страницы физической памяти вызывает задержку в цикле возобновления работы. Для решения проблемы механизм супервыборки наделяет высо-

кими приоритетами те страницы, которые понадобятся при выходе из режима ожидания.

- **Быстрое переключение между пользователями** — целью является получение точной информации о приоритетах и состоянии памяти для каждого пользователя, чтобы переключение на другого пользователя приводило к немедленному переходу в рабочее состояние нужного пользовательского сеанса и не вносило больших задержек в плане доступа к отсутствующим страницам.

Все эти сценарии имеют различные цели, но все они направлены на достижение главной цели: минимизации или полной ликвидации ошибок страниц физической памяти.

Сценарии жестко запрограммированы, и механизм супервыборки управляет ими через API-функции `NtSetSystemInformation` и `NtQuerySystemInformation`, которые контролируют состояние системы. Для супервыборки используется специальный информационный класс `SystemSuperfetchInformation`, предназначенный для управления компонентами ядра и генерирования запросов — на запуск и завершение, на получение информации от сценария или привязку к сценарию одного или нескольких наборов данных трассировки.

Каждый сценарий описывается файлом плана, который, как минимум, содержит список страниц, связанных со сценарием. Значения приоритетов страниц назначаются также в соответствии с определенными правилами (см. далее). При запуске сценария диспетчер сценариев отвечает за реакцию на события путем создания списка страниц, которые требуется разместить в памяти с учетом их приоритетов.

Приоритеты страниц и перебалансировка

Ранее уже было показано, как диспетчер памяти реализует систему приоритетов страниц, чтобы определить, из какого списка ожидающих страниц страницы должны повторно использоваться для заданной операции и в какой список попадет заданная страница. Этот механизм полезен, если процессы и программные потоки могут иметь связанные приоритеты, благодаря которым процесс дефрагментации не засоряет список ожидающих страниц и/или не «крадет» страницы у интерактивных фоновых процессов. Тем не менее его истинная сила проявляется в механизмах назначения приоритетов и перебалансировки, не требующих ручного ввода данных в приложениях или жесткого кодирования информации о степени важности процессов.

Механизм супервыборки назначает страничные приоритеты на основе внутренней оценки каждой страницы, которая отчасти основана на востребованности страницы. Востребованность вычисляется по количеству использований страницы за относительные промежутки времени, например за час, за день или за неделю. Отслеживается также время использования, при этом записывается, как долго к заданной странице не было обращений. И наконец, в конечной оценке учитыва-

ются, например, такие данные, как место, откуда эта страница поступила (из какого списка), и другие закономерности доступа.

Далее конечная оценка преобразуется в значение приоритета, который может быть в диапазоне от 1 до 6 (значение 7 используется для других целей, о которых рассказывается далее). Первыми повторному использованию подвергаются страницы с низкими приоритетами из списка ожидающих страниц, как показано в эксперименте «Просмотр списков ожидающих страниц, расставленных по приоритетам». Приоритет 5 присваивается, как правило, обычным приложениям, а приоритет 1 — приложениям, выполняемым в фоновом режиме. Подобные приложения могут помечаться таким приоритетом сторонними разработчиками. И наконец, приоритет 6 требуется для того, чтобы как можно дольше удержать определенное количество особо важных страниц от повторного использования. Все остальные приоритеты расставляются на основе оценки каждой страницы.

Поскольку механизм супервыборки постепенно «изучает» пользовательскую систему, он может начать вообще без данных истории и медленно накапливать информацию о том, как различные пользователи обращаются к разным страницам. Однако все это может требовать существенного времени обучения, когда появляется новое приложение, пользователь или пакет обновления. Вместо этого за счет внутреннего инструментария Microsoft имеет возможность подготовить механизм супервыборки к сохранению нужных данных, а затем превратить их в заранее выстроенные данные трассировки. Перед поставкой Windows команда разработчиков механизма супервыборки провела трассировку наиболее распространенных приемов и закономерностей применения программ и данных, с которыми, вероятно, сталкиваются все пользователи: раскрытие стартового меню, открытие панели управления или работа с диалоговым окном открытия и сохранения файлов. Затем эти трассировочные данные были сохранены в файлах истории (поставляемых в виде ресурсов в файле Sysmain.dll) и использованы для предварительного заполнения специального списка с приоритетом 7, в который входят наиболее важные данные и страницы из которого очень редко подвергаются повторному использованию. Страницы с приоритетом 7 являются файловыми страницами, сохраняемыми в памяти даже после выхода из процесса и даже между перезагрузками (путем повторного заполнения при следующей загрузке). Наконец, страницы с приоритетом 7 являются статическими в том смысле, что их приоритет никогда не меняется, и механизм супервыборки никогда не станет динамически загружать страницы с приоритетом 7, не входящие в статический, заранее подготовленный набор.

Список приоритетов загружается в память (или заполняется заново) перенастройщиком баланса, но сам акт перебалансировки фактически осуществляется как механизмом супервыборки, так и диспетчером памяти. Как было показано ранее, механизм разбиения по приоритетам списков ожидающих страниц реализован в диспетчере памяти, и такие решения, как выбор страниц, удаляемых из списка первыми, и наделение страниц безусловной защитой, основываются на номере приоритета. Перенастройщик баланса выполняет свою работу не путем самостоятельной перебалансировки памяти, а путем изменения приоритетов, что заставляет

диспетчер памяти приступить к решению поставленных задач. Перенастройщик баланса также, если нужно, отвечает за чтение актуальных страниц с диска, чтобы они могли присутствовать в памяти (осуществляет предвыборку). Затем он назначает приоритет, на основе которого каждый агент оценивает каждую страницу, после чего диспетчер памяти может гарантировать, что каждая страница будет обработана согласно своей важности.

Перенастройщик баланса может также работать независимо от других агентов, например, если он заметит, что распределение страниц по страничным спискам ведется не оптимально или что количество повторно использованных страниц с различными приоритетами отрицательно сказывается на быстродействии системы. Кроме того, перенастройщик баланса имеет возможность при необходимости сократить рабочий набор, что может потребоваться для создания соответствующего ресурса страниц, который будет использоваться для заранее заполняемых механизмом супервыборки данных кэша. Как правило, перенастройщик баланса берет маловостребованные страницы (страницы с низким приоритетом, обнуленные страницы, страницы с достоверным содержимым, но не входящие ни в один из рабочих наборов и ставшие невостребованными) и создает более полезный набор страниц в памяти исходя из тех ресурсов, которые он сам себе выделил. После того как перенастройщик баланса решает, какие страницы поместить в память и какой уровень приоритета им нужен для загрузки (а также какие страницы можно безболезненно выбросить), он выполняет чтение с диска для их предвыборки. При этом он работает в связке со схемами назначения приоритетов диспетчера ввода/вывода, чтобы операции ввода/вывода выполнялись с очень низким приоритетом и не мешали работе пользователя.

Важно заметить, что фактическое потребление памяти механизмом предвыборки полностью зависит от ожидающих страниц, и, как уже упоминалось при рассмотрении страничной динамики, память, отведенная под ожидающие страницы, является доступной, поскольку она может быть в любое время повторно использована другим механизмом выделения памяти как свободная. Иными словами, если механизм супервыборки осуществит предвыборку «не тех данных», на пользователе это практически никак не скажется, поскольку такая память может повторно использоваться по мере необходимости и фактически не потребляет ресурсов.

И наконец, перенастройщик баланса периодически запускается, чтобы убедиться в том, что страницы с большим приоритетом действительно недавно использовались. Поскольку такие страницы будут довольно редко (или, скорее всего, вообще не будут) использоваться повторно, важно не тратить их на данные, которые большую часть времени остаются неиспользуемыми, но в определенные периоды времени используются достаточно интенсивно. При обнаружении подобной ситуации перенастройщик баланса запускается еще раз, чтобы сместить такие страницы вниз в списках приоритетов.

В дополнение к перенастройщику баланса в различные аспекты работы механизма предвыборки вовлекается специальный агент, который называется *агентом запуска приложений*. Он пытается спрогнозировать запуски приложений и создать модель на базе цепей Маркова, описывающую вероятность запусков определенных

приложений на основе фактов запусков других приложений в определенный сегмент времени. Эти сегменты времени поделены на четыре периода времени: утро, полдень, вечер и ночь, приблизительно по 6 часов в каждом, отдельно отслеживаются будние и выходные дни. Например, если в субботний и воскресный вечера пользователь обычно запускает Outlook (для отправки электронной почты) после того, как запустил Word (для написания писем), агент запуска приложений, скорее всего, осуществит предвыборку Outlook, основываясь на высокой вероятности его запуска после Word по вечерам в выходные дни.

Поскольку объем памяти сегодняшних систем достаточно велик, в среднем более 2 Гбайт (хотя супервыборка также хорошо работает и на системах с небольшими объемами памяти), фактически реальный объем памяти, которая часто требуется процессам, для оптимальной производительности должен быть резидентным, т. е. управляться поднабором всего его объема памяти, и механизм супервыборки зачастую способен разместить все требуемые страницы в оперативной памяти. А когда он этого сделать не может, такие технологии, как ReadyBoost и ReadyDrive, помогут обойтись без лишних обращений к жесткому диску.

Устойчивое функционирование

Последний механизм, улучшающий функциональность супервыборки, касается *устойчивого функционирования* (robust performance), или просто *устойчивости* (robustness). Компонент устойчивости, управляемый службой супервыборки пользовательского режима, но в итоге реализованный в ядре (в виде Pf-процедур), при доступе к конкретному файлу отслеживает те операции ввода/вывода, которые могли бы нанести ущерб производительности системы из-за заполнения списков ожидающих страниц ненужными данными. Например, если процесс скопирует большой файл из файловой системы, список ожидающих страниц заполнится содержимым файла, даже если к файлу больше никогда не будет обращений (или их не будет в течение длительного времени). Это приведет к тому, что окажутся выброшенными все остальные данные с тем же приоритетом (а если бы это произошло с какой-нибудь интерактивной полезной программой, то вполне возможно, что ее приоритет был бы по меньшей мере не ниже 5).

Механизм супервыборки реагирует на две конкретные схемы ввода/вывода:

- **Последовательный доступ к файлам.** При такой схеме ввода/вывода система перебирает все данные файла.
- **Последовательный доступ к каталогам.** При такой схеме система перебирает все файлы в каталоге.

Когда механизм супервыборки обнаруживает, что определенный объем данных (выше некоего внутреннего порога) заполнил список ожидающих страниц, он агрессивно снижает приоритет тех страниц, которые были использованы для отображения данного файла (делая их таким образом более устойчивыми), но только в рамках целевого процесса (не нанося тем самым ущерба другим приложениям).

Этим страницам, которые можно назвать *устойчивыми*, фактически назначается новый приоритет, равный 2.

Поскольку данный компонент супервыборки реагирует на ситуацию, а не работает на упреждение, на повышение устойчивости может уйти некоторое время. Поэтому при следующем запуске процесса с подобным поведением механизм супервыборки начинает его отслеживать. Если он выясняет, что процесс всегда повторяет отслеживаемый тип последовательного доступа, данный факт фиксируется, и компонент супервыборки не ждет, как будет развиваться ситуация, а сразу повышает устойчивость файловых страниц, когда они отображаются на память. С этого момента устойчивым для будущих операций доступа к файлам считается весь процесс.

Однако, действуя подобным образом, компонент супервыборки потенциально может нанести урон многим вполне легитимным приложениям или пользовательским сценариям, ориентированным на последовательный доступ. Например, используя утилиту Strings.exe, разработанную в Sysinternals, можно искать строку во всех исполняемых файлах, находящихся в каталоге. Если таких файлов будет много, компонент супервыборки, скорее всего, проведет повышение устойчивости. В результате при следующем запуске программы Strings.exe с другим параметром поиска она будет работать так же медленно, как и в первый раз, хотя должна была бы выполняться намного быстрее. Чтобы предотвратить подобное развитие событий, компонент супервыборки хранит список отслеживаемых процессов, а также внутренний жестко закодированный список исключений. Если впоследствии обнаруживается, что процесс повторно обращается к устойчивым файлам, механизм повышения устойчивости процесса отключается, чтобы процесс мог вернуться к ожидаемой модели поведения.

Говоря о повышении устойчивости вообще — и об оптимизациях супервыборки в частности — важно понимать, что компонент супервыборки постоянно отслеживает типичные закономерности работы и обновляет свое понимание системы, благодаря чему ему удается избежать выборки бесполезных данных. Хотя изменения в повседневной пользовательской активности или в поведении приложений могут привести к тому, что компонент супервыборки необоснованно «засорит» кэш бесполезными данными или выбросит полезные, он быстро адаптируется к любым изменениям характерных схем работы. Если действия пользователя носят непостоянный и случайный характер, в худшем случае система поведет себя так же, как вела бы себя вообще без супервыборки. Если у компонента супервыборки появляются какие-либо сомнения или он не в состоянии надежно отслеживать данные, он успокаивается и не вносит никаких изменений ни в процесс, ни в страницу.

ReadyBoost

Хотя сегодня оперативная память стала намного доступнее и относительно дешевле, чем десять лет назад, она все еще дороже таких устройств хранения данных, как жесткие диски. К сожалению, современные жесткие диски содержат множество подвижных частей, не отличаются особой прочностью и, что еще важнее,

достаточно медлительны по сравнению с оперативной памятью — особенно при позиционировании. Поэтому хранение данных супервыборки на диске — такая же неудачная затея, как выгрузка страницы с ее последующей подгрузкой по ошибке страницы физической памяти.

У твердотельных дисков некоторые недостатки сглажены, но они дороже и попрежнему медленнее оперативной памяти. В то же время интересный компромисс предлагают переносные твердотельные носители, такие как флэш-диски с USB-интерфейсом (USB Flash Disk, UFD), карты CompactFlash и Secure Digital. Они дешевле оперативной памяти и более емкие, но ввиду отсутствия механических частей время выборки данных у них меньше, чем у жестких дисков.

ПРИМЕЧАНИЕ Практически карты CompactFlash и Secure Digital почти всегда подключаются через USB-адаптер, поэтому система рассматривает их как флэш-диски с интерфейсом USB.

Особенно затратным по времени является произвольный дисковый ввод/вывод, потому что время выхода головки дискового накопителя на нужную дорожку с учетом задержки на подход нужного сектора при вращении диска, если взять обычный жесткий диск настольной системы, обычно составляет примерно 10 миллисекунд — целая вечность для современных процессоров с тактовой частотой 3–4 ГГц. В то же время флэш-память может обслуживать произвольное чтение более чем в 10 раз быстрее, чем обычный жесткий диск. Поэтому в Windows включен специальный компонент под названием *ReadyBoost*, позволяющий задействовать устройства хранения данных на основе флэш-памяти в плане создания на них промежуточного уровня кэширования между памятью и дисками.

Служба ReadyBoost (не путайте с ReadyBoot!) реализована с помощью драйвера (%SystemRoot%\System32\Drivers\Rdyboost.sys), который отвечает за запись кэшированных данных в устройство энергонезависимой оперативной памяти (Non-Volatile Random Access Memory, NVRAM). Когда UFD-диск вставляется в системный разъем, ReadyBoost просматривает устройство, чтобы определить его характеристики производительности, и сохраняет результаты тестирования в разделе HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Emdmgmt реестра. (Строка «*Emd*» сокращение от *External Memory Device* (внешнее запоминающее устройство) — рабочее название ReadyBoost в ходе разработки.)

Если размер нового устройства находится в диапазоне от 256 Мбайт до 32 Гбайт, скорость произвольного чтения данных объемом 4 Кбайт равна 2,5 Мбайт в секунду и выше, а скорость произвольной записи данных объемом 512 Кбайт составляет 1,75 Мбайт в секунду и выше, то служба ReadyBoost спросит, хотите ли вы выделить часть пространства под кэширование диска. При вашем согласии ReadyBoost создаст в корневом каталоге устройства файл ReadyBoost.sfcache, который будет использоваться для хранения кэшированных страниц.

После инициализации кэширования служба ReadyBoost перехватывает все запросы на операции чтения и записи на томах локального жесткого диска (например, C:\)

и копирует любые считываемые или записываемые данные в кэшируемый файл, создаваемый службой. Впрочем, есть и исключения — например, данные, которые давно не считывались, или данные, которые принадлежат запросам Volume Snapshot. Данные, хранящиеся на кэшируемом диске, обычно сжимаются в соотношении 2:1, следовательно, кэшированный файл размером 4 Гбайт будет, скорее всего, содержать 8 Гбайт данных. В ходе записи каждый блок кодируется по алгоритму AES (Advanced Encryption Standard — улучшенный стандарт шифрования), при этом случайный ключ сеанса генерируется во время каждой начальной загрузки, чтобы гарантировать закрытость данных в кэше после удаления устройства из системы.

Когда служба ReadyBoost обнаруживает произвольные операции чтения, которые могут быть удовлетворены из кэша, она обслуживает их оттуда, но поскольку у жесткого диска последовательное считывание выполняется эффективнее, чем у флэш-памяти, она позволяет операциям считывания в рамках схемы последовательного доступа проводиться непосредственно с диска, даже если данные находятся в кэше. Кроме того, при выполнении масштабной операции ввода/вывода чтение проводится не из кэша флэш-памяти, а из дискового кэша.

Один из недостатков флэш-памяти заключается в том, что пользователь может в любой момент ее удалить, а это означает, что система никогда не сможет хранить важные данные исключительно на этом носителе (как было показано ранее, операции записи всегда сначала направляются во вторичное хранилище). В следующем разделе рассматривается родственная технология под названием ReadyDrive, которая обладает дополнительными преимуществами и позволяет решить проблему.

ReadyDrive

ReadyDrive — это Windows-технология, использующая возможности приводов на гибридных жестких дисках (Hybrid Hard Disk Drives, H-HDD). H-HDD-диски обладают встроенной энергонезависимой флэш-памятью (также известной как NVRAM). Обычно H-HDD-диски включают в себя от 50 до 512 Мбайт кэша.

Под управлением ReadyDrive флэш-память действует не просто как автоматический прозрачный кэш, а как кэш в оперативной памяти для большинства жестких дисков. В данном случае Windows использует команды ATA-8, определяющие, что находящиеся на диске данные будут содержаться во флэш-памяти. Например, Windows при выключении системы сохранит в кэше данные начальной загрузки, что позволит ускорить повторный запуск. Также там сохраняются фрагменты файла данных гибернации, когда система переходит в этот режим, что позволяет в последующем выйти из него намного быстрее. Поскольку кэш-память остается активной, даже когда диск не вращается, Windows может воспользоваться флэш-памятью в качестве кэша при записи на диск, что позволяет отказаться от раскрутки диска, когда система работает от автономного источника электропитания. Отказ от вращения диска позволяет сэкономить энергию, затрачиваемую на дисковый привод при обычном использовании.

Еще одним потребителем технологии ReadyDrive является компонент супервыборки, поскольку эта технология предлагает те же преимущества, что и ReadyBoost, но с не-

которыми функциональными усовершенствованиями (например, для нее не требуется внешняя флэш-память, поэтому она может работать постоянно). Поскольку кэш находится на реально имеющемся физическом жестком диске (который пользователь вряд ли сможет удалить из работающего компьютера), контроллеру жесткого диска обычно не приходится учитывать возможность исчезновения данных, и он может отказаться от записи на реальный диск, а использовать только кэш.

Отражение процессов

Довольно часто случается, что процесс демонстрирует проблемное поведение, но поскольку он работает в рамках какой-нибудь службы, его приостановка для создания полного дампа памяти или интерактивная отладка нежелательны. Для сокращения времени приостановки процесса при создании дампа можно воспользоваться мини-дампом, в котором сохраняются регистры и стеки программных потоков наряду со страницами памяти, на которые ссылаются регистры. Однако этот тип дампа содержит очень ограниченный объем информации, которого во многих случаях вполне достаточно для диагностики аварийных ситуаций, но недостаточно для выявления источников общих проблем. При использовании механизма отражения процессов целевой процесс приостанавливается только на время, достаточное для получения мини-дампа, при этом создается его приостановленная клонированная копия, после чего целевому процессу разрешается продолжить выполнение, а на базе клона генерируется более объемный дамп, охватывающий всю доступную процессу память пользовательского режима.

В процессе отражения используется ряд компонентов инфраструктуры диагностики Windows (Windows Diagnostic Infrastructure, WDI), с помощью которых осуществляется захват минимальных дампов памяти для их эвристической идентификации на причастность к подозрительному поведению. Например, компонент диагностики утечек памяти (Memory Leak Diagnoser) из состава имеющихся в Windows средств выявления и разрешения проблем истощения ресурсов (Resource Exhaustion Detection and Resolution, RADAR) генерирует отраженный дамп памяти процесса, уличенного в создании утечки закрытой виртуальной памяти. Полученный дамп готов к отправке в компанию Microsoft через систему отчетов об ошибках (Windows Error Reporting, WER) для последующего анализа. Имеющаяся в WDI эвристическая система обнаружения зависших процессов делает то же самое для процессов, которые взаимно блокируют друг друга. Однако поскольку работа этих компонентов носит эвристический характер, они не могут точно выявлять сбойные процессы, чтобы на длительный период времени приостанавливать их выполнение или вообще останавливать.

Для реализации механизма отражения процессов предназначена функция `RtlCreateProcessReflection` из файла Ntdll.dll. Она работает следующим образом:

1. Сначала функция создает раздел общей памяти.
2. Раздел общей памяти заполняется параметрами.

3. Раздел общей памяти отображается на текущий и целевой процессы.
4. Создаются два объекта событий, которые дублируются в целевой процесс, чтобы текущий и целевой процессы могли синхронизировать свои операции.
5. После этого в целевой процесс вызовом функции `RtlpCreateUserThreadEx` внедряется программный поток. Этот поток предназначен для вызова функции `RtlpProcessReflectionStartup` из файла Ntdll.dll. Поскольку внутри адресного пространства каждого процесса файл Ntdll.dll отображен на один и тот же адрес, произвольно сгенерированный при начальной загрузке системы, текущий процесс может просто передать адрес функции, полученный из его собственного варианта отображения файла Ntdll.dll.
6. Если код, вызвавший `RtlCreateProcessReflection`, указал, что ему нужен дескриптор клонированного процесса, функция `RtlCreateProcessReflection` ждет, пока завершится работа удаленного потока; в противном случае она возвращает управление вызвавшему ее коду.
7. Программный поток, внедренный в целевой процесс, назначает дополнительный объект события, предназначенный для синхронизации с клонированным процессом, как только тот будет создан.
8. Затем он вызывает функцию `RtlCloneUserProcess`, передавая ей параметры, полученные из отображения общей с исходным процессом памяти.
9. При наличии параметра `RtlCreateProcessReflection`, определяющего создание клона, если процесс не выполняется в загрузчике, не работает с кучами, не изменяет блок окружения процесса (Process Environment Block, PEB) и не изменяет локальное хранилище волокна (fiber-local storage), функция `RtlCreateProcessReflection` перед продолжением работы получает соответствующие блокировки. Это может пригодиться для отладки, поскольку копия дампа памяти со структурой данных останется в согласованном состоянии.
10. Выполнение функции `RtlCloneUserProcess` завершается вызовом функции пользовательского режима `RtlpCreateUserProcess`, отвечающей за создание общего процесса и передачу флагов, показывающих, что новый процесс должен быть клоном текущего процесса. Функция `RtlpCreateUserProcess`, в свою очередь, вызывает функцию `ZwCreateUserProcess`, чтобы отправить ядру запрос на создание процесса.

При создании клонированного процесса функция `ZwCreateUserProcess` выполняет код почти по тем же маршрутам, что и при создании нового процесса, только вместо функции `PspAllocateProcess`, предназначенной для создания объекта процесса и инициализации программного потока, она вызывает функцию `MmInitializeProcessAddressSpace` с флагом, указывающим на то, что адресом должна быть копия (полученная копированием при записи) целевого процесса, а не исходное адресное пространство процесса. Для фактического клонирования адресного пространства диспетчер памяти использует ту же поддержку, которую он предоставляет API-функции *fork* сервисных функций для UNIX-приложений. Как

только целевой процесс продолжит выполнение, любые изменения, вносимые им в адресное пространство, будут видны только ему и не видны клону, что позволяет адресному пространству клона оставаться согласованным, соответствуя моменту представления целевого процесса.

Выполнение клона начинается с той точки, которая находится сразу за вызовом функции `RtlpCreateUserProcess`. Если создание клона пройдет успешно, его поток получит код возврата `STATUS_PROCESS_CLONED`, в то время как клонирующий поток получит код возврата `STATUS_SUCCESS`. Затем клонированный процесс синхронизируется с целевым процессом и в качестве заключительного действия вызывает функцию, дополнительно переданную функции `RtlCreateProcessReflection`, которая должна быть реализована в файле Ntdll.dll. К примеру, RADAR определяет функцию `RtlDetectHeapLeaks`, выполняющую эвристический анализ куч процесса и отправляющую отчет о результатах обратно потоку, который вызвал функцию `RtlCreateProcessReflection`. Если функция не указана, поток приостанавливается или прекращает свою работу в зависимости от флага, переданного функции `RtlCreateProcessReflection`.

Когда RADAR и WDI используют механизм отражения процессов, они вызывают функцию `RtlCreateProcessReflection` с запросом к ней вернуть им дескриптор клонированного процесса и с запросом к клону приостановиться после инициализации. Затем они генерируют мини-дамп целевого процесса, что приостанавливает этот процесс на время генерации дампа, а потом генерируют более сложный дамп клонированного процесса. После завершения генерации дампа клона они прекращают выполнение клона. Целевой процесс может выполняться во временном окне между завершением создания мини-дампа и созданием клона, но в большинстве случаев какие-либо несоответствия не оказывают влияния на поиск и устранение неисправностей. По этой же схеме действует и утилита Procdump из пакета Sysinternals при задании ключа `-r`, который заставляет ее создать дамп целевого процесса.

Заключение

Эта глава посвящена реализации управления виртуальной памятью Windows в диспетчере памяти. Как и в большинстве других современных операционных систем, каждому процессу предоставляется доступ к закрытому адресному пространству, что позволяет защитить память одного процесса от воздействия других процессов, но при этом процессы могут эффективно и безопасно работать с общей памятью. Описываются также и такие нетривиальные возможности диспетчера памяти, как включение отображаемых файлов и выделение разреженной памяти. Подсистема среды Windows открывает доступ приложениям к большинству возможностей диспетчера памяти через Windows API.

В следующей главе рассматривается важнейший компонент любой операционной системы — система ввода/вывода.

Глава 6
Подсистема ввода/вывода

Подсистема ввода/вывода в Windows состоит из набора компонентов исполнительной системы, которые совместно управляют устройствами и предоставляют приложениям и системе интерфейсы к этим устройствам. В этой главе мы, прежде всего, поговорим о том, какие цели, ставившиеся при проектировании подсистемы ввода/вывода, оказали влияние на ее реализацию. Затем мы перейдем к рассмотрению ее компонентов, в том числе диспетчера ввода/вывода, диспетчера устройств, поддерживающих технологию Plug and Play (PnP), и диспетчера электропитания. Будут проанализированы структура и компоненты подсистемы ввода/вывода и различные типы драйверов устройств. Вы познакомитесь с основными структурами данных, которые описывают устройства, драйверы устройств и запросы на ввод и вывод. Затем мы рассмотрим этапы обработки этих запросов. Завершают главу сведения о механизмах распознавания устройств, установки драйверов и управления электропитанием.

Компоненты подсистемы ввода/вывода

Подсистема ввода/вывода в Windows проектировалась как абстрактный интерфейс приложений для аппаратных (физических) и программных (виртуальных, или логических) устройств, обладающий определенными функциональными возможностями:

- Унифицированные средства безопасности и именования устройств для защиты общих ресурсов (модель безопасности Windows описывается в главе 7).
- Высокопроизводительный асинхронный пакетный ввод/вывод служит для поддержки масштабируемых приложений.
- Специальные службы позволяют писать драйверы устройств на высокоуровневом языке и упрощают их перенос на машины с другой архитектурой.
- Многоуровневая модель и расширяемость обеспечивают возможность добавлять драйверы, меняющие поведение других драйверов или устройств без необходимости модификации последних.
- Динамические загрузка и выгрузка драйверов устройств позволяют выполнять данные процедуры по запросу, экономя системные ресурсы.

- Поддержка технологии Plug and Play обеспечивает обнаружение и установку драйверов для нового оборудования и выделение им нужных аппаратных ресурсов, давая приложениям возможность находить и задействовать интерфейсы устройств.
- Подсистема управления электропитанием позволяет системе или отдельным устройствам переходить в состояния с низким энергопотреблением.
- Поддерживается установка различных файловых систем, в том числе FAT (и ее разновидностей FAT32 и exFAT), CDFS, UDF, ReFS (Resilient File System) и основной файловой системы Windows — NTFS. (Типы и архитектура файловых систем подробно рассматриваются в главе 13 части 2.)
- Благодаря поддержке технологии WMI (Windows Management Instrumentation — инструментарий управления Windows) и средствам диагностики управление драйверами и текущий контроль осуществляются при помощи WMI-приложений и WMI-сценариев (подробно технология WMI рассматривается в главе 9 части 2).

Для реализации этой функциональности в подсистеме ввода/вывода Windows предусмотрен ряд компонентов исполнительной подсистемы и драйверов устройств (рис. 6.1).

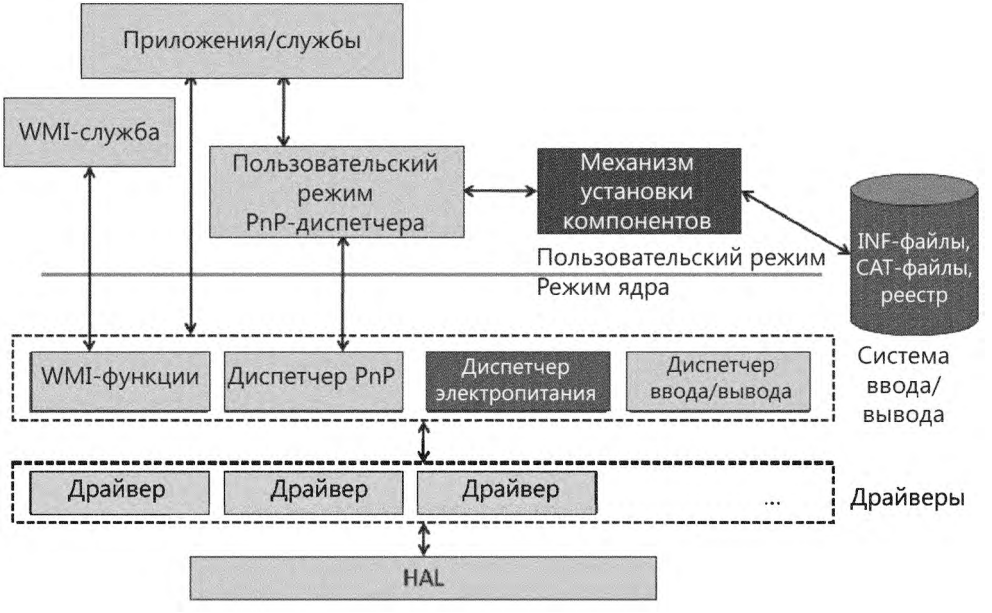


Рис. 6.1. Компоненты подсистемы ввода/вывода

- Центральное место в подсистеме ввода/вывода занимает диспетчер ввода/вывода; он соединяет приложения и системные компоненты с виртуальными,

логическими и физическими устройствами, создает поддерживающую драйверы устройств инфраструктуру.

- *Драйвер* устройства, как правило, предоставляет интерфейс ввода/вывода к конкретному типу устройства. Он представляет собой программный модуль, интерпретирующий высокоуровневые команды (такие, как команды чтения или записи) и выполняющий низкоуровневые команды, связанные с устройством, например запись в регистр управления. Драйверы устройств принимают от диспетчера ввода/вывода команды, предназначенные для управляемых ими устройств, и уведомляют диспетчер о выполнении этих команд. Данный диспетчер часто используется драйверами устройств для пересылки команд ввода/вывода другим драйверам, задействованным в реализации интерфейса того же устройства и участвующим в управлении им.

- PnP-диспетчер работает совместно с диспетчером ввода/вывода и такой разновидностью драйверов устройств, как *драйвер шины*. Он управляет выделением аппаратных ресурсов, а также распознает устройства и реагирует на их подключение или отключение. Именно PnP-диспетчер и драйверы шин обеспечивают загрузку соответствующего драйвера при обнаружении нового устройства. Если нужный драйвер устройства отсутствует, компоненты исполнительной системы, отвечающие за поддержку технологии PnP, вызывают сервисные функции установки устройств PnP-диспетчера в пользовательском режиме.

- Диспетчер электропитания также тесно связан с диспетчером ввода/вывода и PnP-диспетчером. Он управляет переходами в различные состояния энергопотребления как самой системы, так и отдельных драйверов устройств.

- Процедуры поддержки инструментария управления Windows (WMI), называемые WMI-провайдером модели драйверов в Windows (Windows Driver Model, WDM), позволяют драйверам устройств выступать в роли провайдеров, взаимодействуя с WMI-службой в пользовательском режиме через провайдера WDM WMI.

- *Реестр* представляет собой базу данных с описанием основных подключенных к подсистеме устройств, а также параметров инициализации драйверов и конфигурации (подробно они рассматриваются в главе 9 части 2).

- INF-файлы, которые можно узнать по расширению .inf, управляют установкой драйверов. Они связывают аппаратные устройства с драйверами, управляющими этими устройствами. Содержимое такого файла состоит из инструкций (напоминающих инструкции языков сценариев), которые описывают собственно устройство, исходное и целевое положение файлов драйвера, вносимые в реестр при установке драйвера изменения и сведения о зависимостях драйвера. Удостоверяющие файлы драйверов цифровые подписи, проверенные лабораторией WHQL (Microsoft Windows Hardware Quality Lab), хранятся в файлах с расширением .cat. Цифровые подписи также применяются для предотвращения взлома драйвера или его INF-файла.

- Уровень аппаратных абстракций (Hardware Abstraction Layer, HAL) изолирует драйверы от специфических особенностей конкретных процессоров и контроллеров прерываний, поддерживая прикладные программные интерфейсы, скрывающие межплатформенные различия. В сущности, HAL является драйвером шины для устройств на материнской плате компьютера, которые не управляются другими драйверами.

Диспетчер ввода/вывода

Центральным элементом подсистемы ввода/вывода является диспетчер ввода/вывода (I/O manager), задающий инфраструктуру (или модель) для доставки драйверам устройств запросов на ввод и вывод. Данная подсистема имеет пакетное управление. Большинство запросов представлены именно пакетами запросов на ввод/вывод (I/O Request Packets, IRP), передаваемыми от одного компонента системы к другому. (В разделе «Быстрый ввод/вывод» вы познакомитесь с исключением из этого правила, когда IRP-пакеты не используются.) Подобное проектное решение позволяет отдельному программному потоку приложения одновременно управлять целым набором запросов на ввод и вывод. Такая структура данных, как IRP-пакет, содержит информацию, полностью описывающую запрос на ввод и вывод (подробно эта тема рассматривается в разделе «Пакеты запросов ввода/вывода» далее в этой главе).

Диспетчер ввода/вывода представляет операции ввода и ввода в памяти в виде IRP-пакетов. Указатель на IRP передается нужному драйверу, а после завершения операции пакет удаляется. Драйвер, получивший IRP, выполняет указанную в пакете операцию и возвращает пакет диспетчеру ввода/вывода, либо сигнализируя о завершении операции, либо с целью передачи пакета другому драйверу для дальнейшей обработки.

В дополнение к созданию и уничтожению IRP-пакетов диспетчер ввода/вывода предоставляет различным драйверам общий код, который они используют при обработке ввода/вывода. Подобное объединение задач упрощает драйверы и делает их более компактными. В частности, диспетчер ввода/вывода предоставляет функцию, позволяющую драйверу вызывать другие драйверы. Также он управляет буферами запросов на ввод и вывод, обеспечивает поддержку тайм-аута для драйверов и регистрирует загруженные в операционную систему устанавливаемые файловые системы. Диспетчер ввода/вывода содержит почти сотню функций, которые могут вызываться драйверами устройств.

Также диспетчер ввода/вывода предоставляет гибкие сервисные функции ввода/вывода, позволяющие подсистемам окружения — например, Windows и POSIX (вторая в настоящее время не поддерживается) — реализовывать соответствующие функции ввода/вывода. В частности, сюда относится поддержка асинхронного ввода/вывода, которая предоставляет разработчикам возможность создавать высокопроизводительные масштабируемые серверные приложения.

Унифицированный модульный интерфейс драйверов позволяет диспетчеру ввода/вывода вызывать их даже при отсутствии сведений об их структуре и внутреннем

устройстве. Операционная система обрабатывает запросы на ввод и вывод так, будто они адресованы файлам; запрос к виртуальному файлу преобразуется драйвером в запрос к конкретному устройству. Драйверы могут вызывать друг друга (через диспетчер ввода/вывода), обеспечивая многоуровневую независимую обработку запроса на ввод или вывод.

Кроме обычных функций открытия, закрытия, чтения и записи подсистема ввода/вывода в Windows предоставляет ряд расширенных возможностей — например, асинхронного, прямого, буферизованного и фрагментированного ввода/вывода. Эти механизмы рассматриваются в разделе «Типы ввода/вывода» далее в этой главе.

Стандартная обработка ввода/вывода

Большая часть операций ввода и вывода не требует участия всех компонентов подсистемы ввода/вывода. Как правило, запрос на ввод или вывод поступает от приложения, выполняющего соответствующую операцию (например, чтение данных с устройства); такие операции обрабатываются диспетчером ввода/вывода, одним или несколькими драйверами устройств и HAL.

Как уже упоминалось, в операционной системе Windows программные потоки выполняют операции ввода/вывода с виртуальными файлами. Термин «виртуальный файл» относится к любому источнику или приемнику запроса на ввод/вывод, который рассматривается как файл (это может быть устройство, файл, папка, канал или почтовая ячейка). Типичный клиент пользовательского режима вызывает функции `CreateFile` или `CreateFile2` для получения дескриптора виртуального файла. Имя функции выбрано не идеально — она работает не только с файлами, но и вообще с любой *символической ссылкой* в каталоге диспетчера объектов с именем GLOBAL??. Суффикс «File» в именах `CreateFile*` в действительности обозначает объект виртуального файла (`FILE_OBJECT`) — сущность, создаваемую исполнительной подсистемой в результате вызова этих функций. На рис. 6.2 изображен снимок экрана программы WinObj из пакета Sysinternals для каталога GLOBAL??.

Как видно из рис. 6.2, имя — такое, как C:, — в действительности является символической ссылкой на внутреннее имя в каталоге Device диспетчера объектов (в данном случае \Device\HarddiskVolume7). (За дополнительной информацией о диспетчере объектов и пространстве имен диспетчера объектов обращайтесь к главе 8 части 2.) Все имена в каталоге GLOBAL?? могут использоваться в аргументах `CreateFile(2)`. Клиенты режима ядра — например, драйверы устройств — могут воспользоваться аналогичной функцией `ZwCreateFile` для получения дескриптора виртуального файла.

ПРИМЕЧАНИЕ Высокоуровневые абстракции — такие, как .NET Framework и Windows Runtime, — используют собственные API для работы с файлами и устройствами (например, класс `System.IO.File` в .NET или класс `Windows.Storage.StorageFile` в WinRT). Тем не менее все эти API в конечном счете вызывают функцию `CreateFile(2)` для получения дескриптора, который они используют в своей внутренней реализации.

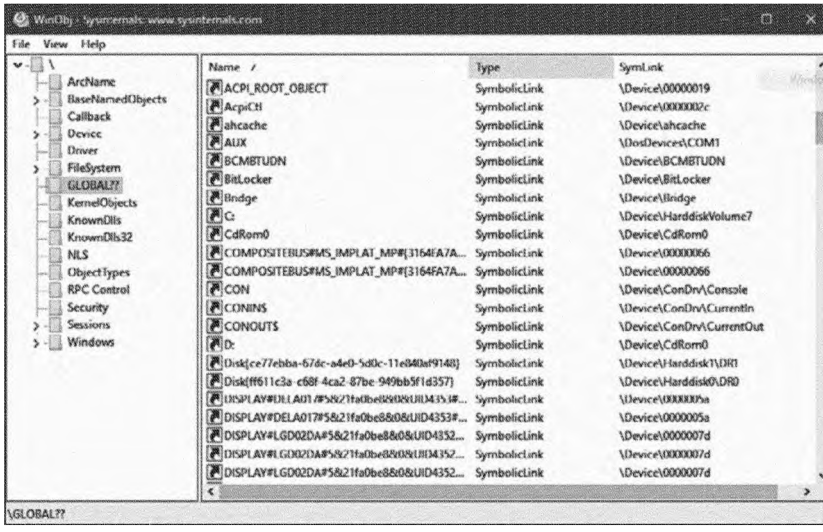


Рис. 6.2. Каталог GLOBAL?? диспетчера объектов

ПРИМЕЧАНИЕ Каталог диспетчера объектов GLOBAL?? иногда обозначается более старым именем DosDevices. Имя DosDevices работает, потому что оно определяется как символическая ссылка на *GLOBAL??* в корне пространства имен диспетчера объектов. В коде драйверов для ссылок на каталог GLOBAL?? обычно используется строка ??.

Операционная система абстрагирует все запросы ввода/вывода как операции с виртуальными файлами, так как диспетчер ввода/вывода ни с чем другим работать не умеет. При этом за преобразование файловых команд (открытие, закрытие, чтение, запись) в команды для конкретного устройства отвечает драйвер. Подобная абстракция обеспечивает единый программный интерфейс для всех устройств. Приложения пользовательского режима вызывают документированные функции, которые, в свою очередь, обращаются к внутренним функциям подсистемы ввода/вывода для чтения из файла, записи в файл и прочих операций. Диспетчер ввода/вывода динамически направляет эти адресованные виртуальным файлам запросы к соответствующим драйверам устройств. Рисунок 6.3 демонстрирует базовую схему обработки запроса на ввод/вывод. (Другие типы запросов ввода/вывода — например, запросы на запись — обрабатываются аналогично; они просто используют другие API.)

Далее мы рассмотрим эти компоненты более подробно, поговорим о различных типах драйверов, их структуре, загрузке, инициализации, а также способах обработки ими запросов на ввод/вывод. Затем вы познакомитесь с функциями и ролью PnP-диспетчера и диспетчера электропитания.

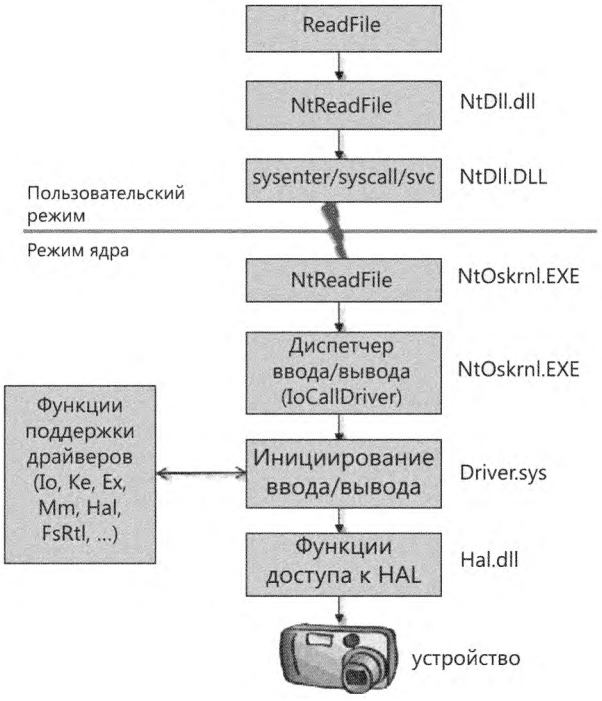


Рис. 6.3. Последовательность обработки типичного запроса ввода/вывода

IRQL и отложенные вызовы процедур

Прежде чем двигаться дальше, необходимо рассмотреть две очень важные концепции ядра Windows, которые играют важную роль в системе ввода/вывода: *уровни запросов прерываний*, или IRQL (Interrupt Request Level), и *отложенные вызовы процедур*, или *DPC* (Deferred Procedure Calls). Подробное изложение этих концепций откладывается до главы 8, но в этом разделе мы предоставим достаточно информации, чтобы вы понимали механику обработки ввода/вывода, изложенную ниже.

IRQL

Термин «IRQL» имеет два разных значения, которые сходятся в некоторых ситуациях.

- **IRQL — приоритет, назначаемый источнику прерываний от физического устройства.** Число задается HAL (при содействии контроллера прерываний, к которому подключаются устройства, требующие обслуживания прерываний).

- **У каждого центрального процессора имеется собственный уровень IRQL.** Его следует рассматривать как регистр процессора (хотя в современных процессорах этот вариант реализации не используется).

Фундаментальное правило IRQL гласит, что код с более низким IRQL не может вмешиваться в работу кода с более высоким IRQL, и наоборот — код с более высоким IRQL не может вытеснять код, работающий с более низким IRQL. Вскоре мы рассмотрим примеры того, как это происходит на практике. Список IRQL для архитектур, поддерживаемых Windows, приведен на рис. 6.4. Не путайте IRQL с приоритетами программных потоков. Собственно, приоритеты потоков имеют смысл только при значениях IRQL меньших 2.

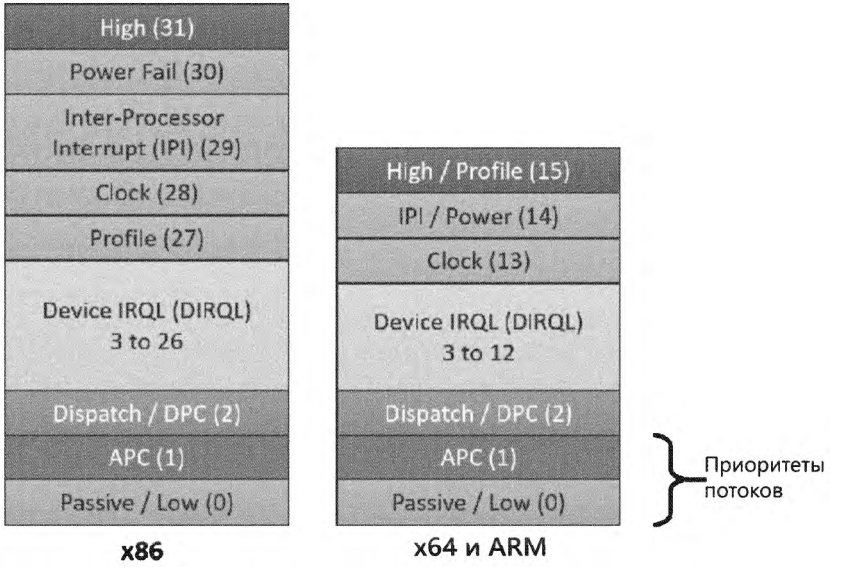


Рис. 6.4. Уровни IRQL

ПРИМЕЧАНИЕ Не путайте IRQL с IRQ (запросами прерываний, Interrupt Request). IRQ — линия, соединяющая устройства с контроллером прерываний. За дополнительной информацией о прерываниях, IRQ и IRQL обращайтесь к главе 8.

Обычно уровень IRQL процессора равен 0. Это означает, что в системе не происходит «ничего особенного», а планировщик ядра, который планирует потоки на основании приоритетов, работает так, как описано в главе 4. В пользовательском режиме значение IRQL может быть равно только 0. Повысить IRQL из пользовательского режима невозможно. (Поэтому в документации пользовательского режима концепция IRQL вообще не упоминается.)

Код режима ядра может повышать и понижать IRQL текущего процессора при помощи функций `KeRaiseIrql` и `KeLowerIrql`. Однако большинство функций, фик-

сированных по времени, вызываются с повышением IRQL до некоторого ожидаемого уровня; вскоре вы убедитесь в этом, когда мы будем рассматривать типичную обработку операций ввода/вывода драйвером.

Самые важные уровни IRQL в контексте ввода/вывода:

- **Passive (0).** Определяется макросом `PASSIVE_LEVEL` в заголовочном файле WDK wdm.h. Это нормальный уровень IRQL, при котором планировщик ядра работает нормально (см. подробное описание в главе 4).
- **Dispatch/DPC (2) (DISPATCH_LEVEL).** Уровень IRQL, на котором работает планировщик ядра. Это означает, что если поток поднимает текущий уровень IRQL до 2 (или выше), поток фактически получает бесконечный квант и не может быть вытеснен другим потоком. По сути, планировщик не может активизироваться на текущем процессоре, пока уровень IRQL не упадет ниже 2. Отсюда следует:
  - Пока IRQL находится на уровне 2 и выше, любое ожидание по объектам диспетчера ядра (мьютексам, семафорам и событиям) приведет к сбою системы. Дело в том, что ожидание подразумевает, что поток может войти в состояние ожидания и на том же процессоре может быть запланирован другой поток. Но поскольку планировщик не находится на нужном уровне, это не произойдет; вместо этого произойдет фатальная ошибка (единственное исключение — если тайм-аут ожидания равен нулю, т. е. ожидание не запрашивается).
  - Ошибки страниц обрабатываться не могут. Ошибка страницы требует переключения контекста на подсистему записи измененных страниц; однако переключения контекста запрещены, поэтому в системе произойдет сбой. Таким образом, код, выполняющийся на уровне IRQL 2 и выше, сможет обращаться только к невыгружаемой памяти — обычно памяти, выделенной из невыгружаемого пула, которая по определению всегда размещается в физической памяти.
- **Device IRQL (3–26 на x86; 3–12 на x64 и ARM) (DIRQL).** Эти уровни закрепляются за аппаратными преобразованиями. При поступлении прерывания диспетчер вызывает соответствующую функцию обслуживания прерывания (ISR, Interrupt Service Routine) и повышает ее уровень IRQL до уровня соответствующего прерывания. Поскольку это значение всегда выше `DISPATCH_LEVEL` (2), все правила, связанные с IRQL 2, применимы и к DIRQL.

Выполнение на конкретном уровне IRQL маскирует прерывания с этим и более низкими уровнями IRQL. Например, ISR с IRQL 8 заблокирует вмешательство (на этом процессоре) с IRQL 7 и ниже. А точнее, никакой код пользовательского режима не сможет выполняться, потому что он всегда выполняется с IRQL 0. Отсюда следует, что выполнение на более высоких уровнях IRQL в общем случае нежелательно; впрочем, есть несколько конкретных ситуаций (они будут рассмотрены в этой главе), в которых это имеет смысл и, более того, необходимо для нормального функционирования системы.

Отложенные вызовы процедур

Отложенный вызов процедуры, или *DPC* (Deferred Procedure Call), – объект, инкапсулирующий вызов функции на уровне IRQL DPC_LEVEL (2). Объекты DPC существуют прежде всего для выполнения действий после прерывания, так как выполнение на уровне DIRQL маскирует (а следовательно, задерживает) другие прерывания, ожидающие обработки. Типичная ISR-функция выполняет минимум возможной работы – обычно она читает состояния устройства, приказывает ему остановить сигнал прерывания, а затем передает дальнейшую обработку на более низкий уровень IRQL (2) запросом DPC. Термин «*отложенный*» в названии означает, что DPC не выполняется немедленно – да и не может, потому что текущий уровень IRQL выше 2. Но когда ISR вернет управление, при отсутствии ожидающих обработки прерываний уровень IRQL процессора падает до 2, и он выполняет накопившиеся вызовы DPC (возможно, только этот один). На рис. 6.5 представлен упрощенный пример последовательности событий, происходящих при прерывании от аппаратных устройств (которые асинхронны по своей природе, т. е. могут поступить в любой момент) при нормальном выполнении кода на уровне IRQL 0 на некоторых процессорах.

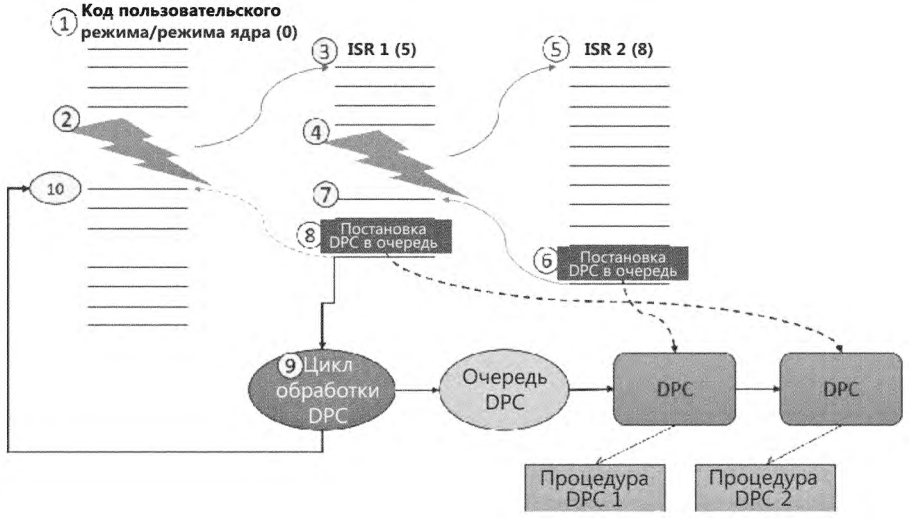


Рис. 6.5. Пример обработки прерываний и DPC

Краткая сводка последовательности событий на рис. 6.5.

1. Некий код пользовательского режима или режима ядра выполняется тогда, когда процессор находится на уровне 0 (на этом уровне проходит большая часть времени).
2. Поступает аппаратное прерывание на уровне IRQL 5 (напомним, что минимальное значение Device IRQL равно 3). Так как 5 больше 0 (текущий уровень

IRQL), состояние процессора сохраняется, IRQL повышает до 5, и вызывается обработчик ISR, связанный с прерыванием. Обратите внимание: переключение контекста при этом не происходит; работает тот же поток, который теперь выполняет код ISR. (Если поток находился в пользовательском режиме, он переключается в режим ядра при поступлении прерывания.)

3. ISR 1 начинает выполняться, когда процессор работает на уровне IRQL 5. На этот момент любые прерывания с IRQL 5 и ниже вмешиваться в обработку не могут.

4. Предположим, поступает новое прерывание с IRQL 8. Предположим, система решает, что оно должно быть обработано тем же процессором. Так как 8>5, выполнение снова прерывается, состояние процессора сохраняется, IRQL повышается до 8, и процессор переходит к ISR 2. Заметьте также, что выполнение происходит в том же потоке. Никакое переключение контекста при этом невозможно, потому что планировщик потоков не может активизироваться при IRQL уровня 2 и выше.

5. Выполняется обработчик ISR 2. До его завершения ISR 2 хотелось бы выполнить дополнительную обработку на более низком уровне IRQL, чтобы прерывания с IRQL менее 8 тоже могли быть обработаны.

6. Напоследок ISR 2 вставляет правильно инициализированный объект DPC со ссылкой на функцию драйвера, которая выполняет всю последующую обработку после закрытия прерывания, вызовом функции `KeInsertQueueDpc`. (Типичное содержание этой последующей обработки рассматривается в следующем разделе.) Затем ISR возвращает управление, и восстанавливается состояние процессора, сохраненное перед входом в ISR 2.

7. На этой стадии IRQL падает до предыдущего уровня (5), и процессор продолжает выполнение обработчика ISR 1, прерванного ранее.

8. Непосредственно перед завершением ISR 1 ставит в очередь собственный объект DPC для выполнения своей последующей обработки. Объекты DPC ставятся в очередь DPC, которую мы еще не рассматривали. ISR 1 возвращает управление, восстанавливая состояние процессора, сохраненное перед началом выполнения ISR 1.

9. В этот момент уровень IRQL должен был бы упасть до старого нулевого значения, чтобы начать обработку прерываний. Но ядро замечает наличие необработанных DPC, поэтому IRQL уменьшается до уровня 2 (`DPC_LEVEL`), и запускается цикл обработки DPC, который перебирает накопленные DPC и последовательно вызывает каждую процедуру DPC. Когда очередь DPC опустеет, обработка DPC завершается.

10. Наконец, IRQL уменьшается до 0, состояние процессора снова восстанавливается, и возобновляется выполнение изначально прерванного исходного кода пользовательского режима или режима ядра. И снова следует напомнить, что вся описанная обработка происходит в одном потоке (каким бы он ни был).

Этот факт подразумевает, что ISR и процедуры DPC не должны зависеть от конкретного потока (а следовательно, части конкретного процесса) для выполнения своего кода. Поток может быть любым; важность этого факта будет рассмотрена в следующем разделе.

Приведенное описание немного упрощено. В нем не упоминается важность DPC, возможность обработки DPC другими процессорами для ускорения и т. д. Эти подробности не важны для темы, рассматриваемой в этом разделе. Тем не менее они полностью описаны в главе 8.

Драйверы устройств

Для интеграции с диспетчером ввода/вывода и прочими компонентами одноименной подсистемы драйвер устройства должен быть написан в соответствии с правилами, установленными для указанного типа драйверов устройств и выполняемых этим драйвером операций. В этом разделе рассматриваются типы поддерживаемых в Windows драйверов устройств и их внутренняя структура.

ПРИМЕЧАНИЕ Большинство драйверов устройств написано на языке C. Начиная с Windows Driver Kit 8.0, появилась возможность безопасного написания драйверов на C++ из-за специальной поддержки C++ в режиме ядра новыми компиляторами C++. Использовать язык ассемблера не рекомендуется из-за появляющихся сложностей с портированием драйвера между аппаратными архитектурами, поддерживаемыми Windows (x86, x64 и ARM).

Типы драйверов устройств

В операционной системе Windows поддерживается широкий спектр типов драйверов устройств и сред разработки. Последние могут различаться даже для драйверов устройств одного типа. Это зависит от особенностей устройств, для которых предназначен драйвер.

Кроме того, драйверы делятся на работающие в пользовательском режиме и в режиме ядра. В Windows поддерживаются два типа драйверов пользовательского режима.

- **Драйверы принтеров подсистемы Windows** переводят аппаратно-независимые запросы на графические операции в понятные принтеру команды, которые затем передаются драйверу порта в режиме ядра. Например, драйвер порта принтера универсальной последовательной шины (USB) называется Usbprint.sys.
- **Драйверы, являющиеся компонентами среды UMDF** (User-Mode Driver Framework), как понятно из их названия, работают с оборудованием в режиме пользователя. С библиотекой поддержки UMDF в режиме ядра они общаются через механизм ALPC. Более подробно мы поговорим об этом в разделе «Среда UMDF».

В этой главе в основном рассматриваются драйверы, работающие в режиме ядра. Их можно разбить на следующие категории.

- **Драйверы файловой системы** (File system drivers) принимают запросы к файлам на ввод/вывод и на их основе выдают более конкретные запросы к драйверам запоминающих или сетевых устройств.
- **Драйверы PnP** (Plug and Play drivers) работают с оборудованием и объединяются с диспетчером электропитания и PnP-диспетчером. В эту категорию входят драйверы запоминающих устройств, видеоадаптеров, устройств ввода и сетевых адаптеров.
- **Драйверы без поддержки Plug and Play** (Non-Plug and Play drivers) включают в себя также расширения ядра и делают систему более функциональной. Как правило, они не интегрированы с PnP-диспетчером или с диспетчером электропитания, так как не связаны с физическими аппаратными устройствами. К этой категории относятся драйверы протоколов и сетевого API. У программы Process Monitor из пакета Sysinternals также имеется драйвер — пример драйвера без поддержки PnP.

Драйверы режима ядра подразделяются на группы в зависимости от модели и роли в обслуживании запросов к устройствам.

WDM-драйверы

WDM-драйверы являются драйверами устройств, соответствующими модели *WDM* (Windows Driver Model). WDM поддерживает управление электропитанием, технологию Plug and Play и инструментарий управления Windows. Большинство драйверов PnP-устройств соответствует модели WDM. Драйверы данной категории делятся на три типа.

- **Драйверы шины** (bus drivers) управляют логической или физической шиной. Это могут быть шины PCMCIA, PCI, USB и IEEE 1394. Драйвер шины отвечает за распознавание подключенных к шине устройств и оповещение о них PnP-диспетчера, а также за управление электропитанием шины. Обычно эти драйверы предоставляются компанией Microsoft в поставке системы.
- **Функциональные драйверы** (function drivers) управляют устройствами конкретного типа. Драйверы шины предоставляют устройства функциональным драйверам через PnP-диспетчер. Функциональным называется драйвер, экспортирующий в операционную систему рабочий интерфейс устройства. В общем случае именно он лучше всего осведомлен о работе устройства.
- **Фильтрующие драйверы** (filter drivers) могут располагаться как выше, так и ниже *функционального* и *шинного* драйверов. Они дополняют или меняют поведение устройства или другого драйвера. Например, служебная программа для перехвата ввода с клавиатуры может быть реализована на базе фильтрующего драйвера клавиатуры, работающего поверх функционального драйвера клавиатуры.

На рис. 6.6 изображен *узел устройства* с драйвером шины, который создает объект физического устройства (PDO), низкоуровневые фильтры, функциональный драйвер, создающий объект функционального устройства (FDO, Functional Device

Object), и высокоуровневые фильтры. Обязательными являются только уровни PDO и FDO; различные фильтры могут существовать, но их может и не быть.

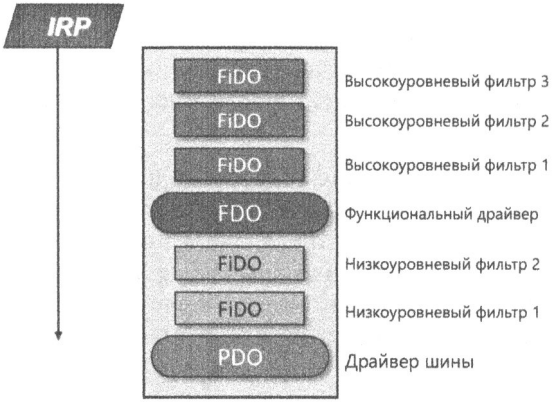


Рис. 6.6. Узел устройства WDM

Ни один WDM-драйвер не отвечает полностью за все аспекты управления устройством. Драйвер шины занимается отслеживанием состава устройств на шине (путем подключения или отключения), помогая PnP-диспетчеру регистрировать эти устройства, обращаясь к относящимся к шине конфигурационным регистрам и в некоторых случаях управляя электропитанием подключенных устройств. К аппаратной части устройства обычно обращается только функциональный драйвер.

Многоуровневые драйверы

Поддержка отдельного устройства часто реализуется целым набором драйверов, каждый из которых предоставляет часть функциональности, необходимой для корректной работы. Кроме WDM-драйверов шины, функциональных и фильтрующих драйверов, поддержка аппаратного обеспечения может обеспечиваться еще и другими компонентами.

- **Драйверы классов** (class drivers) устройств отвечают за обработку ввода/вывода для устройств конкретного класса, таких как жесткий диск, клавиатура или компакт-диск, со стандартизированными аппаратными интерфейсами, позволяющими одному драйверу обслуживать устройства от различных производителей.
- **Драйверы мини-классов** (miniclass drivers) реализуют обработку ввода/вывода, заданную производителем для определенного класса устройств. К примеру, несмотря на наличие стандартного драйвера класса элементов питания от Microsoft, интерфейсы источников бесперебойного питания (Uninterruptible Power Supplies, UPS) и элементов питания портативных компьютеров у различных производителей различаются настолько, что не обойтись без мини-класса. Принадлежащие к данной категории драйверы по сути представляют собой динамически подключаемые библиотеки (DLL) режима ядра и не умеют

напрямую обрабатывать IRP-пакеты — они приводятся в действие драйвером класса и именно оттуда импортируют нужные функции.

- **Драйверы портов** (port drivers) обрабатывают запросы на ввод и вывод в соответствии с типом порта ввода/вывода, например SATA. Они реализуются как библиотеки функций режима ядра, а не как драйверы устройств. Практически все они написаны в Microsoft, ведь, как правило, интерфейсы стандартизированы таким образом, что различные поставщики могут пользоваться одним и тем же драйвером порта. Но иногда возникает необходимость написать собственную версию такого драйвера для специализированного аппаратного обеспечения. В некоторых случаях в понятие «порт ввода/вывода» включается еще и логический порт. К примеру, NDIS является сетевым драйвером «порта».
- **Драйверы мини-портов** (miniport drivers) преобразуют обобщенный запрос ввода/вывода о типе порта в запрос о типе адаптера. По сути, они являются истинными драйверами устройств и импортируют функции, предоставляемые драйвером порта. Драйверы мини-порта пишутся сторонними производителями и предоставляют интерфейс для драйвера порта. Как и драйверы мини-класса, они являются динамически подключаемыми библиотеками (DLL) режима ядра и не умеют напрямую обрабатывать IRP-пакеты.

На рис. 6.7 представлен простой пример, который показывает, как драйверы устройств работают во взаимодействии с иерархией уровней. Драйвер файловой

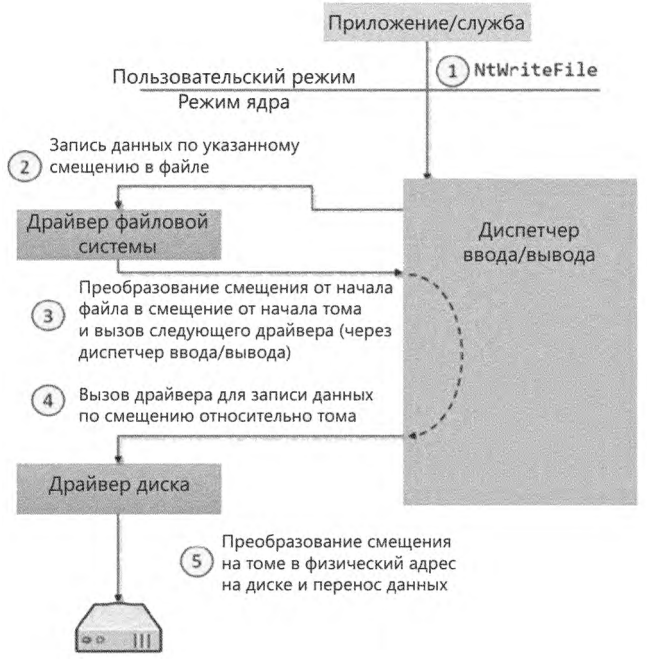


Рис. 6.7. Многоуровневое представление драйвера файловой системы и драйвера диска

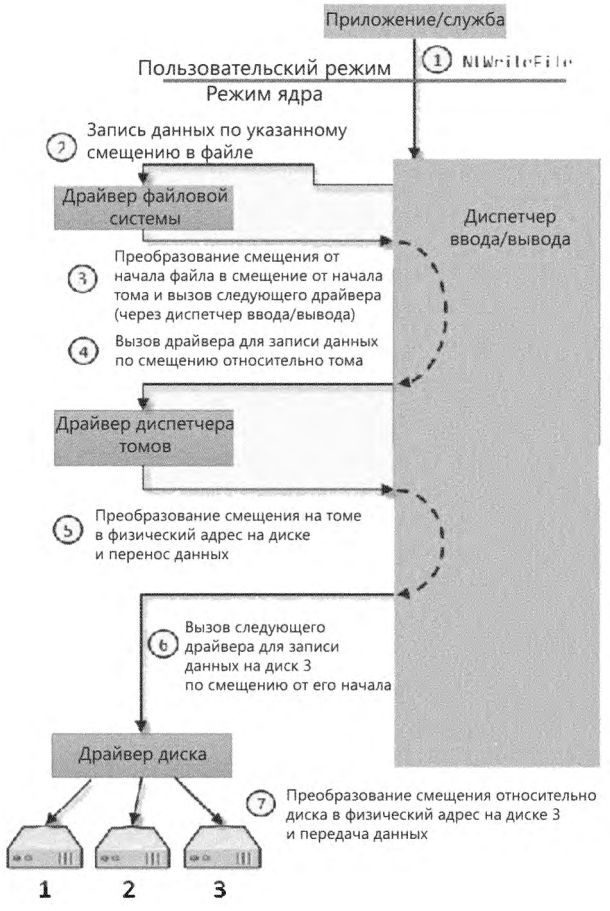


Рис. 6.8. Добавление промежуточного драйвера

системы принимает запрос на запись данных в определенное место конкретного файла. Он преобразуется в запрос на запись определенного количества байтов по определенному «логическому» адресу на диске. Затем данный запрос (через диспетчер ввода/вывода) передается простому драйверу диска. Последний преобразует его в физический адрес на диске и позиционирует головки дискового устройства для записи данных.

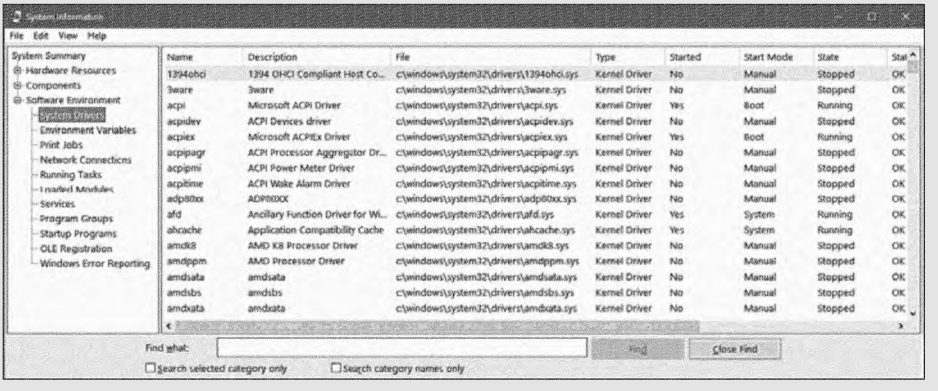
Рисунок иллюстрирует разделение обязанностей между драйверами. Диспетчер ввода/вывода получает запрос на запись относительно начала конкретного файла и передает его драйверу файловой системы, который преобразует содержащуюся в запросе информацию в адрес начала записи (границу сектора на диске) и количество записываемых байтов. Драйвер файловой системы передает запрос драйверу

диска через диспетчер ввода/вывода, а драйвер диска преобразует его в физический адрес на диске и осуществляет передачу данных.

Так как все драйверы — устройств и файловой системы — предоставляют операционной системе одну и ту же инфраструктуру, в иерархию можно добавить новый драйвер без изменения существующих драйверов или подсистемы ввода/вывода. Например, добавлением драйвера можно превратить несколько дисков в один большой диск. Такой драйвер диспетчера логических томов располагается между драйвером файловой системы и драйвером диска, как показано на диаграмме, представленной на рис. 6.8. (Диаграмма стека драйверов устройств, а также описание драйверов диспетчера томов приведен в главе 12 части 2.)

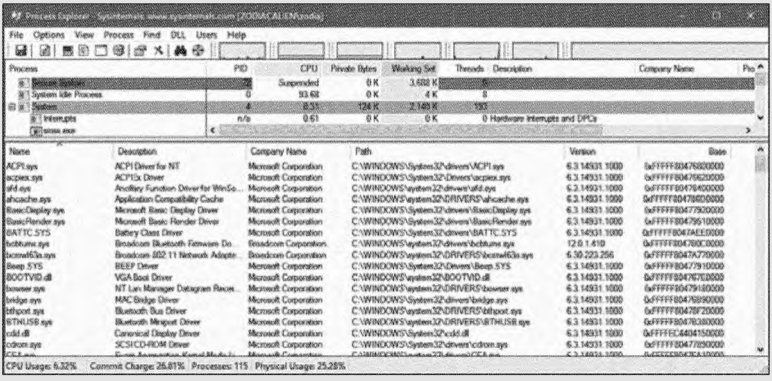
ЭКСПЕРИМЕНТ: ПРОСМОТР СПИСКА ЗАГРУЖЕННЫХ ДРАЙВЕРОВ

Для просмотра списка зарегистрированных в системе драйверов достаточно запустить служебную программу Msinfo32.exe через диалоговое окно Run (Запуск), доступное в меню Start (Пуск). Выберите в разделе Software Environment строку System Drivers — справа появится список всех драйверов системы. Загруженные драйверы отмечены словом Yes в столбце Started.



Список драйверов хранится в подразделах раздела реестра HKLM\System\CurrentControlSet\Services. Этот раздел используется как для драйверов, так и для служб; и те и другие могут запускаться диспетчером служб (SCM, Service Control Manager). Чтобы отличить драйвер от службы для каждого подраздела, проще всего обратиться к параметру Type. Малые значения (1, 2, 4, 8) соответствуют драйверам, а значения 16 (0x10) и 32 (0x20) соответствуют службам Windows. За дополнительной информацией о подразделе Services обращайтесь к главе 9.

Список загруженных драйверов для режима ядра можно просмотреть с помощью приложения Process Explorer. После его запуска выделите строку System в столбце Process и выберите в меню View команду Lower Pane View ▸ DLLs.



В приложении Process Explorer перечислены все загруженные драйверы, их имена, сведения о версии (включая название компании и описание), указан адрес загрузки (предполагается, что вы настроили режим вывода соответствующих столбцов в диалоговом окне Process Explorer).

Ну и наконец, при изучении аварийного дампа (или снимка работающей системы) при помощи отладчика ядра аналогичный список можно получить командой `lm kv`:

```
kd> lm kv
start    end        module name
80626000 80631000   kdcom      (deferred)
    Image path: kdcom.dll
Image name: kdcom.dll
    Browse all global symbols  functions  data
    Timestamp:        Sat Jul 16 04:27:27 2016 (57898D7F)
    CheckSum:         0000821A
    ImageSize:        0000B000
    Translations:     0000.04b0 0000.04e4 0409.04b0 0409.04e4
81009000 81632000   nt         (pdb symbols)          e:\symbols\ntkrpamp.
pdb\A54DF85668E54895982F873F58C984591\ntkrpamp.pdb
    Loaded symbol image file: ntkrpamp.exe
    Image path: ntkrpamp.exe
    Image name: ntkrpamp.exe
    Browse all global symbols  functions  data
    Timestamp:        Wed Sep 07 07:35:39 2016 (57CF991B)
    CheckSum:         005C6B08
    ImageSize:        00629000
    Translations:     0000.04b0 0000.04e4 0409.04b0 0409.04e4
81632000 81693000   hal        (deferred)
    Image path: halmacpi.dll
    Image name: halmacpi.dll
    Browse all global symbols  functions  data
    Timestamp:        Sat Jul 16 04:27:33 2016 (57898D85)
    CheckSum:         00061469
    ImageSize:        00061000
    Translations:     0000.04b0 0000.04e4 0409.04b0 0409.04e4
8a800000 8a84b000   FLTMGR     (deferred)
    Image path: \SystemRoot\System32\drivers\FLTMGR.SYS
```

```
    Image name: FLTMGR.SYS
    Browse all global symbols  functions  data
    Timestamp:        Sat Jul 16 04:27:37 2016 (57898D89)
    CheckSum:         00053B90
    ImageSize:        0004B000
    Translations:     0000.04b0 0000.04e4 0409.04b0 0409.04e4
...
```

Структура драйвера

Запуском драйверов устройств занимается подсистема ввода/вывода. Драйверы состоят из набора процедур, вызываемых для обработки различных этапов запроса на ввод или вывод. Основные процедуры драйвера показаны на рис. 6.9.

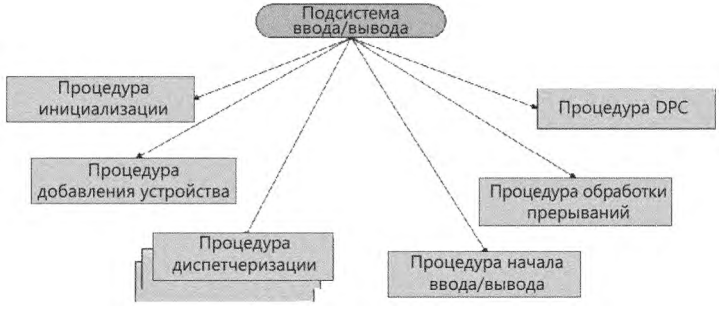


Рис. 6.9. Основные процедуры драйвера устройства

- **Процедура инициализации.** При загрузке драйвера в операционную систему диспетчер ввода/вывода выполняет процедуру инициализации, заданную для точки входа драйвера `GSDriverEntry` средствами WDK. Точка входа включает режим защиты компилятора от ошибок переполнения стека (называемых *cookie*), а затем вызывает процедуру `DriverEntry`, которую и должен реализовать разработчик драйвера. Именно она регистрирует остальные процедуры драйвера в диспетчере ввода/вывода и при необходимости выполняет всю глобальную инициализацию драйвера.
- **Процедура добавления устройства.** Драйверы PnP-устройств реализуют процедуру добавления устройства. PnP-диспетчер при обнаружении устройства, за которое отвечает конкретный драйвер, посылает этому драйверу уведомление. В рамках данной процедуры драйвер, как правило, создает объект, представляющий устройство (но об этом мы поговорим чуть позже).
- **Процедуры диспетчеризации.** Основные точки входа для драйвера устройства: открытие, закрытие, чтение, запись, операции PnP и т. д. Диспетчер ввода/вывода, вызванный для выполнения запроса на ввод или вывод, генерирует IRP-пакет и через одну из процедур диспетчеризации вызывает драйвер.

- **Процедура начала ввода/вывода.** Драйвер использует процедуру начала ввода/вывода, чтобы инициировать передачу данных на устройство или с него. Эта процедура определена только в драйверах, ставящих входящие запросы на ввод/вывод в очередь через диспетчер ввода/вывода. Последний сериализует IRP-пакеты для драйвера, убеждаясь, что драйвер обрабатывает за один раз только один пакет. Драйверы умеют обрабатывать несколько пакетов одновременно, но для устройств, которые не в состоянии обеспечить параллельную работу с набором запросов на ввод и вывод, требуется сериализация.

- **Процедура обработки прерываний.** Когда устройство приостанавливает свою работу, диспетчер прерываний ядра передает управление процедуре обработки прерываний (Interrupt Service Routine, ISR). В модели ввода/вывода Windows процедуры обработки прерываний работают на уровне запросов прерываний устройств (Device Interrupt Request Level, DIRQL), поэтому они выполняют минимум действий во избежание блокировки прерываний более низкого уровня (см. предыдущий раздел). Обычно ISR-процедура ставится в очередь отложенного вызова процедур (DPC) для выполнения на более низком IRQL-уровне (на уровне DPC/dispatch). (Процедуры обработки прерываний поддерживаются только на устройствах, управляемых прерываниями, например, в драйвере файловой системы они не поддерживаются.)

- **Процедура DPC.** Основную часть обработки прерывания, оставшейся после ISR-процедуры, выполняет процедура DPC. Она работает на уровне IRQL 2, что можно считать своего рода компромиссом между высоким уровнем DIRQL и низким уровнем Passive (0). Типичная DPC-процедура инициирует завершение одной операции ввода/вывода и начало следующей такой операции из очереди рассматриваемого устройства.

Многие драйверы устройств обладают дополнительными процедурами, которые не показаны на рис. 6.9.

- **Процедуры завершения ввода/вывода.** Многоуровневый драйвер может иметь одну или несколько процедур завершения ввода/вывода (I/O completion routines), уведомляющих об окончании обработки IRP-пакета драйвером более низкого уровня. Например, диспетчер ввода/вывода вызывает процедуру завершения ввода/вывода драйвера файловой системы, после того как драйвер устройства заканчивает передачу данных в файл или прием данных из файла. Эта процедура уведомляет драйвер файловой системы об удачном или неудачном завершении операции или об ее отмене, а также позволяет данному драйверу произвести освобождение ресурсов.

- **Процедуры отмены ввода/вывода.** Если операция ввода/вывода допускает отмену, драйвер может определить одну или несколько процедур отмены ввода/вывода (cancel I/O routines). Получив IRP-пакет для запроса, допускающего отмену, драйвер связывает процедуру отмены с IRP-пакетом, и если на одном из этапов обработки пакета появится неотменяемая операция, процедура может измениться или вообще исчезнуть. Если выдавший запрос на ввод или вывод

программный поток завершается до окончания обработки этого запроса или отменяет операцию (например, вызовом функции `CancelIo` или `CancelIoEx`), диспетчер ввода/вывода выполняет связанную с IRP процедуру отмены, если таковая имеется. Процедура отмены отвечает за все действия по высвобождению ресурсов, выделенных на обработку IRP, а также за завершение IRP со статусом отмены.

- **Процедуры быстрой диспетчеризации.** Драйверы, которые могут пользоваться диспетчером кэша (подробно он рассматривается в главе 14), например драйверы файловой системы, обычно имеют процедуры быстрой диспетчеризации (fast dispatch routines), позволяющие ядру при обращении к драйверу обходить стандартную обработку ввода/вывода. К примеру, такие операции, как чтение или запись, можно быстро выполнить путем прямого доступа к кэшированным данным, не прибегая к диспетчеру ввода/вывода, генерирующему дискретные операции. Процедуры быстрой диспетчеризации также используются как механизмы обратного вызова из диспетчера памяти и диспетчера кэша в драйверы файловой системы. К примеру, при создании раздела диспетчер памяти выполняет обратный вызов драйвера файловой системы для монопольного захвата файла.
- **Процедура выгрузки.** Любые системные ресурсы, которыми пользуется драйвер, освобождает процедура выгрузки (unload routine), после чего диспетчер ввода/вывода получает возможность удалить их из памяти. В частности, освобождаются все ресурсы, выделенные процедурой инициализации (`DriverEntry`). Драйвер может загружаться и выгружаться в процессе работы системы, если он поддерживает такую возможность, но процедура выгрузки вызывается только после закрытия всех обработчиков файла для устройства.
- **Процедура уведомления о завершении работы системы.** Эта процедура позволяет драйверу провести завершающие действия при завершении работы системы.
- **Процедуры регистрации ошибок.** При неожиданных ошибках (например, при появлении на диске поврежденного блока) принадлежащие драйверу процедуры регистрации ошибок (error-logging routines) уведомляют диспетчер ввода/вывода, который фиксирует произошедшее в файле журнала ошибок.

Объекты драйверов и устройств

При открытии программным потоком дескриптора файлового объекта (этот процесс описывается далее в разделе «Обработка ввода/вывода») диспетчер ввода/вывода должен определить по имени этого объекта, к какому драйверу следует обратиться для обработки запроса. Более того, диспетчер ввода/вывода должен быть в состоянии найти данную информацию, когда программный поток в следующий раз воспользуется тем же самым дескриптором. Это достигается с помощью следующих объектов.

- **Объект драйвера** представляет отдельный драйвер в системе (структура `DRIVER_OBJECT`). Именно он дает диспетчеру ввода/вывода адрес процедур диспетчеризации (точек входа) всех драйверов.
- **Объект устройства** представляет физическое или логическое устройство в системе и описывает его характеристики (структура `DEVICE_OBJECT`) — например, границы выравнивания буферов и адреса очередей входящих IRP-пакетов. Именно он является точкой назначения для всех операций ввода/вывода, так как именно с ним взаимодействует дескриптор.

Диспетчер ввода/вывода создает объект драйвера при загрузке нового драйвера и вызывает процедуру инициализации (`DriverEntry`), которая фиксирует точки входа этого драйвера в атрибутах объекта.

После загрузки драйвер при помощи процедур `IoCreateDevice` и `IoCreateDeviceSecure` создает объекты устройств, представляющие логические или физические устройства или даже логический интерфейс или конечную точку драйвера. Но большинство PnP-драйверов пользуется для создания объектов устройств собственными процедурами добавления устройств, запуская их, когда от PnP-диспетчера поступает сигнал о появлении управляемого ими устройства. А вот драйверы, не отвечающие спецификации Plug and Play, создают объект устройства при вызове диспетчером ввода/вывода процедуры их инициализации. После того как последний объект устройства для драйвера будет удален, а ссылок на драйвер не останется, диспетчер ввода/вывода производит выгрузку этого драйвера.

Отношения между объектом драйвера и его объектами устройств изображены на рис. 6.10.

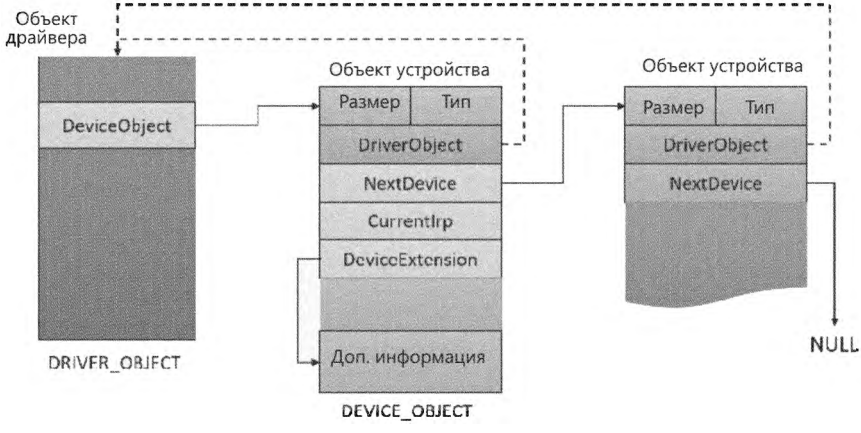


Рис. 6.10. Объект драйвера и его объекты устройств

В поле `DeviceObject` объекта драйвера хранится указатель на его первый объект устройства. В поле `NextDevice` структуры `DEVICE_OBJECT` хранится указатель на

следующий объект устройства, и т. д. до последнего поля, содержащего указатель NULL. Каждый объект устройства содержит обратный указатель на свой объект драйвера в поле `DriverObject`. Все стрелки на рис. 6.10 строятся функциями создания устройств (`IoCreateDevice` или `IoCreateDeviceSecure`). Указатель `DeviceExtension` используется драйвером для выделения дополнительного блока памяти, связанного с каждым объектом устройства под его управлением.

ПРИМЕЧАНИЕ Не путайте объекты драйверов с объектами устройств. Объект драйвера представляет поведение драйвера, а отдельные объекты устройств представляют конечные точки коммуникаций. Например, в системе с четырьмя последовательными портами будет существовать только один объект драйвера (и один двоичный образ драйвера), но четыре экземпляра объектов устройств. Каждый объект устройства представляет один последовательный порт, который может открываться независимо, не влияя на другие последовательные порты. Для физических устройств каждое устройство также представляет отдельный набор аппаратных ресурсов: портов ввода/вывода, областей ввода/вывода, отображаемых в память, и линий прерываний. Система Windows ориентирована на устройства, а не на драйверы.

В момент создания объекта устройства драйвер может присвоить ему имя. В этом случае объект включается в пространство имен диспетчера объектов. Имя объекта задается явно или автоматически генерируется диспетчером ввода/вывода. Объекты устройств по умолчанию оказываются в папке `\Device` пространства имен, недоступного приложениям через Windows API.

ПРИМЕЧАНИЕ Некоторые драйверы помещают объекты устройств в папки, отличные от \Device. Например, драйвер IDE-контроллера создает объекты устройств для IDE-портов и IDE-каналов в папке \Device\Ide. Архитектура систем хранения данных и способ использования драйверами запоминающих устройств объектов устройств рассматриваются в главе 12.

Чтобы предоставить приложениям доступ к объекту устройства, драйвер должен создать в папке \GLOBAL?? символьную ссылку на имя этого объекта в папке \Device. (Эта задача решается функцией `IoCreateSymbolicLink`.) Драйверы, не поддерживающие технологию Plug and Play, и драйверы файловой системы обычно выбирают для символьной ссылки общеизвестное имя (например, \Device\HarddiskVolume2). Но такие имена не работают в средах с динамически меняющимся составом оборудования, поэтому PnP-драйверы предоставляют через функцию `IoRegisterDeviceInterface` один или несколько интерфейсов. Для этого указывается глобальный уникальный идентификатор (GUID), который определяет тип предоставляемой функциональности. 128-разрядные GUID-идентификаторы генерируются программами uuidgen и guidgen, входящими в наборы WDK и Windows SDK. С учетом числового диапазона, предоставляемого 128 битами, можно говорить о том, что каждый создаваемый GUID-идентификатор практически гарантированно является уникальным.

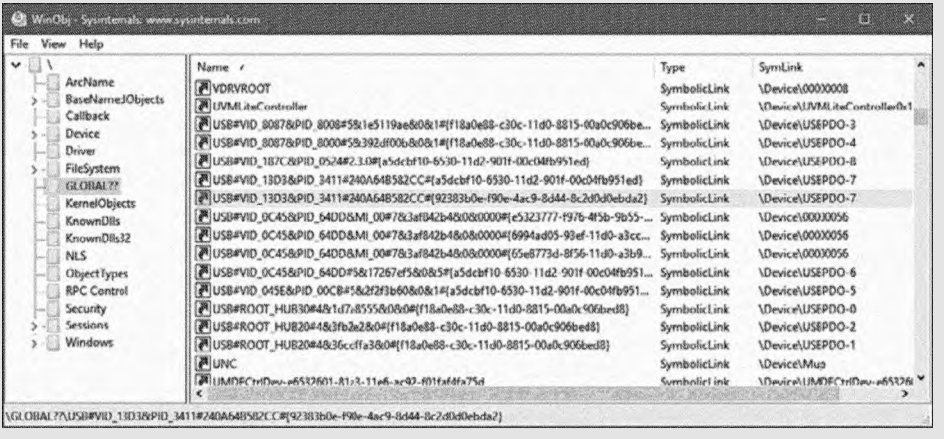
Функция `IoRegisterDeviceInterface` генерирует символьную ссылку, связанную с экземпляром объекта устройства. Но прежде чем диспетчер ввода/вывода действительно создаст ссылку, драйвер должен вызывать эту функцию, чтобы разре-

шить использование интерфейсом устройства. Драйверы обычно делают это при получении от PnP-диспетчера пакета, инициирующего работу устройства, в данном случае — пакета `IRP_MJ_PNP` (основной код функции) с `IRP_MN_START_DEVICE` (дополнительный код функции). Пакеты IRP более подробно рассматриваются в разделе «Пакеты запросов ввода/вывода» этой главы.

Приложение, желающее открыть объект устройства, интерфейсы которого представлены GUID-идентификатором, может вызвать PnP-функции настройки в пространстве пользователя, например `SetupDiEnumDeviceInterfaces` для перечисления доступных для данного GUID-идентификатора интерфейсов и получения списка нужных символьных ссылок. Для доступа к дополнительной информации об устройстве, например к его автоматически сгенерированному имени, приложение вызывает функцию `SetupDiGetDeviceInterfaceDetail` для всех перечисленных в `SetupDiEnumDeviceInterfaces` устройств. Зная имя устройства, приложение может обратиться к Windows-функции `CreateFile` или `CreateFile2`, чтобы открыть устройство и получить его дескриптор.

ЭКСПЕРИМЕНТ: ПРОСМОТР ОБЪЕКТОВ УСТРОЙСТВ

Для просмотра содержимого папки \Device в пространстве имен диспетчера объектов можно воспользоваться служебной программой WinObj от Sysinternals или командой `!object` отладчика ядра. На следующем снимке вы видите присвоенную диспетчером ввода/вывода символьную ссылку, указывающую на объект устройства с автоматически сгенерированным именем.



Команда `!object` отладчика ядра для папки \Device выводит следующую информацию:

```
1: kd> !object \device
Object: 8200c530  Type: (8542b188) Directory
    ObjectHeader: 8200c518 (new version)
    HandleCount: 0  PointerCount: 231
    Directory Object: 82007d20  Name: Device
```

```
Hash Address  Type                    Name
---- -------  ----                    ----
 00  d024a448 Device                  NisDrv
     959afc08 Device                  SrvNet
     958beef0 Device                  WUDFLpcDevice
     854c69b8 Device                  FakeVid1
     8befec98 Device                  RdpBus
     88f7c338 Device                  Beep
     89d64500 Device                  Ndis
     8a24e250 SymbolicLink            ScsiPort2
     89d6c580 Device                  KsecDD
     89c15810 Device                  00000025
     89c17408 Device                  00000019
 01  854c6898 Device                  FakeVid2
     88f98a70 Device                  Netbios
     8a48c6a8 Device                  NameResTrk
     89c2fe88 Device                  00000026
 02  854c6778 Device                  FakeVid3
     8548fee0 Device                  00000034
     8a214b78 SymbolicLink            Ip
     89c31038 Device                  00000027
 03  9c205c40 Device                  00000041
     854c6658 Device                  FakeVid4
     854dd9d8 Device                  00000035
     8d143488 Device                  Video0
     8a541030 Device                  KeyboardClass0
     89c323c8 Device                  00000028
     8554fb50 Device                  KMDF0
 04  958bb040 Device                  ProcessManagement
     97ad9fe0 SymbolicLink            MailslotRedirector
     854f0090 Device                  00000036
     854c6538 Device                  FakeVid5
     8bf14e98 Device                  Video1
     8bf2fe20 Device                  KeyboardClass1
     89c332a0 Device                  00000029
     89c05030 Device                  VolMgrControl
     89c3a1a8 Device                  VMBus
...
```

Если указать для команды `!object` объект-папку диспетчера объектов, отладчик ядра выведет дамп содержимого папки в том виде, в котором он представлен в диспетчере объектов. Для ускорения поиска объекты в папке хранятся в хеш-таблице, основой для которой служит хеш имен объектов. Именно поэтому вы получаете перечень объектов, хранящихся в каждом гнезде хеш-таблицы папки.

Как показано на рис. 6.10, объект устройства ссылается на свой объект драйвера, благодаря чему диспетчер ввода/вывода узнает, процедуру какого драйвера следует вызвать при получении очередного запроса на ввод или вывод. Объект устройства используется для нахождения объекта драйвера, представляющего драйвер, обслуживающий устройство. После этого происходит обращение к объекту драйвера через указанный в исходном запросе номер функции. Каждый номер функции соответствует точке входа драйвера.

С объектом драйвера часто связывается несколько объектов устройств. При выгрузке драйвера из системы диспетчер ввода/вывода прибегает к очереди объектов устройств, чтобы определить, на какие устройства повлияет удаление данного драйвера.

ЭКСПЕРИМЕНТ: ПРОСМОТР ОБЪЕКТОВ ДРАЙВЕРОВ И УСТРОЙСТВ

Вывести объекты драйверов и устройств на экран позволяют команды отладчика ядра `!drvobj` и `!devobj` соответственно. Показанный в этом разделе пример относится к объекту драйвера класса клавиатуры и его единственному объекту устройства:

```
1: kd> !drvobj kbdclass
Driver object (8a557520) is for:
 \Driver\kbdclass
Driver Extension List: (id , addr)

Device Object list:
9f509648  8bf2fe20  8a541030
1: kd> !devobj 9f509648
Device object (9f509648) is for:
 KeyboardClass2 \Driver\kbdclass DriverObject 8a557520
Current Irp 00000000 RefCount 0 Type 0000000b Flags 00002044
Dacl 82090960 DevExt 9f509700 DevObjExt 9f5097f0
ExtensionFlags (0x00000c00)  DOE_SESSION_DEVICE, DOE_DEFAULT_SD_PRESENT
Characteristics (0x00000100)  FILE_DEVICE_SECURE_OPEN
AttachedTo (Lower) 9f509848 \Driver\terminpt
Device queue is not busy.
```

Обратите внимание, что команда `!devobj` показывает также адреса и имена любых объектов устройств, поверх которых находится просматриваемый вами объект (строка `AttachedTo`), и объекты устройств, расположенные над этим объектом (строка `AttachedDevice`), хотя в данном случае их нет.

Команда `!drvobj` может получать дополнительный аргумент для вывода расширенной информации. Пример вывода с максимальной детализацией:

```
1: kd> !drvobj kbdclass 7
Driver object (8a557520) is for:
 \Driver\kbdclass
Driver Extension List: (id , addr)

Device Object list:
9f509648  8bf2fe20  8a541030

DriverEntry:   8c30a010     kbdclass!GsDriverEntry
DriverStartIo: 00000000
DriverUnload:  00000000
AddDevice:     8c307250     kbdclass!KeyboardAddDevice

Dispatch routines:
[00] IRP_MJ_CREATE                    8c301d80    kbdclass!KeyboardClassCreate
[01] IRP_MJ_CREATE_NAMED_PIPE         81142342    nt!IopInvalidDeviceRequest
```

```
[02] IRP_MJ_CLOSE                       8c301c90    kbdclass!KeyboardClassClose
[03] IRP_MJ_READ                        8c302150    kbdclass!KeyboardClassRead
[04] IRP_MJ_WRITE                       81142342    nt!IopInvalidDeviceRequest
[05] IRP_MJ_QUERY_INFORMATION           81142342    nt!IopInvalidDeviceRequest
[06] IRP_MJ_SET_INFORMATION             81142342    nt!IopInvalidDeviceRequest
[07] IRP_MJ_QUERY_EA                    81142342    nt!IopInvalidDeviceRequest
[08] IRP_MJ_SET_EA                      81142342    nt!IopInvalidDeviceRequest
[09] IRP_MJ_FLUSH_BUFFERS               8c303678    kbdclass!KeyboardClassFlush
[0a] IRP_MJ_QUERY_VOLUME_INFORMATION 81142342    nt!IopInvalidDeviceRequest
[0b] IRP_MJ_SET_VOLUME_INFORMATION   81142342    nt!IopInvalidDeviceRequest
[0c] IRP_MJ_DIRECTORY_CONTROL           81142342    nt!IopInvalidDeviceRequest
[0d] IRP_MJ_FILE_SYSTEM_CONTROL         81142342    nt!IopInvalidDeviceRequest
[0e] IRP_MJ_DEVICE_CONTROL              8c3076d0    kbdclass!KeyboardClassDevice
Control
[0f] IRP_MJ_INTERNAL_DEVICE_CONTROL  8c307ff0    kbdclass!KeyboardClassPass
Through
[10] IRP_MJ_SHUTDOWN                    81142342    nt!IopInvalidDeviceRequest
[11] IRP_MJ_LOCK_CONTROL                81142342    nt!IopInvalidDeviceRequest
[12] IRP_MJ_CLEANUP                     8c302260    kbdclass!KeyboardClassCleanup
[13] IRP_MJ_CREATE_MAILSLOT             81142342    nt!IopInvalidDeviceRequest
[14] IRP_MJ_QUERY_SECURITY              81142342    nt!IopInvalidDeviceRequest
[15] IRP_MJ_SET_SECURITY                81142342    nt!IopInvalidDeviceRequest
[16] IRP_MJ_POWER                       8c301440    kbdclass!KeyboardClassPower
[17] IRP_MJ_SYSTEM_CONTROL              8c307f40    kbdclass!KeyboardClassSystem
Control
[18] IRP_MJ_DEVICE_CHANGE               81142342    nt!IopInvalidDeviceRequest
[19] IRP_MJ_QUERY_QUOTA                 81142342    nt!IopInvalidDeviceRequest
[1a] IRP_MJ_SET_QUOTA                   81142342    nt!IopInvalidDeviceRequest
[1b] IRP_MJ_PNP                         8c301870    kbdclass!KeyboardPnP
```

В выходных данных хорошо виден массив процедур диспетчеризации (см. следующий раздел). Обратите внимание: операции, не поддерживаемые драйвером, ссылаются на функцию `IopInvalidDeviceRequest` диспетчера ввода/вывода.

Адрес в команде `!drvobj` относится к структуре `DRIVER_OBJECT`, а адрес в команде `!devobj` — к структуре `DEVICE_OBJECT`. Содержимое этих структур можно просмотреть непосредственно в отладчике:

```
1: kd> dt nt!_driver_object 8a557520
   +0x000 Type             : 0n4
   +0x002 Size             : 0n168
   +0x004 DeviceObject     : 0x9f509648 _DEVICE_OBJECT
   +0x008 Flags            : 0x412
   +0x00c DriverStart      : 0x8c300000 Void
   +0x010 DriverSize       : 0xe000
   +0x014 DriverSection    : 0x8a556ba8 Void
   +0x018 DriverExtension  : 0x8a5575c8 _DRIVER_EXTENSION
   +0x01c DriverName       : _UNICODE_STRING "\Driver\kbdclass"
   +0x024 HardwareDatabase : 0x815c2c28 _UNICODE_STRING "\REGISTRY\MACHINE\
                             HARDWARE\
DESCRIPTION\SYSTEM"
   +0x028 FastIoDispatch   : (null)
   +0x02c DriverInit       : 0x8c30a010     long  +ffffffff8c30a010
   +0x030 DriverStartIo    : (null)
```

```
   +0x034 DriverUnload     : (null)
   +0x038 MajorFunction    : [28] 0x8c301d80     long  +ffffffff8c301d80
1: kd> dt nt!_device_object 9f509648
   +0x000 Type             : 0n3
   +0x002 Size             : 0x1a8
   +0x004 ReferenceCount   : 0n0
   +0x008 DriverObject     : 0x8a557520 _DRIVER_OBJECT
   +0x00c NextDevice       : 0x8bf2fe20 _DEVICE_OBJECT
   +0x010 AttachedDevice   : (null)
   +0x014 CurrentIrp       : (null)
   +0x018 Timer            : (null)
   +0x01c Flags            : 0x2044
   +0x020 Characteristics  : 0x100
   +0x024 Vpb              : (null)
   +0x028 DeviceExtension  : 0x9f509700 Void
   +0x02c DeviceType       : 0xb
   +0x030 StackSize        : 7 ''
   +0x034 Queue            : <unnamed-tag>
   +0x05c AlignmentRequirement : 0
   +0x060 DeviceQueue      : _KDEVICE_QUEUE
   +0x074 Dpc              : _KDPC
   +0x094 ActiveThreadCount : 0
   +0x098 SecurityDescriptor : 0x82090930 Void
   ...
```

В этих структурах есть несколько интересных полей, которые будут рассмотрены в следующих разделах.

Хранение информации о драйверах в объектах означает, что диспетчеру ввода/вывода не нужны подробности об отдельных драйверах. Диспетчер ввода/вывода просто находит драйвер по указателю, что позволяет легко загружать новые драйверы и создает дополнительную прослойку для улучшения портируемости.

Открытие устройств

Файловый объект — структура данных режима ядра, представляющая дескриптор устройства. Файловые объекты точно соответствуют определению объектов в Windows — это системные ресурсы, доступные двум и более процессам в пользовательском режиме; у них могут быть имена, их безопасность обеспечивается моделью защиты объектов, кроме того, они поддерживают синхронизацию. Операции с ресурсами совместного использования в подсистеме ввода/вывода, как и в других компонентах исполнительной подсистемы Windows, осуществляются в виде объектов. (За дополнительной информацией об управлении объектами обращайтесь к главе 8 части 2.)

Файловые объекты обеспечивают представление ресурсов в памяти, которое напоминает интерфейс, ориентированный на ввод/вывод и реализующий чтение или запись. В табл. 6.1 перечислены некоторые атрибуты файлового объекта. Объявле-

ния конкретных полей и их размеры можно посмотреть в определении структуры `FILE_OBJECT` в файле wdm.h.

Таблица 6.1. Атрибуты файловых объектов

Атрибут	Назначение
Имя файла	Идентифицирует физический файл, на который ссылается файловый объект, переданный в `CreateFile` или `CreateFile2` API
Текущее смещение в байтах	Идентифицирует текущее местоположение внутри файла (только для синхронного ввода/вывода)
Режимы совместного доступа	Указывает, могут ли другие вызывающие программы открывать файл для операций чтения, записи или удаления, когда им пользуется текущая вызывающая программа
Флаги режима открытия	Определяют тип ввода/вывода — синхронный или асинхронный, кэшируемый или нет, с последовательным или прямым доступом и т. п.
Указатель на объект устройства	Определяет тип устройства, на котором находится файл
Указатель на блок параметров тома (Volume Parameter Block, VPB)	Определяет том или раздел, в котором находится файл (для файлов файловой системы)
Указатель на указатели объекта раздела	Указывает на корневую структуру, которая описывает отображенный на память/кэшированный файл. Эта структура также содержит открытую карту кэша, которая через диспетчер кэша идентифицирует, какие части файла кэшированы (или, скорее, отображены на память) и где именно в кэше они находятся
Указатель на закрытую карту кэша	Используется для хранения данных кэша для отдельных дескрипторов, таких как закономерности чтения или приоритет страницы для процесса. Информацию о приоритете страницы вы найдете в главе 5 «Управление памятью»
Перечень пакетов запросов на ввод и вывод (IRP)	При независимом от программного потока вводе/выводе (см. далее) и при связывании файлового объекта с портом завершения (также см. далее) — это список всех связанных с рассматриваемым файловым объектом операций ввода/вывода
Контекст завершения ввода/вывода	Сведения о контексте текущего порта завершения ввода/вывода, если он активен
Расширение для файлового объекта	Сохраняет сведения о приоритете ввода/вывода (см. далее) для рассматриваемого файла, данные о необходимости проверки доступа файлового объекта к общим ресурсам и необязательные расширения этого объекта, хранящие контекстную информацию

Чтобы до определенной степени скрыть используемый файловым объектом код драйвера и повысить функциональность этого объекта без усложнения его структуры, к нему было добавлено поле расширения, которое позволяет определить до шести дополнительных атрибутов. Они перечислены в табл. 6.2.

Таблица 6.2. Расширения файлового объекта

Расширение	Назначение
Параметры транзакций	Содержит блок параметров транзакции с информацией о проведенной операции с файлом. Возвращается процедурой `IoGetTransactionParameterBlock`
Подсказка объекта «устройство»	Идентифицирует объект устройства для фильтрующего драйвера, с которым должен ассоциироваться рассматриваемый файл. Задается процедурой `IoCreateFileEx` или `IoCreateFileSpecifyDeviceObjectHint`
Диапазон блока состояния ввода/вывода	Позволяет приложениям блокировать буфер пользовательского режима в памяти режима ядра для оптимизации асинхронного ввода/вывода. Задается процедурой `SetFileIoOverlappedRange`
Общий	Содержит информацию, связанную с фильтрующим драйвером и добавляемыми к вызывающей программе расширенными параметрами создания (ECP). Задается процедурой `IoCreateFileEx`
Запланированный ввод/вывод файла	Сохраняет сведения о резервировании полосы пропускания (см. далее), которые используются системой хранения для оптимизации, и гарантирует пропускную способность мультимедийных приложений. (См. раздел «Резервирование полосы пропускания (планируемый ввод/вывод)»). Атрибут задается процедурой `SetFileBandwidthReservation`
Символическая ссылка	Добавляется к файловому объекту в момент его создания при прохождении точки подключения или соединения для каталога (или при повторной обработке этого пути фильтром). Хранит данные о предоставленном вызывающей программой пути, в том числе о промежуточных переходах, чтобы в случае относительной символической ссылки была возможность вернуться назад. Подробно символические ссылки в NTFS, точки подключения и соединения для каталогов рассматриваются в главе 13 части 2

При открытии вызывающей программой файла или простого устройства диспетчер ввода/вывода возвращает дескриптор файлового объекта. Прежде чем это произойдет, к драйверу, отвечающему за работу устройства, через функцию диспетчеризации `Create` (`IRP_MJ_CREATE`) подается запрос. Он узнает, можно ли открыть устройство, и позволяет драйверу выполнить всю необходимую инициализацию, если запрос на открытие будет выполнен успешно.

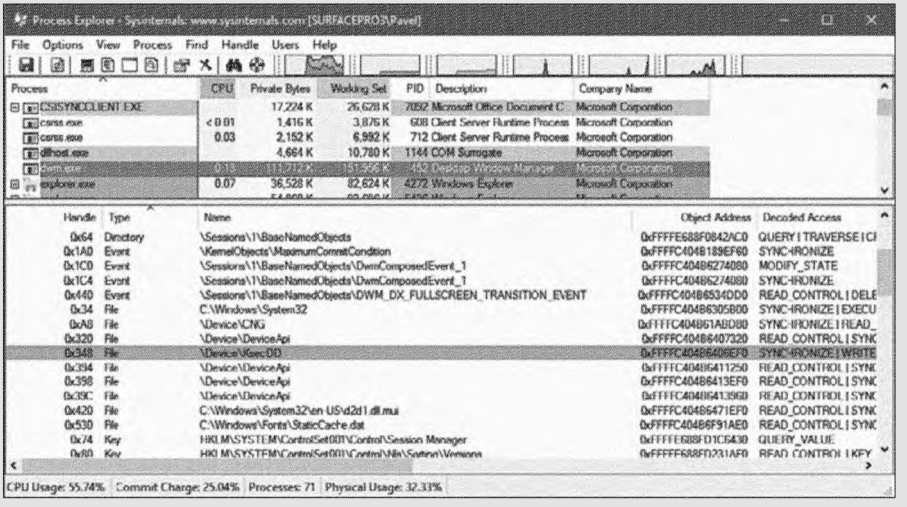
ПРИМЕЧАНИЕ Файловые объекты соответствуют открытым экземплярам файлов, а не самим файлам. В отличие от UNIX-систем, в которых реализованы виртуальные индексные дескрипторы (*vnodes*), в Windows представление файла не определено. Драйверы системных файлов определяют собственные представления.

Файлы, как и прочие объекты исполнительной системы, защищены дескрипторами безопасности, которые включают в себя список управления доступом (Access Control List, ACL). Диспетчер ввода/вывода обращается к подсистеме безопасности, чтобы выяснить, обеспечивает ли процессу ACL файла доступ того типа, который запрошен программным потоком. В случае положительного ответа диспетчер объектов разрешает доступ и связывает права доступа с возвращаемым по-

току дескриптором файла. Если этому или другому программному потоку процесса потребуется проделать над файлом операции, не указанные в исходном запросе, потоку придется еще раз открыть файл уже в рамках нового запроса для получения еще одного дескриптора (или продублировать дескриптор с запрашиваемым уровнем доступа). Это предполагает новую проверку безопасности. (Вопросы защиты объектов рассматриваются в главе 7.)

ЭКСПЕРИМЕНТ: ПРОСМОТР ДЕСКРИПТОРОВ УСТРОЙСТВ

Для любого процесса, открывшего дескриптор устройства, в таблице дескрипторов появится файловый объект. Его можно увидеть в приложении Process Explorer. Выделите процесс и выберите в меню View команду Lower Pane View ▸ Handles. После этого проводится сортировка по столбцу Type и с помощью полосы прокрутки находятся дескрипторы, предоставляющие файловые объекты; они имеют пометку *File*.



В данном примере процесс диспетчера окон рабочего стола (dwm.exe) обладает дескриптором, открытым для созданного драйвером безопасности ядра (Ksecdd.sys) устройства. Для просмотра файлового объекта в отладчике ядра прежде всего требуется определить адрес этого объекта. Следующая команда выводит данные о выделенном на рисунке дескрипторе (со значением 0x348), который принадлежит процессу Dwm.exe с идентификатором 452 (в десятичной записи):

```
lkd> !handle 348 f 0n452

PROCESS ffffc404b62fb780
    SessionId: 1  Cid: 01c4    Peb: b4c3db0000  ParentCid: 0364
    DirBase: 7e607000  ObjectTable: ffffe688fd1c38c0  HandleCount:
             <Data Not Accessible>
    Image: dwm.exe
```

```
Handle Error reading handle count.

0348: Object: ffffc404b6406ef0  GrantedAccess: 00100003 (Audit) Entry:
              ffffe688fd396d20
Object: ffffc404b6406ef0  Type: (ffffc404b189bf20) File
    ObjectHeader: ffffc404b6406ec0 (new version)
        HandleCount: 1  PointerCount: 32767

Так как объект является файловым объектом, данные о нем можно получить командой
!fileobj (обратите внимание: это тот же адрес объекта, который выводится в Pro-
cess Explorer):

lkd> !fileobj ffffc404b6406ef0

Device Object: 0xffffc404b2fa7230   \Driver\KSecDD
Vpb is NULL
Event signalled

Flags:  0x40002
         Synchronous IO
         Handle Created

CurrentByteOffset: 0
```

Файловый объект отличается от остальных объектов исполнительной системы тем, что это не сам ресурс, а только представление совместно используемого ресурса в памяти. Он содержит только данные, связанные с дескриптором объекта, в то время как в самом файле находятся совместно используемые данные или текст. При каждом открытии файла программным потоком создается новый файловый объект с новым набором атрибутов дескриптора. К примеру, для синхронно открываемых файлов атрибут текущего смещения в байтах задает в файле место начала следующей операции чтения или записи с использованием указанного дескриптора. Каждый дескриптор имеет собственное значение этого атрибута, несмотря на совместное использование файла. Кроме того, файловый объект уникален для каждого процесса. Исключением является случай, когда дескриптор дублируется процессом для передачи другому процессу (при помощи функции Windows `DuplicateHandle`) или когда дочерний процесс наследует дескриптор от своего предка. В этих двух ситуациях процессы имеют отдельные дескрипторы, ссылающиеся на один и тот же файловый объект.

Дескриптор файла уникален для процесса, а вот определяемый им физический ресурс — нет. Следовательно, как при доступе к любым общим ресурсам, программным потокам требуется синхронизировать свое обращение к совместно используемым ресурсам (таким, как файлы, папки и устройства). Например, если поток осуществляет запись в файл, при открытии файла он должен получить монопольный доступ на запись, лишив все остальные программные потоки возможности выполнить эту операцию одновременно с ним. Кроме того, если возникнет необходимость в монопольном доступе, поток может заблокировать часть файла на время записи при помощи функции Windows `LockFile`.

При открытии файла в его имя включается имя объекта устройства, на котором находится файл. Например, имя \Device\HarddiskVolume1\Myfile.dat относится к файлу Myfile.dat на диске C:. Подстрока \Device\HarddiskVolume1 является именем внутреннего объекта устройства, представляющего данный диск. При открытии файла Myfile.dat диспетчер ввода/вывода создает файловый объект и сохраняет в нем указатель на объект устройства HarddiskVolume1. Затем он возвращает вызывающей программе дескриптор файла. Впоследствии, как только вызывающая программа воспользуется этим дескриптором, диспетчер ввода/вывода сможет обратиться непосредственно к объекту HarddiskVolume1.

Следует помнить, что внутренние имена устройств неприменимы к Windows-приложениям, вместо этого они должны находиться в специальной папке в пространстве имен диспетчера объектов, которая называется `\GLOBAL??`. Здесь содержатся символьные ссылки на реальные внутренние имена устройств в Windows. Как уже упоминалось, драйверы устройств отвечают за создание ссылок в этой папке, чтобы их устройства были доступны для Windows-приложений. Ссылки можно просмотреть и даже изменить на программном уровне при помощи таких функций Windows, как `QueryDosDevice` и `DefineDosDevice`.

Обработка ввода/вывода

Итак, вы познакомились со структурой и типами драйверов, а также поддерживающими их структурами данных; теперь можно перейти к рассмотрению вопросов, касающихся прохождения по системе запроса на ввод или вывод. Обработка таких запросов состоит из нескольких предсказуемых фаз. Наборы этих фаз зависят от того, какой драйвер — одноуровневый или многоуровневый — управляет устройством, которому адресован запрос. Кроме того, обработка зависит от того, какой ввод/вывод запрошен вызывающей программой — синхронный или асинхронный. Поэтому мы начнем с рассмотрения этих двух типов ввода/вывода, а потом постепенно перейдем к другим.

Типы ввода/вывода

Существуют различные типы запросов ввода/вывода. Более того, диспетчер ввода/вывода позволяет драйверам реализовывать сокращенный интерфейс ввода/вывода, что зачастую сокращает необходимые для обработки данных запросы IRP-пакетов. В этом разделе различные типы запросов будут рассмотрены более подробно.

Синхронный и асинхронный ввод/вывод

Большинство операций ввода/вывода, запрашиваемые приложениями, являются *синхронными* (этот тип предлагается по умолчанию); т. е. программный поток

ждет, когда устройство выполнит операцию с данными и по завершении ввода или вывода вернет код состояния. После этого программа может продолжить работу и немедленно воспользоваться переданными ей данными. В этом простейшем варианте Windows-функции `ReadFile` и `WriteFile` выполняются синхронно. Перед тем как вернуть управление вызывающей программе, они завершают операцию ввода/вывода.

При *асинхронном* вводе/выводе приложение может выдать несколько запросов ввода/вывода и продолжить свою работу, пока устройство выполняет операции ввода/вывода. Это повышает эффективность приложения, позволяя его программному потоку решать другие задачи параллельно с операцией ввода/вывода. Для асинхронного ввода/вывода при вызове функции Windows `CreateFile` или `CreateFile2` следует установить флаг `FILE_FLAG_OVERLAPPED`. Разумеется, после начала операции асинхронного ввода/вывода программный поток не должен получать доступ к запрошенным данным, пока драйвер устройства не закончит их передачу. Выполнение потока следует синхронизировать с завершением запроса ввода/вывода посредством отслеживания дескриптора объекта синхронизации (будь то объект события, порт завершения ввода/вывода или сам файловый объект), который подаст сигнал о завершении операции.

Независимо от типа запроса на ввод или вывод, внутренние операции ввода/вывода, инициированные драйвером на стороне приложения, выполняются асинхронно; т. е. после выдачи запроса на ввод или вывод драйвер устройства должен как можно быстрее вернуть управление подсистеме ввода/вывода. Будет ли управление немедленно возвращено вызывающей программе, зависит от того, для какого именно ввода/вывода — синхронного или асинхронного — был открыт дескриптор. Схема управления для операции чтения показана на рис. 6.3. Обратите внимание, что режим ожидания (который зависит от флага `FILE_FLAG_OVERLAPPED` в файловом объекте) реализуется функцией `NtReadFile` в режиме ядра.

Проверка состояния незавершенной асинхронной операции ввода/вывода в режиме ожидания выполняется макросом `HasOverlappedIoCompleted`, а для получения расширенной информации используются функции `GetOverlappedResult(Ex)`. При использовании портов завершения ввода/вывода для этой цели применяются функции `GetQueuedCompletionStatus(Ex)`.

Быстрый ввод/вывод

При быстром вводе/выводе подсистема ввода/вывода может обойтись без генерирования IRP-пакетов и напрямую обратиться к стеку драйверов для завершения запроса на ввод или вывод. Подробно быстрый ввод/вывод рассматривается в главах 13 и 14 части 2. Драйвер регистрирует свои точки входа для быстрого ввода/вывода, записывая их адреса в структуру, на которую ссылается указатель `PFAST_IO_DISPATCH` его драйверного объекта.

ЭКСПЕРИМЕНТ: ПРОСМОТР ЗАРЕГИСТРИРОВАННЫХ ДРАЙВЕРОМ ПРОЦЕДУР БЫСТРОГО ВВОДА/ВЫВОДА

Команда отладчика ядра `!drvobj` выводит список процедур быстрого ввода/вывода, зарегистрированных драйвером в своем драйверном объекте. Но практический смысл эти процедуры обычно имеют только для драйверов файловой системы, хотя бывают и исключения, например драйверы сетевых протоколов и фильтрующие драйверы шины. Пример таблицы быстрого ввода/вывода для объекта драйвера файловой системы NTFS:

```
lkd> !drvobj \filesystem\ntfs 2
Driver object (ffffc404b2fbf810) is for:
 \FileSystem\NTFS
DriverEntry:   fffff80e5663a030  NTFS!GsDriverEntry
DriverStartIo: 00000000
DriverUnload:  00000000
AddDevice:     00000000

Dispatch routines:
...
Fast I/O routines:
FastIoCheckIfPossible              fffff80e565d6750
NTFS!NtfsFastIoCheckIfPossible
FastIoRead                         fffff80e56526430  NTFS!NtfsCopyReadA
FastIoWrite                        fffff80e56523310  NTFS!NtfsCopyWriteA
FastIoQueryBasicInfo               fffff80e56523140
NTFS!NtfsFastQueryBasicInfo
FastIoQueryStandardInfo            fffff80e56534d20  NTFS!NtfsFastQueryStdInfo
FastIoLock                         fffff80e5651e610  NTFS!NtfsFastLock
FastIoUnlockSingle                 fffff80e5651e3c0  NTFS!NtfsFastUnlockSingle
FastIoUnlockAll                    fffff80e565d59e0  NTFS!NtfsFastUnlockAll
FastIoUnlockAllByKey               fffff80e565d5c50
NTFS!NtfsFastUnlockAllByKey
ReleaseFileForNtCreateSection      fffff80e5644fd90  NTFS!NtfsReleaseForCreate
Section
FastIoQueryNetworkOpenInfo         fffff80e56537750  NTFS!NtfsFastQueryNetwork
OpenInfo
AcquireForModWrite                 fffff80e5643e0c0
NTFS!NtfsAcquireFileForModWrite
MdlRead                            fffff80e5651e950  NTFS!NtfsMdlReadA
MdlReadComplete                    fffff802dc6cd844
nt!FsRtlMdlReadCompleteDev
PrepareMdlWrite                    fffff80e56541a10  NTFS!NtfsPrepareMdlWriteA
MdlWriteComplete                   fffff802dcb76e48
nt!FsRtlMdlWriteCompleteDev
FastIoQueryOpen                    fffff80e5653a520
NTFS!NtfsNetworkOpenCreate
ReleaseForModWrite                 fffff80e5643e2c0
NTFS!NtfsReleaseFileForModWrite
AcquireForCcFlush                  fffff80e5644ca60
NTFS!NtfsAcquireFileForCcFlush
ReleaseForCcFlush                  fffff80e56450cf0
NTFS!NtfsReleaseFileForCcFlush
```

Как видите, файловая система NTFS зарегистрировала свою процедуру `NtfsCopyReadA` как запись `FastIoRead` в таблице быстрого ввода/вывода. По имени записи можно догадаться, что диспетчер ввода/вывода вызывает эту функцию при получении запроса на ввод или вывод, если файл находится в кэше. Если вызов остается безрезультатным, выбирается стандартный путь с формированием IRP-пакета.

Ввод/вывод для файлов, отображенных на память, и кэширование файлов

Операции ввода/вывода с файлами, отображенными на память, — одна из важных задач подсистемы ввода/вывода, которая реализуется совместно подсистемой ввода/вывода и диспетчера памяти. (Отображение файлов на память более подробно рассматривается в главе 5.) Само понятие ввода/вывода для *файлов, отображенных на память* (mapped file I/O), относится к интерпретации файла на диске как части виртуальной памяти процесса. Программа может обращаться к такому файлу как к большому массиву, не прибегая к буферизации или дисковому вводу/выводу. При обращении программы к памяти диспетчер памяти задействует механизм подкачки нужной страницы из файла на диске. Если приложение записывает данные в виртуальное адресное пространство, диспетчер памяти записывает эти данные обратно в файл в процессе стандартной операции подкачки страниц.

Ввод/вывод для файлов, отображенных на память, в режиме пользователя осуществляется с помощью Windows-функций `CreateFileMapping`, `MapViewOfFile` и других функций, связанных с ними. В самой операционной системе такой ввод/вывод применяется при выполнении важных операций, например кэшировании файлов и активации образов. Еще ввод/вывод данного типа активно используется диспетчером кэша. Файловые системы обращаются к этому диспетчеру для отображения данных на виртуальную память, что ускоряет отклик программ, интенсивно использующих ввод и вывод. По мере работы с файлом диспетчер памяти подгружает в память страницы, к которым обращается вызывающая программа. В то время как большинство систем кэширования выделяет под эту процедуру фиксированную область памяти, размер кэша в Windows зависит от объема свободной памяти. Подобная универсальность стала возможной благодаря тому, что диспетчер кэша при автоматическом расширении (или уменьшении) размера кэша опирается на диспетчер памяти. Подробно данный механизм рассматривается в главе 5. Система подкачки страниц диспетчера памяти выполняет часть обязанностей диспетчера кэша, что уменьшает количество проделываемых последним операций. (Внутреннее устройство диспетчера кэша подробно рассматривается в главе 14 части 2.)

Фрагментированный ввод/вывод

В Windows поддерживается также особый вид высокопроизводительного ввода/вывода, который называется *фрагментированным* (scatter/gather I/O). Он реализуется функциями Windows `ReadFileScatter` и `WriteFileGather`. Они дают

приложению возможность в рамках одной операции читать и записывать данные из нескольких буферов в виртуальной памяти в непрерывную область дискового файла, а не использовать отдельный запрос на ввод или вывод для каждого буфера. Чтобы воспользоваться фрагментированным вводом/выводом, файл следует открыть для некэшированного ввода/вывода, пользовательские буферы должны быть выровнены по границам страниц, а ввод/вывод должен быть асинхронным (перекрывающимся). Кроме того, в случае ввода/вывода для запоминающего устройства передаваемые данные нужно выровнять по границам его секторов, при этом их размер должен быть кратен размеру сектора.

Пакеты запросов ввода/вывода

В *пакете запроса ввода/вывода* (I/O Request Packet, IRP) система ввода/вывода хранит информацию, необходимую для обработки запросов на ввод и вывод. При вызове программным потоком прикладного программного интерфейса ввода/вывода диспетчер ввода/вывода создает IRP-пакет, который будет представлять операцию в процессе ее выполнения подсистемой ввода/вывода. По возможности диспетчер ввода/вывода выделяет память под IRP-пакеты в одном из трех невыгружаемых списков уровня процессора.

- **Список малых IRP-пакетов** хранит пакеты в одном блоке стека (о том, что такое блок стека, мы поговорим чуть позже).
- **Список средних IRP-пакетов** содержит пакеты с четырьмя блоками стека (при этом их можно применять для IRP-пакетов, требующих всего два или три блока стека).
- **Список больших IRP-пакетов** содержит IRP-пакеты с более чем четырьмя блоками стека. По умолчанию в последнем случае система сохраняет IRP-пакеты с 14 блоками стека, но раз в минуту это число приводится в соответствие с количеством запрошенных блоков и может быть увеличено до 20 (в зависимости от количества блоков стека, востребованных в последнее время).

Кроме того, для этих списков существуют глобальные резервные списки, что обеспечивает эффективную передачу IRP-пакетов между процессорами. Если IRP-пакет требует больше блоков стека, чем в списке больших IRP-пакетов, диспетчер ввода/вывода выделяет под пакеты память из невыгружаемого пула. Диспетчер ввода/вывода выделяет память для IRP функцией `IoAllocateIrp`, которая также доступна для разработчиков драйверов устройств, потому что в некоторых случаях драйвер может инициировать запросы ввода/вывода напрямую, создавая и инициализируя собственные IRP. После создания и инициализации IRP-пакета диспетчер ввода/вывода сохраняет там указатель на файловый объект вызывающей программы.

На рис. 6.11 показаны некоторые важные поля структуры IRP. Эта структура всегда дополняется одним или несколькими объектами `IO_STACK_LOCATION` (см. следующий раздел).

ПРИМЕЧАНИЕ DWORD-параметр реестра HKLM\System\CurrentControlSet\Session Manager\I/O System `LargeIrpStackLocations`, если он определен, указывает, сколько блоков стека содержится в IRP-пакетах, хранящихся в ассоциативном списке больших IRP-пакетов. Аналогичным образом параметр `MediumIrpStackLocations` в том же разделе может использоваться для изменения размера блоков стека в списке средних IRP-пакетов.

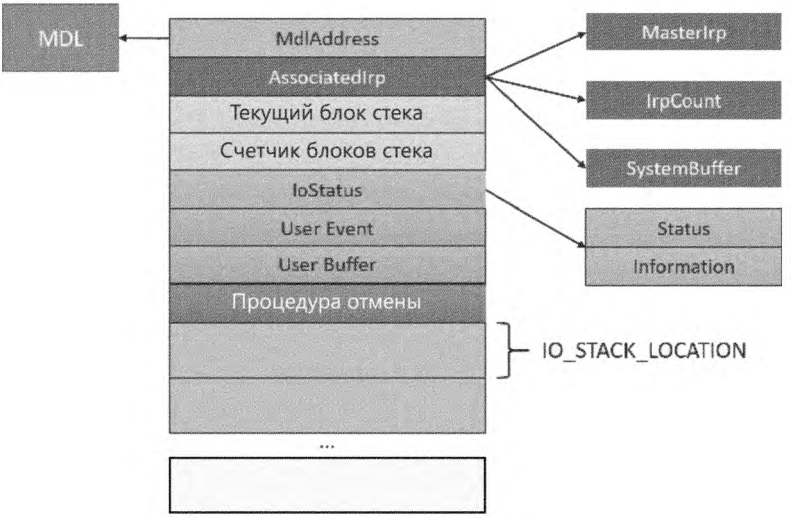


Рис. 6.11. Важные поля структуры IRP

Краткое описание полей:

- **IoStatus** – описание статуса IRP с двумя полями; `Status` (собственно код статуса) и `Information` – полиморфное значение, которое имеет смысл в некоторых ситуациях. Например, для операции чтения или записи это значение (задаваемое драйвером) обозначает количество прочитанных или записанных байтов. Это же значение возвращается в качестве выходного функциями `ReadFile` и `WriteFile`.
- **MdlAddress** – необязательный указатель на список дескрипторов памяти (MDL, Memory Descriptor List). MDL – структура, представляющая информацию о буфере в физической памяти. Использование MDL в драйверах устройств рассматривается в следующем разделе. Если указатель на MDL не востребован, поле содержит NULL.
- **Счетчик блоков стека и текущий блок стека.** В этих полях хранится общее количество завершающих объектов блоков стека ввода/вывода и указатель на текущий блок, к которому должен обращаться драйвер. В следующем разделе блоки стеков ввода/вывода рассматриваются более подробно.
- **Пользовательский буфер.** Указатель на буфер, предоставленный клиентом, инициировавшим операцию ввода/вывода, – например, буфер, переданный функции `ReadFile` или `WriteFile`.

- **Пользовательское событие.** Объект события ядра, использованный с асинхронной операцией ввода/вывода (если он есть). Событие — один из способов оповещения о завершении операции ввода/вывода.
- **Процедура отмены** — функция, вызываемая диспетчером ввода/вывода в случае отмены IRP.
- **AssociatedIrp** — одно из трех полей. Поле `SystemBuffer` используется в том случае, если диспетчер ввода/вывода использовал метод буферизации ввода/вывода для передачи пользовательского буфера драйверу. В следующем разделе рассматривается буферизация ввода/вывода, а также другие способы передачи буферов пользовательского режима драйверам. Поле `MasterIrp` предоставляет механизм создания «главного IRP-пакета», распределяющего свою работу по нескольким субпакетам; главный IRP-пакет считается завершенным только после того, как будут завершены все его субпакеты.

Блоки стека IRP-пакетов

За IRP-пакетом всегда следует один или несколько блоков стека. Количество блоков стека равно количеству уровней устройств в узле устройства, для которого предназначен IRP-пакет. Информация об операции ввода/вывода распределяется между телом IRP-пакета (основная структура) и текущим блоком стека ввода/вывода (где *текущий* означает «предназначенный для конкретного уровня устройств»). На рис. 6.12 показаны важные поля блока стека. При создании IRP-пакета количество запрашиваемых блоков передается функции `IoAllocateIrp`. Затем диспетчер ввода/вывода инициализирует тело IRP-пакета и только первый блок стека, предназначенный для верхнего устройства в узле. Каждый уровень узла устройства отвечает за инициализацию следующего блока стека, если он решит передать IRP-пакет следующему устройству.

Краткая сводка основных полей структуры на рис. 6.12.

- **Основной номер функции** (major function) определяет тип запроса (чтение, запись, создание, PnP и т. д.); он также называется *кодом процедуры диспетчеризации*. Это одна из 28 констант (от 0 до 27) с префиксом `IRP_MJ_` в файле wdm.h. Константа используется диспетчером ввода/вывода для индексирования массива указателей на функции `MajorFunction` в объекте драйвера для перехода к соответствующей функции в драйвере. Большинство драйверов задает процедуру диспетчеризации для обработки подмножества возможных кодов функций, включая операции создания (открытия), чтения, записи, управления вводом/выводом на устройстве, управления электропитанием и PnP-операциями, управления системой (для WMI-команд), очистки и закрытия. Среди типов драйверов, которые обычно заполняют все точки входа (или их большую часть), можно выделить драйверы файловой системы. Напротив, драйвер простого устройства USB с большой вероятностью заполнит только точки входа для открытия, закрытия, чтения, записи и отправ-

ки управляющих кодов ввода/вывода. Диспетчер ввода/вывода заполняет точки входа диспетчеризации, не заполненные драйвером, своей функцией `IopInvalidDeviceRequest`; она заполняет IRP-пакет статусом ошибки, которая указывает, что основная функция, указанная для IRP-пакета, для данного устройства недействительна.

- **Дополнительный номер функции** (minor function) уточняет основной номер для некоторых функций. Например, функции `IRP_MJ_READ` (чтение) и `IRP_MJ_WRITE` (запись) не имеют дополнительного номера. С другой стороны, IRP-пакеты PnP и электропитания всегда имеют дополнительный код функции для уточнения основного. Например, основная функция Plug and Play `IRP_MJ_PNP` слишком расплывчата; точная инструкция задается дополнительной функцией, такой как `IRP_MN_START_DEVICE`, `IRP_MN_REMOVE_DEVICE` и т. д.
- **Параметры** — громадное объединение структур, каждая из которых действительна для конкретной основной функции или комбинации основного/дополнительного кода. Например, для операции чтения (`IRP_MJ_READ`) структура `Parameters.Read` содержит информацию о запросе на чтение (например, размер буфера).
- **FileObject и DeviceObject** содержат указатели на структуры `FILE_OBJECT` и `DEVICE_OBJECT` для данного запроса ввода/вывода.
- **CompletionRoutine** — необязательная функция, которую драйвер может зарегистрировать вызовом `IoSetCompletionRoutine(Ex)`, чтобы она вызывалась при завершении IRP драйвером более низкого уровня. В этот момент драйвер может просмотреть статус завершения IRP и выполнить все необходимые завершающие действия. Он даже может отметить завершение (возвращая специальное значение `STATUS_MORE_PROCESSING_REQUIRED`) и заново отправить IRP-пакет (возможно, с измененными параметрами) узлу устройства или даже другому узлу устройства.

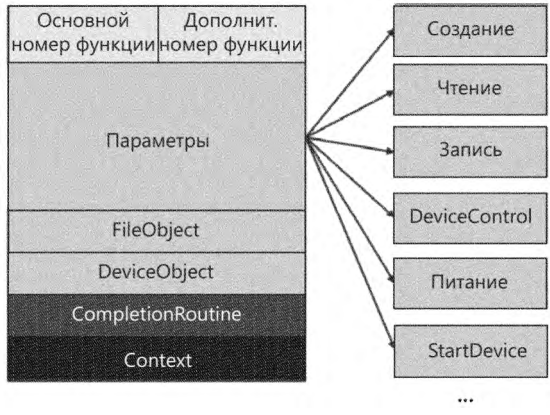


Рис. 6.12. Важные поля структуры IO_STACK_LOCATION

- **Context** — произвольное значение, заданное при вызове `IoSetCompletionRoutine(Ex)`, которое передается в неизменном виде процедуре завершения.

Распределение информации между телом IRP и блоком стека позволяет изменить параметры блока стека для следующего устройства в стеке без потери параметров исходного запроса. Например, реальный IRP-пакет, предназначенный для USB-устройства, часто преобразуется функциональным драйвером в IRP-пакет управления вводом/выводом, у которого аргумент входного буфера указывает на пакет запроса USB (URB, USB Request Packet), «понятный» для драйвера шины USB нижнего уровня. Также стоит заметить, что процедуры завершения могут регистрироваться любым уровнем (кроме самого нижнего), и каждая занимает свое место в блоке стека (процедура завершения хранится в *следующем* блоке стека более низкого уровня).

ЭКСПЕРИМЕНТ: ПРОСМОТР ПРОЦЕДУР ДИСПЕТЧЕРИЗАЦИИ ДРАЙВЕРА

Для получения списка всех функций, определенных драйвером для его процедур диспетчеризации, следует использовать бит 1 (значение 2) с командой отладчика ядра `!drvobj`. Показанный далее результат демонстрирует основные коды функций, поддерживаемых драйвером NTFS. (Этот же эксперимент используется при быстром вводе/выводе.)

```
lkd> !drvobj \filesystem\ntfs 2
Driver object (ffffc404b2fbf810) is for:
 \FileSystem\NTFS
DriverEntry:   fffff80e5663a030                        NTFS!GsDriverEntry
DriverStartIo: 00000000
DriverUnload:  00000000
AddDevice:     00000000

Dispatch routines:
[00] IRP_MJ_CREATE                      fffff80e565278e0  NTFS!NtfsFsdCreate
[01] IRP_MJ_CREATE_NAMED_PIPE           fffff802dc762c80  nt!IopInvalidDeviceRequest
[02] IRP_MJ_CLOSE                       fffff80e565258c0  NTFS!NtfsFsdClose
[03] IRP_MJ_READ                        fffff80e56436060  NTFS!NtfsFsdRead
[04] IRP_MJ_WRITE                       fffff80e564461d0  NTFS!NtfsFsdWrite
[05] IRP_MJ_QUERY_INFORMATION           fffff80e565275f0  NTFS!NtfsFsdDispatchWait
[06] IRP_MJ_SET_INFORMATION             fffff80e564edb80  NTFS!NtfsFsdSetInformation
[07] IRP_MJ_QUERY_EA                    fffff80e565275f0  NTFS!NtfsFsdDispatchWait
[08] IRP_MJ_SET_EA                      fffff80e565275f0  NTFS!NtfsFsdDispatchWait
[09] IRP_MJ_FLUSH_BUFFERS               fffff80e5653c9a0  NTFS!NtfsFsdFlushBuffers
[0a] IRP_MJ_QUERY_VOLUME_INFORMATION    fffff80e56538d10  NTFS!NtfsFsdDispatch
[0b] IRP_MJ_SET_VOLUME_INFORMATION      fffff80e56538d10  NTFS!NtfsFsdDispatch
[0c] IRP_MJ_DIRECTORY_CONTROL           fffff80e564d7080  NTFS!NtfsFsdDirectoryControl
[0d] IRP_MJ_FILE_SYSTEM_CONTROL         fffff80e56524b20  NTFS!NtfsFsdFileSystemControl
[0e] IRP_MJ_DEVICE_CONTROL              fffff80e564f9de0  NTFS!NtfsFsdDeviceControl
[0f] IRP_MJ_INTERNAL_DEVICE_CONTROL     fffff802dc762c80  nt!IopInvalidDeviceRequest
[10] IRP_MJ_SHUTDOWN                    fffff80e565efb50  NTFS!NtfsFsdShutdown
[11] IRP_MJ_LOCK_CONTROL                fffff80e5646c870  NTFS!NtfsFsdLockControl
[12] IRP_MJ_CLEANUP                     fffff80e56525580  NTFS!NtfsFsdCleanup
[13] IRP_MJ_CREATE_MAILSLOT             fffff802dc762c80  nt!IopInvalidDeviceRequest
[14] IRP_MJ_QUERY_SECURITY              fffff80e56538d10  NTFS!NtfsFsdDispatch
[15] IRP_MJ_SET_SECURITY                  fffff80e56538d10  NTFS!NtfsFsdDispatch
```

```
[16] IRP_MJ_POWER                       fffff802dc762c80  nt!IopInvalidDeviceRequest
[17] IRP_MJ_SYSTEM_CONTROL              fffff802dc762c80  nt!IopInvalidDeviceRequest
[18] IRP_MJ_DEVICE_CHANGE               fffff802dc762c80  nt!IopInvalidDeviceRequest
[19] IRP_MJ_QUERY_QUOTA              fffff80e565275f0  NTFS!NtfsFsdDispatchWait
[1a] IRP_MJ_SET_QUOTA                fffff80e565275f0  NTFS!NtfsFsdDispatchWait
[1b] IRP_MJ_PNP                      fffff80e56566230  NTFS!NtfsFsdPnp

Fast I/O routines:
...
```

Все активные IRP-пакеты хранятся в списке IRP-пакетов, связанном с запросившим ввод или вывод программным потоком. (Неактивный пакет, как уже упоминалось, сохраняется в файловом объекте в процессе выполнения программного потока, не зависящего от ввода/вывода — эта тема рассматривается далее в этой главе.) Это позволяет подсистеме ввода/вывода находить и отменять незавершенные IRP-пакеты при завершении выдавшего их потока. Сверх того, пакеты страничного ввода/вывода также связываются с прекратившим свою работу программным потоком (но отменить их невозможно). Это позволяет Windows оптимизировать операции ввода/вывода без привязки к потоку, когда для завершения инициированной текущим программным потоком операции не применяется асинхронный вызов процедур (Asynchronous Procedure Call, APC). Это позволяет обрабатывать ошибки страниц «на месте» вместо выдачи APC-запроса.

ЭКСПЕРИМЕНТ: ПРОСМОТР НЕЗАВЕРШЕННЫХ IRP-ПАКЕТОВ ПРОГРАММНОГО ПОТОКА

Команда `!thread` выводит все связанные с программным потоком IRP-пакеты. То же самое может сделать команда `!process`. Запустите отладчик ядра в режиме локальной или «живой» отладки и выведите список программных потоков процесса explorer.exe:

```
lkd> !process 0 7 explorer.exe
PROCESS ffffc404b673c780
    SessionId: 1  Cid: 10b0    Peb: 00cbb000  ParentCid: 1038
    DirBase: 8895f000  ObjectTable: ffffe689011b71c0  HandleCount: <Data Not
Accessible>
    Image: explorer.exe
    VadRoot ffffc404b672b980 Vads 569 Clone 0 Private 7260.
            Modified 366527. Locked 784.
    DeviceMap ffffe688fd7a5d30
    Token                             ffffe68900024920
    ElapsedTime                       18:48:28.375
    UserTime                          00:00:17.500
    KernelTime                        00:00:13.484
    ...
    MemoryPriority                    BACKGROUND
    BasePriority                      8
    CommitCharge                      10789
```

```
    Job                                ffffc404b6075060

        THREAD ffffc404b673a080  Cid 10b0.10b4  Teb: 0000000000cbc000
            Win32Thread:
ffffc404b66e7090 WAIT: (WrUserRequest) UserMode Non-Alertable
            ffffc404b6760740  SynchronizationEvent
        Not impersonating
...

        THREAD ffffc404b613c7c0  Cid 153c.15a8  Teb: 00000000006a3000
            Win32Thread:
ffffc404b6a83910 WAIT: (UserRequest) UserMode Non-Alertable
            ffffc404b58d0d60  SynchronizationEvent
            ffffc404b566f310  SynchronizationEvent
        IRP List:
            ffffc404b69ad920: (0006,02c8) Flags: 00060800  Mdl: 00000000
...
```

Вы увидите множество программных потоков, большая часть которых имеет IRP-пакеты. Эти пакеты перечислены в разделе IRP `List` сведений о потоке (обратите внимание: отладчик выводит только первые 17 IRP-пакетов для потока, количество незавершенных запросов на ввод и вывод у которого превышает 17). Выберите IRP-пакет командой `!irp` и просмотрите информацию о нем:

```
lkd> !irp ffffc404b69ad920
Irp is active with 2 stacks 1 is current (= 0xffffc404b69ad9f0)
 No Mdl: No System Buffer: Thread ffffc404b613c7c0:  Irp stack trace.
     cmd  flg cl Device   File     Completion-Context
>[IRP_MJ_FILE_SYSTEM_CONTROL(d), N/A(0)]
            5 e1 ffffc404b253cc90 ffffc404b5685620 fffff80e55752ed0-
ffffc404b63c0e00
Success Error Cancel pending
                \FileSystem\Npfs      FLTMGR!FltpPassThroughCompletion
                              Args: 00000000 00000000 00110008 00000000
 [IRP_MJ_FILE_SYSTEM_CONTROL(d), N/A(0)]
            5  0 ffffc404b3cdca00 ffffc404b5685620 00000000-00000000
                \FileSystem\FltMgr
                              Args: 00000000 00000000 00110008 00000000
```

Пакет содержит два блока стека и адресован устройству, принадлежащему драйверу NPFS (Named Pipe File System) — см. главу 10.

Последовательность обработки IRP

IRP-пакеты обычно создаются диспетчером ввода/вывода и передаются первому устройству целевого узла устройств. На рис. 6.13 изображена типичная последовательность обработки IRP для драйверов физических устройств.

Впрочем, IRP-пакеты создаются не только диспетчером ввода/вывода. Диспетчер PnP и диспетчер электропитания также могут создавать IRP-пакеты с основными кодами функций `IRP_MJ_PNP` и `IRP_MJ_POWER` соответственно.

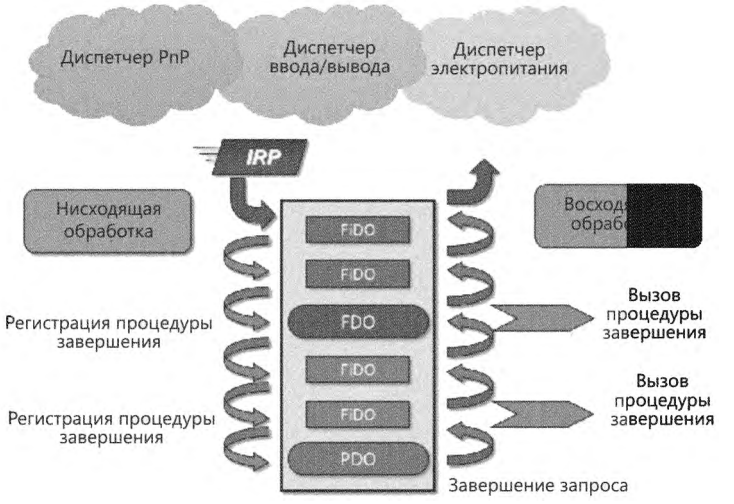


Рис. 6.13. Последовательность обработки IRP

На рис. 6.13 показан пример узла с шестью объектами устройств, образующими иерархию: два высокоуровневых фильтра, FDO, два низкоуровневых фильтра и PDO. Это означает, что IRP-пакет, предназначенный для этого узла, создается с шестью блоками стека — по одному для каждого уровня иерархии. IRP-пакет всегда передается верхнему устройству, даже если дескриптор был открыт для именованного устройства, расположенного ниже в стеке устройств.

Драйвер, получающий IRP-пакет, может выбрать один из следующих вариантов:

- Он может немедленно завершить обработку IRP-пакета вызовом `IoCompleteRequest`. Например, это может произойти из-за того, что IRP-пакет имеет недействительные параметры (недостаточный размер буфера, неверный код управления вводом/выводом и т. д.) или запрошенная операция может быть выполнена немедленно — например, получение информации о статусе устройства или чтение значения из реестра. Драйвер вызывает `IoGetCurrentIrpStackLocation` для получения указателя на блок стека, с которым он должен работать.
- Драйвер может передать IRP-пакет следующему уровню (возможно, после некоторой промежуточной обработки). Например, верхний фильтр может сохранить информацию об операции в журнале и отправить IRP-пакет для нормального выполнения. Прежде чем передавать запрос на следующий уровень вниз, драйвер должен подготовить следующий блок стека ввода/вывода, к которому обратится следующий драйвер. Он может использовать макрос `IoSkipCurrentIrpStackLocation`, если не хочет вносить изменения, или же создать копию вызовом `IoCopyIrpStackLocationToNext` и внести из-

менения в скопированный блок стека, для чего он получает указатель вызовом `IoGetNextIrpStackLocation` и вносит соответствующие изменения. После того как следующий блок стека будет подготовлен, драйвер вызывает `IoCallDriver` для передачи IRP-пакета.

- В дополнение к предыдущему пункту драйвер также может зарегистрировать процедуру завершения вызовом `IoSetCompletionRoutine(Ex)`, прежде чем передавать IRP дальше. Любой уровень, кроме самого нижнего, может зарегистрировать процедуру завершения (на нижнем уровне это не имеет смысла, так как драйвер должен завершить обработку IRP-пакета). После вызова `IoCompleteRequest` драйвером нижнего уровня IRP-пакет перемещается снизу вверх (см. рис. 6.13), вызывая все процедуры завершения в порядке, обратном порядку их регистрации. Источник IRP (диспетчер ввода/вывода, диспетчер PnP, диспетчер электропитания) использует этот механизм для выполнения любых необходимых завершающих действий и, наконец, освобождает IRP.

ПРИМЕЧАНИЕ Поскольку количество устройств в конкретном стеке известно заранее, диспетчер ввода/вывода выделяет один блок стека для каждого драйвера устройства. Однако возможны ситуации, при которых IRP-пакет направляется новому стеку драйверов. Например, это может случиться в сценариях с диспетчером фильтров, при которых один фильтр может перенаправить IRP другому фильтру (скажем, из локальной файловой системы в сетевую файловую систему). Диспетчер ввода/вывода предоставляет API-функцию `IoAdjustStackSizeForRedirection`, которая делает возможной эту функциональность за счет добавления необходимых блоков стека для устройств, присутствующих в стеке перенаправления.

ЭКСПЕРИМЕНТ: ПРОСМОТР СТЕКА УСТРОЙСТВ

Команда `!devstack` отладчика ядра выводит на экран стек для выбранной многоуровневой иерархии объектов устройств, связанной с заданным объектом устройства. В рассматриваемом примере стек связан с объектом устройства \device\keyboardclass0, принадлежащим драйверу класса клавиатуры:

```
lkd> !devstack keyboardclass0
  !DevObj           !DrvObj            !DevExt           ObjectName
> ffff9c80c0424440  \Driver\kbdclass   ffff9c80c0424590  KeyboardClass0
  ffff9c80c04247c0  \Driver\kbdhid     ffff9c80c0424910
  ffff9c80c0414060  \Driver\mshidkmdf  ffff9c80c04141b0  0000003f
!DevNode ffff9c80c0414d30 :
  DeviceInst is "HID\MSHW0029&Col01\5&1599b1c7&0&0000"
  ServiceName is "kbdhid"
```

Строка, связанная с классом `KeyboardClass0`, выделена символом > в первом столбце. Расположенные выше строки относятся к драйверам, работающим поверх драйвера класса клавиатуры, а расположенные ниже — к драйверам, работающим ниже.

ЭКСПЕРИМЕНТ: АНАЛИЗ IRP-ПАКЕТОВ

В этом эксперименте в системе находятся незавершенные IRP-пакеты; нужно узнать их тип, устройство, которому они адресованы, управляющий этим устройством драйвер, выдавший IRP-пакеты программный поток, а также процесс, к которому этот поток относится. Эксперимент лучше проводить в 32-разрядной системе с нелокальной отладкой режима ядра. Вообще говоря, с локальной отладкой он тоже может сработать, но IRP-пакеты могут завершаться в период между выдачей команд, что создает некоторую нестабильность данных.

В системе в любой момент времени присутствует несколько незавершенных IRP-пакетов, поскольку существует множество устройств, которым приложения могут посылать IRP-пакеты, а драйвер обрабатывает запрос только после определенного события, например после получения доступа к данным (например, при чтении в блокирующем режиме из конечной точки сети). Для просмотра незавершенных IRP-пакетов в системе используйте команду `!irpfind` отладчика ядра (получение данных может потребовать времени; можно прервать выполнение после вывода некоторых IRP):

```
kd> !irpfind
Scanning large pool allocation table for tag 0x3f707249 (Irp?) (a5000000 : a5200000)

  Irp    [ Thread ] irpStack: (Mj,Mn)   DevObj  [Driver]         MDL Process
9515ad68 [aa0c04c0] irpStack: ( e, 5)  8bcb2ca0 [ \Driver\AFD] 0xaa1a3540
8bd5c548 [91deeb80] irpStack: ( e,20)  8bcb2ca0 [ \Driver\AFD] 0x91da5c40

Searching nonpaged pool (80000000 : ffc00000) for tag 0x3f707249 (Irp?)

86264a20 [86262040] irpStack: ( e, 0)  8a7b4ef0 [ \Driver\vmbus]
86278720 [91d96b80] irpStack: ( e,20)  8bcb2ca0 [ \Driver\AFD] 0x86270040
86279e48 [91d96b80] irpStack: ( e,20)  8bcb2ca0 [ \Driver\AFD] 0x86270040
862a1868 [862978c0] irpStack: ( d, 0)  8bca4030 [ \FileSystem\Npfs]
862a24c0 [86297040] irpStack: ( d, 0)  8bca4030 [ \FileSystem\Npfs]
862c3218 [9c25f740] irpStack: ( c, 2)  8b127018 [ \FileSystem\NTFS]
862c4988 [a14bf800] irpStack: ( e, 5)  8bcb2ca0 [ \Driver\AFD] 0xaa1a3540
862c57d8 [a8ef84c0] irpStack: ( d, 0)  8b127018 [ \FileSystem\NTFS] 0xa8e6f040
862c91c0 [99ac9040] irpStack: ( 3, 0)  8a7ace48 [ \Driver\vmbus] 0x9517ac40
862d2d98 [9fd456c0] irpStack: ( e, 5)  8bcb2ca0 [ \Driver\AFD] 0x9fc11780
862d6528 [9aded800] irpStack: ( c, 2)  8b127018 [ \FileSystem\NTFS]
862e3230 [00000000] Irp is complete (CurrentLocation 2 > StackCount 1)
862ec248 [862e2040] irpStack: ( d, 0)  8bca4030 [ \FileSystem\Npfs]
862f7d70 [91dd0800] irpStack: ( d, 0)  8bca4030 [ \FileSystem\Npfs]
863011f8 [00000000] Irp is complete (CurrentLocation 2 > StackCount 1)
86327008 [00000000] Irp is complete (CurrentLocation 43 > StackCount 42)
86328008 [00000000] Irp is complete (CurrentLocation 43 > StackCount 42)
86328960 [00000000] Irp is complete (CurrentLocation 43 > StackCount 42)
86329008 [00000000] Irp is complete (CurrentLocation 43 > StackCount 42)
863296d8 [00000000] Irp is complete (CurrentLocation 2 > StackCount 1)
86329960 [00000000] Irp is complete (CurrentLocation 43 > StackCount 42)
89feeae0 [00000000] irpStack: ( e, 0)  8a765030 [ \Driver\ACPI]
8a6d85d8 [99aa1040] irpStack: ( d, 0)  8b127018 [ \FileSystem\NTFS] 0x00000000
8a6dc828 [8bc758c0] irpStack: ( 4, 0)  8b127018 [ \FileSystem\NTFS] 0x00000000
8a6f42d8 [8bc728c0] irpStack: ( 4,34)  8b0b8030 [ \Driver\disk] 0x00000000
8a6f4d28 [8632e6c0] irpStack: ( 4,34) 8b0b8030 [ \Driver\disk] 0x00000000
8a767d98 [00000000] Irp is complete (CurrentLocation 6 > StackCount 5)
8a788d98 [00000000] irpStack: ( f, 0)  00000000 [00000000: Could not read device
object or _DEVICE_OBJECT not found
```

```
]
8a7911a8 [9fdb4040] irpStack: ( e, 0)  86325768 [ \Driver\DeviceApi]
8b03c3f8 [00000000] Irp is complete (CurrentLocation 2 > StackCount 1)
8b0b8bc8 [863d6040] irpStack: ( e, 0)  8a78f030 [ \Driver\vmbus]
8b0c48c0 [91da8040] irpStack: ( e, 5)  8bcb2ca0 [ \Driver\AFD] 0xaa1a3540
8b118d98 [00000000] Irp is complete (CurrentLocation 9 > StackCount 8)
8b1263b8 [00000000] Irp is complete (CurrentLocation 8 > StackCount 7)
8b174008 [aa0aab80] irpStack: ( 4, 0)  8b127018 [ \FileSystem\NTFS] 0xa15e1c40
8b194008 [aa0aab80] irpStack: ( 4, 0)  8b127018 [ \FileSystem\NTFS] 0xa15e1c40
8b196370 [8b131880] irpStack: ( e,31)  8bcb2ca0 [ \Driver\AFD]
8b1a8470 [00000000] Irp is complete (CurrentLocation 2 > StackCount 1)
8b1b3510 [9fcd1040] irpStack: ( e, 0)  86325768 [ \Driver\DeviceApi]
8b1b35b0 [a4009b80] irpStack: ( e, 0)  86325768 [ \Driver\DeviceApi]
8b1cd188 [9c3be040] irpStack: ( e, 0)  8bc73648 [ \Driver\Beep]
...
```

Некоторые IRP завершены и могут быть уничтожены очень скоро — или уже уничтожены, но, поскольку память выделяется из резервных списков, IRP-пакет еще не был заменен новым.

Для каждого IRP-пакета указывается его адрес, за которым следует программный поток, выдавший запрос. Затем в круглых скобках указываются основной и дополнительный коды функций для текущего блока стека. Любой IRP-пакет можно проанализировать командой `!irp`:

```
kd> !irp 8a6f4d28
Irp is active with 15 stacks 6 is current (= 0x8a6f4e4c)
 Mdl=8b14b250: No System Buffer: Thread 8632e6c0:  Irp stack trace.
     cmd  flg cl Device   File     Completion-Context
[N/A(0), N/A(0)]
            0  0 00000000 00000000 00000000-00000000

                        Args: 00000000 00000000 00000000 00000000
 [N/A(0), N/A(0)]
            0  0 00000000 00000000 00000000-00000000

                        Args: 00000000 00000000 00000000 00000000
 [N/A(0), N/A(0)]
            0  0 00000000 00000000 00000000-00000000

                        Args: 00000000 00000000 00000000 00000000
 [N/A(0), N/A(0)]
            0  0 00000000 00000000 00000000-00000000

                        Args: 00000000 00000000 00000000 00000000
 [N/A(0), N/A(0)]
            0  0 00000000 00000000 00000000-00000000

                        Args: 00000000 00000000 00000000 00000000
           >[IRP_MJ_WRITE(4), N/A(34)]
            14 e0 8b0b8030 00000000 876c2ef0-00000000 Success Error Cancel
                \Driver\disk partmgr!PmIoCompletion
                        Args: 0004b000 00000000 4b3a0000 00000002
 [IRP_MJ_WRITE(4), N/A(3)]
          14 e0 8b0fc058 00000000 876c36a0-00000000 Success Error Cancel
               \Driver\partmgr       partmgr!PartitionIoCompletion
                        Args: 4b49ace4 00000000 4b3a0000 00000002
 [IRP_MJ_WRITE(4), N/A(0)]
          14 e0 8b121498 00000000 87531110-8b121a30 Success Error Cancel
               \Driver\partmgr       volmgr!VmpReadWriteCompletionRoutine
```

```
                        Args: 0004b000 00000000 2bea0000 00000002
 [IRP_MJ_WRITE(4), N/A(0)]
           4 e0 8b121978 00000000 82d103e0-8b1220d9 Success Error Cancel
               \Driver\volmgr        fvevol!FvePassThroughCompletionRdpLevel2
                        Args: 0004b000 00000000 4b49acdf 00000000
 [IRP_MJ_WRITE(4), N/A(0)]
           4 e0 8b122020 00000000 82801a40-00000000 Success Error Cancel
               \Driver\fvevol        rdyboost!SmdReadWriteCompletion
                        Args: 0004b000 00000000 2bea0000 00000002
 [IRP_MJ_WRITE(4), N/A(0)]
           4 e1 8b118538 00000000 828637d0-00000000 Success Error Cancel pending
               \Driver\rdyboost      iorate!IoRateReadWriteCompletion
                        Args: 0004b000 3fffffff 2bea0000 00000002
 [IRP_MJ_WRITE(4), N/A(0)]
           4 e0 8b11ab80 00000000 82da1610-8b1240d8 Success Error Cancel
               \Driver\iorate        volsnap!VspRefCountCompletionRoutine
                        Args: 0004b000 00000000 2bea0000 00000002
 [IRP_MJ_WRITE(4), N/A(0)]
           4 e1 8b124020 00000000 87886ada-89aec208 Success Error Cancel pending
               \Driver\volsnap       NTFS!NtfsMasterIrpSyncCompletionRoutine
                        Args: 0004b000 00000000 2bea0000 00000002
 [IRP_MJ_WRITE(4), N/A(0)]
           4 e0 8b127018 a6de4bb8 871227b2-9ef8eba8 Success Error Cancel
               \FileSystem\NTFS           FLTMGR!FltpPassThroughCompletion
                        Args: 0004b000 00000000 00034000 00000000
 [IRP_MJ_WRITE(4), N/A(0)]
           4  1 8b12a3a0 a6de4bb8 00000000-00000000    pending
               \FileSystem\FltMgr
                        Args: 0004b000 00000000 00034000 00000000

Irp Extension present at 0x8a6f4fb4:
```

Перед нами громадный IRP-пакет с 15 блоками стека (текущим является блок 6, выделенный жирным шрифтом; отладчик выделяет его символом >). Для каждого блока стека указывается основной и дополнительный код функции вместе с информацией объекта устройства и адресами процедур завершения.

Теперь нам нужно выяснить, какому объекту устройства адресован IRP-пакет. В этом нам поможет команда `!devobj`, в качестве параметра которой указывается взятый из активного блока стека адрес объекта устройства:

```
kd> !devobj 8b0b8030
Device object (8b0b8030) is for:
 DR0 \Driver\disk DriverObject 8b0a7e30
Current Irp 00000000 RefCount 1 Type 00000007 Flags 01000050
Vpb 8b0fc420 SecurityDescriptor 87da1b58 DevExt 8b0b80e8 DevObjExt
            8b0b8578 Dope
8b0fc3d0
ExtensionFlags (0x00000800)  DOE_DEFAULT_SD_PRESENT
Characteristics (0x00000100)  FILE_DEVICE_SECURE_OPEN
AttachedDevice (Upper) 8b0fc058 \Driver\partmgr
AttachedTo (Lower) 8b0a4d10 \Driver\storflt
Device queue is not busy.
```

Наконец, для получения подробной информации о программном потоке и выдавшем IRP-пакет процессе используется команда `!thread`:

```
kd> !thread 8632e6c0
THREAD 8632e6c0  Cid 0004.0058  Teb: 00000000 Win32Thread: 00000000 WAIT:
(Executive) KernelMode Non-Alertable
    89aec20c  NotificationEvent
IRP List:
    8a6f4d28: (0006,02d4) Flags: 00060043  Mdl: 8b14b250
Not impersonating
DeviceMap                 87c025b0
Owning Process            86264280       Image:         System
Attached Process          N/A            Image:         N/A
Wait Start TickCount      8083           Ticks: 1 (0:00:00:00.015)
Context Switch Count      2223           IdealProcessor: 0
UserTime                  00:00:00.000
KernelTime                00:00:00.046
Win32 Start Address nt!ExpWorkerThread (0x81e68710)
Stack Init 89aecca0 Current 89aebeb4 Base 89aed000 Limit 89aea000 Call 00000000
Priority 13 BasePriority 13 PriorityDecrement 0 IoPriority 2 PagePriority 5
```

Запрос ввода/вывода к одноуровневому драйверу

В этом разделе рассматриваются запросы ввода/вывода к одноуровневому драйверу устройства в режиме ядра. На рис. 6.14 представлен типичный процесс обработки IRP-пакета для такого драйвера.

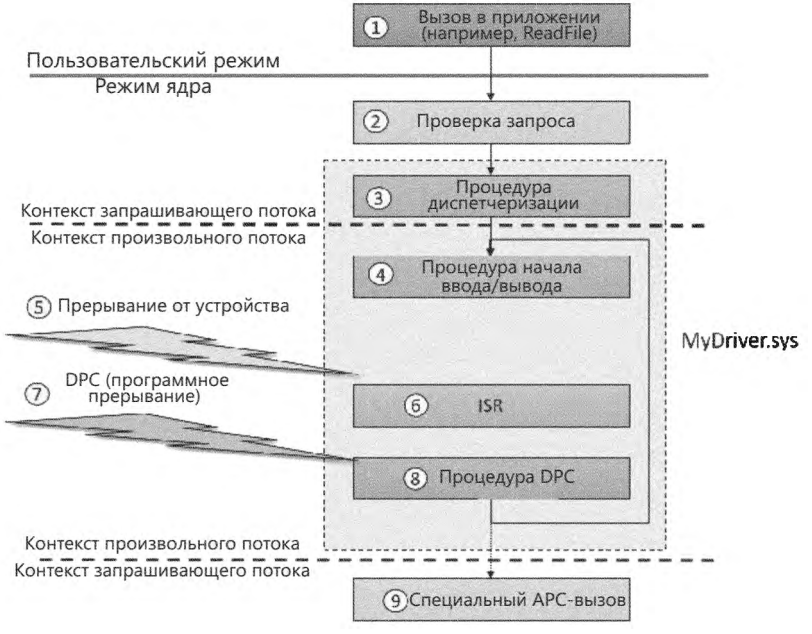


Рис. 6.14. Типичный процесс одноуровневой обработки запросов к физическим устройствам

Прежде чем подробно рассматривать различные фазы на рис. 6.14, стоит сделать несколько общих замечаний.

- На схеме присутствуют горизонтальные разделительные линии двух видов. Первая (сплошная) линия — стандартный разделитель между пользовательским режимом и режимом ядра. Вторая (пунктирная) линия отделяет код, выполняемый в контексте запрашивающего потока, от кода, выполняемого в контексте произвольного потока. Эти контексты определяются следующим образом:
  - В контексте запрашивающего потока выполняется поток, запросивший операцию ввода/вывода. Это важно, потому что, если текущим является поток, выдавший исходный запрос, это означает, что контекст соответствует исходному процессу, — а значит, появляется возможность напрямую обращаться к адресному пространству пользовательского режима с пользовательским буфером, переданным операции ввода/вывода.
  - Контекст произвольного потока означает, что поток, в котором выполняются эти функции, может быть любым. А конкретнее это означает, что он с большой вероятностью отличен от запрашивающего потока, поэтому видимое адресное пространство пользовательского режима будет отличаться от исходного. В этом контексте обращение к пользовательскому буферу с адресом пользовательского режима будет иметь катастрофические последствия. В следующем разделе вы увидите, как решается эта проблема.

ПРИМЕЧАНИЕ Из объяснения фаз на рис. 6.14 станет ясно, почему разделительные линии находятся именно в этих местах.

- Большой прямоугольник, состоящий из четырех блоков (процедура диспетчеризации, процедура начала ввода/вывода, ISR и процедура DPC), обозначает код, предоставленный драйвером. Все остальные блоки предоставляются системой.
- Предполагается, что физическое устройство выполняет операции по одной; это справедливо для многих типов устройств. Даже если устройство может обрабатывать сразу несколько запросов, основная последовательность действий остается неизменной.

Переходим к последовательности событий, представленных на рис. 6.14.

1. Клиентское приложение вызывает функцию Windows API — например, `ReadFile`. `ReadFile` вызывает платформенную функцию `NtReadFile` (в файле Ntdll.dll), что приводит к переходу потока в режим ядра (эти фазы уже были рассмотрены в этой главе).
2. Диспетчер ввода/вывода в своей реализации `NtReadFile` выполняет простейшие проверки запроса (например, проверяется доступность буфера, переданного клиентом, с учетом защиты страницы). Затем диспетчер ввода/вывода находит соответствующий драйвер (с использованием переданного файлового дескриптора), создает и инициализирует IRP-пакет, а затем обращается с вызо-

вом к драйверу в соответствующей процедуре диспетчеризации (в данном примере с индексом `IRP_MJ_READ`), используя вызов `IoCallDriver` с IRP-пакетом.

3. В этой точке драйвер впервые видит IRP-пакет. Вызов почти всегда выполняется в запрашивающем потоке; единственное исключение — если верхний фильтр задержит IRP и позднее вызовет `IoCallDriver` из другого потока. Будем считать, что в нашем случае это *не* так (в большинстве случаев с физическими устройствами этого не происходит; даже если верхние фильтры присутствуют, они выполняют некоторую обработку и немедленно вызывают драйвер нижнего уровня из того же потока). Процедура обратного вызова в драйвере решает две задачи: во-первых, она выполняет дополнительные проверки, которые не могут быть выполнены диспетчером ввода/вывода. Например, драйвер может проверить, достаточен ли размер буфера, предоставленного для операции чтения или записи; для операции `DeviceIoControl` драйвер может проверить, поддерживается ли указанный код управляющей операции. Если любая такая проверка завершится неудачей, драйвер завершает IRP (`IoCompleteRequest`) с кодом ошибки и немедленно возвращает управление. Если же проверки проходят успешно, драйвер вызывает свою процедуру начала ввода/вывода, инициируя выполнение операции. Но если физическое устройство в настоящий момент занято (обработкой предыдущего IRP-пакета), IRP-пакет ставится в очередь, находящуюся под управлением драйвера, а операция возвращает код `STATUS_PENDING` без завершения IRP. Диспетчер ввода/вывода обрабатывает этот сценарий при помощи функции `IoStartPacket`, которая проверяет бит занятости объекта устройства; если устройство занято, то IRP-пакет ставится в очередь (которая также является частью структуры объекта устройства). Если же устройство не занято, то он устанавливает бит занятости и вызывает зарегистрированную процедуру начала ввода/вывода (вспомните соответствующее поле в объекте драйвера, который был инициализирован в `DriverEntry`). Даже если драйвер решит не использовать `IoStartPacket`, логика останется похожей.

4. Если устройство не занято, процедура ввода/вывода вызывается из процедуры диспетчеризации напрямую — это означает, что вызов совершается запрашивающим потоком. Однако на рис. 6.14 показано, что процедура ввода/вывода вызывается в контексте произвольного потока; в общем случае дело обстоит именно так (вы убедитесь в этом, когда мы будем рассматривать процедуру DPC на шаге 8). Цель процедуры ввода/вывода — получение параметров, относящихся к IRP, и использование их для программирования физического устройства (например, запись в порты или регистры с использованием функций HAL для работы с оборудованием — `WRITE_PORT_UCHAR`, `WRITE_REGISTER_ULONG` и т. д.). После завершения процедуры начала ввода/вывода вызов возвращает управление, в драйвере никакой конкретный код не выполняется, оборудование работает и «делает свое дело». Во время работы физического устройства к нему могут поступить новые запросы из того же потока (при использовании асинхронных операций) или других потоков, которые также открыли дескрипторы для устройства. В этом случае процедура диспетчеризации понимает, что устрой-

ство занято, и ставит IRP-пакет в очередь IRP (как упоминалось ранее, это можно сделать вызовом `IoStartPacket`).

5. Завершив текущую операцию, устройство инициирует прерывание. Обработчик режима ядра сохраняет контекст процессора для потока, выполнявшегося на процессоре, который был выбран для обработки прерывания, повышает уровень IRQL этого процессора до уровня IRQL, связанного с прерыванием (DIRQL), и переходит к зарегистрированному ISR-обработчику для устройства.

6. ISR-обработчик, выполняемый на уровне IRQL устройства (см. выше пункт 2), выполняет минимум возможной работы: он приказывает устройству остановить сигнал прерывания и получить данные статуса (или другую необходимую информацию) от физического устройства. В завершение своей работы ISR ставит в очередь DPC-вызов для дальнейшей обработки с более низким уровнем IRQL. Преимущество использования DPC для выполнения основных операций с устройством заключается в том, что до обработки DPC могут произойти все заблокированные прерывания, уровень IRQL которых находится между уровнями Device и Dispatch/DPC (2). В результате прерывания промежуточного уровня будут обслужены быстрее, что приводит к сокращению задержки в системе.

7. После того как прерывание будет снято, ядро замечает, что очередь DPC не пуста; поэтому оно использует программное прерывание на уровне IRQL DPC_LEVEL (2) для перехода к циклу обработки DPC.

8. В какой-то момент DPC-вызов извлекается из очереди и обрабатывается с IRQL 2; обычно при этом выполняются две основные операции:

   - Он получает новый IRP-пакет из очереди (если он есть) и запускает новую операцию для устройства. Это делается прежде всего для того, чтобы устройство не простаивало слишком долго. Если процедура диспетчеризации использовала функцию `IoStartPacket`, то процедура DPC вызывает ее функцию-двойника `IoStartNextPacket`, которая делает именно это. Если IRP-пакет доступен, то процедура начала ввода/вывода вызывается из DPC. Вот почему в общем случае процедура начала ввода/вывода вызывается в контексте произвольного потока. Если IRP-пакетов в очереди нет, то устройство помечается как свободное — т. е. готовое к следующему поступлению запроса.

   - Он завершает IRP-пакет, операция которого только что была завершена драйвером, вызовом `IoCompleteRequest`. С этого момента драйвер не отвечает за IRP-пакет, и обращаться к IRP-пакету не следует, так как он может быть освобожден в любой момент после вызова. Функция `IoCompleteRequest` вызывает все зарегистрированные процедуры завершения. Наконец, диспетчер ввода/вывода освобождает IRP-пакет (на самом деле он использует для этого собственную процедуру завершения).

9. Исходный запрашивающий поток необходимо уведомить о завершении. Так как текущий поток, в котором выполняется DPC, может быть произвольным,

он не совпадает с исходным потоком, с его исходным адресным пространством. Чтобы выполнить код в контексте запрашивающего потока, выдается специальный APC-вызов ядра. APC (асинхронный вызов процедуры) — функция, которая принудительно выполняется в контексте конкретного потока. Когда запрашивающему потоку выделяется процессорное время, сначала выполняется специальный APC-вызов ядра (на уровне IRQL APC_LEVEL=1). Он выполняет все необходимые действия: освобождение потока от ожидания, инициирование события, зарегистрированного для асинхронной операции, и т. д. (За дополнительной информацией об APC обращайтесь к главе 8 части 2.)

И последнее замечание по поводу завершения ввода/вывода: асинхронные функции `ReadFileEx` и `WriteFileEx` позволяют вызывающей стороне задать функцию обратного вызова в параметре. Если вызывающая сторона это сделала, то диспетчер ввода/вывода ставит APC-вызов пользовательского режима в очередь APC вызывающего потока. Это позволяет вызывающей стороне задать функцию, которая должна вызываться при завершении или отмене запроса ввода/вывода. APC-функции завершения пользовательского режима выполняются в контексте запрашивающего потока и срабатывают только тогда, когда поток входит в состояние ожидания с предупреждением (вызовом таких функций, как `SleepEx`, `WaitForSingleObjectEx` или `WaitForMultipleObjectsEx`).

Обращение к буферу в пользовательском адресном пространстве

Как показано на рис. 6.14, в обработке IRP задействованы четыре основные функции драйвера. Некоторым из них может потребоваться доступ к буферу в пользовательском пространстве, представленному клиентским приложением. Когда приложение или драйвер устройства неявно создает IRP-пакет при помощи системных функций `NtReadFile`, `NtWriteFile` или `NtDeviceIoControlFile` (или функций Windows API, соответствующих этим функциям, — `ReadFile`, `WriteFile` или `DeviceIoControl`), указатель на буфер пользователя передается в поле `UserBuffer` тела IRP. Однако прямое обращение к этому буферу возможно только в контексте вызывающего потока (при условии видимости адресного пространства клиентского процесса) и на уровне IRQL 0 (когда подкачка может обрабатываться стандартным образом).

Как было показано в предыдущем разделе, только процедура диспетчеризации удовлетворяет критерию выполнения в контексте запрашивающего потока и на уровне IRQL 0. Причем даже это не всегда так — верхний фильтр может задержать IRP-пакет и не передать его вниз немедленно, что может привести к его передаче вниз с использованием другого потока, и даже когда уровень IRQL процессора будет равен 2 и выше.

Другие три функции (начало ввода/вывода, ISR, DPC) очевидным образом выполняются в произвольном потоке (это может быть любой поток) и с уровнем IRQL 2 (DIRQL для ISR). Прямое обращение к пользовательскому буферу из любой из этих процедур приведет к катастрофическим последствиям, потому что:

- поскольку уровень IRQL равен 2 и выше, подкачка запрещена. Так как буфер (или его часть) может находиться в выгруженной части памяти, обращение к нерезидентной памяти приведет к сбою системы;
- так как поток, в котором выполняются эти функции, может быть любым, а следовательно, видимым может быть адресное пространство произвольного процесса, исходный адрес не имеет смысла, а обращение к нему приведет к нарушению прав доступа или, еще хуже, к обращению к данным произвольного процесса (родительского процесса для потока, который выполнялся на этот момент).

Очевидно, должен существовать безопасный способ обращения к пользовательскому буферу из любой из этих процедур. Диспетчер ввода/вывода предоставляет два механизма, в которых он берет на себя всю черную работу. Это так называемый «буферизованный ввод/вывод» и «прямой ввод/вывод». Третий вариант, который на самом деле трудно назвать вариантом, называется «пассивным вводом/выводом»; здесь диспетчер ввода/вывода не делает ничего особенного и предоставляет драйверу возможность разбираться с проблемой самостоятельно.

Драйвер выбирает нужный метод следующим образом:

- Для запросов чтения и записи (`IRP_MJ_READ` и `IRP_MJ_WRITE`) он устанавливает в поле `Flags` (с использованием логической операции OR, чтобы не затронуть другие флаги) объекта устройства (`DEVICE_OBJECT`) значение `DO_BUFFERED_IO` (для буферизованного ввода/вывода) или `DO_DIRECT_IO` (для прямого ввода/вывода). Если ни один из флагов не установлен, то ни один из механизмов не используется по умолчанию. (`DO` — сокращение от «Device Object», т. е. «объект устройства».)
- Для запросов управления вводом/выводом (`IRP_MJ_DEVICE_CONTROL`) управляющие коды строятся с использованием макроса `CTL_CODE`, где некоторые биты обозначают метод буферизации. Таким образом, метод буферизации может задаваться на уровне управляющих кодом, а это очень полезно.

Ниже каждый метод буферизации будет описан более подробно.

Буферизованный ввод/вывод. При буферизованном вводе/выводе диспетчер ввода/вывода выделяет зеркальный буфер, размер которого равен размеру пользовательского буфера, в невыгружаемом пуле, и сохраняет указатель на новый буфер в поле `AssociatedIrp.SystemBuffer` тела IRP. На рис. 6.15 показаны основные стадии буферизованного ввода/вывода для операции чтения (с операцией записи дело обстоит аналогично).

Драйвер может обратиться к системному буферу (адрес *q* на рис. 6.15) из любого потока и на любом уровне IRQL:

- адрес находится в системном пространстве, а это означает, что он действителен в контексте любого процесса;
- буфер выделяется из невыгружаемого пула, поэтому ошибка страниц невозможна.

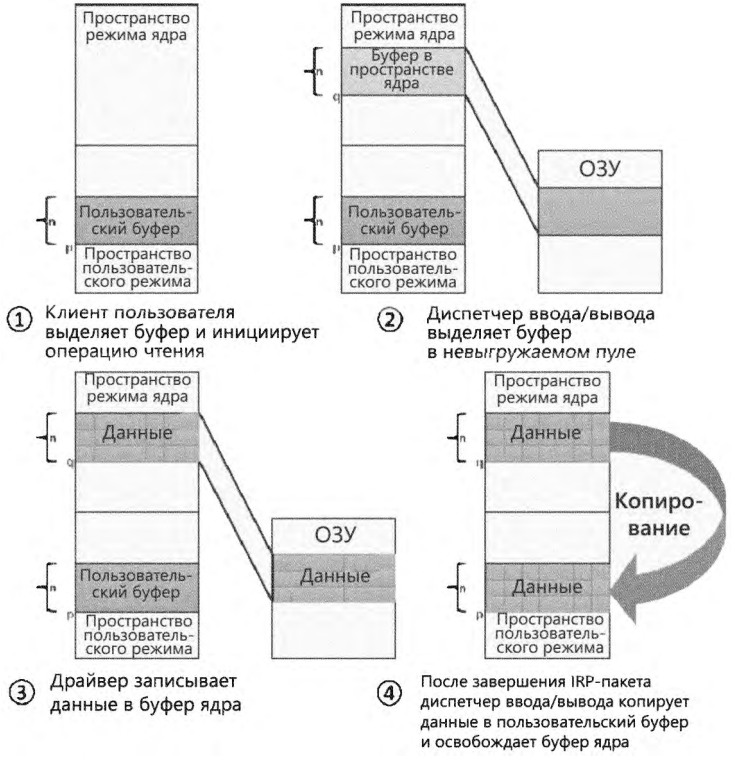


Рис. 6.15. Буферизованный ввод/вывод

Для операций записи диспетчер ввода/вывода копирует данные буфера со стороны вызова в выделенный буфер при создании IRP. Для операций чтения диспетчер ввода/вывода копирует данные из выделенного буфера в пользовательский буфер при завершении IRP (с использованием специального APC-вызова ядра), после чего освобождает выделенный буфер.

Буферизованный ввод/вывод очень прост в использовании, потому что диспетчер ввода/вывода делает практически все за вас. Его основной недостаток — необходимость копирования, которое неэффективно работает для больших буферов. Буферизованный ввод/вывод обычно применяется в тех случаях, когда размер буфера не превышает одной страницы (4 Кбайт) и когда устройство не поддерживает прямой доступ к памяти (DMA), потому что DMA используется для передачи данных с устройства в память или, наоборот, без участия процессора — тогда как с буферизованным вводом/выводом копирование всегда осуществляется процессором, из-за чего DMA становится бессмысленным.

Прямой ввод/вывод предоставляет драйверу возможность обращаться к пользовательскому буферу напрямую, без лишнего копирования. На рис. 6.16 показаны основные стадии прямого ввода/вывода для операции чтения или записи.

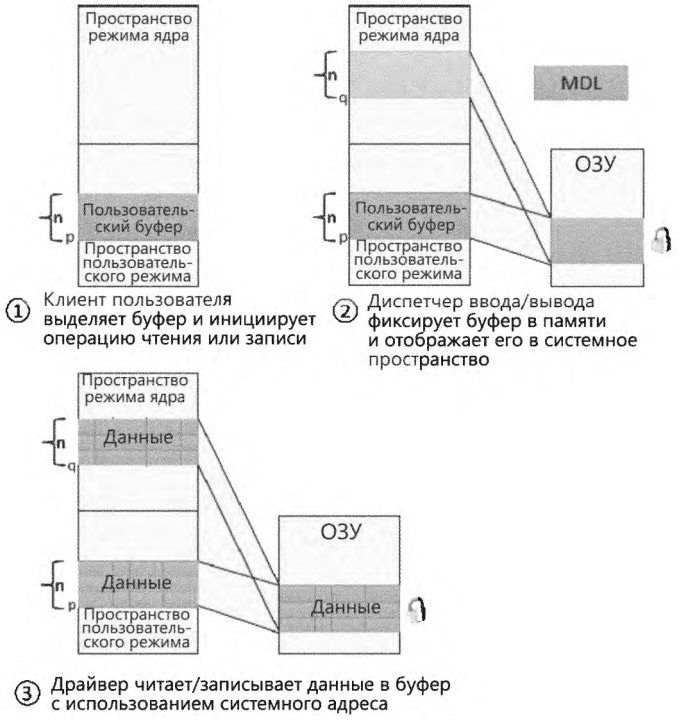


Рис. 6.16. Прямой ввод/вывод

Когда диспетчер ввода/вывода создает IRP-пакет, он фиксирует пользовательский буфер в памяти (т. е. делает его невыгружаемым) вызовом функции `MmProbeAndLockPages` (документированной в WDK). Диспетчер ввода/вывода сохраняет описание памяти в форме *списка дескрипторов памяти*, или *MDL* (Memory Descriptor List) — структуры, которая описывает физическую память, занимаемую буфером. Адрес MDL сохраняется в поле `MdlAddress` тела IRP. Устройствам, поддерживающим DMA, достаточно физических описаний буферов, и для работы таких устройств достаточно MDL. Но если драйвер должен обратиться к содержимому буфера, он может отобразить буфер в системное адресное пространство функцией `MmGetSystemAddressForMdlSafe` с передачей полученной структуры MDL. Полученный указатель (*q* на рис. 6.16) может безопасно использоваться в контексте произвольного потока (это системный адрес) и на любом уровне IRQL (буфер не может выгружаться). Фактически пользовательский буфер отображается в памяти дважды; прямой адрес (*p* на рис. 6.16) может использоваться только в контексте исходного процесса, но второе отображение в системное пространство доступно в любом контексте. Как только IRP-пакет будет завершен, диспетчер ввода/вывода отменяет фиксацию буфера в памяти (так, что он снова может выгружаться из памяти) вызовом `MmUnlockPages` (документированным в WDK).

Прямой ввод/вывод эффективен для больших буферов (размер которых больше одной страницы), потому что он не требует копирования — особенно при передаче данных DMA (по той же причине).

Ввод/вывод без управления (neither I/O). При вводе/выводе без управления диспетчер ввода/вывода не пытается управлять буфером. Вместо этого все управление буфером доверяется драйверу устройства, который может вручную выполнить все действия, выполняемые диспетчером ввода/вывода для других типов управления буфером. В некоторых случаях обращения к буферу в процедуре диспетчеризации оказывается достаточно, и драйвер может обойтись пассивным вводом/выводом. Главное преимущество пассивного ввода/вывода — полное отсутствие непроизводительных затрат.

Драйверы, использующие ввод/вывод без управления для обращения к буферам, которые могут находиться в пользовательском пространстве, должны принять особые меры к тому, чтобы адреса буферов были действительными и не пытались обращаться к памяти режима ядра. Впрочем, передача скалярных значений абсолютно безопасна — хотя лишь очень немногие драйверы смогут обойтись только скалярными значениями. Если не принять должных мер, это может привести к сбоям или появлению уязвимостей в системе безопасности: приложения получат доступ к памяти режима ядра или смогут внедрить свой код в ядро. Функции `ProbeForRead` и `ProbeForWrite`, предоставляемые ядром для драйверов, проверяют, что буфер находится в адресном пространстве полностью в части пользовательского режима. Чтобы избежать ошибок при обращении к недействительным адресам пользовательского режима, драйверы могут обращаться к буферам пользовательского режима, защищенным средствами структурированной обработки исключений (SEH, Structured Exception Handling), выраженными блоками `__try/__except` в C/C++. Эти блоки перехватывают ошибки недействительных обращений и преобразуют их в коды ошибок, возвращаемые приложению. (За дополнительной информацией о SEH обращайтесь к главе 8 части 2). Кроме того, драйверы также должны сохранять все входные данные в буфере ядра, не полагаясь на адреса пользовательского режима, потому что вызывающая сторона всегда может изменить данные втайне от драйвера, даже если сам адрес памяти остается действительным.

Синхронизация

Драйверы должны синхронизировать свой доступ к глобальным данным драйверов и регистрам устройств по двум причинам:

- выполнение драйвера может быть прервано программными потоками с более высоким приоритетом, по истечении выделенного времени или прерыванием с более высоким IRQL-уровнем;
- в многопроцессорных системах Windows может выполнять код драйвера сразу на нескольких процессорах.

Отсутствие синхронизации может привести к повреждению данных, например из-за того, что код драйвера устройства, выполняемый на нулевом IRQL-уровне (passive), при инициации операции ввода или вывода вызывающей программой может быть прерван устройством. Это приводит к выполнению ISR-процедуры драйвера устройства. И если этот драйвер в этот момент вносил какие-то изменения в данные, которые модифицируют ISR, например регистры устройства, память кучи или статические данные, выполнение ISR может сопровождаться повреждением этих данных.

Чтобы избежать подобной ситуации, написанный для Windows драйвер устройства должен синхронизировать свое обращение ко всем данным, к которым возможен доступ на более чем одном IRQL-уровне. Перед попыткой обновления общих данных драйвер устройства должен заблокировать все остальные программные потоки (или все процессоры в многопроцессорной системе), запретив им вносить изменения в рассматриваемую структуру данных.

В однопроцессорной системе синхронизация между двумя и более функциями, выполняемыми на разных уровнях IRQL, реализуется достаточно просто. Такая функция должна просто поднимать IRQL (`KeRaiseIrql`) до наивысшего уровня, на котором выполняются эти функции. Например, для синхронизации между процедурой диспетчеризации (IRQL 0) и процедурой DPC (IRQL 2) процедура диспетчеризации должна поднять IRQL до уровня 2 перед обращением к общим данным. Если возникнет необходимость синхронизации между DPC и ISR, DPC поднимает IRQL до уровня Device (эта информация предоставляется драйверу, когда диспетчер PnP передает драйверу информацию об аппаратных ресурсах, с которыми связан драйвер). В многопроцессорных системах повышения IRQL недостаточно, потому что другая процедура — например, ISR — может обслуживаться на другом процессоре (помните, что IRQL является атрибутом процессора, а не глобальным системным атрибутом).

Для реализации синхронизации с высокими уровнями IRQL между процессорами ядро предоставляет специализированный объект синхронизации: *спин-блокировки* (spinlock). Сейчас мы кратко рассмотрим спин-блокировки в контексте синхронизации драйверов. (Полное описание спин-блокировок откладывается до главы 8.) В принципе спин-блокировка напоминает мьютекс (также подробно рассматривается в главе 8) в том смысле, что она тоже позволяет блоку кода обращаться к общим данным, но работает и используется она иначе. В табл. 6.3 приведена сводка различий между мьютексами и спин-блокировками.

Спин-блокировка — обычный бит памяти, обращение к которым осуществляется атомарной операцией проверки и изменения. Спин-блокировка может принадлежать процессору или быть свободной (т. е. не иметь владельца). Как показано в табл. 6.3, спин-блокировки необходимы при синхронизации на высоких уровнях IRQL (≥ 2), потому что мьютекс не может использоваться в таких случаях из-за необходимости участия планировщика (как говорилось ранее, планировщик не может активизироваться на процессоре с уровнем IRQL 2 и выше). Именно поэтому ожидание спин-блокировки является операцией активного ожидания: поток не может

перейти в нормальное состояние ожидания, потому что это подразумевает пробуждение планировщика и переключение на другой поток на данном процессоре.

Таблица 6.3. Мьютексы и спин-блокировки

	Мьютекс	**Спин-блокировка**
Синхронизация	Один поток из произвольного количества потоков, получивший разрешение, входит в критическую область и обращается к общим данным	Один процессор из произвольного количества процессоров, получивший разрешение, входит в критическую область и обращается к общим данным
Возможность использования на IRQL	< DISPATCH_LEVEL (2)	>= DISPATCH_LEVEL (2)
Тип ожидания	Нормальное (т. е. без расходования тактов процессора во время ожидания)	Активное (т. е. процессор постоянно проверяет состояние спин-блокировки, пока она не освободится)
Принадлежность	Отслеживается поток-владелец, допускается рекурсивное получение	Процессор-владелец не отслеживается, рекурсивное получение приведет к взаимной блокировке

Получение спин-блокировки процессором всегда выполняется в два этапа. Сначала уровень IRQL поднимается до уровня IRQL, на котором должна происходить синхронизация — т. е. до наивысшего уровня IRQL, на котором выполняется синхронизируемая функция. Например, синхронизация между процедурой диспетчеризации IRQL (0) и DPC (2) потребует повышения IRQL до 2; синхронизация между DPC (2) и ISR (DIRQL) потребует повышения IRQL до DIRQL (IRQL для этого конкретного прерывания). Затем происходит попытка получения спин-блокировки посредством атомарной проверки и установки бита спин-блокировки.

ПРИМЕЧАНИЕ Этапы получения спин-блокировки в этом описании несколько упрощены, и из него исключены некоторые детали, не представляющие интереса для обсуждения. Более полное описание спин-блокировок приведено в главе 8 части 2.

Как вы вскоре увидите, функции, получающие спин-блокировки, определяют уровень IRQL для синхронизации.

На рис. 6.17 изображена упрощенная схема двухшагового процесса получения спин-блокировки.

При синхронизации на уровне IRQL 2 — например, между процедурой диспетчеризации и DPC или между DPC и другой процедурой DPC (выполняемой на другом процессоре, конечно) — ядро предоставляет функции `KeAcquireSpinLock` и `KeReleaseSpinLock` (существуют и другие варианты, описанные в главе 8). Эти функции выполняют действия на рис. 6.17 для «соответствующего уровня IRQL» равного 2. В этом случае драйвер должен выделить память для спин-блокировки (`KSPIN_LOCK`,

4 байта в 32-разрядных системах, 8 байтов в 64-разрядных системах) — обычно в расширении устройства (где хранятся данные устройства, находящиеся под управлением драйвера) — и инициализировать ее вызовом `KeInitializeSpinLock`.

Рис. 6.17. Получение спин-блокировки

Для синхронизации между любой функцией (например, DPC или процедурой диспетчеризации) и ISR приходится использовать другие функции. В каждом объекте прерывания (`KINTERRUPT`) содержится спин-блокировка, которая получается перед выполнением ISR (отсюда следует, что один ISR-обработчик не может выполняться параллельно на других процессорах. Синхронизация в этом случае будет осуществляться с этой конкретной спин-блокировкой (создавать другую не надо), которая может быть неявно получена функцией `KeAcquireInterruptSpinLock` и освобождена функцией `KeReleaseInterruptSpinLock`. Другой вариант — использование функции `KeSynchronizeExecution`; ей передается функция обратного вызова, предоставленная драйвером, которая вызывается между получением и освобождением спин-блокировки прерывания.

Вероятно, к этому моменту вы уже поняли, что, хотя ISR-обработчики требуют особого внимания, к любым данным, используемым драйвером устройства, может обратиться тот же драйвер устройства (одна из его функций) с другого процессора. Следовательно, код драйвера устройства должен синхронизировать свое использование любых глобальных или общих данных либо обращения к самому физическому устройству.

Запросы ввода/вывода к многоуровневым драйверам

В разделе «Последовательность обработки IRP» представлены общие средства обработки IRP драйверами устройств, ориентированные на стандартный узел устройств WDM. В предыдущем разделе был описан процесс обработки запроса на ввод/вы-

вод, адресованного простому устройству, управляемому единственным драйвером. Обработка ввода/вывода в случае устройств, работающих с файлами, или запросов к другим многоуровневым драйверам происходит примерно так же, но стоит особо рассмотреть запросы, предназначенные для драйверов файловой системы. На рис. 6.18 показан упрощенный пример прохождения через многоуровневые драйверы асинхронного запроса на ввод/вывод, предназначенного для нефизических устройств. В данном случае рассматривается диск, управляемый файловой системой.

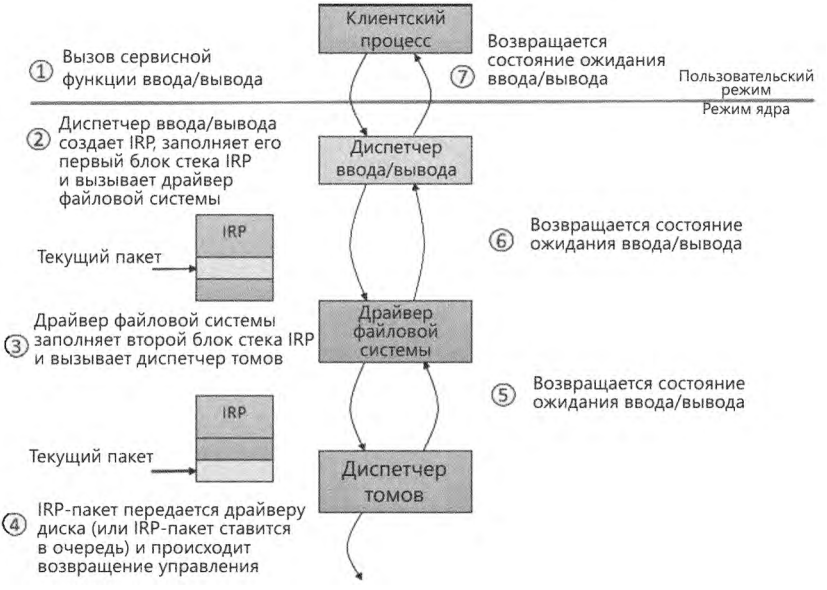


Рис. 6.18. Постановка в очередь асинхронного запроса к многоуровневым драйверам

В очередной раз диспетчер ввода/вывода получает запрос и создает для его представления IRP-пакет. Но на этот раз он передается драйверу файловой системы, который и получает с этого момента полный контроль над операцией ввода/вывода. В зависимости от того, какой тип запроса поступает от вызывающей программы, файловая система посылает драйверу диска уже имеющийся IRP-пакет или генерирует дополнительные IRP-пакеты и поочередно их передает.

Если полученный новый запрос допускает преобразование в один тривиальный запрос к устройству, скорее всего, файловая система будет использовать IRP-пакет повторно. К примеру, при получении от приложения запроса на чтение первых 512 байтов хранящегося на диске файла файловая система NTFS вызовет диспетчер томов и прикажет ему прочитать один сектор тома от начала файла.

После того как DMA-адаптер дискового контроллера завершит передачу данных, этот контроллер генерирует прерывание, вызывая ISR-процедуру, которая выдает запрос на обратный DPC-вызов для завершения IRP-процедуры (рис. 6.19).

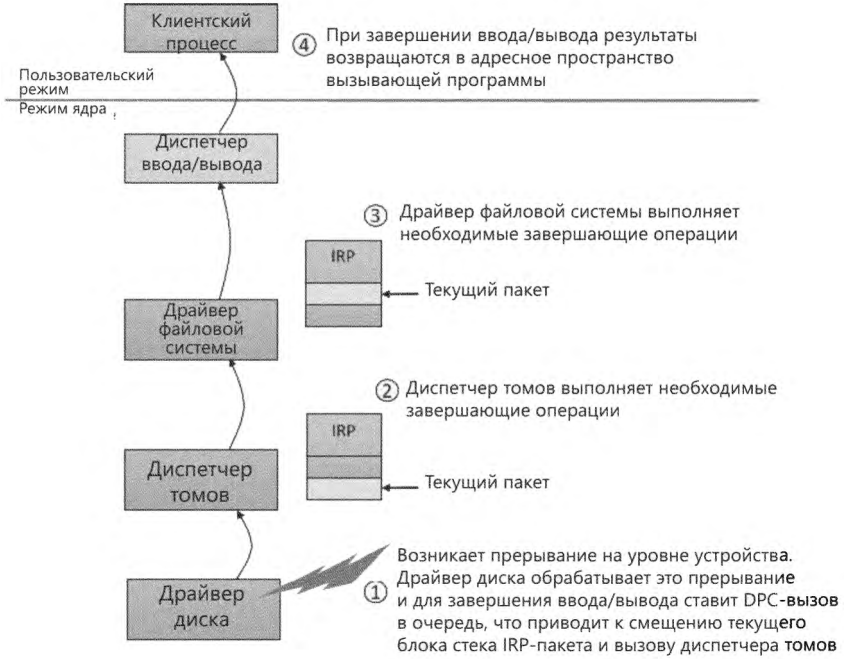


Рис. 6.19. Завершение многоуровневого запроса ввода/вывода

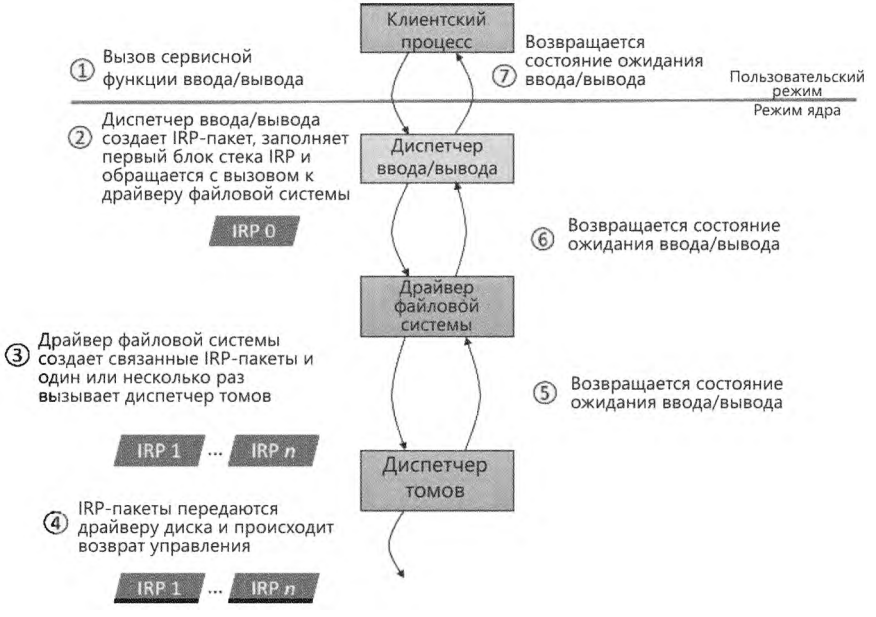


Рис. 6.20. Постановка в очередь связанных IRP-пакетов

Вместо повторного использования одного IRP-пакета файловая система может создать группу связанных IRP-пакетов, которые будут параллельно обрабатываться в рамках одного запроса ввода/вывода. К примеру, если данные, которые требуется считать из файла, разбросаны по диску, драйвер файловой системы может создать несколько IRP-пакетов, каждый из которых будет считывать данные из своего сектора. Организация подобной очереди показана на рис. 6.20.

Драйвер файловой системы передает связанные IRP-пакеты диспетчеру томов, который отправляет их драйверу дискового устройства. Драйвер же ставит их в очередь этого устройства. Они обрабатываются по одному, а за возвращаемыми данными следит драйвер файловой системы. После завершения всех связанных IRP-пакетов подсистема ввода/вывода завершает исходный IRP-пакет и возвращает управление вызывающей программе, как показано на рис. 6.21.

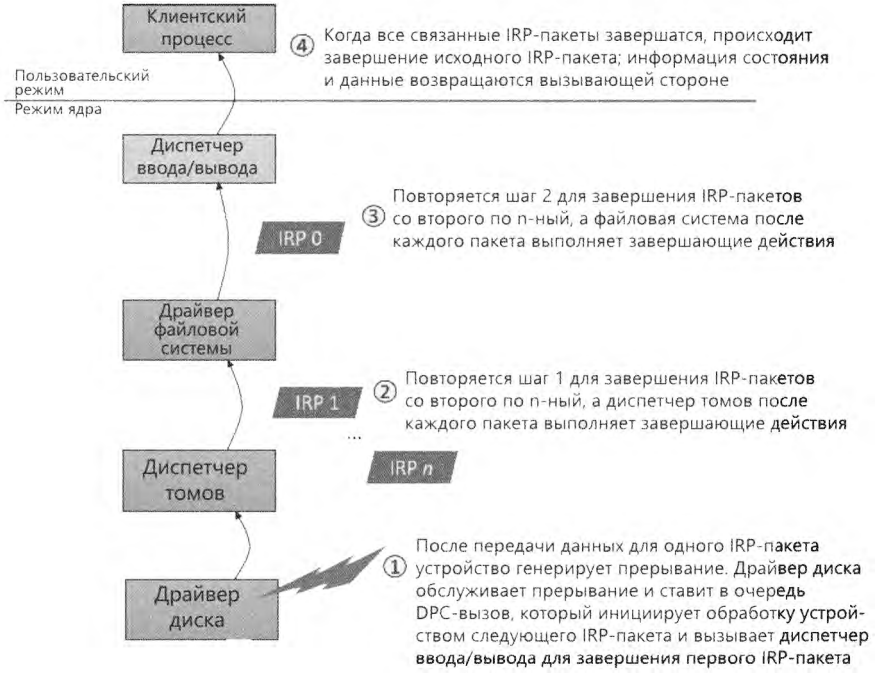


Рис. 6.21. Завершение связанных IRP-пакетов

ПРИМЕЧАНИЕ Все драйверы, управляющие в Windows дисковыми файловыми системами, являются частью как минимум трехуровневого стека драйверов. На верхнем уровне находится драйвер файловой системы, на среднем — диспетчер томов, а завершает все драйвер диска. Между этими драйверами может располагаться любое количество фильтрующих драйверов. В предыдущем примере многоуровневый запрос на ввод/вывод касался только драйверов файловой системы и диспетчера томов. Более подробно управление памятью рассматривается в главе 12 части 2.

Независимый от программных потоков ввод/вывод

В описанных ранее моделях ввода/вывода IRP-пакеты ставились в очередь программного потока, инициировавшего операцию ввода или вывода, а для их завершения диспетчер ввода/вывода выдавал асинхронный вызов, чтобы завершение обработки осуществлялось в контексте процесса и программного потока. Связанная с потоком обработка ввода/вывода обычно в достаточной степени обеспечивает производительность и расширяемость большинства приложений, но в Windows также поддерживается *независимый от программных потоков ввод/вывод* (thread agnostic I/O) с применением двух механизмов:

- использование портов завершения ввода/вывода (см. далее);
- блокировка в памяти буфера пользователя и его отображение на системное адресное пространство.

В первом случае момент проверки завершения ввода/вывода выбирает само приложение, поэтому выдавший запрос на ввод или вывод программный поток уже не важен, так как запрос на завершение может быть выполнен любым другим потоком. В результате IRP-пакет может быть завершен в контексте любого программного потока, имеющего доступ к порту завершения.

Аналогично заблокированная и отображенная в режим ядра версия буфера пользователя вовсе не должна находиться в одном адресном пространстве с инициировавшим запрос программным потоком, ведь ядро имеет доступ к памяти из произвольного контекста. При наличии привилегии `SeLockMemoryPrivilege` приложения могут включить данный режим через функцию `SetFileIoOverlappedRange`.

Как в случае порта завершения ввода/вывода, так и в случае ввода/вывода через заданные функцией `SetFileIoOverlappedRange` буферы файлов диспетчер ввода/вывода связывает IRP-пакеты с породившим их файловым объектом, а не с породившим их программным потоком. Расширение `!fileobj` в отладчике WinDbg выведет на экран полный список IRP-пакетов для файловых объектов, которые используют оба этих подхода.

В следующих разделах вы увидите, каким образом независимый от программного потока ввод/вывод повышает надежность и производительность Windows-приложений.

Отмена ввода/вывода

Существует много вариантов обработки IRP-пакетов и завершения запросов ввода/вывода, но несмотря на это часто операции ввода/вывода не завершаются, а отменяются. Например, при наличии активных IRP-пакетов может возникнуть необходимость в удалении устройства; либо сам пользователь может отменить слишком затянувшуюся операцию на устройстве (скажем, при работе в сети). Кроме того,

возможность отмены ввода/вывода требуется при завершении программного потока и процесса. Ведь потоки не допускают удаления, если есть незавершенные операции ввода или вывода.

Работающий с драйверами диспетчер ввода/вывода должен обрабатывать такие запросы эффективно, надежно и удобно. Драйверы при этом регистрируют для допускающих отмену операций ввода/вывода (обычно это операции, стоящие в очереди, но еще не запущенные на выполнение) процедуру отмены вызовом `IoSetCancelRoutine`. Эта процедура вызывается диспетчером ввода/вывода для отмены операции ввода/вывода. Если драйвер некорректно выполняет свои функции в этом сценарии, в системе могут возникнуть процессы, которые невозможно уничтожить. Визуально они никак не проявляются, но их можно увидеть в диспетчере задач или в приложении Process Explorer.

Отмена ввода/вывода, инициированная пользователем

В большинстве программ для обработки ввода через пользовательский интерфейс (User Interface, UI) выделяется один программный поток, а еще один или несколько потоков предназначается непосредственно для операций с данными, в том числе для их ввода/вывода. В некоторых случаях при попытке пользователя прервать начатую через UI операцию приложение должно отменить ожидающие операции ввода/вывода. Для быстрых операций отмена может и не потребоваться, а вот для операций, занимающих значительное время (например, для передачи больших объемов данных или сетевых операций) в Windows предусмотрена отмена как синхронных, так и асинхронных операций.

- **Отмена синхронного ввода/вывода.** Программный поток может вызвать функцию `CancelSynchronousIo`, которая разрешает отмену даже операций создания (открытия) при условии поддержки со стороны драйвера устройства. Эта функциональность поддерживается рядом драйверов в Windows, в том числе драйверами, управляющими сетевыми файловыми системами (к примеру, MUP, DFS и SMB), которые позволяют отменять операции открытия для сетевых маршрутов.

- **Отмена асинхронного ввода/вывода.** Программный поток может отменить собственные ожидающие выполнения асинхронные операции ввода и вывода вызовом функции `CancelIo`. Она аннулирует все асинхронные операции ввода/вывода для определенного дескриптора файла, какому бы потоку они ни принадлежали, в рамках одного процесса с функцией `CancelIoEx`. Последняя также воздействует на операции, связанные с портами завершения ввода/вывода. В Windows все это происходит в рамках рассмотренного ранее механизма поддержки ввода/вывода, независимого от программных потоков. Система ввода/вывода отслеживает ожидающие выполнения операции ввода/вывода, принадлежащие порту завершения, связывая их с этим портом.

Схема отмены синхронного и асинхронного ввода/вывода демонстрируется на рис. 6.22 и 6.23. (Для драйвера все варианты отмены выглядят одинаково.)

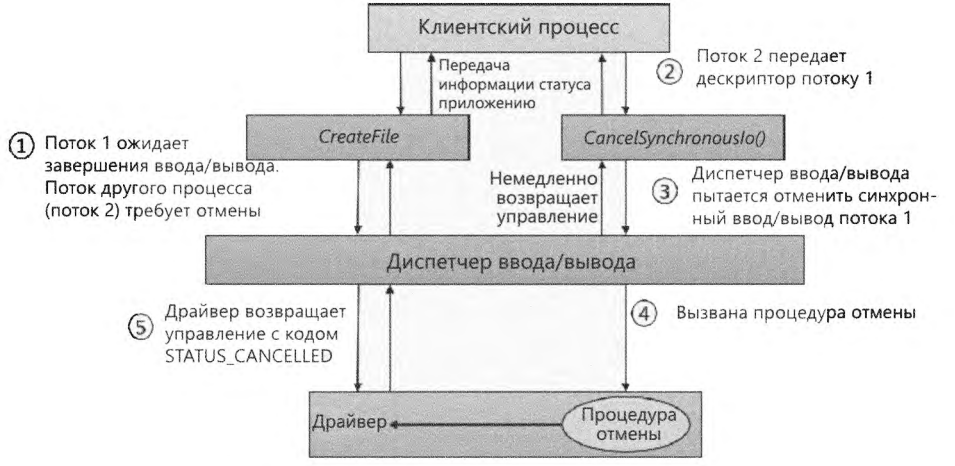


Рис. 6.22. Отмена синхронного ввода/вывода

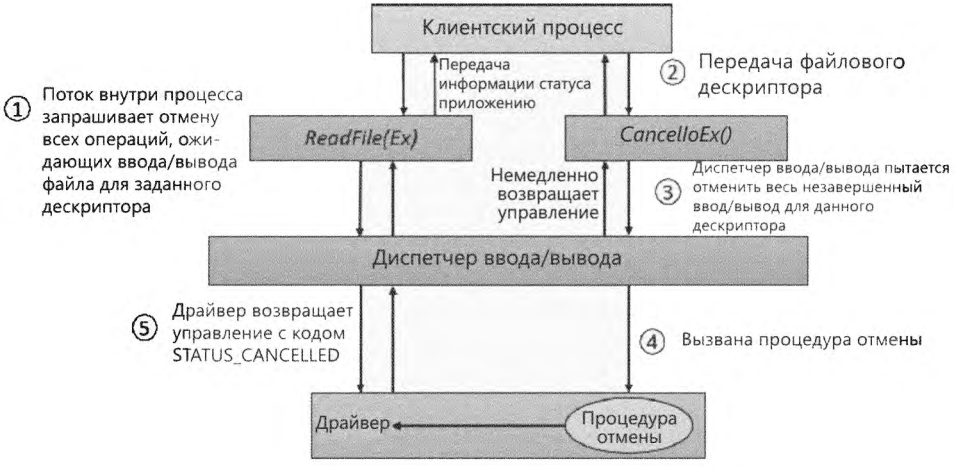


Рис. 6.23. Отмена асинхронного ввода/вывода

Отмена ввода/вывода при завершении программного потока

Другой сценарий, требующий отмены операций ввода/вывода, — прекращение работы программного потока, причем как прямое, так и косвенное вследствие завершения процесса (сопровождаемое завершением всех принадлежащих процессу потоков). Так как с каждым потоком связан список IRP-пакетов, диспетчер ввода/вывода может перебрать этот список, найти в нем допускающие отмену пакеты и аннулировать их. В отличие от функции `CancelIoEx`, которая возвращает управ-

ление, не дожидаясь отмены IRP-пакета, диспетчер процессов не допускает завершения потока до отмены всех операций ввода/вывода. В результате, если драйвер не сможет отменить IRP-пакет, объекты процесса и потока останутся в памяти до завершения работы системы.

ПРИМЕЧАНИЕ Отмену допускают только те IRP-пакеты, для которых драйвером задана одноименная процедура. Диспетчер процессов удаляет программный поток только после отмены или завершения всех связанных с этим потоком операций ввода/вывода.

ЭКСПЕРИМЕНТ: ОТЛАДКА ПРОЦЕССА, КОТОРЫЙ НЕВОЗМОЖНО УДАЛИТЬ

В данном эксперименте мы воспользуемся служебной программой Notmyfault, разработанной в Sysinternals для принудительного создания неубиваемого процесса. Для этого драйвер Myfault.sys (которым пользуется программа Notmyfault.exe) неопределенно долго удерживает IRP-пакет из-за отсутствия для него зарегистрированной процедуры отмены. (Программа Notmyfault подробно рассматривается в главе 15.) Выполните следующие действия.

1. Запустите файл Notmyfault.exe.
2. На экране появляется диалоговое окно Not My Fault. На вкладке Hang установите переключатель Hang with IRP, как показано на иллюстрации, и щелкните на кнопке Hang.

3. Визуально ничего не произойдет, и щелчком на кнопке Cancel вы можете выйти из приложения. Однако при этом процесс Notmyfault будет оставаться в диспетчере задач или в приложении Process Explorer. Попытки его удаления ни к чему не приведут, так как Windows будет бесконечно ждать завершения IRP-пакета, поскольку драйвер Myfault не зарегистрировал процедуру отмены.

4. Для отладки подобных проблем используется отладчик WinDbg. Он поможет выяснить, чем занят поток. Откройте сеанс локальной отладки в режиме ядра и воспользуйтесь командой !process для получения данных о процессе Notmyfault.exe (используется 64-разрядная версия):

```
lkd> !process 0 7 notmyfault64.exe
PROCESS ffff8c0b88c823c0
    SessionId: 1  Cid: 2b04    Peb: 4e5c9f4000  ParentCid: 0d40
    DirBase: 3edfa000  ObjectTable: ffffdf08dd140900  HandleCount: <Data Not
Accessible>
    Image: notmyfault64.exe
    VadRoot ffff8c0b863ed190 Vads 81 Clone 0 Private 493. Modified 8. Locked
0….
        THREAD ffff8c0b85377300  Cid 2b04.2714  Teb: 0000004e5c808000
Win32Thread: 0000000000000000 WAIT: (UserRequest) UserMode Non-Alertable
            fffff80a4c944018  SynchronizationEvent
        IRP List:
            ffff8c0b84f1d130: (0006,0118) Flags: 00060000  Mdl: 00000000
        Not impersonating
        DeviceMap                 ffffdf08cf4d7d20
        Owning Process            ffff8c0b88c823c0       Image:
notmyfault64.exe
...
        Child-SP          RetAddr           : Args to Child
: Call Site
        ffffb881'3ecf74a0 fffff802'cfc38a1c : 00000000'00000100
00000000'00000000 00000000'00000000 00000000'00000000 :
nt!KiSwapContext+0x76
        ffffb881'3ecf75e0 fffff802'cfc384bf : 00000000'00000000
00000000'00000000 00000000'00000000 00000000'00000000 :
nt!KiSwapThread+0x17c
        ffffb881'3ecf7690 fffff802'cfc3a287 : 00000000'00000000
00000000'00000000 00000000'00000000 00000000'00000000 :
nt!KiCommitThreadWait+0x14f
        ffffb881'3ecf7730 fffff80a'4c941fce : fffff80a'4c944018
fffff802'00000006 00000000'00000000 00000000'00000000 :
nt!KeWaitForSingleObject+0x377
        ffffb881'3ecf77e0 fffff802'd0067430 : ffff8c0b'88d2b550
00000000'00000001 00000000'00000001 00000000'00000000 : myfault+0x1fce
        ffffb881'3ecf7820 fffff802'd0066314 : ffff8c0b'00000000
ffff8c0b'88d2b504 00000000'00000000 ffffb881'3ecf7b80 : nt!IopSynchronousSer
viceTail+0x1a0
        ffffb881'3ecf78e0 fffff802'd0065c96 : 00000000'00000000
00000000'00000000 00000000'00000000 00000000'00000000 :
nt!IopXxxControlFile+0x674
        ffffb881'3ecf7a20 fffff802'cfd57f93 : ffff8c0b'85377300
fffff802'cfcb9640 00000000'00000000 fffff802'd005b32f :
nt!NtDeviceIoControlFile+0x56
        ffffb881'3ecf7a90 00007ffd'c1564f34 : 00000000'00000000
00000000'00000000 00000000'00000000 00000000'00000000 :
nt!KiSystemServiceCopyEnd+0x13 (TrapFrame @ ffffb881'3ecf7b00)
```

5. Из трассировки стека видно, что инициировавший ввод/вывод поток ожидает отмены или завершения. Теперь нужно воспользоваться уже знакомой

нам по предыдущим экспериментам командой расширения отладчика `!irp` и попытаться проанализировать проблему. Скопируйте указатель IRP-пакета и исследуйте его при помощи команды `!irp`:

```
lkd> !irp ffff8c0b84f1d130
Irp is active with 1 stacks 1 is current (= 0xffff8c0b84f1d200)
 No Mdl: No System Buffer: Thread ffff8c0b85377300:  Irp stack trace.
     cmd  flg cl Device   File     Completion-Context
>[IRP_MJ_DEVICE_CONTROL(e), N/A(0)]
            5  0 ffff8c0b886b5590 ffff8c0b88d2b550 00000000-00000000
            \Driver\MYFAULT
                    Args: 00000000 00000000 83360020 00000000
```

6. Отсюда видно, что причиной проблемы стал драйвер \Driver\MYFAULT, т. е. Myfault.sys. Его имя (дословно — «моя вина») подчеркивает, что ситуация возникла из-за проблем с драйвером, а не из-за ошибок приложения. К сожалению, хотя вы знаете, какой драйвер создал проблемы, вряд ли вы сможете что-то сделать, кроме перезагрузки системы. Перезагрузка необходима, так как Windows не может допустить, что еще не случившуюся отмену можно просто проигнорировать. IRP-пакет может в любой момент вернуть управление и повредить системную память.

СОВЕТ Если вы столкнетесь с подобной ситуацией в реальной жизни, попробуйте поискать более свежую версию драйвера. Возможно, в ней данная ошибка уже устранена.

Порты завершения ввода/вывода

Для создания высокопроизводительного серверного приложения требуется реализовать эффективную модель программных потоков. Проблемы с производительностью возникают как из-за слишком малого, так и из-за слишком большого количества потоков на сервере. К примеру, если сервер создаст для обработки запросов всего один программный поток, работа клиентов сильно замедлится, так как в любой момент времени будет обрабатываться только один запрос. Конечно, один поток может обрабатывать несколько запросов одновременно, переключаясь с одной операции ввода/вывода на другую, но подобная архитектура крайне сложна и не позволяет пользоваться преимуществами многопроцессорных систем. В качестве другой крайности можно представить создание сервером огромного пула потоков, когда для обработки каждого запроса выделяется отдельный программный поток. Такой сценарий обычно ведет к *пробуксовке* (thrashing): многочисленные потоки порождаются, выполняют обработку данных, блокируются в ожидании ввода/вывода, а затем после завершения обработки запроса блокируются в ожидании нового запроса. Наличие огромного количества потоков уже само по себе приводит к избыточному переключению контекста, возникающего из-за того, что планировщику приходится делить время процессора между всеми активными потоками; такая схема не масштабируется.

Сервер должен, с одной стороны, уменьшить количество переключений контекста, заставив потоки избегать ненужной блокировки, а с другой — обеспечить максимальный параллелизм за счет большого количества потоков. С этой точки зрения идеальна ситуация, когда для каждого процессора выделяется свой активно обрабатывающий клиентские запросы поток, и при этом потоки не блокируются после завершения обработки запроса, если в очереди присутствуют дополнительные запросы. Однако для корректной работы этой оптимальной схемы нужно, чтобы при блокировке потока, обрабатывающего клиентский запрос, в ожидании ввода/вывода (например, когда в рамках обработки он читает файл) у приложения была возможность активировать еще один поток.

Объект IoCompletion

Приложения пользуются объектом `IoCompletion` исполнительной системы, который экспортируется в Windows API как *порт завершения* (completion port) — это центральная точка завершения ввода/вывода, ассоциированная с набором дескрипторов файлов. После того как файл ассоциирован с портом завершения, все заканчиваемые в нем асинхронные операции ввода/вывода помещаются в пакет завершения, который ставится в очередь к данному порту. Ожидание потоком завершения операций ввода/вывода в разных файлах может быть реализовано как ожидание появления в очереди порта завершения указанного пакета. В Windows API для этого служит функция `WaitForMultipleObjects`, но порты завершения обладают таким важным преимуществом, как *параллелизм* — т. е. количество потоков, при помощи которых приложение обслуживает запросы клиентов, контролируется самой системой.

Создавая порт завершения, приложение указывает максимальное количество связанных с этим портом программных потоков, которые могут работать одновременно. Как уже упоминалось, в идеале каждому процессору должен соответствовать один активный поток. Windows использует это значение для управления количеством активных программных потоков. Если количество ассоциированных с портом активных потоков достигает своего максимума, ожидающий выполнения на порте завершения поток не запускается. Он дожидается завершения обработки текущего запроса одним из активных потоков и проверяет наличие в порте еще одного ожидающего пакета. Обнаружив такой пакет, поток извлекает его из очереди и приступает к обработке. При этом отсутствует переключение контекста, т. е. процессоры работают практически на полной мощности.

Применение портов завершения

На рис. 6.24 представлена высокоуровневая схема работы порта завершения. Порт завершения в Windows создается вызовом API-функции `CreateIoCompletionPort`. Заблокированные на порте завершения программные потоки считаются ассоциированными с этим портом и пробуждаются по принципу LIFO (Last In, First Out — последним пришел, первым вышел), т. е. следующий пакет передается потоку, который был заблокирован последним. Блокируемые надолго потоки могут

быть выгружены на диск, поэтому, если с портом связано больше потоков, чем требуется для обработки текущих заданий, система автоматически минимизирует объем памяти, занимаемый слишком долго блокируемыми потоками.

Серверное приложение обычно получает клиентские запросы через конечные точки, задаваемые дескрипторами файлов. В качестве примера можно привести сокеты Windows Sockets 2 (Winsock2) или именованные каналы. Создаваемые коммуникационные конечные точки сервер связывает с портом завершения, а его потоки ждут входящих запросов, вызывая для этого порта функцию `GetQueuedCompletionStatus(Ex)`. Получив пакет из порта завершения, поток активируется и начинает обработку запроса. Во время этой обработки поток многократно блокируется, например из-за необходимости прочитать данные из файла или записать их в файл либо синхронизироваться с другими потоками. Windows обнаруживает подобные действия и распознает, что на порте завершения одним активным потоком стало меньше. И как только поток становится неактивным по причине блокировки, а в очереди присутствует пакет, просыпается другой поток, ожидающий на порте завершения.

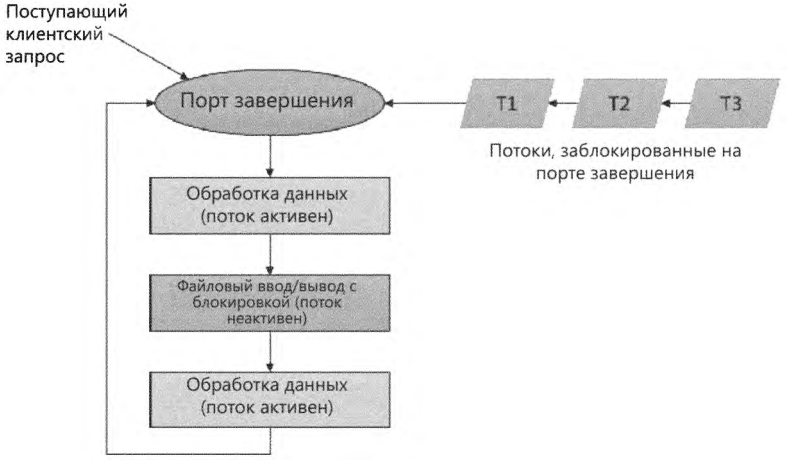


Рис. 6.24. Схема работы порта завершения ввода/вывода

Согласно рекомендациям Microsoft, максимальное количество активных потоков должно быть приблизительно равно количеству процессоров в системе. Имейте в виду, что это число может быть превышено. Представьте, что вы задали максимальное значение количества активных потоков равным 1:

1. При поступлении клиентского запроса активируется выделенный для его обработки поток.
2. Поступает второй запрос, но активировать второй поток, ожидающий на порте, невозможно, так как лимит уже достигнут.
3. Первый поток блокируется из-за ожидания операции ввода или вывода, т. е. он станет неактивным.

4. Второй поток освобождается.
5. Пока второй поток остается активным, завершается файловый ввод/вывод первого потока, т. е. он снова становится активным. С этого момента — и до блокировки одного из потоков — количество активных потоков достигнет 2, т. е. превысит установленный предел на 1. В большинстве случаев количество активных потоков равно предельному или немного превышает его.

API портов завершения также дает серверному приложению возможность ставить в очередь порта завершения самостоятельно определенные пакеты завершения. Это делается с помощью функции `PostQueuedCompletionStatus`. Сервер обычно применяет ее для информирования потоков о внешних событиях, например о необходимости корректного завершения работы операционной системы.

Приложения могут использовать с портами завершения независимый от потоков ввод/вывод, описанный в одном из предыдущих разделов, чтобы избежать связывания потоков с их собственными операциями ввода/вывода, вместо этого ассоциировав с ними объект порта завершения. Порты завершения ввода/вывода в дополнение к другим выгодам, касающимся масштабируемости, позволяют минимизировать переключения контекста. Стандартное завершение ввода/вывода должен выполнить тот же самый поток, который начал операцию, но по окончании ввода или вывода, связанного с портом завершения, диспетчер ввода/вывода использует любой из ожидающих потоков для выполнения операции завершения.

Функционирование порта ввода/вывода

Windows-приложения создают порты завершения вызовом API-функции `CreateIoCompletionPort` с передачей в качестве дескриптора порта завершения значения NULL. Это приводит к выполнению системной функции `NtCreateIoCompletion`. Объект исполнительной системы `IoCompletion` включает в себя объект синхронизации ядра, называемый *очередью ядра* (kernel queue). Соответственно, системная сервисная функция создает объект порта завершения и инициализирует в выделенной для порта памяти объект очереди. (Указатель на порт также ссылается на объект очереди, так как последний располагается в первом поле объекта порта.) Число программных потоков для объекта очереди ядра задается в момент его инициализации потоком. И именно это число передается в функцию `CreateIoCompletionPort`. Для инициализации объекта очереди порта завершения функция `NtCreateIoCompletion` вызывает функцию `KeInitializeQueue`.

При вызове функции `CreateIoCompletionPort` приложением для связывания дескриптора файла с портом выполняется системная сервисная функция `NtSetInformationFile` с дескриптором файла в качестве основного параметра. Класс информации получает значение `FileCompletionInformation`, а дескриптор порта завершения и параметр `CompletionKey` функции `CreateIoCompletionPort` являются значениями данных. Функция `NtSetInformationFile` производит разыменование дескриптора файла для получения файлового объекта и создает структуру данных контекста завершения.

Наконец, функция `NtSetInformationFile` помещает указатель на эту структуру в поле `CompletionContext` файлового объекта. При завершении асинхронной операции ввода/вывода для файлового объекта диспетчер ввода/вывода проверяет, отличается ли значение поля `CompletionContext` от NULL. В случае положительного результата проверки создается пакет завершения, который ставится в очередь порта завершения вызовом функции `KeInsertQueue`. При этом в качестве очереди, в которую помещается пакет, указывается порт. (Как вы помните, объекты порта завершения и очереди имеют один адрес.)

При вызове потоком сервера функции `GetQueuedCompletionStatus` выполняется системная сервисная функция `NtRemoveIoCompletion`. После проверки параметров и преобразования дескриптора порта завершения в указатель на этот порт данная служба вызывает функцию `IoRemoveIoCompletion`, которая в конечном итоге вызывает функцию `KeRemoveQueueEx`. В высокопроизводительных сценариях возможно завершение сразу нескольких операций ввода и вывода, и несмотря на отсутствие блокировки поток будет обращаться к ядру по каждой позиции. API-функции `GetQueuedCompletionStatus` и `GetQueuedCompletionStatusEx` позволяют приложениям получать несколько состояний завершения ввода/вывода одновременно, что сокращает интенсивность обмена пакетами между пользователем и ядром и обеспечивает высокую эффективность. Во внутренней реализации используется функция `NtRemoveIoCompletionEx`, которая вызывает функцию `IoRemoveIoCompletion`, оснащенную счетчиком количества позиций в очереди и передающую это значение функции `KeRemoveQueueEx`.

Как видите, работу портов завершения обеспечивают функции `KeRemoveQueueEx` и `KeInsertQueue`. Они определяют, не нужно ли активировать поток, ожидающий пакета завершения ввода/вывода. Во внутренней реализации объект очереди поддерживает счетчик активных потоков и хранит сведения об их максимальном количестве. Если при вызове потоком функции `KeRemoveQueueEx` число активных потоков равно максимуму или превышает его, он попадет (в порядке LIFO) в список потоков, ожидающих пакета завершения. Список потоков отделен от объекта очереди. В структуре данных блока управления потоком (`KTHREAD`) присутствует указатель на очередь, связанную с объектом очереди; если указатель равен NULL, поток с очередью не связан.

Windows отслеживает потоки, ставшие неактивными из-за ожидания на объектах, отличных от порта завершения, по присутствующему в блоке управления потоком указателю на очередь. Этот указатель проверяют процедуры планировщика, которые могут стать причиной блокировки потока (например, `KeWaitForSingleObject`, `KeDelayExecutionThread` и т. п.). Если его значение отлично от NULL, они вызывают связанную с очередью функцию `KiActivateWaiterQueue`, которая уменьшает счетчик количества ассоциированных с очередью активных потоков. При значениях счетчика меньше максимального и наличии в очереди хотя бы одного пакета завершения пробуждается первый в очереди поток из списка и получает самый старый пакет. И наоборот, при каждом пробуждении связанного с очередью потока планировщик выполняет функцию `KiUnwaitThread`, увеличивающую значение счетчика числа активных потоков.

А результатом вызова API-функции `PostQueuedCompletionStatus` является выполнение системной службы `NtSetIoCompletion`, которая при помощи функции `KeInsertQueue` вставляет определенный пакет в очередь порта завершения.

Схема работы порта завершения показана на рис. 6.25. Несмотря на то что к обработке пакетов завершения готовы два потока, предельное значение 1 допускает активность только одного из них. Поэтому на данном порте завершения блокируются два потока.

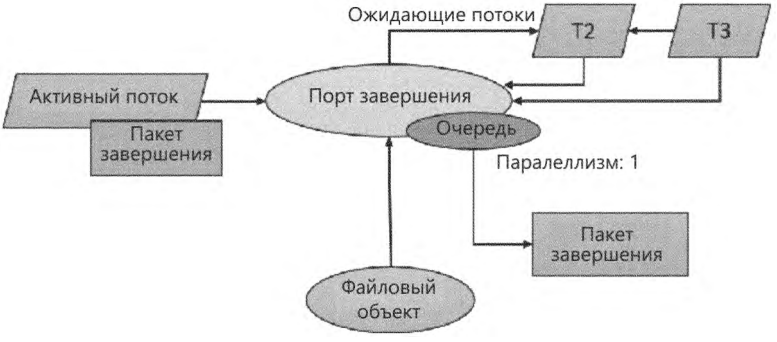


Рис. 6.25. Схема работы порта завершения ввода/вывода

Модель уведомлений для порта завершения ввода/вывода настраивается при помощи API-функции `SetFileCompletionNotificationModes`, позволяющей разработчикам приложений прибегнуть к дополнительной оптимизации, обычно требующей редактирования кода, но обеспечивающей более высокую производительность. Существует три варианта оптимизации режима уведомлений (табл. 6.4). Следует заметить, что все они задаются на уровне дескриптора файла и не модифицируются.

Таблица 6.4. Режимы уведомлений порта завершения ввода/вывода

Режим уведомления	Значение
Пропуск порта завершения при успешном выполнении (`FILE_SKIP_COMPLETION_PORT_ON_SUCCESS=1`)	Диспетчер ввода/вывода не ставит запись завершения в очередь, если соблюдаются три условия. Во-первых, порт завершения должен быть связан с дескриптором файла; во-вторых, файл должен быть открыт для асинхронного ввода/вывода; в-третьих, запрос не должен возвращать значение `ERROR_PENDING`
Пропуск события для дескриптора (`FILE_SKIP_SET_EVENT_ON_HANDLE=2`)	Диспетчер ввода/вывода не задает событие для файлового объекта, если запрос возвращается с кодом успешного завершения или с ошибкой `ERROR_PENDING`, а вызванная функция не является синхронной. Если для запроса предусмотрено явное событие, оно активизируется

Режим уведомления	Значение
Пропуск пользовательского события при быстром вводе/выводе (FILE_SKIP_SET_USER_EVENT_ON_FAST_IO=4)	Диспетчер ввода/вывода не задает явное событие, предусмотренное для запроса, если запрос осуществляется путем быстрого ввода/вывода и управление возвращается с кодом успешного завершения или с ошибкой ERROR_PENDING, а вызываемая функция не является синхронной

Определение приоритетов ввода/вывода

Без приоритетов ввода/вывода фоновые операции (такие, как поисковое индексирование, сканирование вирусов и дефрагментация диска) будут серьезно снижать производительность операций с высоким приоритетом. Если некий процесс в системе выполняет дисковый ввод или вывод, загружающий приложение или открывающий документ пользователь будет сталкиваться с задержкой, так как приоритетной задаче придется ждать доступа к диску. Аналогичная задержка будет сопровождать потоковое воспроизведение мультимедийного контента, например музыки с диска.

В Windows существует два способа задания приоритетов ввода/вывода, позволяющих дать преимущество высокоприоритетным операциям: назначение приоритета отдельным операциям ввода/вывода и резервирование полосы пропускания ввода/вывода.

Приоритеты ввода/вывода

Как показано в табл. 6.5, диспетчер ввода/вывода в Windows поддерживает пять приоритетов ввода/вывода, но используются только три из них. (Возможно, будущие версии Windows будут поддерживать приоритеты High и Low.)

Таблица 6.5. Приоритеты ввода/вывода

Приоритет ввода/вывода	Использование
Critical	Диспетчер памяти
High	Не используется
Normal	Нормальный ввод/вывод приложения
Low	Не используется
Very Low	Запланированные задачи, супервыборка, дефрагментация, индексирование содержимого, фоновые операции

По умолчанию операции ввода/вывода имеют приоритет Normal, а диспетчер памяти в ситуации, когда памяти не хватает и нужно освободить место для данных и кода, записывает на диск измененные данные с приоритетом Critical. Планировщик задач присваивает задачам приоритет Very Low. Аналогичный приоритет

используют приложения, выполняющие фоновые операции. В числе прочего в эту категорию попадают сканирование, выполняемое приложением Windows Defender, и индексирование содержимого рабочего стола.

Стратегии выбора приоритета

Пять вариантов приоритета ввода/вывода делятся на два режима, называемых *стратегиями* (strategies). Существуют стратегии иерархического выбора приоритета (hierarchy prioritization) и выбора приоритета на основе простоя (idle prioritization). В первом случае рассматриваются все варианты приоритетов, кроме Very Low. При этом действуют следующие правила:

- все операции ввода/вывода с приоритетом Critical обрабатываются раньше операций с приоритетом High;
- все операции ввода/вывода с приоритетом High обрабатываются раньше операций с приоритетом Normal;
- все операции ввода/вывода с приоритетом Normal обрабатываются раньше операций с приоритетом Low;
- все операции ввода/вывода с приоритетом Low обрабатываются после операций с более высоким приоритетом.

IRP-пакеты, генерируемые различными приложениями, помещаются в различные очереди в соответствии с их приоритетом, а затем в соответствии с иерархической стратегией определяется очередность выполнения операций.

В то же время в стратегии выбора приоритета на основе простоя для операций ввода/вывода с более высоким приоритетом используется отдельная очередь. Так как приоритет простоя является самым низким из возможных, очередь его операций может застопориться при наличии в системе хотя бы одной операции ввода/вывода с более высоким приоритетом.

Чтобы избежать подобной ситуации, а также для контроля за отсрочкой (имеется в виду частота передачи данных ввода/вывода), в стратегии выбора приоритета на основе простоя используется таймер для слежения за очередью и гарантированной обработки хотя бы одной операции ввода/вывода в единицу времени (обычно полсекунды). Данные, записанные с приоритетом, превышающим приоритет простоя, заставляют диспетчер кэша немедленно сбросить изменения на диск, проигнорировав опережающее чтение, позволяющее превентивно считывать данные из файла, к которому происходит обращение. После завершения последней операции ввода/вывода с приоритетом, отличным от приоритета простоя, следующий запрос на ввод/вывод с приоритетом простоя происходит через 50 миллисекунд. Без этого ввод или вывод с приоритетом простоя возникнет в середине потока с более высоким приоритетом, что приведет к затратному позиционированию.

Если с демонстрационными целями объединить эти стратегии в виртуальную глобальную очередь ввода/вывода, может получиться, к примеру, вариант, пред-

ставленный на рис. 6.26. Следует заметить, что расстановка в каждой из очередей происходит по алгоритму FIFO (First-In, First-Out — первым пришел, первым вышел). Порядок элементов на рисунке представлен только для примера.

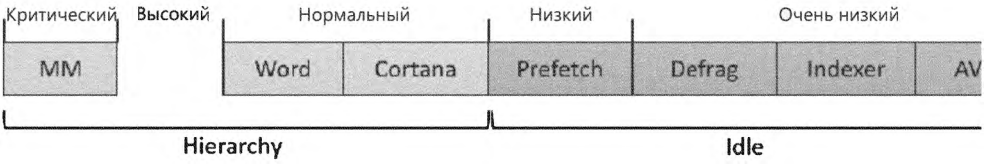


Рис. 6.26. Пример глобальной очереди ввода/вывода

Пользовательские приложения задают приоритет ввода/вывода для трех разных объектов. Функции `SetPriorityClass` (со значением `PROCESS_MODE_BACKGROUND_BEGIN`) и `SetThreadPriority` (со значением `THREAD_MODE_BACKGROUND_BEGIN`) устанавливают приоритет для всех операций ввода/вывода, генерируемых процессом в целом или определенными программными потоками (этот приоритет хранится в IRP-пакете каждого запроса). Эти функции работают только для текущего процесса или потока, они снижают приоритет ввода/вывода до Very Low. Кроме того, они также снижают приоритет планирования до 4, а приоритет памяти до 1. Функция `SetFileInformationByHandle` задает приоритет для конкретного файлового объекта (и хранится он в файловом объекте). Кроме того, драйверы могут задавать приоритет ввода/вывода непосредственно в IRP-пакетах, используя для этого API-функцию `IoSetIoPriorityHint`.

ПРИМЕЧАНИЕ Поле приоритета ввода/вывода в IRP-пакете и/или файловом объекте — не более чем рекомендация. Нет никакой гарантии, что указанное там значение повлияет на поведение драйверов из стека внешней памяти. Может оказаться, что оно этими драйверами даже не поддерживается.

Упомянутые стратегии выбора приоритета реализуются двумя разными типами драйверов. Иерархическая стратегия реализуется драйверами *порта* накопителей, отвечающими за все операции ввода/вывода через определенный порт. К этой группе относятся такие драйверы, как ATA, SCSI или USB. Но из них только драйверы порта ATA (Ataport.sys) и USB (Usbstor.sys) поддерживают данную стратегию, а драйверы порта SCSI и накопителей (Scsiport.sys и Storport.sys) — нет.

ПРИМЕЧАНИЕ Все драйверы порта особо проверяют операции ввода/вывода с приоритетом Critical и перемещают их в начало очереди, даже если полностью механизм иерархии ими не поддерживается. Такой механизм призван поддержать критически важные операции ввода/вывода диспетчера памяти, связанные с подкачкой страниц, которые обеспечивают надежность системы.

Это означает, что такие запоминающие устройства, как жесткие IDE- или SATA-диски, а также USB-диски флэш-памяти будут пользоваться преимуществами при-

оритетов ввода/вывода, в то время как устройства на основе SCSI, Fibre Channel и iSCSI — нет.

В то же время стратегию на основе простоя реализует драйвер класса хранилищ (Classpnp.sys), поэтому она автоматически применима ко всем операциям ввода/вывода, направленным на устройства хранения, в том числе — к SCSI-дискам. Такое разделение гарантирует, что ввод и вывод с приоритетом простоя будут зависеть от алгоритмов задержки, обеспечивая стабильность системы при функционировании с большим количеством подобных операций ввода/вывода, и что приложения, использующие эти операции, смогут работать. Поддержка этой стратегии драйвером класса от Microsoft позволяет избежать проблем с производительностью, которые возникли бы из-за отсутствия такой поддержки у устаревших драйверов порта сторонних производителей.

Рисунок 6.27 демонстрирует упрощенное представление стека драйверов внешней памяти и областей реализации обеих стратегий. Подробно стек драйверов внешней памяти рассматривается в главе 12 части 2.

Рис. 6.27. Реализация приоритетов ввода/вывода в стеке драйверов внешней памяти

Предотвращение инверсии приоритетов ввода/вывода (наследование приоритетов ввода/вывода)

Чтобы избежать инверсии приоритетов ввода/вывода (из-за которой поток с высоким приоритетом может быть заблокирован низкоприоритетным потоком), исполнительный ресурс с функциональностью блокировки (`ERESOURCE`) задействует несколько стратегий. Этот ресурс был выбран для реализации наследования приоритетов ввода/вывода, так как он активно используется файловой системой и драйверами накопителей, где и возникает большинство проблем с инверсией приоритетов. (Исполнительные ресурсы более подробно рассматриваются в главе 8 части 2.)

Если объект `ERESOURCE` получает программный поток с низким приоритетом ввода/вывода и при этом данным объектом владеют ожидающие потоки с нормальным

или более высоким приоритетом, приоритет текущего потока временно повышается до нормального через API-функцию `PsBoostThreadIo`, увеличивающую значение параметра `IoBoostCount` в структуре `ETHREAD`. Она также оповещает Autoboost, если приоритет ввода/вывода потока был повышен или, наоборот, повышение было отменено. (За дополнительной информацией об Autoboost обращайтесь к главе 4.)

Затем объект вызывает API-функцию `IoBoostThreadIoPriority`, перечисляющую все IRP-пакеты в очереди целевого потока (вспомните, что для каждого потока существует список ожидающих IRP-пакетов) и проверяющую, какой из пакетов имеет приоритет ниже целевого (в данном случае целевым является нормальный приоритет). Таким способом идентифицируются ожидающие IRP-пакеты с приоритетом простоя для ввода/вывода. В свою очередь, объект устройства отвечает за идентификацию каждого из таких IRP-пакетов, а диспетчер ввода/вывода проверяет, зарегистрирована ли для приоритета процедура обратного вызова, который разработчики драйвера могут выполнять через API-функцию `IoRegisterPriorityCallback` с установкой флага `DO_PRIORITY_CALLBACK_ENABLED` для объекта устройства. В зависимости от того, осуществляется или нет для IRP-пакета ввод/вывод с подкачкой, данный механизм будет называться *потоковым повышением* (threaded boost) или *повышением подкачки* (paging boost). Наконец, при отсутствии IRP-пакетов, соответствующих данному критерию, но при наличии у потока хотя бы нескольких ожидающих IRP-пакетов процедура производится для них всех вне зависимости от объекта устройства или приоритета и называется *общим повышением* (blanket boosting).

Повышение и понижение приоритетов ввода/вывода

Во избежание задержек, инверсии и других нежелательных сценариев использования приоритетов ввода/вывода в Windows вносятся небольшие изменения в обычные механизмы ввода/вывода. Как правило, это делается путем повышения приоритета данной операции, когда возникает такая необходимость. Данное поведение демонстрируют следующие сценарии.

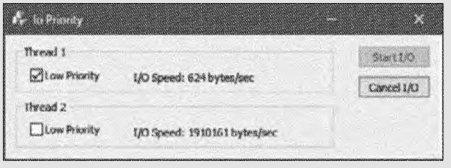
- При вызове драйвера с IRP-пакетом, предназначенным определенному файловому объекту, Windows гарантирует, что в случае запроса из режима ядра данный пакет будет иметь нормальный приоритет, даже если для файлового объекта рекомендован более низкий приоритет ввода/вывода. Это называется *повышением из режима ядра* (kernel boost).
- При чтении из файла подкачки или записи в него (через функции `IoPageRead` и `IoPageWrite`) Windows проверяет, был ли запрос послан из режима ядра, и не прибегает к службе супервыборки (которая всегда использует ввод/вывод с приоритетом простоя). В этом случае IRP-пакет будет иметь нормальный приоритет, даже если приоритет текущего потока ниже. Это называется *повышением подкачки* (paging bump).

В следующем эксперименте мы рассмотрим пример приоритета Very Low, а также просмотрим приоритеты различных запросов с помощью приложения Process Monitor.

ЭКСПЕРИМЕНТ: ПРОИЗВОДИТЕЛЬНОСТЬ ВВОДА/ВЫВОДА С ПРИОРИТЕТАМИ VERY LOW И NORMAL

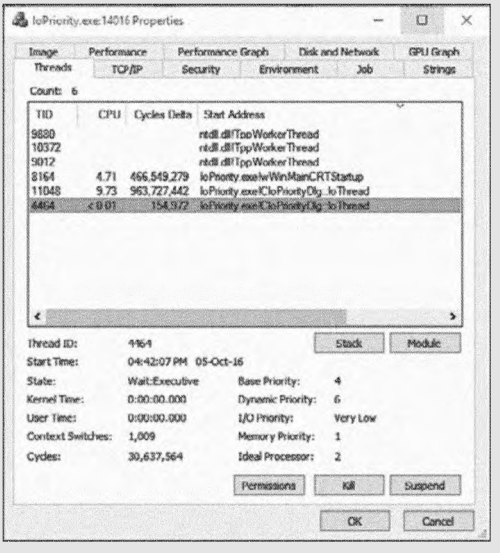
Для определения производительности программных потоков с различными приоритетами ввода/вывода можно воспользоваться приложением IO Priority (включенным в пакет служебных программ для данной книги). Выполните следующие действия.

1. Запустите файл IoPriority.exe.
2. В диалоговом окне для потока Thread 1 установите флажок Low priority.
3. Щелкните на кнопке Start I/O. Вы обнаружите заметное различие в быстродействии двух потоков, как показано на следующем рисунке.



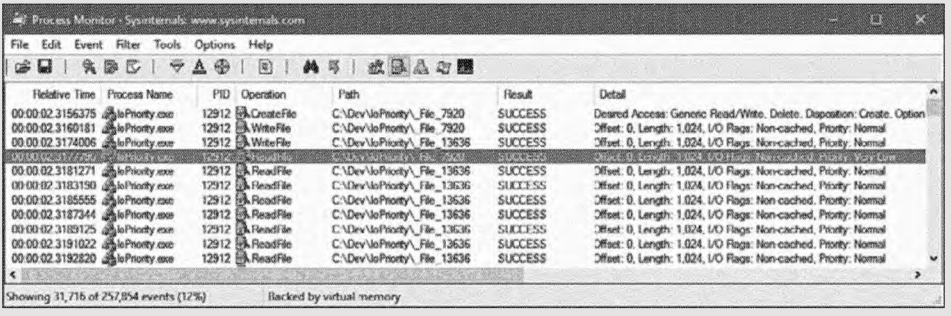
ПРИМЕЧАНИЕ Если оба потока выполняются с приоритетом Low, а система относительно свободна, их производительность будет приблизительно равна производительности одного потока с нормальным приоритетом ввода/вывода. Это связано с тем, что низкоприоритетный ввод/вывод не подвергается искусственному торможению при отсутствии конкуренции со стороны ввода/вывода с более высоким приоритетом.

4. Откройте процесс в Process Explorer и просмотрите информацию о низкоприоритетном потоке:



5. Приложение Process Monitor позволяет отследить операции ввода/вывода, инициируемые программой IO Priority, и посмотреть на рекомендуемые им приоритеты. Запустите Process Monitor, настройте фильтр для IoPriority.exe и повторите эксперимент. В этом приложении потоки читают данные из файла с именем «*_File_+ идентификатор потока*».

6. Прокрутите вывод и найдите запись в _File_1. Результат будет выглядеть так:



7. Видно, что связанный с файлом _File_7920 ввод/вывод имеет приоритет Very Low. Обратившись к столбцам Time Of Day и Relative Time, вы обнаружите, что все операции ввода/вывода разделены промежутком в 0,5 секунды — еще один признак стратегии на основе простоя.

ЭКСПЕРИМЕНТ: АНАЛИЗ ПРОИЗВОДИТЕЛЬНОСТИ ПРИ ПОВЫШЕНИИ/ПОНИЖЕНИИ ПРИОРИТЕТА ВВОДА/ВЫВОДА

Ядро предоставляет несколько внутренних переменных, значение которых можно узнать через недокументированный системный класс `SystemLowPriorityIoInformation`, доступный в функции `NtQuerySystemInformation`. Кроме того, эти системные данные можно узнать через локальный отладчик ядра, даже не прибегая к подобному приложению. Доступны следующие переменные:

- `IoLowPriorityReadOperationCount` и `IoLowPriorityWriteOperationCount`;
- `IoKernelIssuedIoBoostedCount`;
- `IoPagingReadLowPriorityCount` и `IoPagingWriteLowPriorityCount`;
- `IoPagingReadLowPriorityBumpedCount` и `IoPagingWriteLowPriorityBumpedCount`;
- `IoBoostedThreadedIrpCount` и `IoBoostedPagingIrpCount`;
- `IoBlanketBoostCount`.

Для просмотра их значений используйте команду dd отладчика ядра, выводящую значения этих переменных (все значения являются 32-разрядными).

Резервирование полосы пропускания (планирование файлового ввода/вывода)

Резервирование полосы пропускания ввода/вывода в Windows может потребоваться приложениям, которым нужна постоянная производительность ввода/вывода. Вызывая функцию `SetFileBandwidthReservation`, программа воспроизведения мультимедиа просит подсистему ввода/вывода гарантировать чтение данных с устройства с указанной скоростью. Если устройство в состоянии предоставить данные на указанной скорости и это допускается существующим механизмом резервирования, подсистема ввода/вывода сообщает приложению, как быстро должны производиться операции ввода/вывода и насколько большим должен быть размер пакетов.

Подсистема ввода/вывода не обслуживает другие операции ввода/вывода, пока не удовлетворены требования приложений, выполнивших резервирование на целевом накопителе. Рисунок 6.28 иллюстрирует временную развертку операций ввода/вывода, вызванных одним и тем же файлом. Остальным приложениям будут доступны только заштрихованные области. Если полоса пропускания ввода/вывода уже занята, новым операциям придется подождать следующего цикла.

Рис. 6.28. Результат выполнения запросов на ввод и вывод при резервировании полосы пропускания

Как и стратегия иерархического выбора приоритета, резервирование полосы пропускания реализуется на уровне драйвера порта, т. е. подобное резервирование доступно только для IDE-, SATA- и USB-накопителей.

Уведомления о сеансах

Уведомления о сеансах представляют собой специальный класс событий, на которые через механизм асинхронного обратного вызова могут подписываться драйверы. Для этого они используют API-функцию `IoRegisterContainerNotification` и указывают нужный класс уведомлений. До настоящего времени в Windows был реализован всего один класс — `IoSessionStateNotification`. Он позволяет активировать зарегистрированный обратный вызов при изменении состояния рассматриваемого сеанса. Поддерживаются следующие изменения:

- создание или завершение сеанса;
- подключение пользователя к сеансу или отключение от сеанса;
- вход пользователя в сеанс и выход из сеанса.

Если указать объект устройства, принадлежащий определенному сеансу, обратный вызов драйвера будет активен только в рамках этого сеанса. Также можно указать глобальный объект устройства или не указывать его вообще. В этом случае драйвер будет получать уведомления обо всех событиях в системе. Эта возможность особенно полезна для устройств, принимающих участие в предоставляемой через службу терминалов перенаправления функциональности PnP-устройств (таких, как принтер), которая обеспечивает видимость удаленных устройств на шине PnP-диспетчера клиентского узла. Например, если пользователь выходит из сеанса в процессе воспроизведения аудиопотока, драйверу устройства требуется уведомление, чтобы остановить перенаправление аудиопотока от источника.

Программа Driver Verifier

Программа Driver Verifier помогает в поиске и изоляции распространенных ошибок драйверов устройств и другого системного кода в режиме ядра. Компания Microsoft использует программу Driver Verifier для проверки как собственных драйверов устройств производства Microsoft, так и присылаемых производителями для тестирования в лаборатории Windows Hardware Quality Labs (WHQL). Это гарантирует совместимость драйверов с Windows и отсутствие в них распространенных ошибок. (Также существует программа Application Verifier, позволившая повысить качество кода в пользовательском режиме Windows, но ее описание выходит за рамки темы данного издания.)

ПРИМЕЧАНИЕ Главным образом, программа Driver Verifier используется как инструмент для выявления ошибок в коде для разработчиков драйверов устройств, но пригодится она и системным администраторам при крахе системы. О ее роли в анализе системных сбоев рассказывается в главе 15.

Программа Driver Verifier поддерживается несколькими системными компонентами: и в диспетчере памяти, и в диспетчере ввода/вывода, и в HAL присутствуют параметры проверки драйверов, которые можно активировать. Их настройка осуществляется через диспетчер проверки драйверов (%SystemRoot%\System32\Verifier.exe). При запуске программы Driver Verifier без аргументов командной строки появляется интерфейс, показанный на рис. 6.29. (Для включения и отключения Driver Verifier, а также для вывода текущих настроек используется интерфейс командной строки. Чтобы просмотреть набор ключей, введите команду `verifier/?`.)

Driver Verifier Manager различает два вида настроек: стандартные и дополнительные. Деление достаточно условное, но к стандартным настройкам относятся более распространенные возможности, которые, вероятно, должны включаться для каждого тестируемого драйвера, тогда как к дополнительным настройкам относятся менее распространенные или относящиеся к конкретным типам драйверов. При выборе переключателя Create Custom Settings на основной странице мастера

выводится полный список настроек с дополнительным столбцом, в котором указывается, к какой категории относится настройка (рис. 6.30).

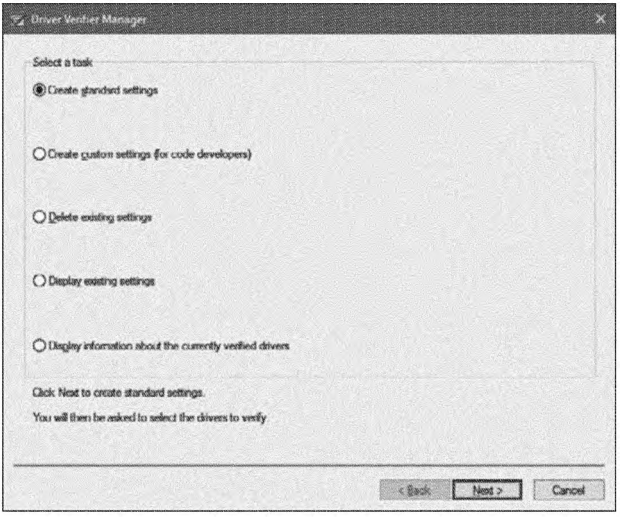


Рис. 6.29. Диспетчер Driver Verifier

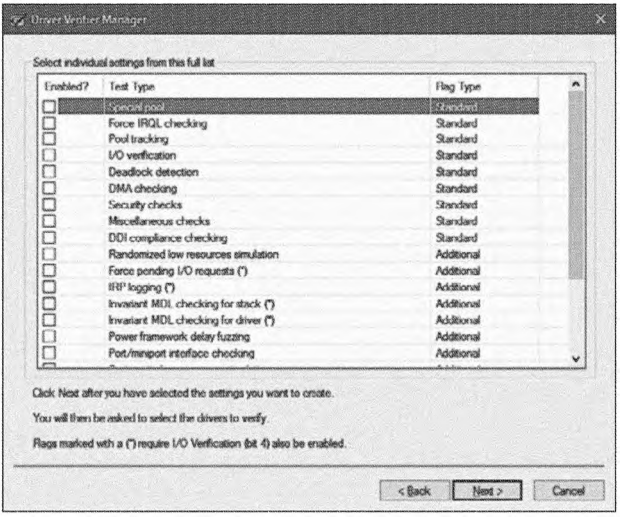


Рис. 6.30. Настройки Driver Verifier

Даже если вы не указали никаких параметров, программа Driver Verifier отслеживает выбранные для проверки драйверы, следя за недопустимыми операциями. К числу таких операций относятся, например, вызовы функций пула памяти ядра при недействительном IRQL-уровне, попытки повторного освобождения свобод-

ной памяти, некорректное освобождение спин-блокировок, ссылки на освобожденные объекты, задержка выключения компьютера более чем на 20 минут и запросы блоков памяти нулевого размера.

Настройки Driver Verifier хранятся в разделе реестра HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management. Параметр `VerifyDriverLevel` содержит битовую маску, которая представляет активные параметры проверки. Параметр `VerifyDrivers` содержит имена отслеживаемых драйверов. (Эти значения не существуют в реестре до того момента, когда вы выберете драйверы для проверки в Driver Verifier Manager.) Если выбрать режим проверки всех драйверов (чего делать никогда не следует, потому что это приведет к значительному замедлению работы системы), параметру `VerifyDrivers` присваивается символ «звездочка» (*). В зависимости от внесенных изменений вам, возможно, придется перезагрузить систему для активизации выбранных проверок.

На ранней стадии процесса загрузки диспетчер памяти читает из реестра параметры, относящиеся к Driver Verifier. По ним он определяет проверяемые драйверы и активные настройки Driver Verifier (учтите, что при загрузке в безопасном режиме все настройки Driver Verifier игнорируются). Если вы выбрали хотя бы один драйвер для проверки, ядро проверяет имя каждого драйвера, загружаемого в память, по списку драйверов, выбранных для проверки. Для каждого драйвера устройства, встречающегося в обоих списках, ядро вызывает функцию `VfLoadDriver`, которая вызывает другие внутренние функции `Vf*` для замены ссылок драйвера на функции ядра ссылками на эквивалентные версии этих функций из Driver Verifier. Например, `ExAllocatePool` заменяется вызовом `VerifierAllocatePool`. Драйвер системы управления окнами (Win32k.sys) также вносит аналогичные изменения для использования эквивалентных функций Driver Verifier.

Параметры проверки, относящиеся к вводу/выводу

Различные параметры проверки, относящиеся к вводу/выводу:

- **I/O Verification** (Проверка ввода/вывода). После установки этого флажка диспетчер ввода/вывода выделяет память под IRP-пакеты для проверенных драйверов из особого пула и следит за их использованием. Кроме того, после завершения обработки IRP-пакета с недействительным статусом или при передаче диспетчеру ввода/вывода недействительного объекта устройства программа Driver Verifier инициирует системный сбой. Также установка этого флажка инициирует отслеживание IRP-пакетов с целью гарантировать корректные пометки их драйверами при асинхронном завершении, корректность управления адресами стека устройств и однократность удаления объектов устройств. Дополнительно программа случайным образом тестирует драйверы, отправляя им имитацию IRP-пакетов, отвечающих за управление электропитанием и WMI, меняя порядок перечисления устройств и в момент их завершения корректируя

статус IRP-пакетов WMI для PnP-устройств и системы электропитания. Таким способом тестируются драйверы, возвращающие неверный статус из своих процедур диспетчеризации. Наконец, программа Driver Verifier распознает ошибки повторной инициализации удаляемых блокировок, пока они удерживаются из-за отложенного удаления устройства.

- **DMA Checking** (Проверка DMA). DMA (Direct Memory Access — прямой доступ к памяти) представляет собой аппаратно поддерживаемый механизм, позволяющий устройствам передавать данные в физическую память и получать их оттуда без участия процессора. Диспетчер ввода/вывода предлагает ряд функций, при помощи которых драйверы инициируют DMA-операции и управляют ими. Установка данного флажка включает режим проверки корректности использования указанных функций и буферов, предоставляемых диспетчером ввода/вывода для DMA-операций.
- **Force Pending I/O Requests** (Принудительная обработка отложенных запросов ввода/вывода). Для многих устройств асинхронные запросы на ввод и вывод завершаются немедленно, поэтому драйверы не всегда умеют корректно обрабатывать отдельные операции асинхронного ввода/вывода. После установки этого флажка диспетчер ввода/вывода в ответ на вызов драйвером функции `IoCallDriver` случайным образом возвращает значение `STATUS_PENDING`, имитируя асинхронное завершение ввода/вывода.
- **IRP Logging** (Протоколирование IRP-пакетов). Данный флажок позволяет отслеживать и фиксировать, каким образом драйвер использует IRP-пакеты. Запись об этом сохраняется как WMI-информация. Служебная программа Dc2wmiparser.exe из WDK позволяет превратить ее в текстовый файл. Примечательно, что записываются только 20 IRP-пакетов для каждого из устройств. Каждый следующий IRP-пакет записывается поверх последнего добавленного пакета. После перезагрузки эта информация удаляется, поэтому, если данные мониторинга нужно проанализировать позже, следует запустить программу Dc2wmiparser.exe.

Параметры проверки, относящиеся к памяти

Ниже перечислены параметры проверки Driver Verifier, относящиеся к памяти. (Некоторые из них также связаны с операциями ввода/вывода.)

Special Pool

В режиме Special Pool функции выделения памяти из пула ограничивают выделяемый блок недействительными страницами, чтобы попытки выхода за ту или иную границу привели к нарушению прав доступа режима ядра, что приведет к сбою системы с идентификацией сбойного драйвера. Режим Special Pool также включает выполнение некоторых дополнительных проверок при выделении или освобождении памяти драйвером. При включении режима Special Pool функции выделения

памяти из пула выделяют область памяти ядра для Driver Verifier. Driver Verifier перенаправляет запросы на выделение памяти, поступающие от проверяемого драйвера, в специальный пул (вместо стандартного пула памяти режима ядра). Когда драйвер устройства выделяет память из специального пула, Driver Verifier выравнивает операции выделения памяти по границам четных страниц. Поскольку Driver Verifier заключает выделенную страницу между недействительными страницами, если драйвер устройства попытается выполнить чтение или запись за концом буфера, драйвер обратится к недействительной странице, и диспетчер памяти инициирует нарушение прав доступа режима ядра.

На рис. 6.31 изображен пример буфера, выделяемого Driver Verifier по поручению драйвера устройства при включенной проверке ошибок переполнения.

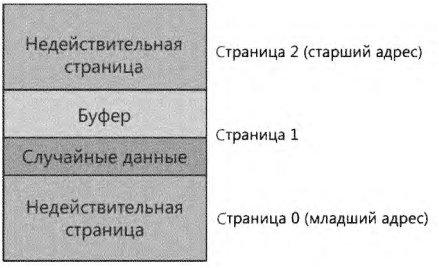


Рис. 6.31. Структура выделения буфера в специальном пуле

По умолчанию Driver Verifier пытается выполнить проверку переполнения. Для этого буфер, используемый драйвером устройства, размещается в конце выделенной страницы, а начало страницы заполняется повторяющимися случайными данными. Хотя включить проверку недозаполнения буфера в Driver Verifier Manager невозможно, проверку можно включить вручную — добавьте в раздел реестра HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management параметр `PoolTagOverruns` типа DWORD и присвойте ему 0 (или запустите программу Gflags.exe и выберите вариант Verify Start в разделе Kernel Special Pool Tag вместо значения по умолчанию Verify End). Когда в Windows включается проверка недозаполнения, Driver Verifier выделяет буфер драйвера в начале страницы (а не в конце).

Конфигурация проверки переполнения также включает некоторые меры по проверке недозаполнения. Когда драйвер освобождает свой буфер, чтобы вернуть память Driver Verifier, Driver Verifier убеждается в том, что закономерность не нарушена. Если же данные изменились, значит, драйвер устройства вышел за границу и записал данные в память за пределами буфера.

При специальном выделении памяти система также проверяет уровень IRQL процессора в момент выделения и освобождения памяти. Эта проверка перехватывает ошибки, встречающиеся в некоторых драйверах устройств: выделение выгружаемой памяти на уровне IRQL DPC/Dispatch и выше.

Вы также можете настроить проверки специального пула вручную; для этого в раздел HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management включается параметр `PoolTag` типа DWORD, представляющий теги выделения, используемые системой для специального пула. Таким образом, даже если программа Driver Verifier не настроена на проверку конкретного драйвера устройства, но тег, связанный драйвером с выделяемой памятью, совпадает с тегом, заданным в параметре реестра `PoolTag`, функции выделения памяти выделяют память из специального пула. Если присвоить `PoolTag` значение 0x2a или универсальную маску (*), вся память, выделяемая драйверами, будет выделяться из специального пула — при условии наличия достаточной визуальной или физической памяти (если свободных страниц не хватает, драйверы возвращаются к выделению памяти из обычного пула).

Pool tracking

При включении режима Pool tracking диспетчер памяти проверяет при выгрузке драйвера, освободил ли драйвер все выделенные блоки памяти. Если он этого не сделал, в системе происходит фатальный сбой с указанием драйвера-нарушителя. Driver Verifier также выводит общую статистику использования пула на вкладке Pool Tracking Driver Verifier Manager (чтобы вызвать ее из основного пользовательского интерфейса, установите переключатель Display Information About the Currently Verified Drivers и дважды щелкните на кнопке Next). Кроме того, можно воспользоваться командой отладчика ядра `!verifier`. Эта команда выводит более подробную информацию, чем Driver Verifier; она будет особенно полезной для разработчиков драйверов.

Режимы Pool Tracking и Special Pool распространяются не только на явные вызовы выделения памяти (такие, как `ExAllocatePoolWithTag`), но и на вызовы других API-функций ядра, которые неявно выделяют память из пула: `IoAllocateMdl`, `IoAllocateIrp` и другие функции создания IRP-пакетов; различные строковые API-функции Rtl-библиотеки; и `IoSetCompletionRoutineEx`.

Также режим Pool Tracking распространяется на функцию начисления квот пулов. Вызов `ExAllocatePoolWithQuotaTag` начисляет в квоту пула текущего процесса количество выделяемых байтов. Если такой вызов совершается из процедуры DPC, процесс, к которому будет применено начисление, непредсказуем, потому что процедуры DPC могут выполняться в контексте любого процесса. Режим Pool Tracking также проверяет вызовы этой функции из контекста процедур DPC.

Программа Driver Verifier также может отслеживать заблокированные страницы памяти; при этом дополнительно отслеживаются страницы, оставшиеся заблокированными после завершения операции ввода/вывода, и генерируется код фатальной ошибки `DRIVER_LEFT_LOCKED_PAGES_IN_PROCESS` вместо `PROCESS_HAS_LOCKED_PAGES` — первый код однозначно сообщает, что за ошибку несет ответственность драйвер. Также при этом указывается функция, ответственная за блокировку страниц.

Force IRQL Checking

Одна из самых распространенных ошибок драйверов устройств — обращение драйвера к выгружаемым данным или коду в тот момент, когда процессор, на котором выполняется драйвер устройства, работает на повышенном уровне IRQL. Диспетчер памяти не может обработать ошибку страницы, когда уровень IRQL находится на уровне DPC/Dispatch и выше. Часто система не обнаруживает обращения к выгружаемым данным при работе процессора на повышенном уровне IRQL, потому что выгружаемые данные в момент обращения физически присутствуют в памяти. Однако в других случаях данные оказываются выгруженными, что приводит к системному фатальному сбою с кодом `IRQL_NOT_LESS_OR_EQUAL` (т. е. «уровень IRQL не был меньше или равен уровню, необходимому для операции — в данном случае обращения к выгружаемой памяти»).

Хотя поиск ошибок такого рода в драйверах устройств обычно является сложной задачей, Driver Verifier упрощает ее. Если включить режим Force IRQL Checking, Driver Verifier принудительно выведет весь выгружаемый код режима ядра и данные из системного рабочего набора каждый раз, когда проверяемый драйвер устройства поднимет уровень IRQL. Для этого используется внутренняя функция `MiTrimAllSystemPagableMemory`. При включении этого режима каждый раз, когда проверяемый драйвер обращается к выгружаемой памяти при повышенном уровне IRQL, система мгновенно обнаруживает нарушение, и в описании системного фатального сбоя указывается драйвер-нарушитель.

Другая распространенная причина сбоев драйверов — некорректное использование IRQL, когда объекты синхронизации являются частью структур данных, которые выгружаются из памяти, а затем используются для ожидания. Объекты синхронизации никогда не должны выгружаться, потому что диспетчер должен обращаться к ним на повышенном уровне IRQL, что приведет к фатальному сбою. Driver Verifier проверяет, какие из следующих структур присутствуют в выгружаемой памяти: `KTIMER`, `KMUTEX`, `KSPIN_LOCK`, `KEVENT`, `KSEMAPHORE`, `ERESOURCE` и `FAST_MUTEX`.

Low Resources Simulation

При включении режима Low Resources Simulation программа Driver Verifier случайным образом выдает ошибки при операциях выделения памяти, выполняемых проверяемыми драйверами устройств. В прошлом многие драйверы устройств писались в предположении, что память ядра всегда доступна; при нехватке этой памяти драйверу устройства не приходилось беспокоиться об этом, потому что в системе все равно происходил фатальный сбой. Но поскольку ситуации нехватки памяти могут действовать временно, а современные мобильные устройства уступают по мощности большим машинам, драйверы устройств должны правильно обрабатывать неудачные попытки выделения памяти, свидетельствующие об исчерпании свободной памяти ядра.

К функциям драйвера, которые будут сопровождаться случайными отказами, относятся `ExAllocatePool*`, `MmProbeAndLockPages`, `MmMapLockedPagesSpecifyCache`, `MmMapIoSpace`, `MmAllocateContiguousMemory`, `MmAllocatePagesForMdl`, `IoAllocateIrp`, `IoAllocateMdl`, `IoAllocateWorkItem`, `IoAllocateErrorLogEntry`, `IOSetCompletionRoutineEx` и различные строковые API-функции Rtl-библиотеки, выделяющие память из пула. Driver Verifier также отказывает в некоторых попытках выделения памяти, выполняемых GDI-функциями ядра (полный список приведен в документации WDK). Также можно задать следующие параметры:

- **вероятность отказа при попытке выделения памяти** (6 % по умолчанию);
- **какие приложения должны быть задействованы в тестировании** (все по умолчанию);
- **задействованные теги пулов** (все по умолчанию);
- **задержка перед началом моделирования сбоев** (по умолчанию 7 минут после загрузки системы — время, достаточное для прохождения критического периода инициализации, в котором условие нехватки памяти может помешать загрузке драйвера устройства).

Все эти настройки можно изменить при помощи параметров командной строки verifier.exe.

После периода задержки Driver Verifier начинает случайным образом отказывать в попытках выделения памяти для проверяемых драйверов устройств. Если драйвер некорректно обрабатывает попытки выделения памяти, скорее всего, это приведет к фатальному сбою системы.

Systematic Low Resources Simulation

По аналогии с режимом Low Resources Simulation, в этом режиме некоторые вызовы ядра и Ndis.Sys (для сетевых драйверов) завершаются отказом, но на систематической основе и с анализом стека вызовов в точке внедрения сбоя. Если драйвер корректно обработал сбой, повторное внедрение сбоев в этот стек вызовов не происходит. Это позволяет разработчику драйвера систематично рассмотреть проблемы, исправить обнаруженную проблему и затем перейти к другой проблеме. Анализ стека вызовов — относительно затратная операция, поэтому проверять более одного драйвера в этом режиме не рекомендуется.

Разное

Некоторые из проверок, отнесенных Driver Verifier к категории «разное» (Miscellaneous), позволяют обнаружить факт освобождения памяти, содержащей некоторые активные системные структуры. Например, Driver Verifier проверяет:

- **Активные рабочие элементы в освобожденной памяти.** Драйвер вызывает функцию `ExFreePool` для освобождения блока из пула, в котором присутству-

ет один или несколько рабочих элементов, поставленных в очередь вызовом `IoQueueWorkItem`.

- **Активные ресурсы в освобожденной памяти.** Драйвер вызывает функцию `ExFreePool` перед вызовом `ExDeleteResource` для уничтожения объекта `ERESOURCE`.
- **Активные резервные списки в освобожденной памяти.** Драйвер вызывает функцию `ExFreePool` перед вызовом `ExDeleteNPagedLookasideList` или `ExDeletePagedLookasideList` для уничтожения резервного списка.

Наконец, при включенной проверке Driver Verifier выполняет некоторые автоматические проверки, которые невозможно включать или отключать по одной. К их числу относятся:

- Вызов `MmProbeAndLockPages` или `MmProbeAndLockProcessPages` для MDL-списка с неправильными флагами. Например, функция `MmProbeAndLockPages` не должна вызываться для MDL-списка, созданного вызовом `MmBuildMdlForNonPagedPool`.
- Вызов `MmMapLockedPages` для MDL-списка с неправильными флагами. Например, функция `MmMapLockedPages` не должна вызываться для MDL-списка, уже отображенного на системный адрес. Другой пример некорректного поведения драйвера — вызов `MmMapLockedPages` для MDL-списка, который еще не был заблокирован.
- Вызов `MmUnlockPages` или `MmUnmapLockedPages` для неполного MDL-списка (созданного вызовом `IoBuildPartialMdl`).
- Вызов `MmUnmapLockedPages` для MDL-списка, не отображенного на системный адрес.
- Создание объектов синхронизации (таких, как события или мьютексы) в памяти `NonPagedPoolSession`.

Driver Verifier — один из представителей арсенала средств проверки и отладки, доступных для разработчиков драйверов устройств. Во многих драйверах устройств, запускаемых с Driver Verifier, обнаруживаются ошибки. Таким образом, применение Driver Verifier повышает качество всего кода режима ядра, выполняемого в Windows.

PnP-диспетчер

PnP-диспетчер является основным компонентом, от которого зависит способность Windows к распознаванию и принятию изменений в аппаратной конфигурации. Пользователю не требуется разбираться в тонкостях настройки устройств при их установке и удалении. Например, именно PnP-диспетчер позволяет помещенному на док-станцию портативному компьютеру с системой Windows автоматически обнаружить дополнительные устройства док-станции и сделать их доступными пользователю.

Поддержка технологии Plug and Play требует взаимодействия на уровнях оборудования, драйверов устройств и операционной системы. В Windows такая поддержка

базируется на промышленных стандартах перечисления и идентификации подключенных к шинам устройств. Например, стандарт USB определяет способ самоидентификации устройств, подключенных к шине USB. На этой основе в Windows реализуются следующие PnP-возможности:

- PnP-диспетчер автоматически распознает установленные устройства. Процесс распознавания включает в себя перечисление присоединенных к системе устройств при загрузке, а также их обнаружение или удаление во время работы системы.
- PnP-диспетчер выделяет аппаратные ресурсы, собирая информацию о требованиях к ним со стороны устройств (прерывания, диапазоны адресов ввода/вывода, регистры ввода/вывода или ресурсы, связанные с шинами). В ходе *арбитража ресурсов* (resource arbitration) PnP-диспетчер распределяет ресурсы между устройствами с учетом их требований. Так как устройства могут быть добавлены в систему после распределения ресурсов на этапе загрузки, PnP-диспетчер должен уметь перераспределять ресурсы в соответствии с потребностями динамически добавляемых устройств.
- Загрузка соответствующих драйверов — еще одна функция PnP-диспетчера. По результатам идентификации устройства он определяет, присутствует ли в системе управляющий этим устройством драйвер. При обнаружении такого драйвера PnP-диспетчер дает диспетчеру ввода/вывода команду его загрузить. Если же нужный драйвер не установлен, PnP-диспетчер режима ядра указывает PnP-диспетчеру пользовательского режима установить устройство, прося пользователя указать местоположение нужных драйверов.
- PnP-диспетчер реализует механизмы, позволяющие приложениям и драйверам обнаруживать изменения в аппаратной конфигурации. Иногда для работы драйверов и приложений требуется определенное устройство, поэтому в Windows включены средства, дающие им возможность запрашивать информацию о наличии, добавлении и удалении устройств.
- PnP-диспетчер предоставляет место для хранения состояния устройств и принимает участие в настройке системы, обновлениях, переносе и автономном управлении образами.
- PnP-диспетчер поддерживает устройства, подсоединенные к сети, например сетевые проекторы и принтеры. Для этого он позволяет специальным драйверам шины распознавать сеть как шину и создавать для работающих на ее основе устройств узлы.

Уровень поддержки технологии Plug and Play

Хотя операционная система Windows полностью поддерживает технологию Plug and Play, конкретный уровень поддержки зависит от подключенных к системе устройств и установленных в ней драйверов. Если хотя бы одно устройство или

драйвер не отвечает стандарту Plug and Play, уровень поддержки технологии Plug and Play может быть снижен. Более того, не поддерживающий этот стандарт драйвер может помешать использованию других устройств системой. В табл. 6.6 перечислены результаты различных комбинаций устройств и драйверов, как с поддержкой технологии Plug and Play, так и без таковой.

Таблица 6.6. Поддержка технологии Plug and Play устройствами и драйверами

Тип устройства	**Тип драйвера**	
	Plug and Play	**Не Plug and Play**
PnP-совместимое	Полная поддержка	Без поддержки
PnP-несовместимое	Частичная поддержка	Без поддержки

Несовместимое со стандартом PnP устройство (например, устаревшая звуковая карта ISA) не поддерживает автоматическое определение. Так как операционная система не в курсе, где физически находится оборудование, определенные операции, например извлечение портативного компьютера из док-станции, переход в спящий режим и гибернация, просто отключены. Но если для такого устройства вручную установить PnP-драйвер, он сможет, по крайней мере, использовать ресурсы, которые ему будет выделять PnP-диспетчер.

К несовместимым с PnP драйверам относятся, например, устаревшие драйверы, разработанные для Windows NT 4. Они могут функционировать в более поздних версиях Windows, но PnP-диспетчер не может динамически перераспределять назначаемые таким устройствам ресурсы. Допустим, устаревшее устройство использует для ввода и вывода диапазоны памяти A и B, а в процессе загрузки PnP-диспетчер выделяет ему диапазон A. Если позже к системе будет добавлено устройство, способное пользоваться только диапазоном A, PnP-диспетчер не сможет перенастроить драйвер первого устройства на диапазон B. В результате второе устройство не получит нужные ему ресурсы, и система не сможет с ним работать. Кроме того, устаревшие драйверы мешают переходу системы в режим сна или гибернации. (О них мы поговорим в разделе «Диспетчер электропитания» далее в этой главе.)

Перечисление устройств

Перечисление устройств (device enumeration) происходит при загрузке системы, возобновлении из гибернации или при получении явного распоряжения (например, при нажатии кнопки Обновить конфигурацию оборудования (Scan for Hardware Changes) в интерфейсе диспетчера устройств). PnP-диспетчер строит дерево устройств (см. далее) и сравнивает его с хранимым текущим деревом, полученным в результате предыдущего перечисления (если оно есть). При загрузке или выходе из гибернации хранимое дерево устройств пусто. Вновь обнаруженные и удаленные устройства

требуют особых действий — например, загрузки соответствующих драйверов (для вновь обнаруженных устройств) и оповещения драйверов об удалении устройств.

PnP-диспетчер начинает перечисление устройств с виртуального драйвера шины с именем *Root*, который представляет всю систему и играет роль драйвера шины для драйверов, не поддерживающих стандарт Plug and Play, а также для уровня аппаратных абстракций (HAL). Последний действует как драйвер шины, перечисляющий напрямую подключенные к материнской плате устройства и такие системные компоненты, как аккумуляторы. Определяя основную шину (обычно это шина PCI) и устройства типа аккумуляторов и вентиляторов, HAL на самом деле полагается на описание оборудования, зафиксированное в реестре программой установки.

Драйвер основной шины перечисляет устройства на ней, по возможности обнаруживая другие шины, драйверы которых инициализируются PnP-диспетчером. Эти драйверы, в свою очередь, могут обнаружить другие устройства, включая вспомогательные шины. Такой рекурсивный процесс — перечисление, загрузка драйвера, дальнейшее перечисление — продолжается до обнаружения и конфигурирования всех устройств в системе.

По мере поступления сообщений от драйверов шин об обнаруженных устройствах PnP-диспетчер формирует внутреннее дерево, называемое *деревом устройств* (device tree), которое отражает взаимосвязи между устройствами. Элементы этого дерева называются *узлами устройств* (devnodes). Каждый узел содержит информацию об объектах устройств, представляющих устройства, а также прочие сведения, относящиеся к PnP-устройствам и записанные в узел PnP-диспетчером. Пример упрощенного дерева устройств представлен на рис. 6.32. Шина PCI является основной шиной системы, к ней подключаются шины USB, ISA и SCSI.

Служебная программа диспетчер устройств, доступная из оснастки Управление компьютером (Computer Management), которая вызывается командой

Рис. 6.32. Пример дерева устройств

Программы ▸ Администрирование (Programs ▸ Administrative Tools) меню Пуск (Start), а также по ссылке Диспетчер устройств (Device Manager) панели управления, демонстрирует список устройств в конфигурации, предлагаемой по умолчанию. Кроме того, можно выбрать в меню Вид (View) диспетчера устройств вариант Устройства по подключению (Devices By Connection) и представить список в виде иерархического дерева, как показано на рис. 6.33.

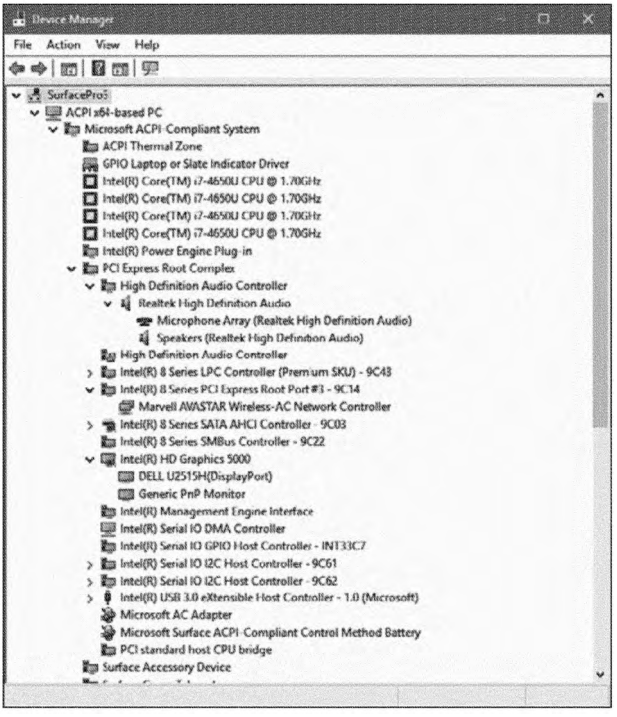


Рис. 6.33. Дерево устройств в диспетчере устройств

ЭКСПЕРИМЕНТ: ПРОСМОТР ДАМПА ДЕРЕВА УСТРОЙСТВ

Чтобы получить более подробную информацию об устройствах, можно воспользоваться командой !devnode отладчика ядра. Передавая параметры 0 1, вы получите дамп внутренних структур узлов дерева. При этом элементы структур выводятся с отступами, отражающими их положение в общей иерархии:

```
lkd> !devnode 0 1
Dumping IopRootDeviceNode (= 0x85161a98)
DevNode 0x85161a98 for PDO 0x84d10390
  InstancePath is "HTREE\ROOT\0"
  State = DeviceNodeStarted (0x308)
  Previous State = DeviceNodeEnumerateCompletion (0x30d)
  DevNode 0x8515bea8 for PDO 0x8515b030
```

```
  DevNode 0x8515c698 for PDO 0x8515c820
    InstancePath is "Root\ACPI_HAL\0000"
    State = DeviceNodeStarted (0x308)
Previous State = DeviceNodeEnumerateCompletion (0x30d)
    DevNode 0x84d1c5b0 for PDO 0x84d1c738
      InstancePath is "ACPI_HAL\PNP0C08\0"
      ServiceName is "ACPI"
      State = DeviceNodeStarted (0x308)
      Previous State = DeviceNodeEnumerateCompletion (0x30d)
      DevNode 0x85ebf1b0 for PDO 0x85ec0210
        InstancePath is "ACPI\GenuineIntel_-_x86_Family_6_Model_15\_0"
        ServiceName is "intelppm"
        State = DeviceNodeStarted (0x308)
        Previous State = DeviceNodeEnumerateCompletion (0x30d)
      DevNode 0x85ed6970 for PDO 0x8515e618
        InstancePath is "ACPI\GenuineIntel_-_x86_Family_6_Model_15\_1"
        ServiceName is "intelppm"
        State = DeviceNodeStarted (0x308)
        Previous State = DeviceNodeEnumerateCompletion (0x30d)
      DevNode 0x85ed75c8 for PDO 0x85ed79e8
        InstancePath is "ACPI\ThermalZone\THM_"
        State = DeviceNodeStarted (0x308)
        Previous State = DeviceNodeEnumerateCompletion (0x30d)
      DevNode 0x85ed6cd8 for PDO 0x85ed6858
        InstancePath is "ACPI\pnp0c14\0"
        ServiceName is "WmiAcpi"
        State = DeviceNodeStarted (0x308)
        Previous State = DeviceNodeEnumerateCompletion (0x30d)
      DevNode 0x85ed7008 for PDO 0x85ed6730
        InstancePath is "ACPI\ACPI0003\2&daba3ff&2"
        ServiceName is "CmBatt"
        State = DeviceNodeStarted (0x308)
        Previous State = DeviceNodeEnumerateCompletion (0x30d)
      DevNode 0x85ed7e60 for PDO 0x84d2e030
InstancePath is "ACPI\PNP0C0A\1"
        ServiceName is "CmBatt"
...
```

Выводимая для каждого узла информация включает в себя параметр `InstancePath`, который представляет собой имя подраздела перечисленного устройства, хранящееся в разделе HKLM\SYSTEM\CurrentControlSet\Enum, и параметр `ServiceName`, соответствующий подразделу драйвера устройства в разделе HKLM\SYSTEM\CurrentControlSet\Services. Для просмотра таких ресурсов, как прерывания, порты и диапазоны памяти, назначенные каждому узлу дерева устройств, используйте команду `!devnode` с параметром `0 3`.

Стеки устройств

Узлы дерева устройств создаются PnP-диспетчером, а драйверные объекты и объекты устройств служат для управления связью между узлами и ее логического представления. Эта связь называется *стеком устройств* (device stack) и может рассматриваться как упорядоченный список пар «объект устройства/драйвер».

Стек устройств строится снизу вверх. На рис. 6.34 показан пример узла устройства (повторение рис. 6.6) с семью объектами устройств (все они управляют одним устройством). Каждый узел устройства содержит как минимум два объекта устройств (PDO и FDO), но их может быть и больше. Стек устройств содержит:

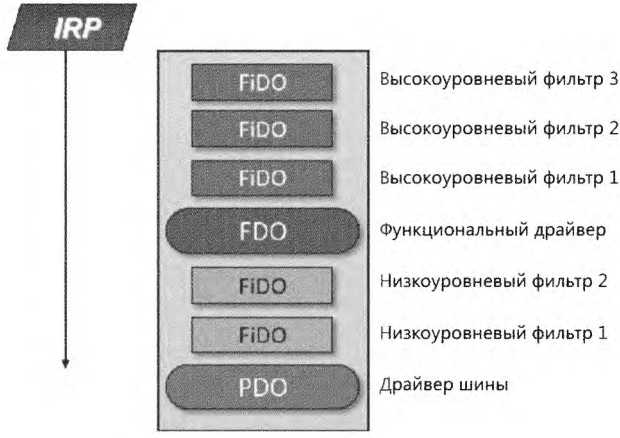


Рис. 6.34. Узел устройства (стек устройств)

- Объект физического устройства (Physical Device Object, PDO) создается драйвером шины по заданию PnP-диспетчера, когда этот драйвер, перечисляя устройства на своей шине, сообщает о наличии какого-либо устройства. PDO представляет физический интерфейс устройства и всегда находится внизу стека устройств.
- Один или несколько необязательных объектов фильтрующих устройств (Filter Device Objects, FiDO), которые находятся между PDO и FDO (см. далее в списке), создаются фильтрующими драйверами шин. Они называются *низкоуровневыми фильтрами* (термин «низкоуровневый» интерпретируется относительно FDO). Они могут использоваться для перехвата IRP-пакетов, исходящих от FDO и передаваемых в направлении драйвера шины (которые могут представлять интерес для фильтров шин).
- Один (и только один) объект функционального устройства (Functional Device Object, FDO) создается так называемым *функциональным драйвером*, который загружает PnP-диспетчер для управления обнаруженным устройством. FDO представляет собой логический интерфейс устройства, обладающий наиболее полной информацией о функциональности, предоставляемой устройством. Функциональный драйвер может выступать и в роли драйвера шины, если к устройству, представляемому объектом функционального устройства, подключены другие устройства. Этот драйвер часто создает интерфейс с объектом физического устройства, соответствующего данному объекту функционального устройства, что позволяет приложениям и другим драйверам открывать устрой-

ство и взаимодействовать с ним. Иногда функциональные драйверы делят на драйверы класса, порта и мини-порта, совместно управляющие вводом/выводом для FDO.

- Один или несколько необязательных объектов фильтрующих устройств (FiDO), расположенных поверх FDO, создаются высокоуровневыми фильтрующими драйверами. Они называются *высокоуровневыми фильтрами* и первыми получают доступ к заголовку IRP для PDO.

ПРИМЕЧАНИЕ Всем объектам устройств на рис. 6.34 присваиваются разные имена, чтобы их было проще распознавать. Тем не менее все они являются экземплярами структур `DEVICE_OBJECT`.

Стеки устройств строятся снизу вверх благодаря возможности диспетчера ввода/вывода формировать слои, поэтому IRP-пакеты перемещаются по стеку сверху вниз. При этом решение о завершении IRP-пакета может быть принято на любом уровне стека (см. раздел «Последовательность обработки IRP» этой главы).

Загрузка драйверов для стека устройств

Как PnP-диспетчер находит правильные драйверы в процессе построения стека устройств? В реестре эта информация хранится в трех разных разделах (и их подразделах), перечисленных в табл. 6.7 (*CSS* — сокращение для *CurrentControlSet*).

Таблица 6.7. Важнейшие разделы реестра для загрузки драйверов PnP

Раздел реестра	Короткое имя	Описание
HKLM\System\CCS\Enum	Раздел оборудования	Настройки известных устройств
HKLM\System\CCS\Control\Class	Раздел класса	Настройки типов устройств
HKLM\System\CCS\Services	Раздел программного обеспечения	Настройки драйверов

Когда драйвер шины выполняет перечисление и обнаруживает новое устройство, он сначала создает PDO для представления обнаруженного физического устройства. Затем он оповещает об этом PnP-диспетчера вызовом функции `IoInvalidateDeviceRelations` (документированной в WDK) со значением из перечисления `BusRelations` и PDO, сообщая тем самым PnP-диспетчеру об обнаружении изменения на шине. В ответ PnP-диспетчер запрашивает у драйвера шины (с помощью IRP-пакета) идентификатор устройства.

Идентификаторы зависят от конкретной шины; например, для шины USB идентификатор составляется из идентификатора производителя (Vendor ID, VID) и идентификатора продукта (Product ID, PID), который производитель присваивает устройству. Для устройства PCI он состоит из похожего идентификатора производителя и идентификатора устройства, однозначно идентифицирующего устройство для

производителя (а также некоторых необязательных компонентов; за дополнительной информацией о форматах идентификаторов устройств обращайтесь к WDK). В совокупности эти идентификаторы образуют то, что в спецификации Plug and Play называют *идентификатором устройства* (device ID). Также PnP-диспетчер запрашивает у драйвера шины идентификатор экземпляра (instance ID), позволяющий различать экземпляры одного и того же устройства. Этот идентификатор либо описывает положение относительно шины (например, порт USB), либо представляет собой полностью уникальный дескриптор (к примеру, серийный номер).

Комбинация идентификаторов устройства и экземпляра дает идентификатор экземпляра устройства (Device Instance ID, DIID), при помощи которого PnP-диспетчер ищет раздел устройства в разделе оборудования (см. табл. 6.7). Имена подразделов этого раздела строятся по схеме <перечислитель><идентификатор_устройства><идентификатор_экземпляра>, где перечислитель — драйвер шины, идентификатор_устройства — тип устройства, а идентификатор_экземпляра однозначно идентифицирует экземпляр одного из нескольких одинаковых устройств.

На рис. 6.35 представлен пример подраздела оборудования для видеокарты Intel. Раздел устройства содержит данные описания, к числу которых принадлежат параметры `Service` и `ClassGUID` (которые берутся из INF-файла драйвера при установке), используемые PnP-диспетчером для поиска драйверов устройства:

- Параметр `Service` ищется в разделе `Software`, а в параметре `ImagePath` хранится путь к драйверу (SYS-файл). На рис. 6.36 показан подраздел программного обеспечения с именем `igfx` (см. рис. 6.35), в котором хранится драйвер видеокарты

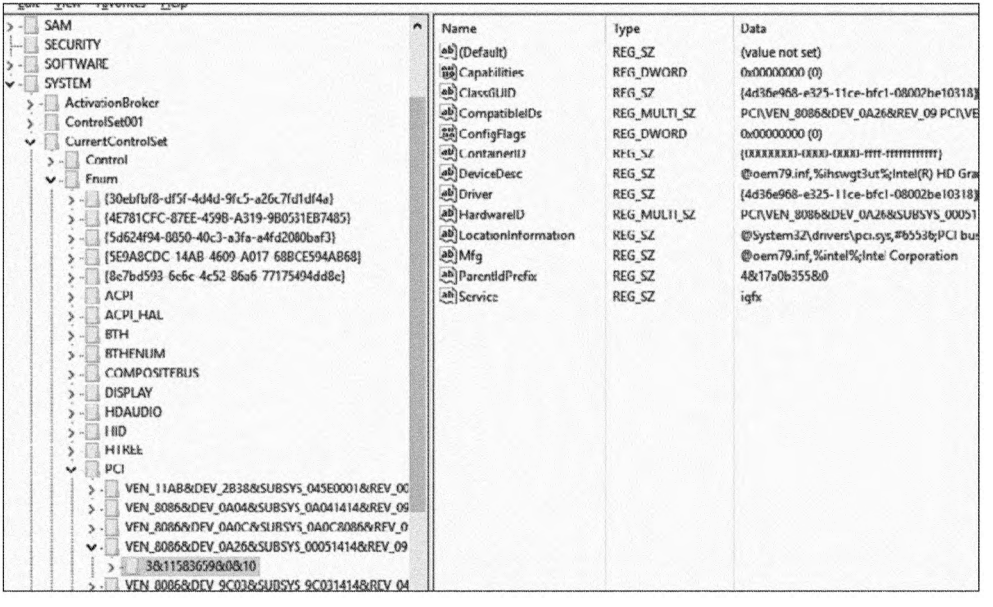


Рис. 6.35. Пример подраздела оборудования

Intel. PnP-диспетчер загружает этот драйвер (если он не загружен ранее) и вызывает его процедуру добавления устройства, где драйвер создает FDO.

- Если в разделе присутствует параметр с именем `LowerFilters`, он содержит список строк драйверов, которые должны загружаться как низкоуровневые драйверы (из подраздела `Software`). PnP-диспетчер загружает эти драйверы перед загрузкой драйвера, связанного с упомянутым выше параметром `Service`.

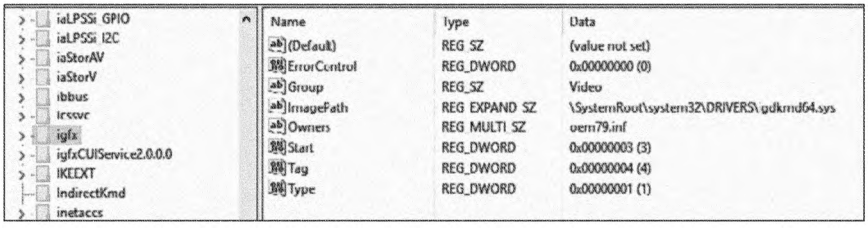


Рис. 6.36. Пример подраздела Software

- Если в разделе присутствует параметр с именем `UpperFilters`, он содержит список строк драйверов (из раздела программного обеспечения — по аналогии с `LowerFilters`). PnP-диспетчер загружает их по аналогии с тем, как загружается драйвер, на который ссылается параметр `Service`.

- Параметр `ClassGUID` представляет общий тип устройства (экран, клавиатура, диск и т. д.) и ссылается на подраздел раздела `Class` (см. табл. 6.7). Раздел представляет значения, применимые ко всем драйверам для устройств этого типа. В частности, если параметры `LowerFilters` и/или `UpperFilters` присутствуют, они интерпретируются так же, как одноименные значения из раздела оборудования данного устройства. Например, это позволяет загружать высокоуровневые фильтры для клавиатурных устройств независимо от конкретной модели клавиатуры или фирмы-производителя. На рис. 6.37 показан раздел класса для клавиатурных устройств. Обратите внимание на удобное имя (`Keyboard`), хотя в данном случае важен только код GUID (решение по конкретному классу предоставляется в установочном INF-файле). В параметре `UpperFilters` указан системный драйвер класса клавиатуры, который всегда загружается как часть узла устройства клавиатуры. (Также обратите внимание на параметр `IconPath`, который определяет значок для типа клавиатурных устройств в интерфейсе диспетчера устройств.)

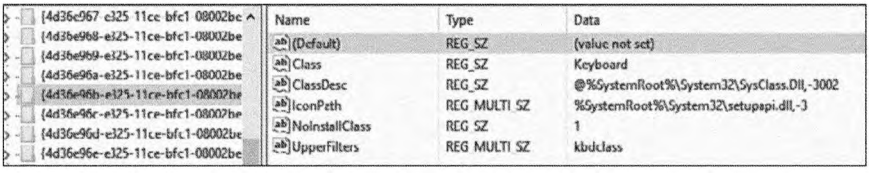


Рис. 6.37. Пример подраздела Software

В итоге загрузка драйверов для узла устройства происходит в следующем порядке:

1. Загружается драйвер шины и создается PDO.
2. Загружаются все низкоуровневые фильтры, перечисленные в разделе экземпляра оборудования, в заданном порядке, с созданием объектов фильтрующих устройств (FiDO на рис. 6.34).
3. Загружаются все низкоуровневые фильтры, перечисленные в соответствующем разделе класса, в заданном порядке, с созданием их объектов FiDO.
4. Загружается драйвер, указанный в параметре `Service`, с созданием его объекта FDO.
5. Загружаются все высокоуровневые фильтры, перечисленные в разделе экземпляра оборудования, в заданном порядке, с созданием объектов фильтрующих устройств (FiDO на рис. 6.34).
6. Загружаются все высокоуровневые фильтры, перечисленные в соответствующем разделе класса, в заданном порядке, с созданием их объектов FiDO.

Для многофункциональных устройств (таких, как многофункциональные принтеры или сотовые телефоны с интегрированными фотокамерами и плеерами) Windows поддерживает идентификаторы контейнеров (container ID), привязываемые к узлу устройств. Это уникальный для конкретного экземпляра физического устройства GUID-код, используемый всеми принадлежащими этому устройству функциональными узлами, как показано на рис. 6.38.

Идентификатор контейнера, как и идентификатор экземпляра, сообщается драйвером шины соответствующего аппаратного обеспечения. После перечисления

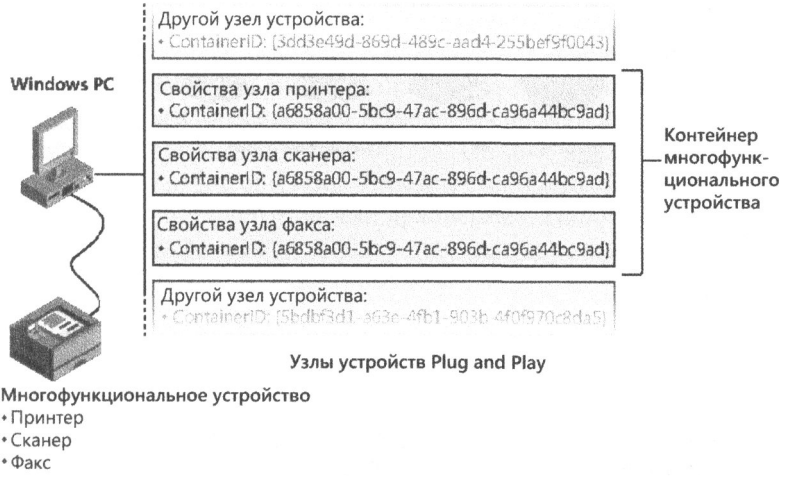


Рис. 6.38. Многофункциональный принтер с уникальным идентификатором с точки зрения PnP-диспетчера

устройства все связанные с одним PDO-объектом узлы начинают использовать один идентификатор контейнера. Так как в Windows уже поддерживаются многие установленные по умолчанию шины, например PnP-X, Bluetooth и USB, большинство драйверов устройств могут просто вернуть связанный с шиной идентификатор, на основе которого Windows генерирует соответствующий идентификатор контейнера. Для других видов устройств и шин драйвер может программно генерировать собственные уникальные идентификаторы.

Наконец, в ситуации, когда драйверы устройств не предоставляют идентификатор контейнера, Windows может сделать обоснованное предположение, по возможности запросив топологию шины через такой механизм, как ACPI. Поняв, является ли определенное устройство потомком другого, относится ли оно к сменным, допускает ли «горячее» подключение и доступно ли пользователю (в отличие от, например, внутренних компонентов материнской платы), Windows получает возможность корректно назначать идентификаторы контейнеров узлам, соответствующим многофункциональному устройству.

Для конечного пользователя выгода от группирования устройств по идентификаторам контейнеров проявляется при их просмотре в окне Устройства и принтеры (Devices And Printers) в современных версиях Windows. В этом окне сканнер, принтер и факс многофункционального принтера представляются как один графический элемент, а не как три разных устройства. К примеру, на рис. 6.39 принтер HP 6830 series идентифицируется как одно устройство.

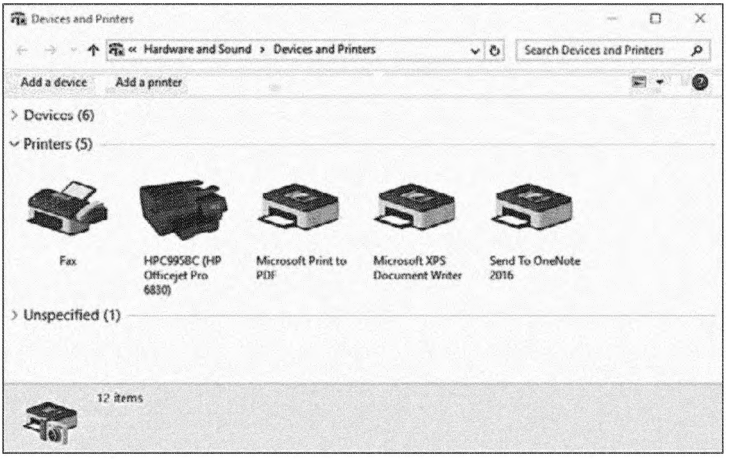


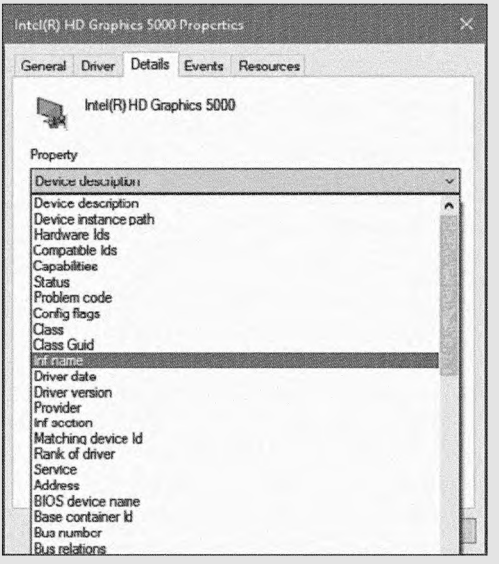
Рис. 6.39. Устройства и принтеры

ЭКСПЕРИМЕНТ: ПРОСМОТР ДЕТАЛЬНЫХ СВЕДЕНИЙ ОБ УЗЛАХ УСТРОЙСТВ В ДИСПЕТЧЕРЕ УСТРОЙСТВ

По умолчанию в диспетчере устройств выводятся подробные сведения об узле устройства на вкладке Сведения (Details). На этой вкладке имеется целый набор

полей, в том числе идентификатор экземпляра устройства для узла, идентификатор устройства, имя службы, фильтры и режимы управления электропитанием.

Ниже показан список на вкладке Сведения (Details), демонстрирующий некоторые типы доступной информации.



Поддержка Plug and Play драйверами

Для поддержки Plug and Play драйвер должен реализовать процедуру диспетчеризации Plug and Play (`IRP_MJ_PNP`), процедуру диспетчеризации управления питанием (`IRP_MJ_POWER` — см. раздел «Диспетчер электропитания» далее в этой главе) и процедуру добавления устройства. Однако драйверы шины должны поддерживать запросы Plug and Play, отличные от тех, которые поддерживаются функциональными или фильтрующими драйверами. Например, когда диспетчер PnP управляет перечислением устройств при загрузке системы, он запрашивает у драйверов шин описание устройств, которые они нашли на соответствующих шинах, при помощи IRP-пакетов PnP.

Функциональные и фильтрующие драйверы готовятся к управлению своими устройствами в процедурах добавления устройства, но они не взаимодействуют с физическими компонентами устройства напрямую. Вместо этого они ожидают, когда PnP-диспетчер отправит команду запуска устройства (дополнительный код IRP-пакета PnP `IRP_MN_START_DEVICE`) для конкретного устройства их процедуре диспетчеризации Plug and Play. Прежде чем отправлять команду запуска устройства, PnP-диспетчер выполняет арбитраж ресурсов, чтобы решить, какие ресурсы

следует выделить устройству. Команда запуска устройства включает выделение ресурсов, определенное PnP-диспетчером в результате арбитража. Получая команду запуска устройства, драйвер может настроить устройство для использования заданных ресурсов. Если приложение попытается открыть устройство, которое еще не прошло запуск, оно получит ошибку с указанием на то, что устройство не существует.

После того как устройство запустится, PnP-диспетчер сможет отправлять драйверу дополнительные команды Plug and Play, включая команды, относящиеся к удалению устройства из системы или переназначению ресурсов. Например, если пользователь вызовет функцию отключения/удаления устройства (рис. 6.40), щелкнув на значке USB-подключения в области уведомлений на панели задач, — например, для извлечения USB-диска, PnP-диспетчер отправит уведомление о запросе на удаление всем приложениям, зарегистрировавшимся на PnP-уведомления для данного устройства. Как правило, приложения используют для регистрации собственные дескрипторы, которые они закрывают во время обработки уведомлений о запросах на удаление. Если ни одно приложение не наложило вето на запрос на удаление, PnP-диспетчер отправляет команду запроса на удаление драйверу, которому принадлежит извлекаемое устройство (`IRP_MN_QUERY_REMOVE_DEVICE`). В этот момент у драйвера есть возможность отказать в удалении или убедиться в том, что все незавершенные операции ввода/вывода с устройством были завершены, чтобы отклонять все последующие запросы ввода/вывода для устройства. Если драйвер подтвердит запрос на удаление и для устройства не осталось открытых дескрипторов, PnP-диспетчер отправляет драйверу команду удаления (`IRP_MN_REMOVE_DEVICE`), чтобы драйвер перестал обращаться к устройству и освободил все ресурсы, выделенные драйвером для устройства.

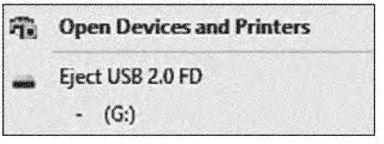


Рис. 6.40. Безопасное извлечение устройства

Когда PnP-диспетчеру потребуется переназначить ресурсы устройства, он сначала предлагает драйверу временно приостановить дальнейшие операции с устройством, отправляя драйверу команду запроса на остановку (`IRP_MN_QUERY_STOP_DEVICE`). Драйвер либо соглашается (если это не приведет к потере или повреждению данных), либо возвращает отказ. Как и в случае с командой запроса на удаление, если драйвер соглашается, драйвер завершает незавершенные операции ввода/вывода и не инициирует для устройства дальнейшие запросы ввода/вывода, которые не могут быть отменены с последующим перезапуском. Драйвер обычно ставит в очередь новые запросы ввода/вывода, чтобы переназначение ресурсов происходило прозрачно для приложений, в настоящее время обращающихся к устройству. Затем PnP-диспетчер отправляет драйверу команду остановки (`IRP_MN_STOP_DEVICE`).

В этот момент PnP-диспетчер может приказать драйверу назначить устройству другие ресурсы, а затем снова отправить драйверу команду запуска для устройства.

Различные команды Plug and Play фактически управляют работой конечного автомата с четко определенной таблицей переходов, показанной на рис. 6.41. (Диаграмма состояний отражает конечный автомат, реализованный функциональными драйверами. Драйверы шины реализуют более сложный автомат.) Каждый переход на рис. 6.41 помечен константой дополнительного кода IRP без префикса `IRP_MN_`. Одно из состояний, которые еще не обсуждались, активизируется командой PnP-диспетчера (`IRP_MN_SURPRISE_REMOVAL`). Эта команда выдается либо при неожиданном удалении устройства пользователем (например, при извлечении карты PCMCIA без использования функции безопасного удаления), либо при сбое устройства. Команда приказывает драйверу немедленно прекратить все взаимодействие с устройством, потому что устройство не связано с системой, и отменить все незавершенные запросы ввода/вывода.

Рис. 6.41. Переходы между состояниями Plug and Play для устройств

Установка драйвера

Если PnP-диспетчер обнаруживает устройство, для которого отсутствует драйвер, он рассчитывает, что установку проведет PnP-диспетчер пользовательского режима. При обнаружении устройства в процессе загрузки системы для него определяется узел устройства, но загрузка откладывается до запуска PnP-диспетчера в режиме пользователя, который реализован в файле Umpnpmgr.dll и выполняется как служба в стандартном хосте Svchost.exe.

Участвующие в установке драйвера компоненты показаны на рис. 6.42. Затемненные области соответствуют компонентам, предоставляемым системой, в то время как светлые объекты являются частью установочных файлов драйвера. Сначала драйвер шины информирует PnP-диспетчер о перечисленном устройстве (этап 1), сообщая его DIID. PnP-диспетчер проверяет, присутствует ли в реестре соответствующий функциональный драйвер. В случае отрицательного результата проверки PnP-диспетчер пользовательского режима уведомляется (этап 2) о новом устройстве через его DIID. Сначала PnP-диспетчер пользовательского режима пытается автоматически установить нужные драйверы. Если в процессе установки появляются диалоговые окна, на которые должен реагировать пользователь, а авторизованный в данный момент пользователь обладает правами администратора, PnP-диспетчер пользовательского режима запускает (этап 3) приложение Rundll32.exe (которое является хостом и для классических приложений .cpl панели управления) для выполнения мастера установки оборудования (%SystemRoot%\System32\Newdev.dll). Если у авторизованного в данный момент пользователя отсутствуют привилегии администратора (или в системе нет пользователей), а установка устройства невозможна без взаимодействия с ним, PnP-диспетчер пользовательского режима откладывает установку до момента, пока в системе не появится привилегированный пользователь. Мастер установки оборудования использует API-функции из Setupapi.dll и CfgMgr32.dll (диспетчер конфигурации) для поиска INF-файлов, соответствующих совместимым с обнаруженным устройством драйверам. При этом пользователю может быть предложено задействовать установочный диск с INF-файлами от производителя. Кроме того, мастер может обнаружить подходящий INF-файл в хранилище драйверов (%SystemRoot%\System32\DriverStore), в котором находятся драйверы, поставляемые вместе с Windows и загруженные через программу Windows Update. Установка выполняется в два этапа. Сначала разработчик сторонних драйверов импортирует пакет драйверов в хранилище, а затем система выполняет установку через процесс %SystemRoot%\System32\Drvinst.exe.

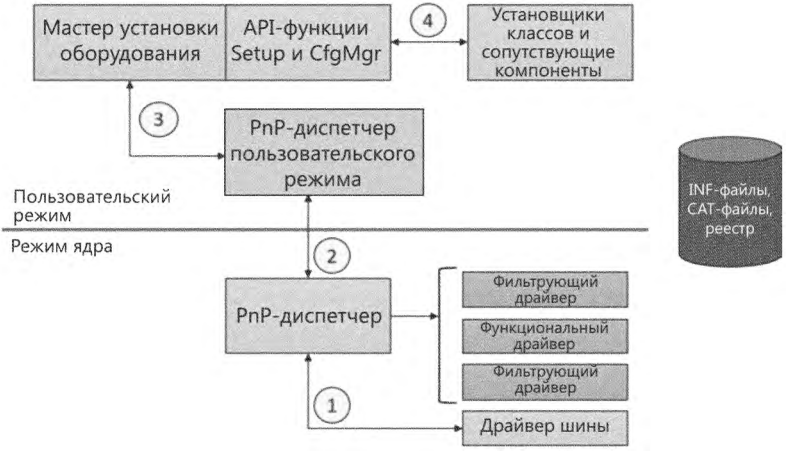


Рис. 6.42. Компоненты, участвующие в установке драйвера

Для поиска драйверов нового устройства процесс установки получает от драйвера шины список идентификаторов оборудования (см. выше) и идентификаторов совместимых устройств. Совместимые устройства имеют более общую природу — например, у USB-мыши конкретного производителя может быть специальная кнопка, которая выполняет некую уникальную операцию, но совместимый идентификатор обобщенной мыши может использовать более общий драйвер, входящий в поставку Windows, если конкретный драйвер недоступен — по крайней мере, он предоставляет базовую, общую функциональность мыши.

Эти идентификаторы описывают все способы идентификации устройства, предусмотренные в установочном файле драйвера (INF). Списки упорядочиваются таким образом, чтобы первыми оказались наиболее специфические характеристики устройства. Если совпадения обнаруживаются в нескольких INF-файлах, применяются следующие правила:

- более точные совпадения обладают более высоким приоритетом, чем менее точные;
- INF-файлы с цифровой подписью обладают более высоким приоритетом, чем неподписанные;
- более новые INF-файлы с цифровой подписью обладают более высоким приоритетом, чем старые файлы с подписью.

ПРИМЕЧАНИЕ Если обнаруженный идентификатор соответствует идентификатору совместимого устройства, мастер установки оборудования может запросить носитель с более актуальными версиями драйверов.

INF-файл определяет местоположение файлов функциональных драйверов и содержит команды, которые вводят нужные данные в раздел перечисления и в раздел класса драйвера и копируют необходимые файлы. INF-файл может заставить мастер установки оборудования (этап 4) запустить DLL установщика класса или компонента, участвующего в установке устройства. Эти модули выполняют операции, характерные для класса или устройства, — например, выводят диалоговые окна, в которых пользователь может задать параметры устройства. Наконец, после загрузки драйверов, образующих узел устройства, строится стек устройства/драйвера (этап 5).

ЭКСПЕРИМЕНТ: ПРОСМОТР INF-ФАЙЛА ДРАЙВЕРА

При установке драйвера или другого программного обеспечения, у которого есть INF-файл, система копирует этот файл в папку %SystemRoot%\Inf. В этой папке всегда присутствует файл Keyboard.inf, представляющий собой INF-файл для драйвера класса клавиатур. Откройте его в приложении Блокнот, и вы увидите примерно следующее (текст после символа «;» представляет собой комментарий):

```
;
; KEYBOARD.INF  -- This file contains descriptions of Keyboard class devices
;
;
; Copyright (c) Microsoft Corporation.  All rights reserved.

[Version]
Signature   ="$Windows NT$"
Class       =Keyboard
ClassGUID   ={4D36E96B-E325-11CE-BFC1-08002BE10318}
Provider    =%MSFT%
DriverVer=06/21/2006,10.0.10586.0

[SourceDisksNames]
3426=windows cd

[SourceDisksFiles]
i8042prt.sys    = 3426
kbdclass.sys    = 3426
kbdhid.sys      = 3426
...
```

Содержимое INF-файла имеет классический формат INI-файлов: имена разделов заключены в квадратные скобки, под ними перечисляются пары «ключ/значение», разделенные знаком =. INF-файл не «выполняется» последовательно, от начала до конца; вместо этого он строится как дерево. Некоторые значения ссылаются на разделы с конкретным именем, на которых продолжается выполнение (за подробностями обращайтесь к документации WDK).

Если поискать в этом файле сочетание *«.sys»*, найдется запись, указывающая PnP-диспетчеру пользовательского режима, что он должен установить драйверы i8042prt.sys и kbdclass.sys:

```
...
[i8042prt_CopyFiles]
i8042prt.sys,,,0x100

[KbdClass.CopyFiles]
kbdclass.sys,,,0x100
...
```

Перед установкой драйвера PnP-диспетчер пользовательского режима проверяет системную политику подписи драйверов. Если параметры, указанные для системы, блокируют установку неподписанных драйверов или предупреждают о попытках их установить, PnP-диспетчер пользовательского режима ищет в INF-файле драйвера запись, указывающую на папку (файл с расширением .cat), в которой содержится цифровая подпись драйвера.

Лаборатория Microsoft WHQL тестирует драйверы, поставляемые с Windows и предоставляемые изготовителями оборудования. Прошедший WHQL-тесты драйвер «подписывается» Microsoft. Это означает, что WHQL вычисляет *хеш*

или уникальное значение для представления файлов драйвера, в том числе его образа. Затем этот хеш подписывается с применением закрытого ключа Microsoft, помещается в CAT-файл и включается в дистрибутив Windows или передается изготовителю оборудования.

ЭКСПЕРИМЕНТ: ПРОСМОТР CAT-ФАЙЛОВ

При установке какого-либо компонента, файлы которого включают в себя CAT-файл, например драйвера, Windows копирует этот файл в папку %SystemRoot%\System32\Catroot. Откройте эту папку в Проводнике и найдите папку с CAT-файлами. В частности, в файлах Nt5.cat и Nt5ph.cat хранятся подписи и хеши страниц для системных файлов Windows.

Открыв один из CAT-файлов, вы увидите диалоговое окно с двумя вкладками. На вкладке Общие (General) представлены сведения о подписи в данном файле. На вкладке Каталог безопасности (Security Catalog) перечислены хеши компонентов, подписанных с помощью этого CAT-файла. На следующем снимке экрана CAT-файла для аудиодрайверов Intel показан хеш для SYS-файла аудиодрайвера. Остальные хеши в этом файле относятся к вспомогательным DLL-библиотекам, поставляемым вместе с драйвером.

При установке драйвера PnP-диспетчер пользовательского режима извлекает из CAT-файла подпись драйвера, расшифровывает ее с применением открытой части пары ключей Microsoft (открытый/закрытый ключ) и сравнивает полученный хеш с хешем файла устанавливаемого драйвера. При совпадении драйвер считается проверенным на соответствие требованиям WHQL. Если же совпадение не обнаружено, PnP-диспетчер пользовательского режима действует в соответствии с параметрами системной политики в отношении подписи драйвера. То есть он может запретить установку, предупредить пользователя о том, что драйвер не подписан, или просто установить драйвер.

ПРИМЕЧАНИЕ У драйверов, устанавливаемых программами, самостоятельно настраивающими реестр и копирующими файлы драйвера в систему, а также у драйверов, динамически загружаемых приложениями, наличие подписей не проверяется. Вместо этого они проверяются в соответствии с политикой подписи кода в режиме ядра (KMCS), рассматриваемой в главе 8. Политика же проверки подписей драйверов распространяется только на драйверы, устанавливаемые с помощью INF-файлов.

ПРИМЕЧАНИЕ PnP-диспетчер пользовательского режима также проверяет, входит ли устанавливаемый драйвер в «черный список», поддерживаемый службой Windows Update. Если драйвер присутствует в списке, установка блокируется с выдачей предупреждения для пользователя. В список вносятся драйверы, заведомо содержащие ошибки или несовместимости, а их установка предотвращается системой.

Общая схема загрузки и установки драйверов

В предыдущем разделе вы узнали, как PnP-диспетчер обнаруживает и загружает драйверы физических устройств. В основном эти драйверы загружаются «по требованию», т. е. драйвер не загружается, пока в нем не возникнет необходимость — в системе появляется устройство, за которое отвечает драйвер; и наоборот, если все устройства, находящиеся под управлением драйвера, будут удалены, драйвер выгружается.

На более общем уровне раздел программного обеспечения в реестре содержит настройки драйверов (а также служб Windows). И хотя для управления службами используется тот же раздел реестра, службы являются программами пользовательского режима, не связанными с драйверами режима ядра (хотя диспетчер служб может использоваться для загрузки как служб, так и драйверов). В этом разделе основное внимание уделяется драйверам; за информацией о службах обращайтесь к главе 9.

Загрузка драйверов

В каждом подразделе раздела программного обеспечения (HKLM\System\CurrentControlSet\Services) хранится набор параметров, управляющих некоторыми аспектами драйвера (или службы). Один из этих параметров — `ImagePath` — уже встречался нам при рассмотрении процесса загрузки драйверов PnP. На рис. 6.36 показан пример раздела драйвера, а в табл. 6.8 приведена сводка важнейших параметров в разделе программного обеспечения драйвера (за полным списком обращайтесь к главе 9 части 2).

Параметр `Start` определяет фазу, в которой происходит загрузка драйвера (или службы). В этом отношении между драйверами и службами есть два основных различия.

- Только для драйверов устройств могут задаваться параметры `Start` со значением `boot-start` (0) или `system-start` (1). Дело в том, что в этих фазах пользовательский режим еще не существует, поэтому службы загружаться не могут.
- Драйверы устройств могут использовать параметры `Group` и `Tag` (не показанные в табл. 6.8) для управления порядком загрузки в фазе начальной загрузки, но в отличие от служб, они не могут задавать параметры `DependOnGroup` или `DependOnService` (за подробностями обращайтесь к главе 9 части 2).

Таблица 6.8. Важнейшие параметры в разделе реестра драйвера

Имя параметра	Описание
`ImagePath`	Путь к файлу образа драйвера (SYS)
`Type`	Указывает, что представляет раздел — службу или драйвер. Значение 1 представляет драйвер, значение 2 представляет драйвер файловой системы (или фильтрующий драйвер). Значения 16 (0x10) и 32 (0x20) представляют службу. За подробностями обращайтесь к главе 9
`Start`	Определяет, когда должен загружаться драйвер. Возможные значения: `0 (SERVICE_BOOT_START)` — драйвер загружается начальным загрузчиком системы. `1 (SERVICE_SYSTEM_START)` — драйвер загружается после инициализации исполнительной среды. `2 (SERVICE_AUTO_START)` — драйвер загружается диспетчером служб. `3 (SERVICE_DEMAND_START)` — драйвер загружается по требованию. `4 (SERVICE_DISABLED)` — драйвер не загружается

В главе 11 части 2 описаны фазы начальной загрузки системы и объясняется, что параметр `Start` драйвера со значением 0 означает, что драйвер должен загружаться загрузчиком операционной системы. Параметр `Start`, равный 1, означает, что диспетчер ввода/вывода загружает драйвер после того, как исполнительные подсистемы завершат инициализацию. Диспетчер ввода/вывода вызывает процедуры инициализации драйвера в порядке загрузки драйверов в фазе начальной загрузки. Как и службы Windows, драйверы используют параметр `Group` из своего раздела реестра для определения группы, к которой они принадлежат; параметр реестра HKLM\SYSTEM\CurrentControlSet\Control\ServiceGroupOrder\List определяет порядок загрузки групп в фазе начальной загрузки.

Для уточнения порядка загрузки драйвер может включить параметр `Tag` для управления своим порядком в группе. Диспетчер ввода/вывода сортирует драйверы в каждой группе в соответствии со значениями параметра `Tag`, определенными в разделах реестра драйверов. Драйверы без этого параметра размещаются в конце списка своей группы. Можно считать, что диспетчер ввода/вывода инициализирует драйверы с низкими числовыми значениями `Tag` до того, как будут инициализированы драйверы с большими числовыми значениями, но это не обязательно. Раздел реестра HKLM\SYSTEM\CurrentControlSet\Control\GroupOrderList определяет приоритет тегов в группе; с этим разделом Microsoft и разработчики драйверов обладают некоторой свободой в переопределении числовых приоритетов.

ПРИМЕЧАНИЕ Параметры `Group` и `Tag` — пережиток ранних версий Windows NT. Они редко применяются на практике. В общем случае драйверы не должны зависеть от других драйверов (только от библиотек ядра, связанных с драйвером, — например, NDIS.sys).

Несколько рекомендаций по назначению параметра `Start`:

- Драйверы, поддерживающие Plug and Play, задают свой параметр `Start` в соответствии с фазой начальной загрузки, в которой они должны загружаться.
- Драйверы (с поддержкой Plug and Play и без нее), которые должны загружаться в ходе начальной загрузки системы, задают параметру `Start` значение `boot-start` (0). Примеры — драйверы системной шины и загрузочной файловой системы.
- Драйвер, который не обязателен для загрузки системы и который обнаруживает устройство, которое не может перечислить драйвер системной шины, задает параметру `Start` значение `system-start` (1). Пример — драйвер последовательного порта, который сообщает PnP-диспетчеру о присутствии стандартных последовательных портов PC, обнаруженных в ходе аппаратной диагностики и сохраненных в реестре.
- Драйвер без поддержки Plug and Play или драйвер файловой системы, наличие которого не обязательно для загрузки системы, задает параметру `Start` значение `auto-start` (2). Пример — драйвер MUP (Multiple UNC Provider), который предоставляет поддержку путей UNC (Universal Naming Convention) для удаленных ресурсов (например, \\RemoteComputerName\SomeShare).
- Драйверы Plug and Play, наличие которых не обязательно для загрузки системы, задает параметру `Start` значение `demand-start` (3). Пример — драйверы сетевых адаптеров.

Для драйверов Plug and Play и драйверов перечисляемых устройств параметр `Start` служит только одной цели — гарантировать, что загрузчик операционной системы загрузит драйвер (если драйвер необходим для успешной загрузки системы). В остальном порядок загрузки для драйверов Plug and Play определяется процессом перечисления устройств диспетчера PnP.

Установка драйвера

Как было показано ранее, драйверам Plug and Play для установки нужен INF-файл. INF-файл включает идентификаторы устройств, с которыми может работать драйвер, и инструкции по копированию файлов и записи значений в реестр. Другим типам драйверов (например, драйверам файловой системы, фильтрам файловой системы и сетевым фильтрам) также необходимы INF-файлы, включающие уникальный набор значений для этого конкретного типа драйвера.

Чисто программные драйверы (вроде того, который использует Process Explorer) тоже могут использовать INF-файл для установки, хотя это и не обязательно. Они

могут устанавливаться вызовом API-функции `CreateService` (или использовать программу-«обертку» вроде sc.exe), как делает Process Explorer после извлечения драйвера из ресурса в исполняемом файле (при выполнении с повышенными разрешениями). Как подсказывает имя, эта API-функция используется для установки как служб, так и драйверов. Аргументы `CreateService` указывают, что именно устанавливается — драйвер или служба (за подробностями обращайтесь к документации Windows SDK). После установки вызов `StartService` загружает драйвер (или службу), а для драйвера, как обычно, вызывается `DriverEntry`.

Чисто программный драйвер обычно создает объект устройства с именем, известным его клиентам. Например, Process Explorer создает устройство с именем PROCEXP152, которое затем используется программой Process Explorer в вызове `CreateFile`; за ним следуют такие вызовы, как `DeviceIoControl`, для отправки запросов драйверу (преобразуемых в IRP-пакеты диспетчером ввода/вывода). На рис. 6.43 изображена символическая ссылка объекта Process Explorer из каталога \GLOBAL?? (в программе WinObj из пакета Sysinternals), которая создается Process Explorer при первом запуске с повышенными привилегиями (напомним, что имена в этом каталоге доступны для клиентов пользовательского режима). Обратите внимание: ссылка указывает на реальный объект устройства в каталоге \Device с тем же именем (что не является строго обязательным).

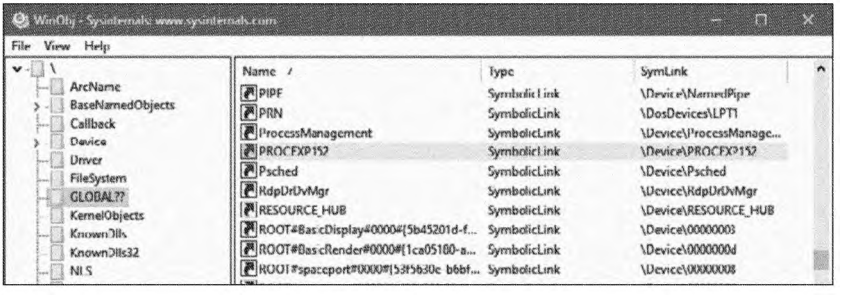


Рис. 6.43. Символическая ссылка и имя устройства Process Explorer

Windows Driver Foundation

Набор инструментальных средств Windows Driver Foundation (WDF) упрощает разработку драйверов Windows для решения таких стандартных задач, как правильная обработка Plug and Play и Power IRP. WDF включает два фреймворка: KernelMode Driver Framework (KMDF) и User-Mode Driver Framework (UMDF). WDF распространяется с открытым кодом (*https://github.com/Microsoft/Windows-Driver-Frameworks*). В табл. 6.9 приведена информация о поддержке версий Windows (для Windows 7 и выше) для KMDF, а в табл. 6.10 — та же информация для UMDF.

В Windows 10 появилась концепция универсальных драйверов, кратко представленная в главе 2. Такие драйверы используют общий набор правил DDI, реали-

Таблица 6.9. Версии KMDF

Версия KMDF	Метод выпуска	Входит в Windows	Системы, в которых работают драйверы
1.9	Windows 7 WDK	Windows 7	Windows XP и выше
1.11	Windows 8 WDK	Windows 8	Windows Vista и выше
1.13	Windows 8.1 WDK	Windows 8.1	Windows 8.1 и выше
1.15	Windows 10 WDK	Windows 10	Windows 10, Windows Server 2016
1.17	Windows 10 версия 1511 WDK	Windows 8 версия 1511	Windows 10 версия 1511 и выше, Windows Server 2016
1.19	Windows 10 версия 1607 WDK	Windows 8 версия 1607	Windows 10 версия 1607 и выше, Windows Server 2016

Таблица 6.10. Версии UMDF

Версия KMDF	Метод выпуска	Входит в Windows	Системы, в которых работают драйверы
1.9	Windows 7 WDK	Windows 7	Windows XP и выше
1.11	Windows 8 WDK	Windows 8	Windows Vista и выше
2.0	Windows 8.1 WDK	Windows 8.1	Windows 8.1 и выше
2.15	Windows 10 WDK	Windows 10	Windows 10, Windows Server 2016
2.17	Windows 10 версия 1511 WDK	Windows 8 версия 1511	Windows 10 версия 1511 и выше, Windows Server 2016
2.19	Windows 10 версия 1607 WDK	Windows 8 версия 1607	Windows 10 версия 1607 и выше, Windows Server 2016

зованных в разных выпусках Windows 10 — от IoT Core до Mobile и настольных версий. Универсальные драйверы могут строиться на базе KMDF, UMDF 2.*x* или WDM. Такие драйверы относительно легко строятся в среде Visual Studio на целевой платформе `Universal`. Компилятор помечает все правила DDI, выходящие за границы режима `Universal`.

В UMDF версий 1.*x* использовалась модель программирования драйверов на базе COM, которая сильно отличалась от программной модели KMDF, использующей C с объектами. Версия UMDF 2 была приближена к KMDF и предоставляет практически идентичный API, что способствует сокращению общих затрат на разработку драйверов WDF; более того, при необходимости драйверы UMDF 2.*x* можно преобразовать в KMDF с небольшим объемом работы. UMDF 1.*x* в этой книге не рассматривается; за дополнительной информацией обращайтесь к документации WDK.

Ниже рассматриваются фреймворки KMDF и UMDF, которые работают практически одинаково независимо от конкретной ОС.

KMDF

Среда WDF (Windows Driver Foundation) уже упоминалась в главе 2. В этом разделе будут представлены компоненты и функциональность, предоставляемые частью фреймворка KMDF, относящейся к режиму ядра. Учтите, что в этом разделе лишь кратко затронута базовая архитектура KMDF. За более подробной информацией по теме обращайтесь к документации Windows Driver Kit.

ПРИМЕЧАНИЕ Почти вся информация, представленная в этом разделе, относится и к UMDF 2.*x*; немногочисленные исключения рассматриваются в следующем разделе.

Структура и функциональность KMDF-драйвера

Сначала посмотрим, какие драйверы или устройства поддерживаются KMDF. В общем случае это любые WDM-совместимые драйверы, умеющие выполнять стандартную обработку ввода/вывода и манипулировать IRP-пакетами. Среда KMDF не подходит драйверам, которые непосредственно не пользуются API-функциями ядра Windows, вместо этого вызывая библиотеки в существующие драйверы порта и классов. Эти типы драйверов не могут использовать KMDF, так как предназначены для обратного вызова реальных WDM-драйверов, обеспечивающих обработку ввода/вывода. Также драйвер может использовать KMDF, если он предоставляет собственные функции диспетчеризации вместо применения драйверов порта или класса, IEEE 1394 и ISA, PCI, PCMCIA и SD Client (для накопителей Secure Digital).

Хотя среда KMDF предоставляет уровень абстракции поверх WDM, описанная ранее базовая структура драйвера применима и к KMDF-драйверам. По сути, KMDF-драйверы должны предоставлять следующие функции.

- **Процедура инициализации.** Подобно любому другому драйверу, у KMDF-драйвера есть инициализирующая его функция `DriverEntry`. На этом этапе KMDF-драйверы подключают инфраструктуру и выполняют действия по собственному конфигурированию и инициализации. Для драйверов, не поддерживающих механизм PnP, именно на этом этапе должен создаваться первый объект устройства.
- **Процедура добавления устройства.** В основе управления KMDF-драйвером лежат события и обратные вызовы (о них мы вкратце поговорим позже). Самым важным для PnP-устройств является функция обратного вызова `EvtDriverDeviceAdd`, так как он получает уведомления, когда PnP-диспетчер в ядре перечисляет одно из устройств драйвера.
- **Одна или несколько процедур EvtIo*.** Подобно процедурам диспетчеризации WDM-драйвера, эти процедуры обратного вызова обрабатывают определенные типы запросов на ввод/вывод из определенной очереди устройства. Как правило, драйвер создает одну или несколько очередей, в которые KMDF помещает

запросы на ввод и вывод к устройствам драйвера. Эти очереди конфигурируются по типу запроса и по типу диспетчеризации.

Простейшему KMDF-драйверу могут понадобиться только процедуры инициализации и добавления устройства, так как обобщенную функциональность для обработки большинства типов ввода/вывода предоставляет сама инфраструктура. В модели KMDF *события* относятся ко времени выполнения, когда драйвер в состоянии ответить или принять участие в происходящем. Эти события не относятся к примитивам синхронизации (синхронизация обсуждается в главе 8), это внутренние события для фреймворка.

Для событий, существенных для функционирования драйвера или требующих специализированной обработки, драйвер регистрирует процедуру обратного вызова. В остальных случаях драйвер позволяет KMDF выполнить заданные по умолчанию универсальные операции. К примеру, в случае события извлечения (`EvtDeviceEject`) драйвер может поддержать эту операцию и предоставить обратный вызов, а может предпочесть KMDF-код, предлагаемый по умолчанию, который сообщит пользователю о невозможности извлечь устройство. Не все события имеют поведение, предлагаемое по умолчанию, и обратные вызовы должны поддерживаться драйвером. Характерным примером является событие `EvtDriverDeviceAdd`, представляющее собой основу любого драйвера PnP-устройства.

ЭКСПЕРИМЕНТ: ПРОСМОТР ДРАЙВЕРОВ KMDF И UMDF 2

Расширение Wdfkd.dll, поставляемое с пакетом инструментов отладки для Windows, предоставляет множество команд для отладки и анализа драйверов и устройств в KMDF (вместо встроенного отладочного расширения в стиле WDM, в котором отсутствует аналогичная информация). Для вывода списка установленных KMDF-драйверов воспользуйтесь командой отладчика `!wdfkd.wdfldr`. Представленный здесь пример демонстрирует результат выполнения этой команды на виртуальной машине Hyper-V в 32-разрядной версии Windows 10 с установленными встроенными драйверами:

```
lkd> !wdfkd.wdfldr
-----------------------------------------------------------------
 KMDF Drivers
-----------------------------------------------------------------
 LoadedModuleList      0x870991ec
---------------------------------
LIBRARY_MODULE  0x8626aad8
  Version       v1.19
  Service       \Registry\Machine\System\CurrentControlSet\Services\Wdf01000
  ImageName     Wdf01000.sys
  ImageAddress  0x87000000
  ImageSize     0x8f000
  Associated Clients: 25

  ImageName                     Ver   WdfGlobals FxGlobals  ImageAddress ImageSize
  umpass.sys                    v1.15 0xa1ae53f8 0xa1ae52f8 0x9e5f0000   0x00008000
  peauth.sys                    v1.7  0x95e798d8 0x95e797d8 0x9e400000   0x000ba000
  mslldp.sys                    v1.15 0x9aed1b50 0x9aed1a50 0x8e300000   0x00014000
  vmgid.sys                     v1.15 0x97d0fd08 0x97d0fc08 0x8e260000   0x00008000
```

```
  monitor.sys                  v1.15 0x97cf7e18 0x97cf7d18 0x8e250000   0x0000c000
  tsusbhub.sys                 v1.15 0x97cb3108 0x97cb3008 0x8e4b0000   0x0001b000
  NdisVirtualBus.sys           v1.15 0x8d0fc2b0 0x8d0fc1b0 0x87a90000   0x00009000
  vmgencounter.sys             v1.15 0x8d0fefd0 0x8d0feed0 0x87a80000   0x00008000
  intelppm.sys                 v1.15 0x8d0f4cf0 0x8d0f4bf0 0x87a50000   0x00021000
  vms3cap.sys                  v1.15 0x8d0f5218 0x8d0f5118 0x87a40000   0x00008000
  netvsc.sys                   v1.15 0x8d11ded0 0x8d11ddd0 0x87a20000   0x00019000
  hyperkbd.sys                 v1.15 0x8d114488 0x8d114388 0x87a00000   0x00008000
  dmvsc.sys                    v1.15 0x8d0ddb28 0x8d0dda28 0x879a0000   0x0000c000
  umbus.sys                    v1.15 0x8b86ffd0 0x8b86fed0 0x874f0000   0x00011000
  CompositeBus.sys             v1.15 0x8b869910 0x8b869810 0x87df0000   0x0000d000
  cdrom.sys                    v1.15 0x8b863320 0x8b863220 0x87f40000   0x00024000
  vmstorfl.sys                 v1.15 0x8b2b9108 0x8b2b9008 0x87c70000   0x0000c000
  EhStorClass.sys              v1.15 0x8a9dacf8 0x8a9dabf8 0x878d0000   0x00015000
  vmbus.sys                    v1.15 0x8a9887c0 0x8a9886c0 0x82870000   0x00018000
  vdrvroot.sys                 v1.15 0x8a970728 0x8a970628 0x82800000   0x0000f000
  msisadrv.sys                 v1.15 0x8a964998 0x8a964898 0x873c0000   0x00008000
  WindowsTrustedRTProxy.sys    v1.15 0x8a1f4c10 0x8a1f4b10 0x87240000   0x00008000
  WindowsTrustedRT.sys         v1.15 0x8a1f1fd0 0x8a1f1ed0 0x87220000   0x00017000
  intelpep.sys                 v1.15 0x8a1ef690 0x8a1ef590 0x87210000   0x0000d000
  acpiex.sys                   v1.15 0x86287fd0 0x86287ed0 0x870a0000   0x00019000
---------------------------------
Total: 1 library loaded
```

Если бы драйверы UMDF 2.*x* были загружены, они бы также присутствовали в этом списке. Это одно из преимуществ библиотеки UMDF 2.*x* (за дополнительной информацией обращайтесь к описанию UMDF в этой главе).

Учтите, что библиотека KMDF реализована в модуле Wdf01000.sys — для текущей версии 1.*x*. Возможно, будущим версиям KMDF будет назначена основная версия 2, а реализация будет храниться в другом модуле ядра Wdf02000.sys. Этот будущий модуль может существовать параллельно с модулем версии 1.*x*, при этом каждый модуль будет загружаться с откомпилированными для него драйверами. Таким образом обеспечивается изоляция и независимость драйверов, построенных для разных основных версий библиотек KMDF.

Объектная модель KMDF

KMDF использует объектную модель со свойствами, методами и событиями, реализованную на языке C; она отчасти похожа на модель ядра, но не использует диспетчер объектов. Вместо этого KMDF осуществляет внутреннее управление своими объектами, предоставляя драйверам доступ к ним через дескрипторы, а сами структуры данных при этом остаются закрытыми. Для каждого типа объектов фреймворк предоставляет процедуры для выполнения операций (они называются *методами*); например, процедура `WdfDeviceCreate` создает устройство. Кроме того, у объектов могут присутствовать специальные поля данных, доступ к которым осуществляется через API `Get/Set` (для модификаций, которые всегда должны завершаться успешно) или `Assign/Retrieve` (для модификаций, которые могут завершиться неудачей); они называются *свойствами*. Например, функция `WdfInterruptGetInfo` возвращает информацию о данном объекте прерывания (`WDFINTERRUPT`).

В отличие от изолированных друг от друга объектов ядра, все KMDF-объекты принадлежат иерархии — большинство типов объектов связано с родителем. Корневым объектом является структура `WDFDRIVER`, описывающая реальный драйвер. По строению и предназначению она аналогична предоставляемой диспетчером ввода/вывода структуре `DRIVER_OBJECT`, а все прочие KMDF-структуры являются ее потомками. Следующим по важности является объект `WDFDEVICE`, относящийся к экземпляру распознанного в системе устройства и создаваемый функцией `WdfDeviceCreate`. Еще раз напомню, что это аналог структуры `DEVICE_OBJECT`, использующейся в модели WDM и в диспетчере ввода/вывода. Поддерживаемые в KMDF типы объектов перечислены в табл. 6.11.

Таблица 6.11. Типы KMDF-объектов

Объект	Тип	Описание
Список потомков	`WDFCHILDLIST`	Список связанных с устройством потомков объекта `WDFDEVICE`. Используется только драйверами шины
Коллекция	`WDFCOLLECTION`	Список объектов одного типа, например отфильтрованная группа объектов `WDFDEVICE`
Отложенный вызов процедуры	`WDFDPC`	Экземпляр DPC-объекта
Устройство	`WDFDEVICE`	Экземпляр устройства
Общий DMA-буфер	`WDFCOMMONBUFFER`	Область памяти, к которой могут обратиться устройство и драйвер при прямом доступе к памяти
Средство включения DMA	`WDFDMAENABLER`	Включает прямой доступ к памяти для драйвера на указанном канале
DMA-транзакция	`WDFDMATRANSACTION`	Экземпляр DMA-транзакции
Драйвер	`WDFDRIVER`	Корневой объект для драйвера; представляет драйвер с его параметрами, обратными вызовами и другими элементами
Файл	`WDFFILEOBJECT`	Экземпляр файлового объекта, который может использоваться в качестве канала между приложением и драйвером
Обобщенный объект	`WDFOBJECT`	Позволяет поместить определенные драйвером нестандартные данные в объектную модель данных инфраструктуры в виде объекта
Прерывание	`WDFINTERRUPT`	Экземпляр прерывания, который должен обработать драйвер
Очередь ввода/вывода	`WDFQUEUE`	Представляет рассматриваемую очередь ввода/вывода
Запрос ввода/вывода	`WDFREQUEST`	Представляет текущий запрос к `WDFQUEUE`

Объект	Тип	Описание
Цель ввода/вывода	WDFIOTARGET	Представляет стек устройств, в который направлен текущий объект WDFREQUEST
Ассоциативные списки	WDFLOOKASIDE	Описывает исполнительный ассоциативный список
Память	WDFMEMORY	Описывает область выгружаемого или невыгружаемого пула
Раздел реестра	WDFKEY	Описывает раздел реестра
Список ресурсов	WDFCMRESLIST	Определяет аппаратные ресурсы, назначенные объекту WDFDEVICE
Список диапазона ресурсов	WDFIORESLIST	Определяет текущий возможный диапазон аппаратных ресурсов объекта WDFDEVICE
Список требований к ресурсам	WDFIORESREQLIST	Содержит массив объектов WDFIORESLIST, описывающий все возможные диапазоны ресурсов для объекта WDFDEVICE
Спин-блокировка	WDFSPINLOCK	Описывает спин-блокировку
Строка	WDFSTRING	Описывает структуру Unicode-строки
Таймер	WDFTIMER	Описывает таймер исполнительной системы (см. главу 8 части 2)
USB-устройство	WDFUSBDEVICE	Определяет один экземпляр USB-устройства
USB-интерфейс	WDFUSBINTERFACE	Определяет один интерфейс данного объекта WDFUSBDEVICE
USB-канал	WDFUSBPIPE	Определяет канал к конечной точке данного объекта WDFUSBINTERFACE
Блокировка ожидания	WDFWAITLOCK	Представляет объект события диспетчера ядра
WMI-экземпляр	WDFWMIINSTANCE	Представляет блок данных для объекта WDFWMIPROVIDER
WMI-провайдер	WDFWMIPROVIDER	Описывает схему WMI для всех поддерживаемых драйвером объектов WDFWMIINSTANCE
Рабочий элемент	WDFWORKITEM	Описывает исполнительный рабочий элемент

К каждому из этих объектов можно присоединить другие KMDF-объекты в качестве потомка. Некоторые объекты могут иметь только одного или двух предков, в то время как другие присоединяются в качестве потомков к любым объектам. Например, объект **WDFINTERRUPT** должен быть связан с объектом **WDFDEVICE**, в то время как объекты **WDFSPINLOCK** и **WDFSTRING** могут иметь в качестве предка любой объект, что обеспечивает точный контроль над их применимостью и уменьшает количество глобальных переменных состояния. На рис. 6.44 представлена иерархия KMDF-объектов.

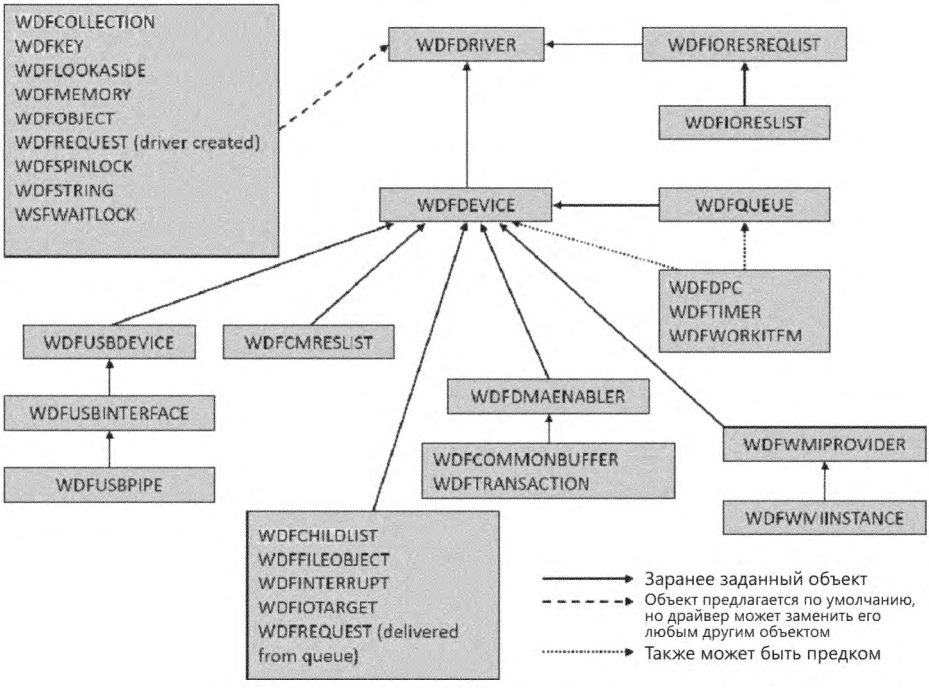


Рис. 6.44. Иерархия KMDF-объектов

Следует заметить, что упомянутые ранее и показанные на рис. 6.44 связи не обязательно должны быть прямыми. Родитель может располагаться где-то в *иерархической цепочке* (hierarchy chain). В результате иерархия влияет не только на местоположение объекта, но и на его время жизни. При создании дочернего объекта счетчик ссылок его предка увеличивается на единицу. Соответственно, уничтожение родительского объекта приводит к исчезновению всех его потомков. Именно поэтому связывание с текущим объектом такого объекта, как `WDFSTRING` или `WDFMEMORY`, вместо заданного по умолчанию объекта `WDFDRIVER`, позволяет при удалении предка автоматически освободить память и информацию о состоянии.

С концепцией иерархии в KMDF тесно связано понятие контекста объекта (object context). Как уже упоминалось, KMDF-объекты непрозрачны, а их местоположение определяется местом в иерархической цепочке, поэтому важно дать драйверам возможность присоединять к объектам собственные данные для отслеживания информации, которая не предоставляется фреймворком. Контекст позволяет всем KMDF-объектам содержать подобную информацию, кроме того, возможны контекстные области набора объектов, позволяющие разным слоям кода внутри одного драйвера взаимодействовать с объектом разными способами. В модели WDM структура данных расширения устройства позволяет связать такую информацию с конкретным устройством, но в KMDF контекстные области могут присутствовать даже у спин-блокировки или строки. Подобная расширяемость

позволяет отвечающим за обработку ввода/вывода библиотекам или уровням кода независимо взаимодействовать с другим кодом, основываясь на контекстной области, с которой они работают.

Наконец, с KMDF-объектами связывается набор атрибутов (табл. 6.12). Обычно им оставляют значения, предлагаемые по умолчанию, но драйвер может переопределить их в момент создания объекта. Для этого нужно задать структуру `WDF_OBJECT_ATTRIBUTES` (аналогичную структуре `OBJECT_ATTRIBUTES` диспетчера объектов, используемой при создании объектов ядра).

Таблица 6.12. Атрибуты объектов KMDF

Атрибут	Описание
`ContextSizeOverride`	Размер контекстной области объекта
`ContextTypeInfo`	Тип контекстной области объекта
`EvtCleanupCallback`	Обратный вызов, уведомляющий драйвер об очистке объекта перед его удалением (могут существовать ссылки на объект)
`EvtDestroyCallback`	Обратный вызов, уведомляющий драйвер о неминуемом удалении объекта (счетчик ссылок равен 0)
`ExecutionLevel`	Описывает максимальный IRQL-уровень, на котором KMDF может активировать обратные вызовы
`ParentObject`	Определяет предка объекта
`SynchronizationScope`	Указывает, следует ли синхронизировать обратный вызов с родителем, очередью или устройством или синхронизация не требуется

Модель ввода/вывода в KMDF

Модель ввода/вывода в KMDF использует уже знакомые нам механизмы WDM (Windows Driver Model). По сути, саму инфраструктуру можно представить как WDM-драйвер, так как для ее абстрагирования и обеспечения функциональности применяются прикладные программные интерфейсы ядра и варианты поведения, характерные для WDM. В KMDF драйвер инфраструктуры определяет собственные процедуры IRP-диспетчеризации в стиле WDM и контролирует все отправленные драйверу IRP-пакеты. После применения одного из трех KMDF-обработчиков ввода/вывода (их мы вкратце опишем далее) запросы упаковываются в соответствующие KMDF-объекты, при необходимости вставляются в соответствующие очереди, и если в данных событиях заинтересован драйвер, происходит обратный вызов драйвера. Процесс ввода/вывода в среде KMDF показан на рис. 6.45.

На основе механизма обработки IRP-пакетов для WDM-драйверов, о котором рассказывалось ранее, KMDF выполняет одно из следующих действий.

- Посылает IRP-пакет обработчику ввода/вывода, выполняющему стандартные операции с устройствами.

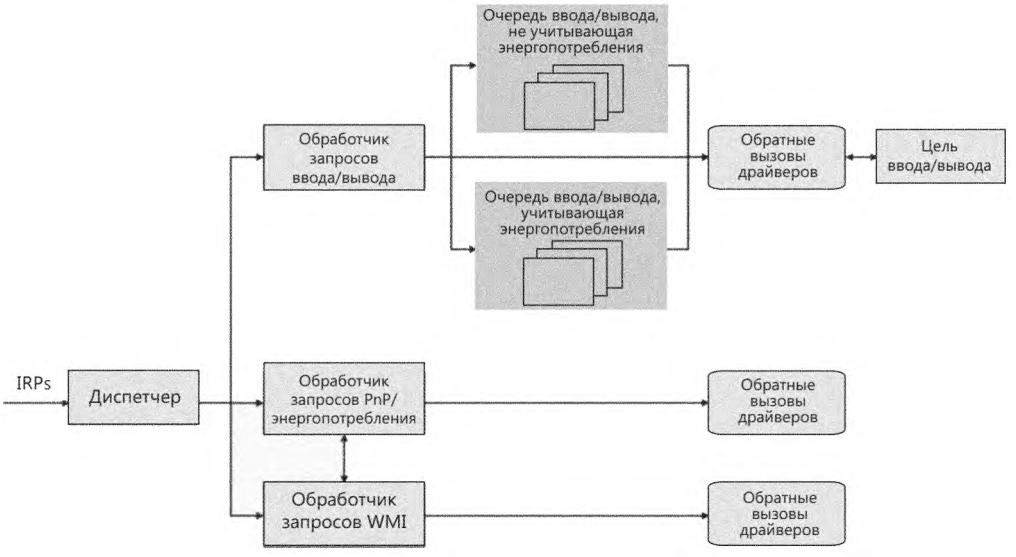


Рис. 6.45. Процесс ввода/вывода и обработки IRP-пакетов в KMDF

- Посылает IRP-пакет обработчику PnP-устройств и электропитания, который обслуживает эти события и уведомляет другие драйверы в случае изменения состояния.
- Посылает IRP-пакет WMI-обработчику, отвечающему за отслеживание и протоколирование.

Затем эти компоненты уведомляют драйвер о любых событиях, для которых они зарегистрировались, потенциально они могут передать запрос другому обработчику для дальнейших операций, а затем запрос завершается при помощи внутренних операций обработчика или в результате вызова драйвера. Если среда KMDF завершила обработку IRP-пакета, но запрос при этом до конца не выполнен, предпринимается одно из следующих действий:

- для драйверов шин и функциональных драйверов IRP-пакет завершается с кодом `STATUS_INVALID_DEVICE_REQUEST`;
- для фильтрующих драйверов запрос передается нижележащему драйверу.

Обработка ввода/вывода в KMDF основана на механизме очередей (объект `WDFQUEUE`, а не `KQUEUE`, который обсуждался ранее в этой главе). Очереди в KMDF представляют собой масштабируемые контейнеры запросов на ввод и вывод (упакованные как объекты `WDFREQUEST`). Они предоставляют богатый набор функций, выходящий за пределы сортировки ожидающих запросов на ввод и вывод для данного устройства. К примеру, очереди отслеживают активные в данный момент запросы и поддерживают отмену ввода/вывода, параллельный ввод/вывод (возможность выполнять и завершать более одного запроса на ввод/вывод одновре-

менно) и синхронизацию ввода/вывода (как отмечено в списке атрибутов объектов в табл. 6.12). Типичный KMDF-драйвер создает по меньшей мере одну очередь и связывает с каждой из созданных им очередей одно или несколько событий и ряд возможностей из следующего списка.

- Обратные вызовы, зарегистрированные со связанными с данной очередью событиями.
- Состояние управления электропитанием для очереди. В KMDF поддерживаются очереди, как учитывающие, так и не учитывающие энергопотребление. Для первой из них обработчик ввода/вывода пробуждает устройство, когда это нужно (и когда это возможно), включает таймер простоя, если у устройства отсутствует очередь запросов ввода/вывода, и вызывает процедуры отмены ввода/вывода при выходе системы из рабочего состояния.
- Метод диспетчеризации для очереди. Доставка запросов на ввод и вывод в KMDF осуществляется в последовательном, параллельном или ручном режимах. Последовательные запросы доставляются по одному (KMDF ждет завершения драйвером предыдущего запроса), в то время как в параллельном режиме доставка запросов драйверу происходит сразу же. В ручном режиме драйвер сам извлекает из очереди запросы на ввод и вывод.
- Возможность очереди принимать буферы нулевой длины, например входящие запросы, не содержащие данных.

ПРИМЕЧАНИЕ Метод диспетчеризации влияет только на количество одновременных активных запросов в очереди драйвера. Он не указывает, последовательно или параллельно будут осуществляться обратные вызовы событий. Это поведение определяется через описанный ранее атрибут объекта области синхронизации. Соответственно, даже при отключенном режиме параллельности в параллельной очереди могут присутствовать несколько входящих запросов.

Взяв за основу механизм очередей, обработчик ввода/вывода в KMDF при получении запроса на создание, закрытие, очистку, запись, чтение или управление устройством (IOCTL) может выполнять несколько операций.

- В случае запроса на создание драйвер может запросить немедленное уведомление через функцию `EvtDeviceFileCreate` или же он может для получения этих запросов сформировать очередь в отличном от ручного режиме. Если ни один из этих методов не используется, KMDF просто отправляет код успешного завершения, указывая, что по умолчанию приложения смогут открывать дескрипторы KMDF-драйверов, не поддерживающих их собственный код.
- В случае запросов на очистку и закрытие драйвер немедленно уведомляется через обратные вызовы функций `EvtFileCleanup` и `EvtFileClose`, если таковые зарегистрированы. В противном случае инфраструктура просто отправляет код успешного завершения.
- Наконец, рис. 6.46 иллюстрирует ход выполнения запроса ввода/вывода для самых распространенных операций (чтение, запись и управление вводом/выводом).

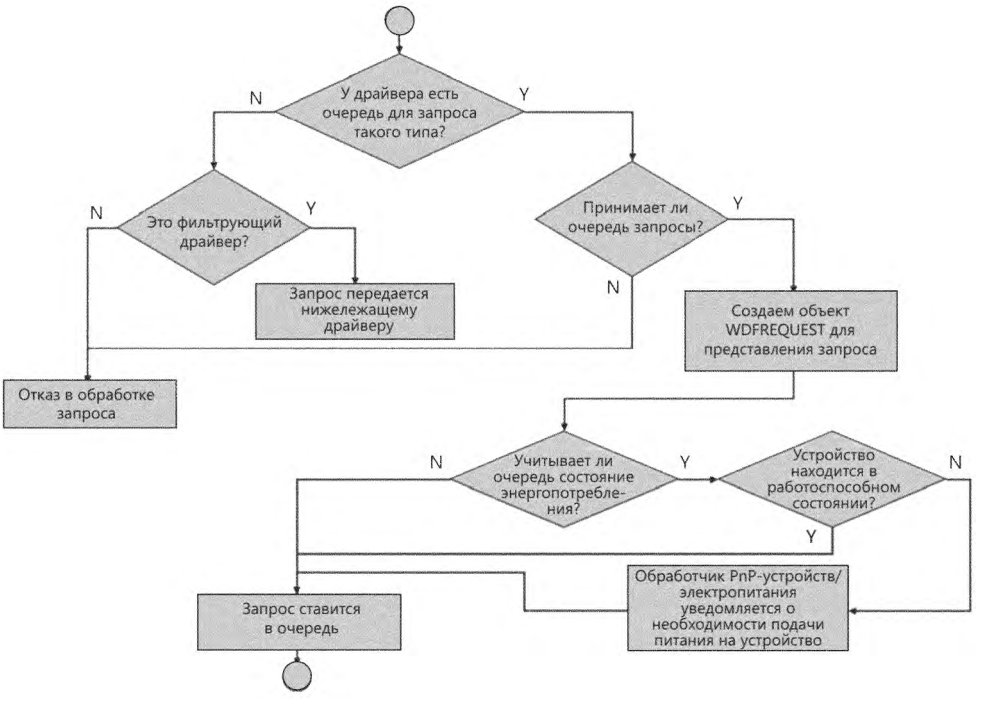


Рис. 6.46. Обработка в KMDF запросов ввода/вывода для чтения, записи и IOCTL

UMDF

В Windows появляется все больше драйверов, работающих в пользовательском режиме; эти драйверы используют фреймворк UMDF (User-Mode Driver Framework), являющийся частью WDF. Фреймворк UMDF версии 2 согласуется с KMDF в отношении объектной модели, модели программирования и модели ввода/вывода. Тем не менее эти фреймворки не идентичны из-за некоторых внутренних различий между пользовательским режимом и режимом ядра. Например, некоторые объекты KMDF из табл. 6.12 в UMDF не существуют — `WDFCHILDLIST`, DMA-объекты, `WDFLOOKASIDELIST` (эти списки могут создаваться только в режиме ядра), `WDFIORESLIST`, `WDFIORESREQLIST`, `WDFDPC` и объекты WMI. Тем не менее многие основные объекты и концепции KMDF в равной степени применимы к UMDF 2.x.

UMDF обладает рядом преимуществ перед KMDF.

- Драйверы UMDF выполняются в пользовательском режиме, и все необработанные исключения приводят к сбою хост-процесса UMDF, но не к сбою всей системы.
- Хост-процесс UMDF выполняется с правами учетной записи локальной службы, которая обладает чрезвычайно ограниченными привилегиями на локальной

машине, и получает только анонимный доступ к сетевым подключениям. Таким образом сокращается поверхность атаки системы безопасности.

- Выполнение в пользовательском режиме означает, что IRQL всегда находится на уровне 0 (`PASSIVE_LEVEL`). Таким образом, драйвер всегда может инициировать ошибки страниц и использовать объекты диспетчера для синхронизации (события, мьютексы и т. д.).
- Драйверы UMDF проще в отладке, чем драйверы KMDF, потому что отладочная конфигурация не требует двух разных машин (виртуальных или физических).

Главный недостаток UMDF — повышение задержки из-за необходимости согласования переходов между режимом ядра и пользовательским режимом (см. далее). Кроме того, некоторые типы драйверов (например, драйверы высокопроизводительных PCI-устройств) попросту не предназначены для выполнения в пользовательском режиме, и поэтому написать их на базе UMDF не удастся.

Среда UMDF специально разработана для поддержки так называемых классов устройств протокола (protocol device classes). Этот термин относится к устройствам, использующим один и тот же стандартизированный, универсальный протокол и предлагающим на его базе специальную функциональность. К таким протоколам в настоящее время относятся IEEE 1394 (FireWire), USB, Bluetooth и TCP/IP. Любые устройства, работающие на этих шинах (или подсоединенные к сети), являются потенциальными кандидатами для UMDF. В качестве примера можно привести переносные музыкальные проигрыватели, PDA, сотовые телефоны, обычные камеры и веб-камеры и т. п. Еще двумя UMDF-потребителями являются SideShow-совместимые устройства (дополнительные мониторы) и среда Windows Portable Device (WPD), поддерживающая сменные USB-накопители. Наконец, как и в среде KMDF, в UMDF можно реализовать чисто программные драйверы (например, для виртуального устройства).

В отличие от KMDF-драйверов, работающих как драйверные объекты, представляющие файл образа (с расширением .sys), UMDF-драйверы работают в хост-процессе драйвера (%SystemRoot%\System32\WUDFHost.exe), аналогичном хост-процессу служб. Хост-процесс включает в себя сам драйвер, инфраструктуру драйвера для пользовательского режима (реализованную как DLL-библиотека) и среду выполнения (отвечающую за диспетчеризацию ввода/вывода, загрузку драйверов, управление стеком устройств, коммуникацию с ядром и пул потоков).

Как и в ядре, каждый UMDF-драйвер работает как часть стека, в котором может находиться целый набор управляющих устройством драйверов. Так как код режима пользователя не имеет доступа к адресному пространству ядра, в UMDF входит несколько компонентов для обеспечения этого доступа через специализированный интерфейс. Этот доступ реализуется через фрагмент UMDF в режиме ядра, использующий усовершенствованный локальный вызов процедур (Advanced Local Procedure Call, ALPC), который взаимодействует со средой исполнения в хост-процессах драйвера режима пользователя (ALPC описывается в главе 8 части 2). Архитектура модели UMDF-драйвера показана на рис. 6.47.

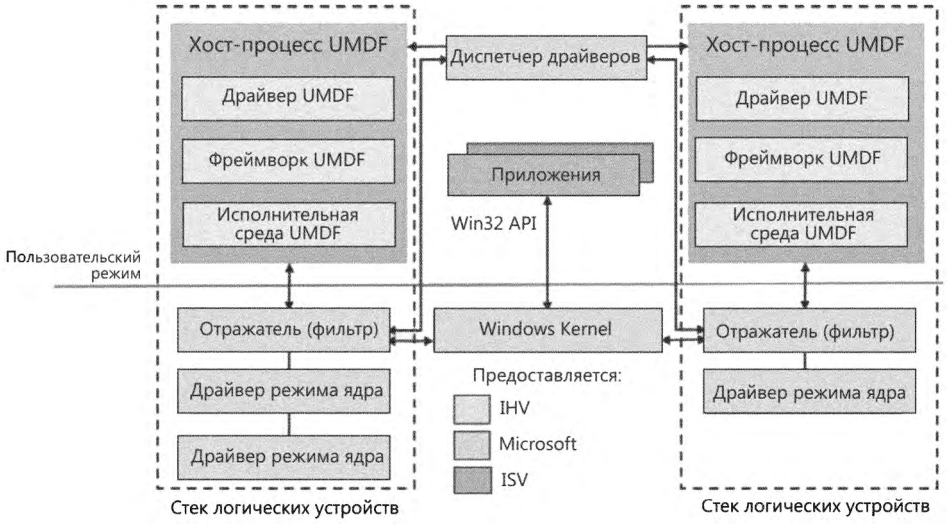


Рис. 6.47. UMDF-архитектура

На рис. 6.47 показаны два стека устройств, управляющих двумя разными устройствами, каждый с UMDF-драйвером, работающим внутри собственного хост-процесса. Диаграмма демонстрирует, что в архитектуре принимают участие следующие компоненты:

- **Приложения** являются клиентами драйверов. Это стандартные Windows-приложения, использующие для реализации ввода/вывода те же самые API-интерфейсы, что и KMDF- или WDM-устройства. Приложения не подозревают, что общение с устройствами происходит на базе UMDF и вызовы передаются диспетчеру ввода/вывода режима ядра.
- **Ядро Windows** (диспетчер ввода/вывода) формирует IRP-пакеты для операций, как и любое стандартное устройство.
- **Отражателем** (reflector) называется стандартный фильтрующий WDM-драйвер, расположенный на вершине стека каждого устройства, управляемого UMDF-драйвером (`%SystemRoot%\System32\Drivers\WUDFRd.Sys`). Он отвечает за взаимодействие ядра с хост-процессом драйвера пользовательского режима. Относящиеся к управлению электропитанием, PnP-устройствами и стандартному вводу/выводу IRP-пакеты через ALPC направляются в хост-процесс. Это позволяет UMDF-драйверу отвечать на запросы ввода/вывода и выполнять работу, а также поддерживать модель PnP, обеспечивая перечисление и установку PnP-устройств, а также управление PnP-устройствами. Кроме того, отражатель следит за хост-процессами драйвера, гарантируя адекватную скорость отклика на запросы и предотвращая зависание драйверов и приложений.
- **Диспетчер драйверов** отвечает за запуск и остановку хост-процессов драйвера в зависимости от имеющихся UMDF-устройств, а также за управление содержа-

щейся на них информацией. Также он отвечает за ответ на сообщения отражателя и применение их к соответствующему хост-процессу (примером может служить реакция на установку устройства). Диспетчер драйверов функционирует как стандартная Windows-служба, реализованная в библиотеке %SystemRoot%\System32\WUDFsvc.dll (со стандартным хостом Svchost.exe), и настраивается на автоматический запуск после установки первого UMDF-драйвера для устройства. Для всех хост-процессов драйвера запускается только один экземпляр диспетчера драйверов, который и обеспечивает функционирование UMDF-драйверов.

- **Хост-процесс** предоставляет пространство адресов и среду выполнения для реального драйвера (WUDFHost.exe). Он запускается в локальной учетной записи службы, не относясь при этом к Windows-службам и не будучи управляемым диспетчером управления службами (SCM). Им управляет диспетчер драйверов. Также хост-процесс отвечает за предоставление стека устройств пользовательского режима для фактического оборудования, видимого всем приложениям в системе. В текущем выпуске UMDF каждый экземпляр устройства имеет собственный стек устройств, запущенный в отдельном хост-процессе. В будущем набор экземпляров сможет совместно пользоваться одним хост-процессом. Хост-процессы являются дочерними процессами диспетчера драйверов.
- **Драйверы режима ядра.** Если для устройства, управляемого UMDF-драйвером, требуется определенная поддержка ядра, можно написать дополнительный драйвер режима ядра для решения этой задачи. В этом случае устройство будет управляться как UMDF-, так и KMDF-драйверами (или WDM-драйверами).

Чтобы посмотреть, как работает UMDF, вставьте в USB-разъем флэш-накопитель с каким-либо содержимым. Запустите приложение Process Explorer, и вы увидите

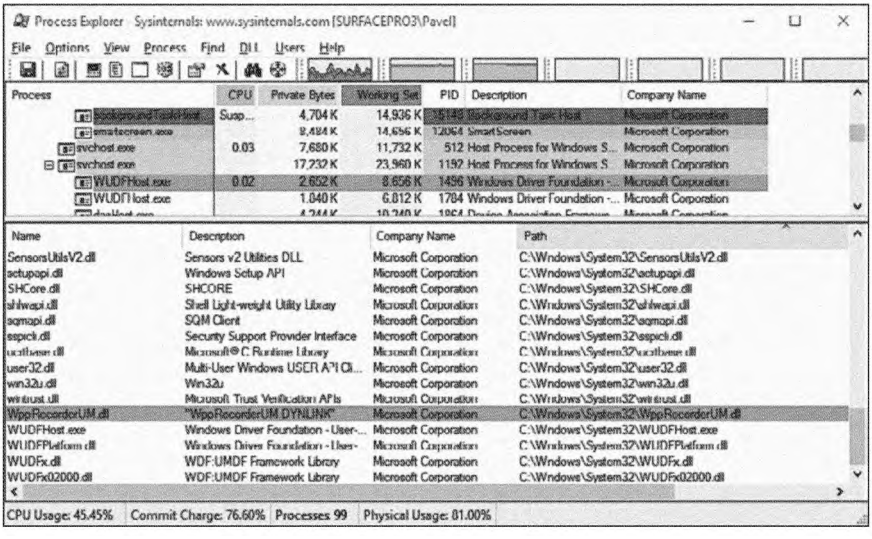


Рис. 6.48. DLL в хост-процессе среды UMDF

процесс WUDFHost.exe, соответствующий хост-процессу драйвера. Переключитесь на вывод DLL-библиотек и воспользуйтесь прокруткой, чтобы найти библиотеки, аналогичные показанным на рис. 6.48.

Можно выделить три основных компонента, соответствующих описанной архитектуре:

- **WUDFHost.exe** – исполняемый файл хоста UMDF;
- **WUDFx02000.dll** – DLL-библиотека фреймворка UMDF 2.*x*;
- **WUDFPlatform.dll** – исполнительная среда.

Диспетчер электропитания

Как вы уже знаете, функциональность Plug and Play в Windows требует поддержки со стороны системного оборудования. Точно так же и управлению электропитанием требуется аппаратная поддержка, отвечающая спецификации ACPI (Advanced Configuration and Power Interface), которая сейчас является частью UEFI (Unified Extensible Firmware Interface. (Со спецификацией ACPI можно ознакомиться на странице *http://www.uefi.org/specifications*.)

Этот стандарт определяет различные уровни энергопотребления для системы и устройств. В табл. 6.13 перечислены шесть состояний, от S0 (полностью активное, или рабочее, состояние) до S5 (полное отключение). Каждое из них характеризуется следующими параметрами.

- **Энергопотребление** (power consumption). Объем энергии, потребляемой компьютером.
- **Возобновление работы программного обеспечения** (software resumption). Состояние программного обеспечения при переходе компьютера в «более активное» состояние.
- **Аппаратная задержка** (hardware latency). Время, необходимое для возвращения компьютера в полностью активное состояние.

Таблица 6.13. Состояния энергопотребления системы

Состояние	Энергопотребление	Возобновление работы ПО	Аппаратная задержка
S0 (полностью активное состояние)	Максимальное	–	Отсутствует
S1 (засыпание)	Меньше, чем S0, но больше, чем S2	Система возобновляет работу с прерванной точки (возвращается в состояние S0)	Менее 2 секунд

Состояние	Энергопотребление	Возобновление работы ПО	Аппаратная задержка
S2 (засыпание)	Меньше, чем S1, но больше, чем S3	Система возобновляет работу с прерванной точки (возвращается в состояние S0)	2 и более секунды
S3 (засыпание)	Меньше, чем S2; процессор отключен	Система возобновляет работу с прерванной точки (возвращается в состояние S0)	Аналогично состоянию S2
S4 (гибернация)	Ток подается только на кнопку включения электропитания, происходит активизация электроники	Система перезапускается из сохраненного файла гибернации и возобновляет работу с прерванной точки (возвращается в состояние S0)	Длительная и неопределенная
S5 (полное отключение)	Ток подается только на кнопку включения электропитания	Система загружается «с нуля»	Длительная и неопределенная

В состояниях ждущего режима (S1–S4) компьютер кажется отключенным, так как его энергопотребление снижено. Но при этом он сохраняет в памяти и на диске всю информацию, необходимую для возвращения в состояние S0. В состояниях S1–S3 для сохранения содержимого памяти требуется столько энергии, чтобы при переходе в состояние S0 (пробуждение компьютера пользователем или устройством) работа системы возобновилась с прерванной точки.

При переходе системы в состояние S4 диспетчер электропитания сохраняет сжатое содержимое памяти в файле Hiberfil.sys, который называется файлом гибернации. Он достаточно велик, чтобы вместить несжатое содержимое памяти, и располагается в скрытом файле в корневом каталоге системного тома. (Сжатие позволяет минимизировать дисковый ввод/вывод, а также ускорить переход в сон и выход из сна.) Сохранив содержимое памяти, диспетчер электропитания отключает компьютер. При последующем включении происходит обычный процесс загрузки, отличающийся только тем, что диспетчер загрузки Bootmgr проверяет сохраненный в файле гибернации действительный образ памяти. Если в этом файле присутствуют данные о состоянии системы, Bootmgr запускает приложение %SystemRoot%\System32\Winresume.exe, которое считывает содержимое файла в память и возобновляет работу системы с точки, зафиксированной в файле гибернации.

В системах с гибридным спящим режимом (по умолчанию это только настольные ПК) запрос пользователя на переход в спящий режим фактически представляет собой комбинацию состояний S3 и S4: пока компьютер переходит в спящий режим, на диск записывается аварийный файл гибернации. В отличие от обычного файла гибернации, содержащего данные практически для всей активной памяти, в аварийный файл входят только данные, которые нельзя выгрузить позднее. Это ускоряет приостановку работы системы (так как на диск нужно записывать меньше данных). После этого драйверы уведомляются о переходе в состояние S4, что позволяет им выбрать нужную конфигурацию и сохранить состояние системы как при обычной

гибернации. Далее система переходит в спящий режим, как после обычного перехода. Однако при включении электропитания она, по сути, оказывается в состоянии S4 — пользователь может включить компьютер, и Windows восстановится из аварийного файла гибернации.

ПРИМЕЧАНИЕ Гибернацию можно полностью отключить и сэкономить часть дискового пространства. Для этого следует выполнить команду `powercfg /h` из командной строки с повышенными привилегиями.

Компьютер никогда не совершает прямых переходов из состояния S1 в S4. Сначала ему нужно перейти в состояние S0. Как показано на рис. 6.49, переход системы из состояний S1–S5 в S0 называется *пробуждением* (waking), а переход из состояния S0 в любое из состояний S1–S5 — *засыпанием* (sleeping).

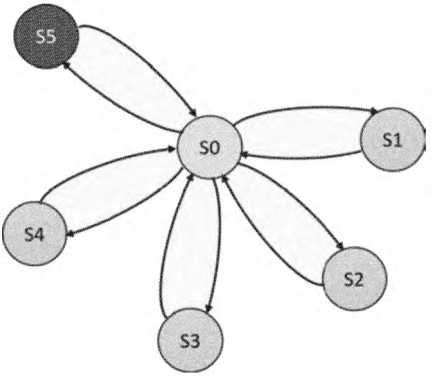


Рис. 6.49. Переходы системы между различными состояниями энергопотребления

ЭКСПЕРИМЕНТ: СИСТЕМНЫЕ СОСТОЯНИЯ ЭНЕРГОПОТРЕБЛЕНИЯ

Чтобы просмотреть информацию о поддерживаемых состояниях энергопотребления, откройте окно командной строки с повышенными привилегиями и введите команду `powercfg /a`. Результат должен выглядеть примерно так:

```
C:\WINDOWS\system32>powercfg /a
The following sleep states are available on this system:
    Standby (S3)
    Hibernate
    Fast Startup

The following sleep states are not available on this system:
    Standby (S1)
        The system firmware does not support this standby state.

    Standby (S2)
        The system firmware does not support this standby state.
```

```
    Standby (S0 Low Power Idle)
        The system firmware does not support this standby state.

    Hybrid Sleep
        The hypervisor does not support this standby state.
```

Обратите внимание: в ждущем состоянии S3 доступна гибернация. Давайте отключим гибернацию и выполним команду повторно:

```
C:\WINDOWS\system32>powercfg /h off

C:\WINDOWS\system32>powercfg /a

The following sleep states are available on this system:
    Standby (S3)

The following sleep states are not available on this system:
    Standby (S1)
        The system firmware does not support this standby state.

    Standby (S2)
        The system firmware does not support this standby state.

    Hibernate
        Hibernation has not been enabled.

    Standby (S0 Low Power Idle)
        The system firmware does not support this standby state.

    Hybrid Sleep
        Hibernation is not available.
        The hypervisor does not support this standby state.

    Fast Startup
        Hibernation is not available.
```

Хотя система может пребывать в одном из шести состояний энергопотребления, для устройств стандарт ACPI определяет четыре состояния: от D0 до D3. В состоянии D0 устройство полностью включено, состояние D3 соответствует полному отключению. Стандарт ACPI позволяет драйверам и устройствам самостоятельно определять состояния D1 и D2, но при этом в состоянии D1 должно потребляться столько же или меньше энергии, чем в состоянии D0, а в состоянии D2 устройство должно потреблять столько же или меньше энергии, чем в состоянии D1.

В Windows 8 (и выше) состояние D3 разбито на два подсостояния, «D3-горячее» и «D3-холодное». В D3-горячем состоянии устройство в целом отключено, но оно не отсоединено от главного источника питания, а контроллер его родительской шины может обнаружить присутствие устройства на шине. В D3-холодном состоянии устройство отсоединено от главного источника питания, а контроллер шины не может обнаружить устройство. Это состояние предоставляет еще одну возможность для экономии питания. На рис. 6.50 представлены состояния устройства и возможные переходы.

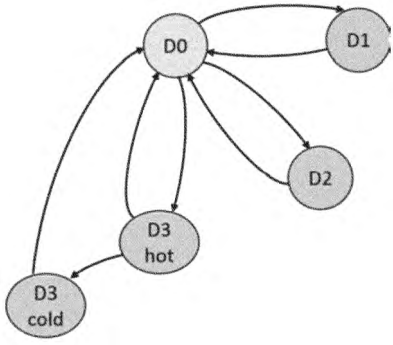


Рис. 6.50. Переходы между состояниями энергопотребления устройств

До выхода Windows 8 устройства могли перейти в D3-горячее состояние только при полной активности системы (S0). Переход в D3-холодное состояние происходил неявно при переходе системы в спящее состояние. Начиная с Windows 8, появилась возможность выбрать для устройства D3-холодное состояние при полностью активном состоянии. Драйвер, управляющий устройством, не может перевести устройство в D3-холодное состояние напрямую; вместо этого он переводит устройство в D3-горячее состояние, после чего (в зависимости от других устройств на той же шине, переходящих в D3-горячее состояние) драйвер шин и встроенное ПО могут решить перевести все устройства в D3-холодное состояние. Решение о переводе устройств в D3-холодное состояние зависит от двух факторов: (1) фактических возможностей драйвера шины и встроенного ПО и (2) драйвера, который должен инициировать переход в D3-холодное состояние — либо в установочном INF-файле, либо динамическим вызовом функции `SetD3DColdSupport`.

Компания Microsoft в сочетании с крупными OEM-производителями оборудования, определила серию эталонных спецификаций управления питанием. В них определяются состояния питания устройств, необходимые для устройств конкретного класса (для основных классов устройств: экран, сеть, SCSI и т. д.). У некоторых устройств не существует промежуточных состояний между полным включением и полным отключением, вследствие чего эти состояния остаются неопределенными.

Режим ожидания с подключением и текущий режим ожидания

Возможно, вы заметили в приведенном выше эксперименте еще одно системное состояние, которое называется «режимом ожидания» — Standby (`S0 Low Power Idle`). Хотя это состояние и не относится к числу официальных состояний ACPI, оно является модификацией S0, известной под названием *режима ожидания с подключением* (Connected Standby) в Windows 8.*x* и выше. В Windows 10 (настольные и мобильные версии) это состояние было доработано и получило названием *текущего режима ожидания* (Modern Standby). «Нормальное» состояние ожидания (S3 выше) иногда называется *традиционным* (Legacy Standby).

Основная проблема с традиционным состоянием ожидания заключается в том, что система не работает, а следовательно, например, если пользователь получит сообщение электронной почты, система не сможет принять его без пробуждения в состояние S0. Возможно это или нет — зависит от конфигурации и возможностей устройства. Даже если система пробудится для получения сообщения, она не вернется в спящее состояние немедленно. Текущий режим ожидания решает обе проблемы.

Системы с поддержкой текущего режима ожидания обычно переходят в это состояние при получении приказа о переходе в режим ожидания. Формально система все еще находится в состоянии S0 — процессор активен, а код может выполняться. Тем не менее настольные процессы (не являющиеся UWP-приложениями) приостанавливаются, как и UWP-приложения (большинство из них не выполняется на переднем плане и все равно приостанавливается), но фоновые задачи, созданные UWP-приложениями, продолжают выполняться. Например, у почтового клиента может быть фоновая задача, которая периодически проверяет наличие новых сообщений.

Нахождение в текущем режиме ожидания также означает, что система сможет очень быстро активизироваться в полноценном состоянии S0 (иногда это называется «быстрым включением»). Следует заметить, что не все системы поддерживают текущий режим ожидания; все зависит от чипсета и других компонентов платформы (например, система, на которой проводился последний эксперимент, не поддерживает текущий режим ожидания, а следовательно, поддерживает традиционный режим ожидания).

За дополнительной информацией о текущем режиме ожидания обращайтесь к документации Windows Hardware по адресу *https://msdn.microsoft.com/en-us/library/windows/hardware/mt282515(v=vs.85).aspx*.

Работа диспетчера электропитания

Политика управления электропитанием в Windows определяется диспетчером электропитания и отдельными драйверами устройств. Владельцем системной политики управления электропитанием является диспетчер электропитания. Это означает, что он принимает решение о состоянии энергопотребления системы на конкретный момент. При необходимости выключения или перехода к засыпанию (ждущий режим) или ко сну (спящий режим) диспетчер указывает устройствам, поддерживающим управление электропитанием, что они должны перейти в соответствующее состояние.

Решения о переходе в другое состояние энергопотребления принимаются исходя из следующих факторов:

- уровень активности системы;
- уровень заряда аккумулятора;
- наличие запросов приложений на выключение или переход в ждущий или спящий режим;

- действия пользователя, например нажатие кнопки включения электропитания;
- настройки электропитания в Панели управления.

Часть получаемой PnP-диспетчером при перечислении устройств информации связана с поддержкой устройствами системы управления электропитанием. Драйвер сообщает, поддерживает ли устройство состояния D1 и D2, а также то, какие задержки сопровождают переход из состояний D3–D0 в состояние D1 (последняя часть данных является необязательной). Чтобы диспетчеру электропитания было проще определить время перехода в другое состояние энергопотребления, драйверы шин возвращают таблицу соответствия между системными состояниями (S0–S5) и состояниями, поддерживаемыми конкретным устройством.

В таблице указывается состояние устройства с наименьшим энергопотреблением для каждого системного состояния, а также напрямую отражены различные варианты энергопотребления при нахождении компьютера в спящем или ждущем режиме. Например, табл. 6.14 представляет собой возможную таблицу соответствий для шины, поддерживающей все четыре возможных состояния устройств. Большинство драйверов полностью выключает свои устройства (D3) при выходе системы из состояния S0, чтобы на время простоя машины свести к минимуму энергопотребление. Однако некоторые устройства, например сетевые адаптеры, поддерживают способность вывода системы из состояния сна. О наличии подобной способности также сообщается при перечислении устройств.

Таблица 6.14. Пример таблицы соответствия системных состояний и состояний устройства

Состояние системы	Состояние устройства
S0 (полностью активное состояние)	D0 (полностью активное состояние)
S1 (засыпание)	D1
S2 (засыпание)	D2
S3 (засыпание)	D3
S4 (гибернация)	D3 (полное отключение)
S5 (полное отключение)	D3 (полное отключение)

Участие драйверов в управлении электропитанием

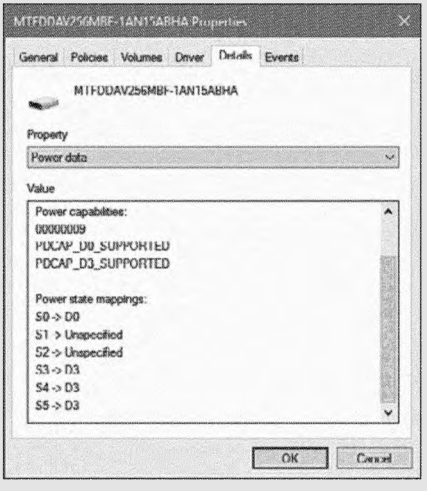
Принимая решение о переходе в другое состояние, диспетчер электропитания посылает команды процедуре драйвера, отвечающей за диспетчеризацию электропитания (`IRP_MJ_POWER`). Управлять устройством могут несколько драйверов, но только один из них назначается проводником политики управления электропитанием устройства (обычно этот драйвер управляет FDO). Этот драйвер определяет состояние устройства в зависимости от состояния энергопотребления системы.

Например, при переходе из состояния S0 в S3 драйвер может принять решение о переводе устройства из состояния D0 в D1.

Вместо прямого оповещения других участвующих в управлении устройством драйверов, драйвер, являющийся проводником политики управления электропитанием, делает это через диспетчер электропитания, вызывая функцию `PoRequestPowerIrp`. Диспетчер электропитания реагирует отправкой соответствующей команды отвечающим за диспетчеризацию электропитания процедурам драйверов. Такое поведение позволяет диспетчеру контролировать количество активных команд управления электропитанием в системе в определенный момент времени. Например, некоторым устройствам в системе для включения может требоваться много электроэнергии. Диспетчер электропитания следит за тем, чтобы такие устройства не включались одновременно.

ЭКСПЕРИМЕНТ: ПРОСМОТР СООТВЕТСТВИЙ СОСТОЯНИЙ ЭЛЕКТРОПИТАНИЯ В ДРАЙВЕРЕ

Посмотреть соответствие состояний электропитания в драйвере и в системе можно через диспетчер устройств. Откройте для устройства диалоговое окно Свойства (Properties), перейдите на вкладку Сведения (Details), откройте раскрывающийся список и выберите Сведения о питании (Power Data). В этом окне также выводятся данные о текущем состоянии электропитания устройства, возможностях электропитания и состояниях, допускающих пробуждение системы.



Для многих команд управления электропитанием предусмотрены соответствующие команды-запросы. Например, при переходе системы в ждущий режим диспетчер электропитания сначала опрашивает устройства о допустимости такого перехода. Устройство, занятое выполнением критических по времени операций или взаимодействующее с другим устройством, может отклонить запрос, и система останется в прежнем состоянии.

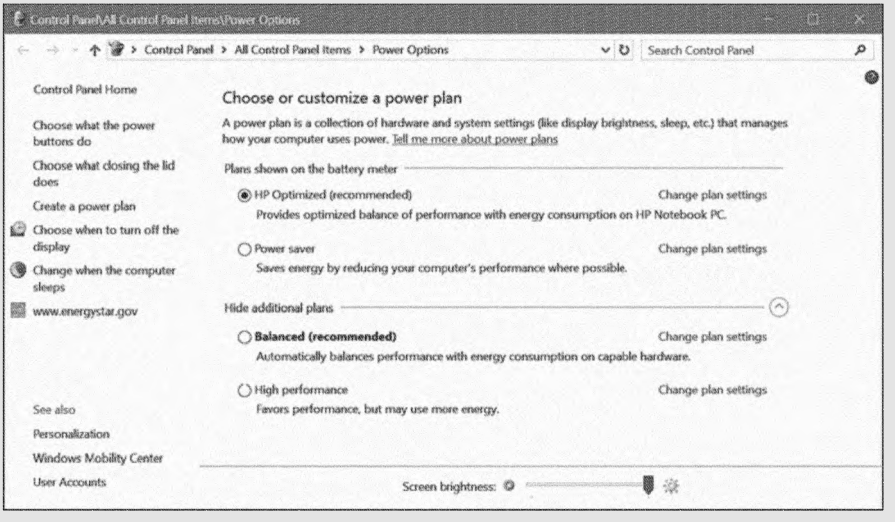
ЭКСПЕРИМЕНТ: ПРОСМОТР СИСТЕМНОЙ ПОЛИТИКИ И ВОЗМОЖНОСТЕЙ УПРАВЛЕНИЯ ЭЛЕКТРОПИТАНИЕМ

Возможности своего компьютера в плане управления электропитанием можно узнать при помощи команды !pocaps отладчика ядра. Вот пример выводимых этой командой данных для портативного компьютера с Windows 10 на платформе x64:

```
lkd> !pocaps
PopCapabilities @ 0xfffff8035a98ce60
  Misc Supported Features:  PwrButton SlpButton Lid S3 S4 S5 HiberFile FullWake
VideoDim
  Processor Features:       Thermal
  Disk Features:
  Battery Features:         BatteriesPresent
    Battery 0 - Capacity:        0  Granularity:        0
    Battery 1 - Capacity:        0  Granularity:        0
    Battery 2 - Capacity:        0  Granularity:        0
  Wake Caps
    Ac OnLine Wake:         Sx
    Soft Lid Wake:          Sx
    RTC Wake:               S4
    Min Device Wake:        Sx
    Default Wake:           Sx
```

Строка Misc Supported Features говорит о том, что кроме S0 система поддерживает состояния S3, S4 и S5 (состояния S1 и S2 не реализованы) и имеет действительный файл гибернации, в котором можно сохранить содержимое системной памяти при переходе в спящий режим (состояние S4).

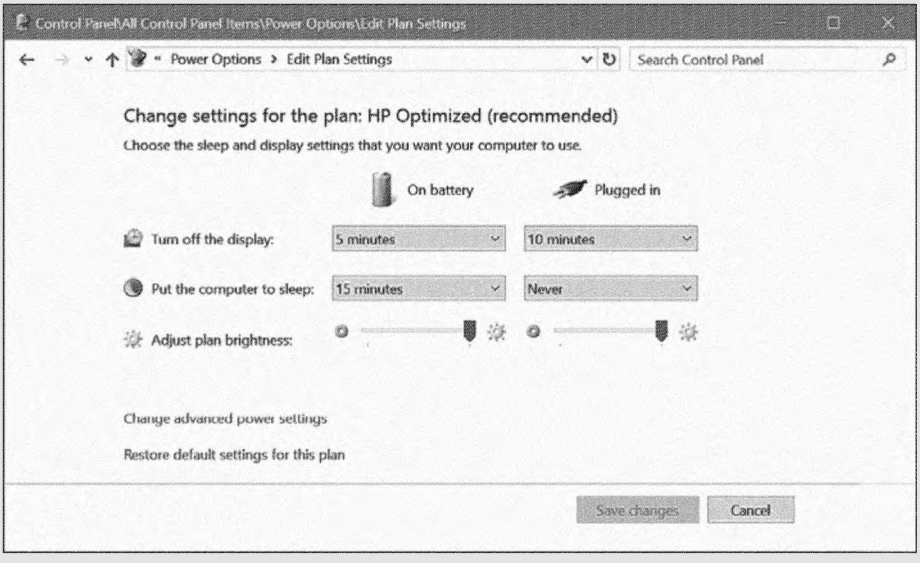
Показанная ниже страница Электропитание (Power Options), которая открывается после выбора одноименной команды в панели управления, позволяет настраивать различные аспекты системной политики управления электропитанием. Точный перечень доступных для настройки параметров зависит от возможностей системы в части управления электропитанием, с которыми мы только что познакомились.



Учтите, что OEM-производители могут добавлять схемы управления электропитанием. Список этих схем выводится командой powercfg /list:

```
C:\WINDOWS\system32>powercfg /list

Existing Power Schemes (* Active)
-----------------------------------
Power Scheme GUID: 381b4222-f694-41f0-9685-ff5bb260df2e  (Balanced)
Power Scheme GUID: 8759706d-706b-4c22-b2ec-f91e1ef6ed38  (HP Optimized
(recommended)) *
Power Scheme GUID: 8c5e7fda-e8bf-4a96-9a85-a6e23a8c635c  (High performance)
Power Scheme GUID: a1841308-3541-4fab-bc81-f71556f20b4a  (Power saver)
```

Меняя заранее заданный план управления электропитанием, можно устанавливать интервалы простоя, по истечении которых отключается монитор, останавливаются жесткие диски, происходит переход в ждущий (состояние S3 из предыдущего эксперимента) и в спящий (состояние S4) режимы. Кроме того, параметр Настройка плана электропитания (Change Plan Settings) позволяет задать поведение системы при нажатии кнопок включения электропитания или перехода в спящий режим, а также при закрытии крышки ноутбука.



Ссылка Изменить дополнительные параметры питания (Change Advanced Power Settings) непосредственно влияет на системную политику управления электропитанием. Параметры этой политики можно посмотреть при помощи команды `!popolicy` отладчика ядра. Вот как выглядит информация, выводимая этой командой для рассматриваемой системы:

```
lkd> !popolicy
SYSTEM_POWER_POLICY (R.1) @ 0xfffff8035a98cc64
  PowerButton:         Sleep  Flags: 00000000   Event: 00000000
  SleepButton:         Sleep  Flags: 00000000   Event: 00000000
```

```
LidClose:              None  Flags: 00000000   Event: 00000000
Idle:                 Sleep  Flags: 00000000   Event: 00000000
OverThrottled:         None  Flags: 00000000   Event: 00000000
IdleTimeout:              0  IdleSensitivity:        90%
MinSleep:                S3  MaxSleep:               S3
LidOpenWake:             S0  FastSleep:              S3
WinLogonFlags:            1  S4Timeout:               0
VideoTimeout:           600  VideoDim:                0
SpinTimeout:           4b0   OptForPower:             0
FanTolerance:             0% ForcedThrottle:          0%
MinThrottle:              0% DyanmicThrottle:      None (0)
```

Первые строки описывают реакцию системы на нажатие кнопок включения электропитания и перехода в спящий режим. В данной системе обе эти кнопки переводят систему в спящий режим. С другой стороны, при закрытии крышки ноутбука не происходит ничего. Показанные в конце значения тайм-аутов выражаются в секундах (они представлены в шестнадцатеричной системе). Они соответствуют указанным на странице Электропитание (Power Options) параметрам. К примеру, тайм-аут для монитора равен 600, что означает его отключение после 600 секунд, или 10 минут, простоя (из-за ошибки в отладчике значение выводится в десятичном виде). Жесткий диск перестает работать после тайм-аута 0x4b0, что соответствует 1200 секундам, или 20 минутам.

Управление электропитанием устройств со стороны драйверов и приложений

Драйвер не только отвечает на команды диспетчера электропитания, связанные с изменением состояния системы, но и сам может управлять состоянием энергопотребления своих устройств. В некоторых случаях он может снижать энергопотребление управляемого им устройства, если оно неактивно в течение некоторого времени. В качестве примеров можно вспомнить мониторы, поддерживающие режим уменьшения яркости и функцию остановки дисков. Драйвер может как самостоятельно распознавать простаивающее устройство, так и пользоваться для этого механизмами, предоставляемыми диспетчером электропитания. Во втором случае устройство регистрируется в диспетчере электропитания, вызывая функцию `PoRegisterDeviceForIdleDetection`.

Эта функция сообщает диспетчеру электропитания пороговые интервалы простоя устройства и указывает, в какое состояние следует перевести устройство в этом случае. Драйвер задает два тайм-аута: первый для энергосберегающей конфигурации, второй для максимально производительной. После вызова функции `PoRegisterDeviceForIdleDetection` драйвер должен уведомить диспетчер электропитания об активности устройства через функцию `PoSetDeviceBusy` или `PoSetDeviceBusyEx`, а затем снова зарегистрироваться, чтобы распознавать ситуации простоя и при необходимости включать и выключать устройства. API-интерфейсы `PoStartDeviceBusy` и `PoEndDeviceBusy` доступны и в более новых версиях Windows; это упрощает программную логику, необходимую для реализации желаемого поведения.

Несмотря на возможность управлять собственным состоянием электропитания, устройство не может влиять на состояние электропитания системы и предотвращать переходы в другие состояния. К примеру, если плохо написанный драйвер не поддерживает состояния низкого энергопотребления, он может оставить устройство полностью включенным или полностью его отключить, не мешая переходу системы в состояние низкого энергопотребления. Ведь диспетчер электропитания просто *уведомляет* драйвер о переходе, не спрашивая его *согласия* на это действие. Драйверы получают IRP-пакет запроса энергопотребления (`IRP_MN_QUERY_POWER`), когда система готовится к переходу в состояние с более низким энергопотреблением. Драйвер может наложить вето на запрос, однако диспетчер энергопотребления не обязан подчиниться; он может отложить переход по мере возможности (например, если устройство работает на аккумуляторе, заряд которого еще далек от критического), и все же попытка перехода в состояние гибернации никогда не может завершиться неудачей.

За управление электропитанием главным образом отвечают драйверы и ядро, но приложения тоже имеют возможность внести свой вклад. Процессы пользовательского режима могут регистрироваться для различных уведомлений о состояниях электропитания, например о разряженности аккумулятора, о переключении с аккумулятора (DC) на питание от сети (AC) или о начале перехода системы в другое состояние. Однако, как и драйверы, приложения не в состоянии воспрепятствовать этим операциям. У них есть до двух секунд на очистку состояния, необходимого для перехода в спящий режим.

Инфраструктура управления электропитанием

Начиная с Windows 8, ядро предоставляет инфраструктуру для управления состоянием электропитания для отдельных компонентов (иногда называемых *функциями*) устройств. Например, представьте, что у аудиоустройства имеются компоненты воспроизведения и записи; если компонент воспроизведения активен, а компонент записи — нет, было бы полезно перевести компонент записи в состояние пониженного энергопотребления. Инфраструктура управления электропитанием PoFx (Power Management Framework) предоставляет API, который используется драйверами для обозначения состояний энергопотребления и требований компонентов. Все компоненты должны поддерживать состояние полной активности, которое обозначается F0.

F-состояния с высокими номерами обозначают состояния пониженного энергопотребления, в котором может находиться компонент; более высоким номерам соответствует более низкое энергопотребление и большее время перехода в F0. Обратите внимание: управление F-состояниями имеет смысл только при нахождении устройства в состоянии D0, потому что в более высоких D-состояниях оно вообще не работает.

Владелец политики электропитания устройства (обычно FDO) должен зарегистрироваться в PoFx вызовом функции `PoFxRegisterDevice`. При вызове драйвер передает следующую информацию.

- Количество компонентов в устройстве.
- Набор обратных вызовов, которые могут быть реализованы драйвером для уведомления о различных событиях со стороны PoFx — например, переключении в активное состояние или состояние бездействия, переходе устройства в состояние D0 и отправки кодов управления питанием (за дополнительной информацией обращайтесь к документации WDK).
- Для каждого компонента — количество поддерживаемых F-состояний.
- Для каждого компонента — самое глубокое F-состояние, из которого может пробудиться компонент.
- Для каждого компонента и для каждого F-состояния — время, необходимое для возврата из этого состояния в F0, минимальное время пребывания компонента в F-состоянии, с которым переход становится оправданным, и номинальное энергопотребление компонента в данном F-состоянии. Также можно указать, что энергопотребление пренебрежимо мало и может не учитываться при принятии PoFx решения об одновременном пробуждении нескольких компонентов.

PoFx использует эту информацию — в сочетании с информацией от других устройств и общесистемной информацией о состоянии энергопотребления (например, текущего профиля энергопотребления) — для принятия разумных решений о том, в каком F-состоянии должен находиться тот или иной компонент. Проблема в том, как согласовать две противоречивые цели: (1) добиться того, чтобы бездействующие компоненты потребляли как можно меньше энергии, и (2) позаботиться о том, чтобы компонент переходил в состояние F0 достаточно быстро, чтобы компонент воспринимался как постоянно подключенный.

Драйвер должен оповещать PoFx о том, что компонент должен быть активным (состояние F0), вызовом `PoFxActivateComponent`. Иногда после этого вызова PoFx активизирует соответствующий обратный вызов, сообщая драйверу о том, что компонент находится в состоянии F0. И наоборот, когда драйвер определяет, что компонент в настоящее время не нужен, он вызывает `PoFxIdleComponent`, на который PoFx реагирует переводом компонента в F-состояние с пониженным энергопотреблением и уведомлением драйвера о завершении перехода.

Управление производительностью

Только что описанные механизмы позволяют компоненту, находящемуся в состоянии бездействия (состояния, отличные от F0), потреблять меньше энергии, чем в состоянии F0. Но некоторые компоненты даже в состоянии F0 могут потреблять меньше энергии в зависимости от работы, выполняемой устройством. Например,

видеокарта может использовать меньше энергии при отображении в основном статического изображения, тогда как при воспроизведении 3D-изображения с частотой 60 кадров в секунду энергопотребление существенно возрастет.

В Windows 8.*x* такие драйверы должны реализовать специальный алгоритм выбора состояния производительности и обращаться с уведомлением к подсистеме ОС, которая называется PEP (Platform Extension Plug-in). PEP относится к конкретной линейке процессоров или однокристальных систем (SoC), вследствие чего код драйвера плотно привязывается к PEP.

Windows 10 расширяет PoFx API в области управления состоянием производительности, давая возможность коду драйвера использовать стандартные API-функции и не беспокоиться о конкретной реализации PEP для платформы. Для каждого компонента PoFx предоставляет следующие типы состояний производительности:

- Конечное число состояний по частоте (Гц), пропускной способности (бит в секунду) и непрозрачного числа, имеющего смысл для драйвера.
- Непрерывная шкала состояний между минимумом и максимумом (частота, пропускная способность или специальное значение).

Например, видеокарта может определить конечное множество частот, на которых она может работать, косвенно влияющих на энергопотребление. Аналогичные наборы производительности могут определяться и для ее пропускной способности, если это уместно.

Чтобы зарегистрироваться в PoFx для управления состоянием производительности, драйвер должен сначала зарегистрировать устройство в PoFx (`PoFxRegisterDevice`), как описано в предыдущем разделе. Затем драйвер вызывает `PoFxRegisterComponentPerfStates` и передает подробную информацию производительности (конечные значения или диапазон, частота, пропускная способность, специальное значение) и обратный вызов, в котором фактически происходит изменение состояния.

Когда драйвер решает, что компонент должен изменить состояние производительности, он вызывает `PoFxIssuePerfStateChange` или `PoFxIssueComponentPerfStateChangeMultiple`. Эти вызовы приказывают PEP перевести компонент в заданное состояние (для заданного индекса или значения в зависимости от того, используются ли конечные состояния или диапазон). Драйвер мог также указать, что вызов должен быть синхронным, асинхронным или «произвольным» (в этом случае решение принимает PEP). В любом случае PoFx в конечном итоге активизирует обратный вызов, зарегистрированный драйвером, с состоянием производительности, которое может совпадать с запрошенным — однако последнее также может быть отклонено PEP. Если запрос принят, драйвер обращается с соответствующими вызовами к оборудованию для внесения запрашиваемых изменений. Если PEP отклоняет запрос, драйвер может повторить попытку с новым вызовом одной из упоминавшихся функций. Перед активизацией обратного вызова драйвера может быть выполнен только один вызов.

Запросы на изменение режима электропитания

Несмотря на отсутствие у приложений и драйверов возможности запретить уже начавшийся переход в состояние сна, некоторые сценарии требуют механизма отключения этого перехода при определенном взаимодействии пользователя с системой. Например, при просмотре пользователем фильма компьютер вроде бы простаивает (так как в течение 15 минут отсутствуют перемещения указателя мыши и клавиатурный ввод), поэтому медиа-проигрыватель должен иметь возможность временно запрещать переход ко сну до завершения своей работы. Скорее всего, вы вспомните и другие варианты экономии электроэнергии, которые система предпринимает в подобных случаях, например отключение или затемнение экрана. В старых версиях Windows API-функция пользовательского режима `SetThreadExecutionState` могла контролировать систему, уведомляя диспетчер электропитания о том, что пользователь все еще присутствует в системе. Но эта функция была лишена средств диагностики и не обеспечивала достаточную детализацию запросов на управление электропитанием. Кроме того, драйверы не могли посылать собственные запросы, а пользовательским приложениям приходилось аккуратно обрабатывать свою потоковую модель, так как запросы приходили не на уровне процессов или системы, а на уровне программных потоков.

В настоящее время в Windows поддерживаются объекты, представляющие собой запросы на управление электропитанием, реализуемые ядром и определяемые диспетчером объектов. С помощью программы WinObj, с которой мы познакомились в главе 8, можно просмотреть объекты типа `PowerRequest` в папке `\ObjectTypes` или же воспользоваться командой `!object` отладчика ядра, применив ее к типу объекта `\ObjectTypes\PowerRequest`.

Запросы на управление электропитанием генерируются приложениями пользовательского режима через API-функцию `PowerCreateRequest`, а затем включаются или выключаются API-функциями `PowerSetRequest` и `PowerClearRequest` соответственно. В ядре драйверы пользуются функциями `PoCreatePowerRequest`, `PoSetPowerRequest` и `PoClearPowerRequest`. Так как в данном случае дескрипторы не применяются, была реализована функция `PoDeletePowerRequest`, удаляющая ссылку на объект (в то время как в пользовательском режиме можно воспользоваться функцией `CloseHandle`).

В Power Request API могут использоваться четыре типа запросов.

- **Системный запрос.** Запрос этого типа запрещает системе автоматически переходить в спящий режим из-за таймера бездействия (хотя пользователь все равно может перейти в спящий режим — например, закрыв крышку ноутбука).
- **Экранный запрос.** Запрос этого типа делает то же, что и системный запрос, но для экрана.
- **Запрос режима отсутствия.** Это разновидность нормального поведения спящего режима Windows (S3), который используется для перевода компьютера в режим полной активности с отключенным экраном и звуком, вследствие чего пользователю кажется, что машина на самом деле находится в спящем

состоянии. Такое поведение обычно используется только специализированными приставками или медиацентрами, если передача данных должна продолжаться даже при нажатии пользователем физической кнопки Sleep.

- **Запрос необходимости выполнения.** Такие запросы (появившиеся в Windows 8 и Server 2012) требуют, чтобы процессы UWP-приложений продолжали выполнение даже в том случае, если в обычных условиях оно было бы завершено PLM (Process Lifecycle Manager); расширенный промежуток времени зависит от таких факторов, как настройки политики электропитания. Такой тип запроса поддерживается только в системах с поддержкой текущего режима ожидания; в противном случае запрос интерпретируется как системный.

ЭКСПЕРИМЕНТ: ПРОСМОТР ЗАПРОСОВ НА УПРАВЛЕНИЕ ЭЛЕКТРОПИТАНИЕМ

К сожалению, объекты ядра запроса на управление электропитанием, созданные такими вызовами, как `PowerCreateRequest`, недоступны в общедоступных символических именах. Тем не менее программа Powercfg предоставляет возможность просмотра запросов без использования отладчика ядра. Ниже приведен результат работы программы во время воспроизведения видео и потокового аудио по сети на ноутбуке с системой Windows 10:

```
C:\WINDOWS\system32>powercfg /requests
DISPLAY:
[PROCESS] \Device\HarddiskVolume4\Program Files\WindowsApps\Microsoft.
ZuneVideo_10.16092.10311.0_x64__8wekyb3d8bbwe\Video.UI.exe
Windows Runtime Package: Microsoft.ZuneVideo_8wekyb3d8bbwe

SYSTEM:
[DRIVER] Conexant ISST Audio (INTELAUDIO\FUNC_01&VEN_14F1&DEV_50F4&SUBSYS_
         103C80D3&R
EV_1001\4&1a010da&0&0001)
An audio stream is currently in use.
[PROCESS] \Device\HarddiskVolume4\Program Files\WindowsApps\Microsoft.
ZuneVideo_10.16092.10311.0_x64__8wekyb3d8bbwe\Video.UI.exe
Windows Runtime Package: Microsoft.ZuneVideo_8wekyb3d8bbwe

AWAYMODE:
None.

EXECUTION:
None.

PERFBOOST:
None.

ACTIVELOCKSCREEN:
None.
```

В выходных данных приведены шесть типов запросов (вместо четырех, описанных выше). Последние два типа — PERFBOOST и ACTIVELOCKSCREEN — объявляются среди внутренних типов запросов на управление питанием в заголовках ядра; в настоящее время они не используются.

Заключение

Подсистема ввода/вывода определяет модель обработки ввода/вывода в Windows и выполняет общие для набора драйверов функции. Основной сферой ее ответственности является создание IRP-пакетов, представляющих запросы на ввод и вывод, и передача этих пакетов через различные драйверы с возвращением результатов вызывающему программному потоку после завершения ввода/вывода. Диспетчер ввода/вывода локализует драйверы и устройства через объекты подсистемы ввода/вывода, в том числе объекты драйверов и устройств. Для повышения быстродействия подсистема ввода/вывода Windows работает асинхронно, обеспечивая приложения пользовательского режима как синхронным, так и асинхронным вводом/выводом.

К драйверам устройств относятся не только традиционные драйверы, управляющие аппаратными устройствами, но и драйверы файловой системы, сетевые драйверы, а также многоуровневые фильтрующие драйверы. Все они имеют общую структуру и используют одинаковые механизмы взаимодействия друг с другом и с диспетчером ввода/вывода. Интерфейсы подсистемы ввода/вывода позволяют писать драйверы на высокоуровневом языке, что ускоряет их разработку и улучшает портируемость. Благодаря наличию общей структуры драйверы могут располагаться друг над другом, обеспечивая модульность и уменьшая дублирование функций между драйверами. Кроме того, благодаря наличию общего стандарта Windows DDI драйверы могут использоваться для разных устройств и форм-факторов без изменения кода.

Наконец, роль PnP-диспетчера состоит в том, чтобы совместно с драйверами устройств динамически распознавать оборудование и формировать внутреннее дерево устройств, упрощающее их перечисление и установку драйверов. Диспетчер электропитания по возможности переводит устройства в состояния с пониженным энергопотреблением для экономии электроэнергии и продления срока работы аккумуляторов.

Следующая глава посвящена одному из важнейших аспектов современных компьютеров: безопасности.

Глава 7
Безопасность

В любой среде, предоставляющей доступ к одним и тем же физическим или сетевым ресурсам сразу нескольким пользователям, остро стоит вопрос предотвращения неавторизованного доступа к конфиденциальным данным. Операционная система, наряду с отдельными пользователями, должна иметь возможность защитить файлы, память и настройки конфигурации от нежелательного просмотра и изменения. Средства безопасности операционной системы включают в себя такие вполне очевидные механизмы, как учетные записи, пароли и защита файлов. Они также включают в себя такие менее заметные механизмы, как защита операционной системы от повреждения, недопущение осуществления ряда действий (например, перезагрузки компьютера) со стороны менее привилегированных пользователей и запрещение неблагоприятного воздействия пользовательских программ на программы других пользователей или на операционную систему.

В данной главе будет рассмотрено, каким образом каждый аспект архитектуры и реализации Microsoft Windows влияет на выполнение жестких требований обеспечения безопасности.

Оценка безопасности

Наличие программного обеспечения (включая операционные системы), оцениваемого с точки зрения вполне определенных стандартов, помогает правительству, корпорациям и домашним пользователям защитить конфиденциальные и персональные данные, хранящиеся в компьютерных системах. Текущий стандарт оценки безопасности, используемый в Соединенных Штатах и во многих других странах, называется общими критериями (Common Criteria, CC). Но чтобы понять, какие возможности по обеспечению безопасности встроены в Windows, полезно будет узнать историю системы оценки безопасности, повлиявшей на конструкцию Windows, критерии оценки заслуживающих доверия компьютерных систем — Trusted Computer System Evaluation Criteria (TCSEC).

Критерии оценки заслуживающих доверия компьютерных систем

Национальный центр компьютерной безопасности (National Computer Security Center, NCSC) был основан в 1981 году как часть национального агентства безопасности (National Security Agency, NSA) министерства обороны США (U.S. Department of Defense, DoD). Одной из целей NCSC было создание диапазона оценок безопасности, показанного в табл. 7.1, используемого для индикации степени защиты коммерческих операционных систем, сетевых компонентов и предложения заслуживающих доверия приложений. Эти оценки безопасности, которые можно найти по адресу *http://csrc.nist.gov/publications/history/dod85.pdf*, были определены в 1983 году и обычно обозначаются термином «*Оранжевая книга*».

Таблица 7.1. Оценочные уровни стандарта TCSEC

Оценочный уровень	Описание
A1	Проверенная архитектура (Verified Design)
B3	Домены безопасности (Security Domains)
B2	Структурированная защита (Structured Protection)
B1	Защита с использованием грифа секретности (Labeled Security Protection)
C2	Защита управляемого доступа (Controlled Access Protection)
C1	Защита избирательного доступа (считается устаревшим) (Discretionary Access Protection)
D	Минимальная защита (Minimal Protection)

Стандарт TCSEC состоит из оценочных показателей «уровней доверия», где самые высокие уровни основываются на нижних уровнях и выстраиваются путем добавления более жестких требований защиты и проверок. Ни одна из операционных систем не отвечает требованиям A1 или оценочному уровню «Verified Design» — проверенной архитектуре. Хотя некоторые операционные системы получили одну из оценок B-уровня, вполне достаточным и самым высоким оценочным уровнем практически для всех операционных систем общего назначения является C2.

Перечисленные далее требования оценки безопасности C2 считались основными, и они по-прежнему считаются основными требованиями для любой защищенной операционной системы.

- **Механизм безопасного входа** в систему, который требует уникальной идентификации пользователей и получения ими полномочий доступа к компьютеру только после того, как они тем или иным способом пройдут аутентификацию.
- **Управление избирательным доступом**, позволяющее владельцу ресурса (например, файла) определить, кто может получить доступ к ресурсу и что он при этом может с ним сделать. Владелец наделяет правами, разрешающими различные виды доступа, отдельного пользователя или группу пользователей.

- **Контроль безопасности**, позволяющий обнаруживать и записывать события, относящиеся к вопросам безопасности, или любые попытки создания системных ресурсов, а также обращения к ним или их удаления. В идентификаторах входа сохраняется информация, идентифицирующая пользователя, что упрощает отслеживание любого пользователя, пытающегося выполнить неавторизованное действие.
- **Защита от повторного использования объекта**, которая не позволяет пользователям просматривать данные, удаленные другим пользователем, или не позволяет обращаться к памяти, которая ранее была использована, а затем освобождена другим пользователем. Например, в некоторых операционных системах можно создать новый файл определенной длины, а затем проверить содержимое файла для просмотра данных, находящихся на том месте диска, которое было выделено файлу. Эти данные могут содержать конфиденциальную информацию, которая была сохранена в файле другого пользователя, но затем была удалена. Защита от повторного использования объекта закрывает эту потенциальную дыру безопасности путем инициализации всех объектов, включая файлы и память перед их выделением пользователю.

Windows также отвечает двум требованиям безопасности B-уровня.

- **Механизм доверенного маршрута** предотвращает возможность перехвата пользовательских имен и паролей троянскими программами при попытке входа в систему. Используемый в Windows механизм доверенного маршрута появился в форме комбинации входа в систему Ctrl+Alt+Delete, которая не может быть перехвачена непривилегированными приложениями. Эта последовательность клавиш, известная также как *защищенная последовательность переноса внимания* — SAS (Secure Attention Sequence), всегда выводит управляемый системой экран безопасности Windows (если пользователь уже вошел в систему) или экран входа в систему, чтобы можно было легко распознать троянские программы. (Если групповая политика это позволяет, SAS-последовательность может быть также отправлена программным способом через SendSAS API.) При вводе SAS троянская программа, предоставляющая поддельное диалоговое окно, будет обойдена.
- **Доверенное средство управления**, которое требует поддержки ролей отдельных учетных записей для осуществления административных функций. Например, для администрирования (администраторов), для учетных записей пользователей, отвечающих за резервное копирование компьютера, и для обычных пользователей предоставляются отдельные учетные записи.

Windows отвечает всем этим требованиям благодаря своей подсистеме безопасности и связанным с ней компонентам.

Общие критерии

В январе 1996 года США, Великобритания, Германия, Франция, Канада и Нидерланды опубликовали совместно разработанную спецификацию оценки безопасности под названием «Общие критерии оценки безопасности информационных си-

стем» — Common Criteria for Information Technology Security Evaluation (CCITSE). Спецификация CCITSE, которую обычно называют просто общими критериями — Common Criteria (CC), — является общепризнанным международным стандартом для оценки безопасности программных продуктов. Главная страница CC находится по адресу *www.niap-ccevs.org/cc-scheme/*.

Стандарт CC обладает большей гибкостью по сравнению с оценками доверительности TCSEC и имеет структуру, более близкую к стандарту ITSEC, чем к стандарту TCSEC. Стандарт CC включает понятие профиля защиты — Protection Profile (PP), используемое для сбора требований безопасности в свободно определяемые и сопоставляемые наборы, и понятие задания по безопасности — Security Target (ST), содержащего набор требований по безопасности, которые могут быть выдвинуты на основе PP. Стандарт CC также определяет диапазон из семи оценочных уровней доверия — Evaluation Assurance Levels (EAL), которые показывают уровень доверия к сертификации. Таким образом, CC (подобно предшествующему ему стандарту ITSEC) удаляет связь между функциональностью и уровнем гарантий, которая была представлена в TCSEC и в ранних схемах сертификации.

Операционные системы Windows 2000, Windows XP, Windows Server 2003 и Windows Vista Enterprise добились сертификации Common Criteria под профилем защиты контролируемого доступа — Controlled Access Protection Profile (CAPP). Это примерно соответствует оценочному уровню TCSEC C2. Все они получили оценку EAL 4+, где плюс означает «исправление недостатков». EAL 4 является высшим уровнем, признаваемым за пределами национальных границ.

В марте 2011 года операционные системы Windows 7 и Windows Server 2008 R2 получили оценку соответствия требованиям к профилю защиты, установленным правительством США для операционных систем общего назначения в сетевой среде (GPOSPP) (*http://www.commoncriteriaportal.org/files/ppfi les/pp_gpospp_v1.0.pdf*). Сертификация включает гипервизор Hyper-V. И снова Windows добилась оценочного уровня доверия EAL-4+. Отчет об оценке можно найти на веб-сайте *http://www.commoncriteriaportal.org/files/epfiles/st_vid10390-vr.pdf*, а описание задания по безопасности, дающее подробные сведения об удовлетворенных требованиях, — по адресу *http://www.commoncriteriaportal.org/files/epfiles/st_vid10390-st.pdf*. Аналогичная сертификация для Windows 10 и Windows Server 2012 R2 была проведена в июне 2016 года. Отчет можно найти по адресу *http://www.commoncriteriaportal.org/files/epfiles/cr_windows10.pdf*.

Системные компоненты безопасности

В реализацию системы безопасности Windows входят следующие компоненты и базы данных (все упоминаемые файлы находятся в каталоге %SystemRoot%\System32, если только не указано иное).

- **Монитор безопасности** — Security reference monitor (SRM). Компонент, который находится в исполняющей системе Windows (Ntoskrnl.exe) и отвечает за

определение структуры данных маркеров доступа для представления контекста безопасности, выполнение проверки безопасности доступа к объектам, работу с привилегиями (правами пользователей) и генерирование любых итоговых сообщений проверки безопасности.

- **Подсистема проверки подлинности локальной системы безопасности** — Local Security Authority subsystem (LSASS). Процесс пользовательского режима запускает образ Lsass.exe, который отвечает за политику безопасности локальной системы (которая, например, определяет пользователей, обладающих правом входить в систему, политики паролей, привилегии, предоставленные пользователям и группам, и настройки проверки безопасности системы), аутентификацию пользователя и отправку сообщений проверки безопасности журналу событий (Event Log). Основная часть этих функциональных возможностей реализована в загружаемой LSASS библиотеке службы авторизации локальных пользователей Lsasrv.dll.

- **LSAIso.exe.** LSASS (при соответствующей настройке в поддерживаемых системах Windows 10 и Server 2016) используется для хранения хешей маркеров безопасности пользователей — вместо хранения самих маркеров в памяти LSASS. Данная функция также обозначается термином «Охранник учетных данных» (Credential Guard) — см. раздел «Охранник учетных данных» этой главы. Поскольку Lsaiso.exe также является трастлетом (изолированным процессом пользовательского режима), выполняемым на уровне VTL 1, ни один обычный процесс — и даже нормальное ядро — не может обратиться к адресному пространству этого процесса. LSASS хранит зашифрованный двоичный объект хеша пароля, необходимый для взаимодействия с Lsaiso (через ALPC).

- **База данных политики LSASS.** База данных, в которой содержатся настройки политики безопасности локальной системы. Эта база данных хранится в реестре в ACL-защищенной области в разделе HKLM\SECURITY. Она включает информацию, которая, в частности, определяет, каким доменам доверена аутентификация попыток входа в систему, у кого есть права на доступ к системе и каким образом осуществляется этот доступ (интерактивные, сетевые или служебные регистрации), кому и какие привилегии назначены и какие виды проверки безопасности следует выполнять. В базе данных политики LSASS также хранятся «секреты», включающие информацию о входе в систему, использующуюся для кэшированных входов в домены и входов в службы Windows под учетной записью пользователя. Дополнительные сведения о службах Windows даны в главе 9.

- **Администратор учетных данных в системе защиты** — Security Accounts Manager (SAM). Служба, отвечающая за управление базой данных, содержащей имена пользователей и определения групп на локальной машине. SAM-служба, реализованная в виде библиотеки Samsrv.dll, загружается в процесс LSASS.

- **База данных SAM.** База данных, содержащая определения локальных пользователей и групп, наряду с их паролями и другими атрибутами. На контроллерах доменов служба SAM не хранит сведений о пользователях, определенных на до-

мене, но хранит информацию определения учетной записи и пароля системного администратора, имеющего право на восстановление системы. Эта база данных хранится в реестре в разделе HKLM\SAM.

- **Active Directory.** Служба каталогов, которая содержит базу данных, хранящую информацию об объектах в домене. *Домен* является совокупностью компьютеров и связанных с ними групп безопасности, которые управляются как единое целое. Active Directory хранит информацию об объектах в домене, куда включаются пользователи, группы и компьютеры. В Active Directory хранится информация о паролях и привилегиях для пользователей и групп домена, которая реплицируется на компьютерах, назначаемых в качестве контроллеров домена. Сервер Active Directory, реализованный в виде библиотеки Ntdsa.dll, запускается в контексте процесса LSASS. Дополнительные сведения об Active Directory приведены в главе 10 части 2.

- **Пакеты аутентификации** (Authentication packages). Включают DLL-библиотеки, код которых выполняется как в контексте процесса LSASS, так и в контекстах клиентских процессов, и реализуют имеющуюся в Windows политику аутентификации. DLL-библиотека аутентификации отвечает за аутентификацию пользователей, реализуя ее путем проверки соответствия введенного имени пользователя и пароля; если проверка проходит успешно, процессу LSASS возвращается информация, уточняющая детали идентификации пользователя с точки зрения безопасности, которые используются LSASS для генерации маркера доступа.

- **Интерактивный диспетчер входа в систему (Winlogon).** Процесс пользовательского режима, в котором выполняется образ Winlogon.exe, отвечающий за реагирование на SAS и за управление интерактивными сеансами входа в систему. К примеру, Winlogon создает первый пользовательский процесс при входе пользователя в систему.

- **Пользовательский интерфейс входа в систему (LogonUI).** Процесс пользовательского режима, выполняющий образ LogonUI.exe, который предоставляет пользователям интерфейс, используемый ими для самоидентификации в системе. LogonUI использует поставщиков учетных данных для запроса учетных данных пользователя с использованием различных механизмов.

- **Поставщики учетных данных** — Credential providers (CP). Внутрипроцессные COM-объекты, запускающие процесс LogonUI (стартующий по запросу со стороны Winlogon при выполнении SAS) и используемые для получения имени пользователя и пароля, PIN-кода смарткарты или биометрических данных (например, отпечатка пальца). К числу стандартных CP относятся библиотеки authui.dll, SmartcardCredentialProvider.dll, BioCredProv.dll и FaceCredentialProvider.dll — реализация распознавания лиц, появившаяся в Windows 10.

- **Служба сетевого входа (Netlogon).** Служба Windows (Netlogon.dll, работает под управлением стандартного хоста SvcHost) создает защищенный канал к контроллеру домена, по которому отправляются запросы безопасности — на-

пример, интерактивный вход в систему (если на контроллере домена запущена Windows NT 4) или проверка аутентификации диспетчера LAN Manager и диспетчера NT LAN Manager (v1 и v2). Netlogon также используется для входов в систему со стороны Active Directory.

- **Драйвер устройства безопасности ядра (KSecDD, Kernel Security Device Driver).** Библиотека функций режима ядра (%SystemRoot%\System32\Drivers\Ksecdd.sys), реализующая интерфейсы усовершенствованной системы вызова локальных процедур — ALPC (Advanced Local Procedure Call), которые используются для обмена данными в пользовательском режиме с LSASS другими компонентами безопасности режима ядра, включая такой компонент, как шифрующая файловая система — Encrypting File System (EFS).
- **AppLocker.** Механизм, позволяющий администраторам определять, какие исполняемые файлы, DLL-библиотеки и сценарии могут использоваться конкретными пользователями и группами. AppLocker состоит из драйвера (%SystemRoot%\System32\Drivers\AppId.sys) и службы (AppIdSvc.dll), выполняемой в процессе SvcHost.

На рис. 7.1 показаны взаимоотношения между некоторыми из этих компонентов и баз данных, которыми они управляют.

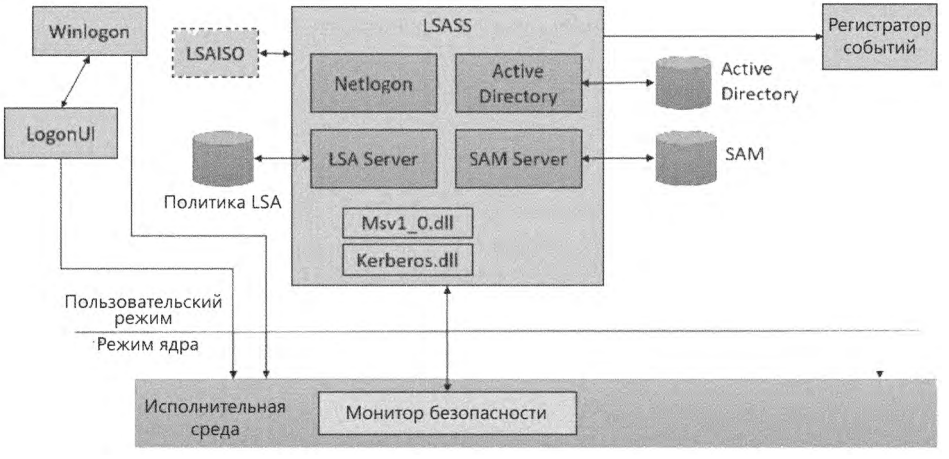


Рис. 7.1. Компоненты системы безопасности Windows

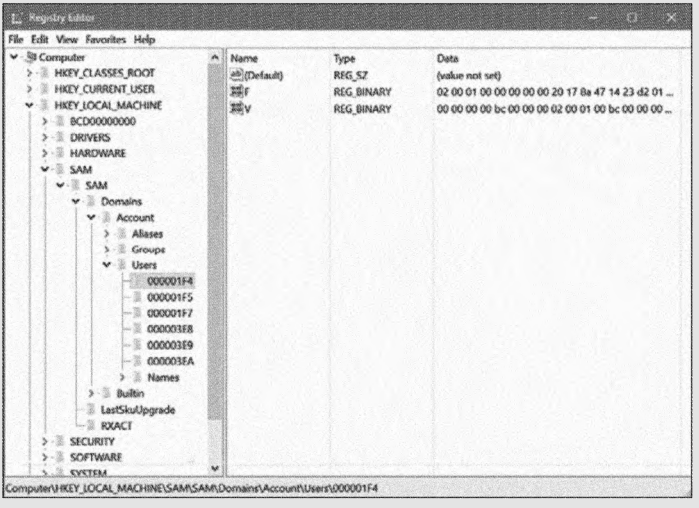
ЭКСПЕРИМЕНТ: ВНУТРИ HKLM\SAM И HKLM\SECURITY

Дескрипторы безопасности, связанные с разделами реестра `SAM` и `Security`, блокируют доступ к ним под любой учетной записью, отличной от учетной записи локальной системы. Один из способов получения доступа к этим разделам для их исследования заключается в сбросе их защиты, но это может ослабить безопасность системы. Другой способ основан на выполнении Regedit.exe под

учетной записью локальной системы. Это можно сделать путем использования средства PsExec, входящего в набор Windows Sysinternals, с ключом `-s`:

```
C:\>psexec -s -i -d c:\windows\regedit.exe
```

Ключ `-i` приказывает PsExec выполнить исполняемый файл в интерактивном окне. Без него процесс будет выполняться в неинтерактивном окне на невидимом рабочем столе. Ключ `-d` просто указывает, что команда PsExec не должна ожидать завершения целевого процесса.



Монитор безопасности SRM, выполняющийся в режиме ядра, и процесс LSASS, выполняющийся в пользовательском режиме, обмениваются данными с помощью средства ALPC, рассмотренного в главе 8. В ходе инициализации системы SRM создает порт `SeRmCommandPort`, к которому подключается LSASS. Когда запускается процесс LSASS, он создает ALPC-порт `SeLsaCommandPort`. SRM подключается к этому порту, приводя к созданию закрытых коммуникационных портов. SRM создает общий раздел памяти для сообщений, превышающих размер 256 байт, передавая дескриптор в запросе на подключение. После того как SRM и LSASS подключатся друг к другу в ходе инициализации системы, больше они соответствующие коммуникационные порты не прослушивают. Поэтому более поздний пользовательский процесс не имеет возможности успешно подключиться к любому из этих портов для нанесения какого-либо вреда, поскольку запрос на подключение никогда не будет завершен.

Безопасность на основе виртуализации

Ядро часто называют *доверенным* из-за присущего ему более высокого уровня привилегий и изоляции от приложений пользовательского режима. Тем не менее каждый месяц пишутся бесчисленные сторонние драйверы — по заявлениям компании

Microsoft ежемесячно наблюдается около миллиона уникальных хешей драйверов! Все эти драйверы могут содержать сколько угодно уязвимостей, не говоря уже о намеренно вредоносном коде режима ядра. В такой ситуации представление о том, что ядро представляет собой небольшой защищенный компонент, а приложения пользовательского режима «защищены» от атаки, совершенно очевидно является нереализованным идеалом. При таком состоянии дел полное доверие к ядру невозможно, а ключевые приложения пользовательского режима, которые могут содержать исключительно ценные пользовательские данные, открыты для злонамеренного использования со стороны вредоносных приложений пользовательского режима (использующих дефекты в компонентах режима ядра) или вредоносных программ режима ядра.

Как упоминалось в главе 2, Windows 10 и Server 2016 включают архитектуру безопасности на основе виртуализации (VBS), которая предоставляет дополнительный ортогональный уровень доверия: виртуальные уровни безопасности (VTL). В этом разделе вы узнаете о том, как Охранник учетных данных и Device Guard используют VTL для защиты пользовательских данных и формирования дополнительного уровня безопасности, основанного на аппаратных средствах доверия, для работы с цифровыми подписями. В конце этой главы вы также увидите, как механизм защиты KPP (Kernel Patch Protection) предоставляется на базе компонента PatchGuard и дополняется технологией HyperGuard на базе VBS.

Напомним, что обычный код пользовательского режима и код ядра выполняются на уровне VTL 0 и ничего не знают о существовании VTL 1. Это означает, что любые компоненты, размещенные на уровне VTL 1, скрыты и недоступны для кода VTL 0. Если вредоносная программа сможет проникнуть в обычное ядро, она все равно не получит доступ к чему-либо на уровне VTL 1 — даже к коду пользовательского режима, выполняемому на уровне VTL 1 (который называется «изолированным пользовательским режимом»). На рис. 7.2 изображены основные компоненты VBS, которые будут рассматриваться в этом разделе:

- технологии HVCI (Hypervisor-Based Code Integrity) и KMCI (Kernel-Mode Code Integrity), на основе которых работает Device Guard;
- LSA (Lsass.exe) и изолированная LSA (Lsalso.exe), на основе которых работает Охранник учетных данных.

Кроме того, вспомните, что в главе 3 была представлена реализация трастлетов, выполняемых в IUM.

Конечно, как и любой доверенный компонент, уровень VTL 1 также делает некоторые предположения относительно того, заслуживают ли доверия компоненты, от которых он зависит. По этой причине для правильного функционирования VTL 1 механизм безопасной загрузки Secure Boot (а следовательно, встроенное ПО), защищенный гипервизор и аппаратные элементы — такие, как IOMMU и Intel Management Engine, — должны быть защищены от уязвимостей, доступных из VTL 0. За дополнительной информацией об аппаратной цепочке сертификатов и технологиях безопасности, относящихся к загрузке, обращайтесь к главе 11 части 2.

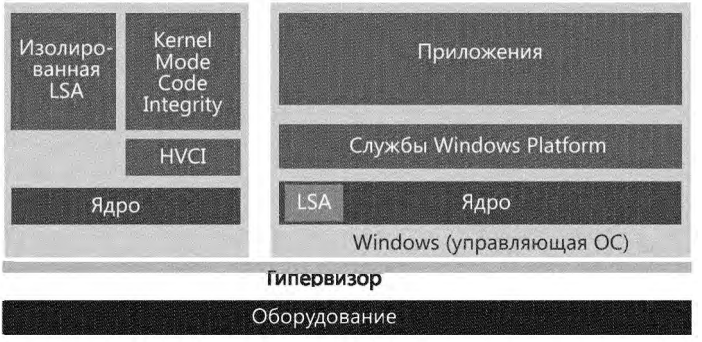


Рис. 7.2. Компоненты VBS

Охранник учетных данных

Чтобы понять границы безопасности и защиты, предоставляемых Охранником учетных данных, важно понимать различные компоненты, предоставляющие доступ к ресурсам и данным пользователя или функциональность входа в сетевой среде.

- **Пароль** — основной вид учетных данных, используемых интерактивными пользователями для идентификации на компьютере. Пароль используется для выполнения аутентификации и как основа для других компонентов модели. Пожалуй, это самый ценный компонент идентификационных данных пользователя.
- **Односторонняя функция NT (NTOWF, NT One-Way Function).** Хеш, используемый унаследованными компонентами для идентификации пользователя (после успешного ввода пароля) с использованием протокола NTLM (NT LAN Manager). Хотя современные сетевые системы не используют NTLM для аутентификации пользователя, этот механизм продолжает использоваться многими локальными компонентами, а также некоторыми унаследованными сетевыми компонентами (например, посредниками аутентификации на базе NTLM). Так как NTOWF представляет собой хеш MD4, его алгоритмическая сложность перед лицом современного оборудования и отсутствие защиты от повторяемости означает, что перехват хеша приведет к моментальному взлому (а возможно, даже восстановлению пароля).
- **Билет на получение билетов (TGT, Ticket-Granting Ticket).** Эквивалент NTOWF, использующий намного более современный механизм удаленной аутентификации: Kerberos. Он используется по умолчанию в доменах на базе Windows Active Directory и принудительно используется в Server 2016. TGT и соответствующий ключ предоставляются локальной машине после успешного входа (как и NTOWF в NTLM); перехват обоих компонентов приводит к мгновенной компрометации учетных данных пользователя, причем повторное использование и восстановление пароля будут невозможны.

Без включения Охранника учетных данных некоторые из этих компонентов учетных данных аутентификации пользователей присутствуют в памяти LSASS.

ПРИМЕЧАНИЕ Чтобы включить Охранника учетных данных в версиях Windows 10 Enterprise и Server 2016, откройте редактор групповой политики (gpedit.msc), откройте узел Конфигурация компьютера (Computer Configuration), выберите пункт Административные шаблоны (Administrative Templates), откройте папку Система (System), выберите пункт Device Guard и выберите команду Turn on Virtualization Based Security. В левой верхней части открывшегося диалогового окна включите режим Enabled. Наконец, выберите один из вариантов Enabled в поле со списком Credential Guard Configuration.

Защита пароля

Пароль, зашифрованный с применением локального симметричного ключа, хранится для предоставления функциональности SSO (Single Sign-On) по таким протоколам, как дайджест-проверка подлинности (WDigest, используется для аутентификации на базе HTTP со времен Windows XP) или Terminal Services/RDP. Так как эти протоколы используют текстовую аутентификацию, пароль должен храниться в памяти; появляется возможность обратиться к нему из внедренного кода, отладчика или других низкоуровневых средств с последующей расшифровкой. Охранник учетных данных не может изменить природу этих принципиально небезопасных протоколов. А следовательно, единственное возможное решение, примененное Охранником учетных данных, заключается в блокировке функциональности SSO для таких протоколов. Это приводит к потере совместимости и заставляет пользователя проходить аутентификацию заново.

Очевидно, предпочтительное решение заключается в полном отказе от использования пароля — Windows Hello (см. раздел «Windows Hello» далее в этой главе) предоставляет такую возможность. Аутентификация с применением биометрических учетных данных (таких, как распознавание лица или отпечатков пальцев пользователя) снимает необходимость во вводе пароля, защищает интерактивные учетные данные от аппаратных регистраторов ввода с клавиатуры (кейлогеров), программ перехватчиков режима ядра и средств фальсификации пользовательского режима. Если пользователю не нужно вводить пароль, то и похитить этот пароль не удастся. Другая разновидность защищенных учетных данных — комбинация смарткарты и PIN-кода. Хотя PIN-код может быть похищен в процессе ввода, смарткарта становится физическим элементом, ключ которого не может быть перехвачен без сложной атаки на аппаратном уровне. Это разновидность *двухфакторной аутентификации*, существующей во множестве других реализаций.

Защита ключа NTOWF/TGT

Даже с защитой интерактивных учетных данных успешный вход приводит к тому, что центр распространения ключей (KDC, Key Distribution Center) возвращает

TGT и его ключ, а также NTOWF для унаследованных приложений. Позднее пользователь просто использует NTOWF для обращения к унаследованным ресурсам, а TGT и его ключ — для генерирования билета службы. В дальнейшем последний может использоваться для обращения к удаленным ресурсам (например, файлам в сетевой папке), как показано на рис. 7.3.

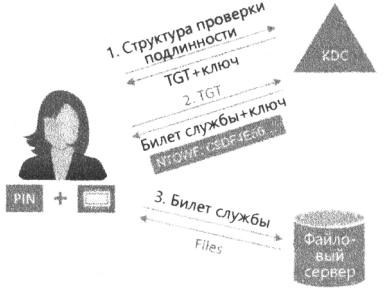


Рис. 7.3. Обращение к удаленным ресурсам

Таким образом, если NTOWF или TGT с ключом (хранимым в LSASS) окажется в руках атакующего, доступ к ресурсам будет возможен даже без смарткарты, PIN-кода, лица или отпечатка пальца пользователя. Защита LSASS от доступа со стороны атакующего — один из возможных вариантов, для реализации которого можно воспользоваться архитектурой PPL (Protected Process Light), описанной в главе 3.

LSASS можно настроить для выполнения в защищенном режиме, для чего параметру `RunAsPPL` типа DWORD в разделе `HKLM\System\CurrentControlSet\Consol\Lsa` присваивается значение 1. (Этот режим не используется по умолчанию, так как допустимые сторонние поставщики аутентификации [DLL-библиотеки] загружаются и выполняются в контексте LSASS, что было бы невозможно при защищенном выполнении LSASS.) К сожалению, хотя этот способ защиты предохраняет NTOWF и ключ TGT от атак из пользовательского режима, он не обеспечивает защиты от атак из режима ядра или атак пользовательского режима, использующих уязвимости в миллионах ежемесячно выпускаемых драйверов. Охранник учетных данных решает эту проблему за счет использования другого процесса `Lsaiso.exe`, выполняемого в форме трастлета на уровне VTL 1. Таким образом, секретные данные пользователя хранятся в памяти этого процесса, а не в памяти LSASS.

Безопасность передачи данных

Как показано в главе 2, уровень VTL 1 обладает минимальной поверхностью атаки, так как он не содержит ни полного обычного ядра «NT», ни драйверов или обращений к вводу/выводу устройств. Соответственно, изолированная реализация LSA, являющаяся трастлетом VTL 1, не может напрямую взаимодействовать с KDC. За это отвечает процесс LSASS, который служит посредником

и обеспечивает реализацию протокола; он взаимодействует с KDC, обеспечивая аутентификацию пользователя, получение TGT, ключа и NTOWF, а также взаимодействие с файловым сервером с использованием билетов службы. И здесь вроде бы возникает проблема: билет TGT и его ключ/NTOWF временно проходят через LSASS в процессе аутентификации, и LSASS получает доступ к TGT и ключу для генерирования билетов службы. Возникают два вопроса: как LSASS получает и отправляет секретные данные от изолированного LSA и как помешать атакующему сделать то же самое?

Чтобы ответить на первый вопрос, вспомните, что в главе 3 рассказывалось о том, какая функциональность доступна для трастлетов. Одна из таких областей — ALPC (Advanced Local Procedure Call) — поддерживается защищенным ядром посредством опосредованных вызовов `NtAlpc*` к нормальному ядру. Затем изолированная среда пользовательского режима реализует поддержку RPC-библиотеки времени выполнения (Rpcrt4.dll) по протоколу ALPC, что позволяет приложениям VTL 0 и VTL 1 передавать данные с использованием локальных RPC-вызовов, как и у любых других приложений и функций. На рис. 7.4 показан процесс Lsaiso.exe в Process Explorer, захвативший дескриптор порта `LSA_ISO_RPC_SERVER ALPC`. Он используется для взаимодействия с процессом Lsass.exe (за дополнительной информацией о ALPC обращайтесь к главе 8 части 2).

Чтобы ответить на второй вопрос, необходимо в определенной степени разбираться в криптографических протоколах и моделях «вызов/ответ». Если вы знакомы с основными концепциями технологии SSL/TLS и ее применения в интернет-коммуникациях для предотвращения атак типа «злоумышленник в середине» (MitM, Man-in-the-Middle), KDC и протокол изолированной LSA можно рассматривать аналогичным образом. Хотя LSASS занимает то же место, что и посредник, он видит только зашифрованный трафик между KDC и изолированной LSA без возможности понимать его содержимое. Так как изолированная LSA устанавливает локальный «сеансовый ключ», который существует только в VTL 1, а затем использует защищенный протокол для передачи сеансового ключа, зашифрованного с другим ключом, которым обладает только KDC, в результате KDC может ответить TGT и ключом, зашифрованным с применением сеансового ключа изолированной LSA. Таким образом, LSASS видит зашифрованное сообщение для KDC (которое не может расшифровать) и зашифрованное сообщение от KDS (которое не может расшифровать).

Эта модель может использоваться даже для защиты унаследованной аутентификации NTLM, основанной на модели «вызов/ответ». Например, когда пользователь выполняет вход с текстовыми учетными данными, LSA отправляет их изолированной LSA, которая затем шифрует их с сеансовым ключом и возвращает зашифрованные учетные данные LSASS. Позднее, когда потребуется аутентификация NTLM «вызов/ответ», LSASS отправляет запрос NTLM и ранее зашифрованные учетные данные изолированной LSA. На этой стадии ключ шифрования имеется только у изолированной LSA, поэтому она расшифровывает учетные данные и генерирует ответ NTML на основании запроса.

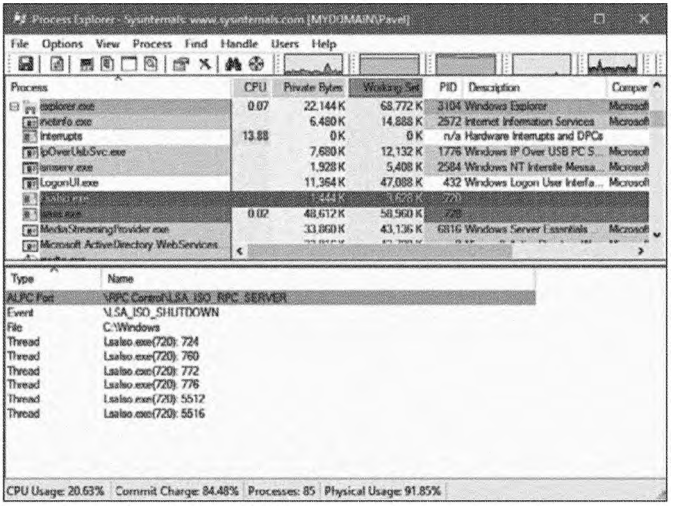


Рис. 7.4. Lsaiso.exe и порт ALPC

Однако следует учесть, что в этой модели существуют четыре возможные атаки.

- Если злоумышленник получит физический доступ к машине, текстовый пароль может быть перехвачен в процессе ввода или при отправке изолированной LSA (если подсистема LSASS уже скомпрометирована). Защитой от такой атаки может стать Windows Hello.
- Как упоминалось ранее, NTML не обеспечивает защиты от повторного воспроизведения. Таким образом, если ответ NTLM будет перехвачен, он может быть повторно воспроизведен для того же запроса. Кроме того, если нападающий сможет взломать LSASS после входа, он сможет захватить зашифрованные учетные данные и заставить изолированную LSA сгенерировать новые ответы NTLM для произвольных запросов NTLM. Тем не менее такая атака будет работать только до перезагрузки, потому что в этот момент изолированная LSA генерирует новый сеансовый ключ.
- В ситуации с входом Kerberos данные NTOWF (не зашифрованные) могут быть перехвачены и использованы повторно, как и при стандартной атаке передачи хеша (pass-the-hash). Еще раз подчеркнем, что для этого требуется уже скомпрометированная машина (или физический перехват сетевого трафика).
- Пользователь с физическим доступом к системе может отключить Охранника учетных данных. В такой ситуации используется унаследованная модель аутентификации, что открывает возможность для проведения более старых моделей атак.

Блокировка UEFI

Так как отключение Охранника учетных данных (которое в конечном итоге сводится к простму изменению параметра реестра) выполняется тривиально для

атакующего, механизмы Secure Boot и UEFI могут предотвратить отключение Охранника учетных данных физически отсутствующим администратором (например, вредоносной программой с административными правами). Для этого Охранник учетных данных включается с блокировкой UEFI (UEFI lock). В этом режиме в память встроенного ПО записывается переменная EFI, и выполняется перезагрузка. При перезагрузке загрузчик Windows, который все еще работает в режиме EFI Boot Services, записывает переменную времени выполнения EFI (недоступную для чтения и записи после выхода из режима EFI Boot Services) для регистрации факта включения Охранника учетных данных. Кроме того, регистрируется режим BCD (Boot Configuration Database).

При загрузке ядро автоматически переписывает необходимый параметр реестра Охранника учетных данных при наличии режима BCD и/или переменной времени выполнения UEFI. Если параметр BCD будет удален атакующим, система BitLocker (если она включена) и удаленная аттестация на базе TPM (если она включена) обнаружит изменение и потребует физического ввода административного ключа восстановления перед загрузкой, после чего режим BCD будет восстановлен на основании переменной времени выполнения UEFI. Если же удалена переменная времени выполнения UEFI, загрузчик Windows восстановит ее на основании загрузочной переменной UEFI. Соответственно, без специального кода для удаления загрузочной переменной UEFI — что может быть сделано только в режиме EFI Boot Services — отключить Охранника учетных данных в режиме блокировки UEFI не удастся.

Единственный код, который на это способен, размещается в специальном двоичном файле компании Microsoft с именем SecComp.efi. Он должен быть загружен администратором, который затем загружает компьютер с альтернативного устройства на базе EFI и выполняет код вручную (для чего потребуется как ключ восстановления BitLocker, так и физический доступ) или же изменяет BCD (для чего необходим ключ восстановления BitLocker). При перезагрузке SecComp.efi потребует подтверждения в режиме UEFI (что может быть сделано только физическим пользователем).

Политики аутентификации и защищенный Kerberos

Использование модели безопасности «безопасно, если взлом уже не произошел перед входом или выполнен физическим администратором» определенно является шагом вперед перед традиционной моделью безопасности, не использующей Охранника учетных данных. Тем не менее некоторым предприятиям и организациям может потребоваться более сильная гарантия безопасности: что даже скомпрометированная машина не может использоваться для подделки или воспроизведения учетных данных пользователя, а если учетные данные пользователя были скомпрометированы, они не могут быть использованы за пределами некоторых систем. Благодаря технологии Server 2016, называемой «политиками аутентификации» (Authentication Policies), и защищенному Kerberos, Охранник учетных данных может работать в режиме повышенной безопасности.

В этом режиме защищенное ядро VTL 1 получает специальный ключ-идентификатор машины с использованием TPM (также возможно использование файла на диске, но это создает потенциальный дефект безопасности). Затем ключ используется для генерирования ключа TGT машины во время исходной операции присоединения к домену (очевидно, важно проследить за тем, чтобы машина находилась в надежном состоянии в процессе подготовки), и этот ключ TGT отправляется KDC. После настройки, когда пользователь выполняет вход со своими учетными данными, они объединяются с учетными данными машины (к которым имеет доступ только изолированная LSA), образуя ключ проверки происхождения (proof-of-origin). Тогда KDC отвечает NTOWF, пользовательским TGT и его ключом после шифрования с ключом проверки происхождения. В этом режиме предоставляются две гарантии безопасности.

- **Пользователь выполняет аутентификацию с известной машины.** Если пользователь (али атакующий) обладает исходными учетными данными и попытается использовать их на другой машине, учетные данные машины на базе TPM будут другими.
- **Ответ NTLM/билет пользователя поступает от изолированной LSA и не был сгенерирован вручную в LSASS.** Тем самым гарантируется, что Охранник учетных данных активен на машине — даже если физический пользователь отключил его каким-то образом.

К сожалению, если машина будет скомпрометирована таким образом, что ответ KDC, зашифрованный с применением ключа проверки происхождения и содержащий пользовательский TGT и его ключ, будет перехвачен, становится возможным его сохранение и использование для запроса билетов, зашифрованных с сеансовым ключом, от изолированной LSA. Затем данные могут быть отправлены на файловый сервер (к примеру) для обращения к нему — до того, как перезагрузка уничтожит сеансовый ключ. Соответственно, в системах с Охранником учетных данных рекомендуется выполнять перезагрузку при каждом выходе пользователя. В противном случае атакующий сможет выдавать действительные билеты даже при отсутствии пользователя.

Будущие усовершенствования

Как обсуждалось в главе 2 и главе 3, защищенное ядро VTL 1 в настоящее время совершенствуется и дополняется поддержкой специализированных классов оборудования PCI и USB, взаимодействие с которыми может осуществляться исключительно через гипервизор и код VTL 1 с использованием SDF (Secure Device Framework). В сочетании с BioIso.exe и FsIso.exe — новыми трастлетами для безопасного получения биометрических данных и видеокадров (с веб-камеры) компонент на базе ядра VTL 0 не сможет перехватить содержимое попыток аутентификации Windows Hello (которое рассматривается как безопасное по сравнению с текстовым паролем, но формально может быть перехвачено при помощи специально написанного драйвера). После выдачи учетных данных Windows Hello они на

аппаратном уровне гарантированно не будут доступны для VTL 0. В этом режиме необходимость участия LSASS в аутентификации Windows Hello отпадает. Изолированная LSA будет получать учетные данные прямо от изолированных служб, поставляющих биометрические данные или кадры.

ПРИМЕЧАНИЕ SDF (Secure Driver Framework) — аналог WDF для драйверов VTL 1. Этот фреймворк не предназначен для открытого распространения; Microsoft предоставляет его только партнерским компаниям для создания драйверов VTL 1.

Device Guard

Если Охранник учетных данных защищает учетные данные пользователей, Device Guard решает совершенно иную задачу: защиту машины пользователя от разного рода программных и аппаратных атак. Device Guard использует сервисы Windows Code Integrity, такие как KMCS (Kernel-Mode Code Signing) и UMCI (User-Mode Code Integrity), и укрепляет их при помощи HVCI (HyperVisor Code Integrity). (За дополнительной информацией о Code Integrity обращайтесь к главе 8 части 2.)

Механизм Device Guard допускает полную настройку благодаря CCI (Custom Code Integrity) и политикам цифровой подписи, которые определяются администратором организации и обеспечиваются механизмом защищенной загрузки. Эти политики, описанные в главе 8, основаны на списках включения/исключения, основанных на криптографически достоверной информации (такой, как хеши SHA-2 или выдаваемых сертификатов) вместо путей к файлам или имен файлов, как в политиках AppLocker (см. раздел «AppLocker» далее в этой главе).

Мы не будем описывать здесь различные способы определения и настройки политик Code Integrity; вместо этого мы покажем, как Device Guard обеспечивает соблюдение этих политик посредством следующих гарантий.

- **Если включена цифровая подпись кода режима ядра, то загружаться может только подписанный код независимо от компрометации самого режима ядра.** Поскольку процесс загрузки ядра оповещает защищенное ядро на уровне VTL 1 при загрузке драйвера, то успешная загрузка происходит только при подтверждении подписи HVCI.
- **Если включена цифровая подпись кода режима ядра, то подписанный код не может изменяться после загрузки — даже самим ядром.** Это объясняется тем, что страницы с исполняемым кодом будут помечаться как доступные только для чтения при помощи механизма SLAT (Second Level Address Translation) гипервизора (эта тема более подробно рассматривается в главе 8 части 2).
- **Если включена цифровая подпись кода режима ядра, то динамическое размещение кода в памяти запрещается (повторение первых двух пунктов).** Дело в том, что ядро не дает возможности выделения исполняемых записей в таблице страниц SLAT, даже несмотря на то, что таблицы страниц ядра сами могут помечать такой код как исполняемый.

- **Если включена цифровая подпись кода режима ядра, код времени выполнения UEFI не может модифицироваться даже другим кодом времени выполнения UEFI или самим ядром.** Кроме того, защищенное ядро должно уже проверить, что этот код был подписан на момент загрузки. (От этого зависит работа Device Guard.) Кроме того, данные времени выполнения UEFI не могут быть исполняемыми. Для этого система читает весь код и данные времени выполнения UEFI, проверяет наличие правильных разрешений и дублирует их в записях таблиц страниц SLAT, которые защищаются в VTL 1.
- **Если включена цифровая подпись кода режима ядра, выполняться может только подписанный код режима ядра (кольцо 0).** Снова может показаться, что это повторение первых трех пунктов, но представьте подписанный код кольца 3. Такой код действителен с точки зрения UMCI и прошел авторизацию как исполняемый код в записях таблиц страниц SLAT. Защищенное ядро использует функциональность MBEC (Mode-Based Execution Control), если она реализована на аппаратном уровне — при этом SLAT дополняется битом разрешения исполнения пользовательского режима/режима ядра или программной эмуляцией этой функциональности в гипервизоре, называемой RUM (Restricted User Mode).
- **Если включена цифровая подпись кода пользовательского режима, загружаться могут только подписанные образы пользовательского режима.** Это означает, что подписаны должны быть все файлы исполняемых процессов (.exe), а также загружаемые ими библиотеки (.dll).
- **Если включена цифровая подпись кода пользовательского режима, ядро не разрешает приложениям пользовательского режима разрешать запись в существующие исполняемые страницы с кодом.** Очевидно, код пользовательского режима не может выделять исполняемую память или изменять существующую память без разрешения ядра. Это означает, что ядро может устанавливать собственные правила контроля. Но даже в случае компрометации ядра SLAT следит за тем, чтобы страницы пользовательского режима не могли стать исполняемыми без ведома и одобрения защищенного ядра и чтобы запись в такие исполняемые страницы была невозможна.
- **Если включена цифровая подпись кода пользовательского режима, а политика подписи требует жестких гарантий, динамическое размещение кода в памяти запрещено.** Здесь проявляется важное отличие от режима ядра: по умолчанию подписанному коду пользовательского режима разрешено выделение дополнительной исполняемой памяти для поддержки JIT-сценариев, если только в сертификате приложения не присутствует специальный объект EKU (Enhanced Key Usage), дающий разрешение на динамическое генерирование кода. В настоящее время этот объект EKU присутствует в образе NGEN.EXE (.NET Native Image Generation), который разрешает исполняемым файлам .NET, содержащим только IL-код, функционировать даже в этом режиме.
- **Если включен ограниченный режим языка PowerShell, то все сценарии PowerShell, которые используют динамические типы, отражение или другие**

средства языка, разрешающие исполнение произвольного кода и/или маршалинг к функциям Windows/.NET API, также должны быть подписаны. Тем самым предотвращается выход из ограниченного режима вредоносных сценариев PowerShell.

Записи таблиц страниц SLAT защищены на уровне VTL 1 и содержат «эталонные» разрешения для заданной страницы памяти. Блокируя доступ к биту разрешения исполнения по мере необходимости и/или к биту разрешения записи для существующих исполняемых страниц, Device Guard перемещает весь контроль за подписыванием кода на уровень VTL 1 (библиотека SKCI.DLL — Secure Kernel Code Integrity).

Кроме того, даже при отсутствии настроенной поддержки на машине, при включенном Охраннике учетных данных Device Guard работает в третьем режиме, следя за тем, чтобы все трастлеты имели особую подпись Microsoft с сертификатом, включающим EKU изолированного пользовательского режима. В противном случае атакующий с привилегиями кольца 0 может атаковать обычный механизм KMCS и загрузить вредоносный трастлет для атаки компонента изолированной LSA. Кроме того, все меры контроля подписи кода пользовательского режима активны для трастлетов, выполняющихся в режиме жестких гарантий.

Наконец, важно понимать, что для оптимизации быстродействия механизм HVCI не проводит повторную аутентификацию каждой отдельной страницы при выходе системы из гибернации (спящий режим S4). В некоторых случаях данные сертификатов могут быть недоступны. Даже в таком случае данные SLAT должны быть воссозданы; это означает, что записи таблицы страниц SLAT хранятся в самом файле гибернации. Следовательно, гипервизор должен проследить за тем, чтобы файл гибернации не был никоим образом изменен. Для этого файл гибернации шифруется с ключом локальной машины, хранящимся в TPM.

К сожалению, без TPM этот ключ приходится хранить в переменной времени выполнения UEFI, что позволит локальному атакующему расшифровать файл гибернации, внести изменения и снова зашифровать его.

Защита объектов

Суть управления избирательным доступом и проверки безопасности заключается в защите объектов и в регистрации доступа. В число защищаемых в Windows объектов входят файлы, устройства, почтовые ящики, каналы (именованные и безымянные), задания, процессы, потоки, события, ключевые события, пары событий, мьютексы, семафоры, общие разделы памяти, порты завершения ввода/вывода, LPC-порты, таймеры ожиданий, маркеры доступа, тома, станции окон, рабочие столы, сетевые ресурсы, службы, разделы реестра, принтеры, объекты Active Directory и т. д. — теоретически все, что управляется диспетчером объектов исполняющей системы. На практике те объекты, которые не открыты пользовательскому режиму (например, объекты драйверов), обычно не защищены. Код режима ядра считается надежным и обычно использует интерфейсы к диспетчеру объектов, не

выполняющие проверки доступа. Поскольку системные ресурсы, открытые пользовательскому режиму (и поэтому требующие проверки безопасности), реализованы как объекты в режиме ядра, диспетчер объектов Windows играет ключевую роль в обеспечении безопасности объектов.

Для просмотра данных защиты объектов можно воспользоваться программой Winobj из пакета Sysinternals (рис. 7.5). На рис. 7.6 изображена страница свойств Security объекта раздела в сеансе пользователя. Хотя наиболее часто с защитой объектов ассоциируются такие ресурсы, как файлы, Windows использует аналогичную модель и такой же механизм безопасности, как и для файлов в файловой системе и для объектов исполняющей системы. Что же касается вопросов управления доступом, то объекты исполняющей системы отличаются от файлов только в методах доступа, поддерживаемых каждым типом объектов.

Как вы увидите далее, то, что показано на рис. 7.6, на самом деле является принадлежащим объекту списком управления избирательным доступом — discretionary access control list, или DACL. Списки DACL будут подробно рассмотрены в следующем разделе.

Вы можете воспользоваться Process Explorer для просмотра свойств безопасности объектов, сделав двойной щелчок на дескрипторе на нижней панели (настроенной для просмотра дескрипторов). У этого способа есть дополнительное преимущество: при этом выводятся неименованные объекты. Страница свойств одинакова в обоих программах, так как сама страница предоставляется Windows.

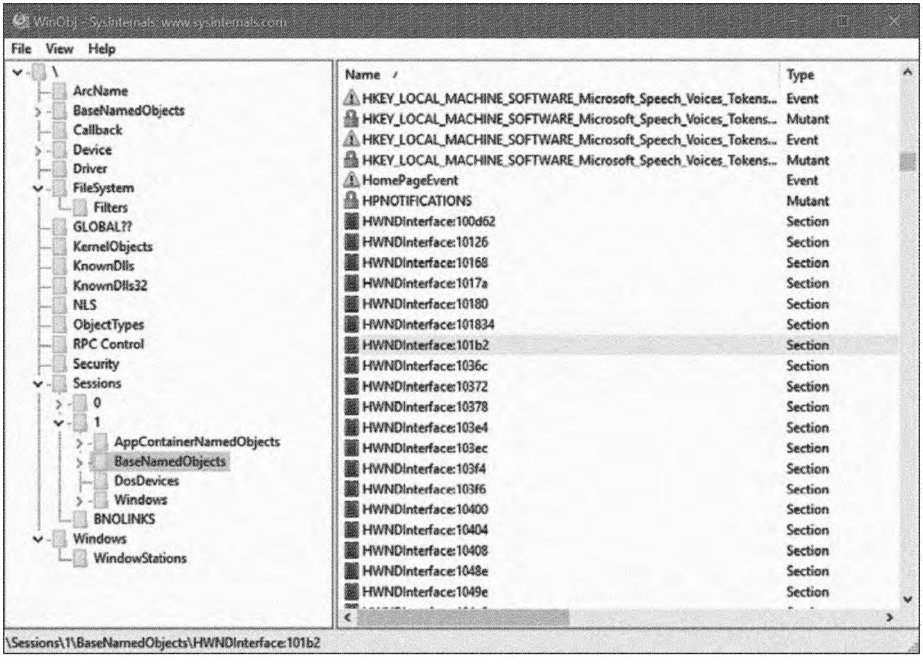


Рис. 7.5. WinObj с выделенным объектом раздела

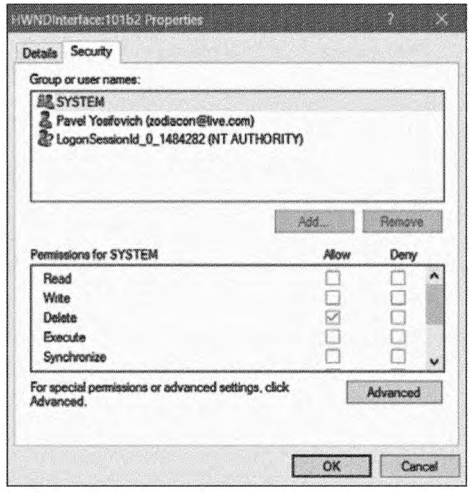


Рис. 7.6. Объект исполняющей системы и его дескриптор безопасности в программе Winobj

Чтобы контролировать возможность работы с объектом, система безопасности должна сначала убедиться в идентичности каждого пользователя. Такая необходимость гарантии пользовательской идентичности является причиной того, что Windows требует аутентифицированного входа в систему перед получением доступа к любым системным ресурсам. Когда процесс запрашивает дескриптор объекта, диспетчер объектов и система безопасности используют идентификатор безопасности запрашивающего и дескриптор безопасности объекта, чтобы определить, может ли запрашивающему быть присвоен дескриптор, предоставляющий процессу желаемый доступ к объекту.

Как станет ясно при дальнейшем рассмотрении вопроса, поток может получить другой контекст безопасности, нежели его процесс. Этот механизм называется *заимствованием прав* (impersonation), и когда поток заимствует чьи-либо права, механизм проверки безопасности использует контекст безопасности потока, а не того процесса, которому принадлежит поток. Если поток ничьих прав не заимствует, проверка безопасности возвращается обратно к использованию контекста безопасности того процесса, который владеет потоком. Важно иметь в виду, что все потоки в процессе используют общую таблицу дескрипторов, поэтому, когда поток открывает объект, даже если он заимствует чьи-либо права, доступ к объекту имеют все потоки процесса.

Иногда проверки идентичности пользователя недостаточно системе для предоставления доступа к ресурсу, который должен быть доступен согласно учетной записи. Логично полагать, что есть четкое различие между службой, выполняемой под учетной записью Alice и неизвестным приложением, загруженным пользователем Alice через интернет. Windows достигает подобной изоляции в рамках работы одного и того же пользователя с помощью механизма целостности Windows (Windows

integrity mechanism), в котором реализуются уровни целостности. Механизм целостности Windows используется с повышениями привилегий, реализуемыми системой управления учетными записями пользователей — User Account Control (UAC), в изоляции привилегий пользовательского интерфейса — User Interface Privilege Isolation (UIPI) и контейнерах AppContainer (см. далее в этой главе).

Проверки прав доступа

Модель безопасности Windows требует, чтобы перед открытием объекта поток указал, какого типа действия он хочет совершить над объектом. Для выполнения проверок прав доступа, основанных на желаемом потоком доступе, диспетчер объектов вызывает SRM, и если доступ предоставляется, процессу потока назначается дескриптор, с которым поток (или другие потоки процесса) может выполнять дальнейшие операции над объектом.

Одним из событий, заставляющих диспетчер объектов выполнять проверку безопасности доступа, является открытие процессом существующего объекта с использованием его имени. Когда объект открывается по имени, диспетчер объектов выполняет поиск указанного объекта в пространстве имен диспетчера объектов. Если объект не будет найден во вторичном пространстве имен (например, в пространстве имен диспетчера конфигурации реестра или в пространстве имен драйверов файловой системы), диспетчер объектов сразу же после обнаружения объекта вызывает внутреннюю функцию `ObpCreateHandle`. Как следует из имени данной функции, `ObpCreateHandle` создает запись в таблице дескрипторов процесса, которая связывается с объектом. Сначала функция `ObpCreateHandle` вызывает функцию `ObpGrantAccess`, чтобы посмотреть, имеет ли поток разрешение на доступ к объекту, если он такое разрешение имеет, функция `ObpCreateHandle` вызывает функцию исполняющей системы `ExCreateHandle` для создания записи в таблице дескрипторов процесса. Для инициализации проверки безопасности доступа функция `ObpGrantAccess` вызывает функцию `ObCheckObjectAccess`.

Функция `ObpGrantAccess` передает функции `ObCheckObjectAccess` мандат безопасности потока, открывающего объект, типы доступа к объекту, запрошенному потоком (чтение, запись, удаление и т. д.), и указатель на объект. Сначала функция `ObCheckObjectAccess` блокирует дескриптор безопасности объекта и контекст безопасности потока. Блокировка дескриптора безопасности объекта не позволяет другим потокам системы изменять параметры безопасности объекта в ходе проверки прав доступа. Блокировка контекста безопасности потока не позволяет другим потокам (из этого же процесса или из других процессов) изменять идентификационные данные о безопасности потока в ходе проверки прав доступа. Затем функция `ObCheckObjectAccess` вызывает метод безопасности объекта, чтобы получить настройки безопасности этого объекта (за информацией о методах объектов обращайтесь к главе 8). Вызов метода безопасности может задействовать функцию в других компонентах исполняющей системы. Тем не менее многие объекты исполняющей системы полагаются на исходную поддержку диспетчера безопасности системы.

Когда компонент исполняющей системы, определяющий объект, не хочет переопределять политику безопасности SRM по умолчанию, он делает пометку, что тип объекта имеет исходную настройку безопасности. При вызове метода безопасности объекта SRM сначала проверяет, не установлены ли для объекта исходные настройки безопасности. Объект с исходными настройками безопасности сохраняет информацию о своей безопасности в своем заголовке, и его методом безопасности является `SeDefaultObjectMethod`. Объект, который не зависит от исходных настроек безопасности, должен управлять своей собственной информацией о безопасности и предоставлять особый метод безопасности. К объектам, зависящим от исходных настроек безопасности, относятся мьютексы, события и семафоры. В качестве примера объекта, переписывающего настройки безопасности по умолчанию, можно привести файловый объект. Диспетчер ввода/вывода, который определяет тип файлового объекта, имеет драйвер файловой системы, в котором находится файл, управляющий безопасностью его файлов (или отказывающийся от этого управления). Таким образом, когда система запрашивает данные безопасности файлового объекта, представляющего файл на томе NTFS, метод безопасности файлового объекта диспетчера ввода/вывода извлекает информацию о настройках безопасности файла, используя драйвер файловой системы. Но следует заметить, что функция `ObCheckObjectAccess` не выполняется, когда файлы открыты, поскольку они находятся во вторичном пространстве имен. Система вызывает метод безопасности файлового объекта только тогда, когда поток явным образом запрашивает информацию о безопасности файла или устанавливает для него настройки безопасности (например, с помощью Windows-функции `SetFileSecurity` или `GetFileSecurity`).

После получения информации о безопасности объекта функция `ObCheckObjectAccess` вызывает SRM-функцию `SeAccessCheck`. Функция `SeAccessCheck` относится к функциям, формирующим основу модели безопасности Windows. Среди прочих входных параметров функция `SeAccessCheck` принимает информацию о безопасности объекта, идентификационные данные о безопасности потока в том виде, в котором они извлечены функцией `ObCheckObjectAccess`, и запрашиваемый потоком тип доступа. Функция `SeAccessCheck` возвращает значение `true` или `false` в зависимости от того, предоставлен или нет потоку доступ к запрашиваемому объекту.

Рассмотрим пример: предположим, поток хочет знать о выходе из некоторого процесса (или завершении его по другим причинам). Он должен получить дескриптор целевого процесса, вызывая API-функцию `OpenProcess` с передачей двух важных аргументов: уникального идентификатора процесса (будем считать, что он известен или был получен каким-то образом) и маски доступа, определяющей операции, которые поток хочет выполнить с использованием возвращенного дескриптора. Ленивые разработчики могут просто передать маску доступа `PROCESS_ALL_ACCESS`, указав, что процессу должны быть предоставлены все возможные права доступа. Возможен один из двух результатов.

- Если вызывающему потоку могут быть предоставлены все разрешения, он получает действительный дескриптор и может вызвать `WaitForSingleObject`,

чтобы ожидать выхода из процесса. Тем не менее другой поток процесса (возможно, обладающий даже меньшими привилегиями) может использовать тот же дескриптор для выполнения других операций с процессом — например, предварительно завершить его вызовом `TerminateProcess`, потому что дескриптор разрешает все возможные операции с процессом.

- Вызов может завершиться неудачей, если вызывающий поток не обладает достаточными привилегиями для получения всех возможных прав доступа. В таком случае результатом будет недействительный дескриптор, означающий отсутствие прав доступа к процессу. Это печально, потому что потоку было достаточно запросить только маску доступа `SYNCHRONIZE`. Такой запрос обладает куда большими шансами на успех, чем `PROCESS_ALL_ACCESS`.

Отсюда следует простой вывод: поток должен запрашивать ровно те права доступа, которые ему необходимы — не больше и не меньше.

Еще одним событием, заставляющим диспетчер объектов проводить проверку прав доступа, является ссылка процесса на объект с использованием существующего дескриптора. Такие ссылки зачастую проводятся опосредованно, например, когда процесс вызывает API-функцию Windows для работы с объектом и передает ей дескриптор объекта. Например, поток, открывающий файл, может запросить право на чтение файла. Если у потока есть право на такой доступ к объекту, как предписано его контекстом безопасности и настройками безопасности файла, диспетчер объектов создает дескриптор, который представляет файл в таблице дескрипторов того процесса, которому принадлежит поток. Типы доступа предоставляются потокам процесса через дескриптор, который хранится с дескриптором диспетчера объектов.

Впоследствии поток может попытаться провести запись в файл, использовав Windows-функцию `WriteFile`, передав ей в качестве параметра дескриптор файла. Системная сервисная функция `NtWriteFile`, которую функция `WriteFile` вызывает через библиотеку `Ntdll.dll`, использует функцию диспетчера объектов `ObReferenceObjectByHandle` (документированную в WDK), чтобы получить из дескриптора указатель на объект файла. Функция `ObReferenceObjectByHandle` получает в параметре информацию о доступе, который хочет получить к объекту вызывающий процесс. После поиска записи дескриптора в таблице дескрипторов процесса функция `ObReferenceObjectByHandle` сравнивает запрошенный доступ с доступом, предоставленным во время открытия файла. В данном примере функция `ObReferenceObjectByHandle` покажет, что операция записи должна быть отклонена, поскольку вызывающий процесс не получил доступ по записи при открытии файла.

Функции безопасности Windows также позволяют Windows-приложениям определять свои собственные закрытые объекты и вызывать службы SRM (через API-функции пользовательского режима AuthZ, которые будут рассмотрены чуть позже), чтобы задействовать модель безопасности Windows в отношении таких объектов. Многие функции режима ядра, которые диспетчер объектов и другие компоненты исполняющей системы используют для защиты своих собственных объектов, экспортируются в виде API-функций Windows, работающих в пользовательском

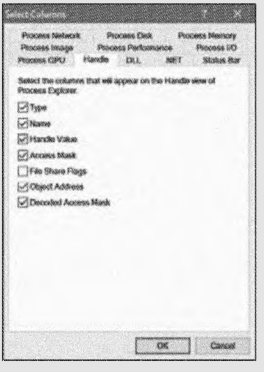
режиме. Аналогом функции `SeAccessCheck` в пользовательском режиме является API-функция AuthZ `AccessCheck`. Таким образом, Windows-приложения могут воспользоваться гибкостью модели безопасности и явным образом интегрироваться с имеющимися в Windows интерфейсами аутентификации и администрирования.

По сути модель безопасности SRM представляет собой уравнение с тремя входными параметрами: идентификационные данные безопасности потока, доступ, который поток желает получить к объекту, и настройки безопасности объекта. На выходе получаем либо «да», либо «нет», что показывает, дает модель безопасности потоку запрошенный доступ или не дает. В следующих разделах входные параметры рассматриваются более подробно, после чего дается описание алгоритма проверки доступа, реализованного в модели.

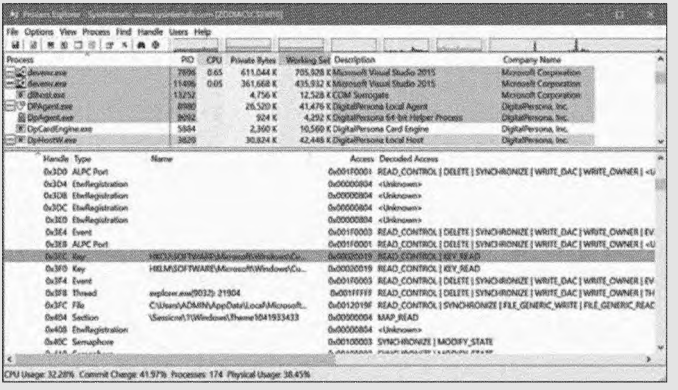
ЭКСПЕРИМЕНТ: ПРОСМОТР МАСОК ДОСТУПА ДЛЯ ДЕСКРИПТОРОВ

Программа Process Explorer может отображать маски доступа, связанные с открытыми дескрипторами. Выполните следующие действия.

1. Откройте Process Explorer.
2. Откройте меню View, выберите команду Lower Pane View и выберите вариант Handles, чтобы на нижней панели выводилась информация о дескрипторах.
3. Щелкните правой кнопкой мыши на столбцах заголовков на нижней панели. Выберите команду Select Columns, чтобы открыть диалоговое окно, изображенное на следующей иллюстрации:



4. Установите флажки Access Mask и Decoded Access Mask (последний присутствует только в версии 16.10 и выше). Щелкните на кнопке OK.
5. Выберите в списке процессов строку Explorer.exe и просмотрите дескрипторы на нижней панели. С каждым дескриптором связана маска доступа с описанием прав, предоставляемых при использовании этого дескриптора. Чтобы вам было проще интерпретировать биты маски доступа, в столбце Decoded Access Mask выводится текстовое представление масок доступа для многих типов объектов:



Обратите внимание: права доступа делятся на обобщенные (например, `READ_CONTROL` и `SYNCHRONIZE`) и конкретные (например, `KEY_READ` и `MODIFY_STATE`). Большинство конкретных прав представляет собой сокращенные версии определений из заголовков Windows (например, `MODIFY_STATE` вместо `EVENT_MODIFY_STATE`, `TERMINATE` вместо `PROCESS_TERMINATE`).

Идентификаторы безопасности

Вместо использования имен (которые могут быть уникальными, а могут и не быть), для идентификации всего, что производит в системе действия, Windows использует идентификаторы безопасности — SID (Security Identifiers). SID-идентификаторы имеются у пользователей, а также у локальных и доменных групп, у локальных компьютеров, доменов, участников доменных групп и служб. SID представляет собой числовое значение переменной длины, состоящее из номера версии SID-структуры, 48-разрядное значение идентификатора полномочий и переменное количество 32-разрядных кодов значений подчиненных полномочий или относительных идентификаторов — RID (Relative Identifier). Значение полномочий идентифицирует агента, выдавшего SID, и этим агентом обычно является локальная система Windows или домен. Значения подчиненных полномочий идентифицируют представителей, имеющих отношение к выдавшему полномочия, а RID-идентификаторы являются просто способом, применяемым в Windows для создания уникальных SID на основе общего базового SID. Из-за большой длины SID-идентификаторов Windows старается сгенерировать внутри каждого SID по-настоящему случайное значение, практически невозможно, чтобы Windows выдала один и тот же SID на машине или домене или где-либо еще дважды.

В текстовом формате SID выводятся с префиксом S, и его различные компоненты отделены друг от друга дефисами:

```
S-1-5-21-1463437245-1224812800-863842198-1128
```

В данном SID номером версии служит цифра 1, значением идентификатора полномочий служит цифра 5 (полномочия безопасности Windows), а затем следуют четыре значения подчиненных полномочий плюс один RID (1128), который составляет

оставшуюся часть SID. Это SID домена, а у локального компьютера домена будет SID с тем же номером версии, значением идентификатора полномочий и количеством значений подчиненных полномочий.

При установке Windows программа установки назначает компьютеру SID машины. Windows назначает SID-идентификаторы локальным учетным записям, имеющимся на компьютере. Каждый SID локальной учетной записи создается на основе исходного компьютерного SID и в конце имеет RID. RID-идентификатор для пользовательских учетных записей и групп начинается с 1000 и становится больше на 1 с каждым новым пользователем или группой. По аналогии с этим Dcpromo.exe (Domain Controller Promote) — утилита, используемая для создания нового домена Windows, — повторно использует SID компьютера, повышаемого до контроллера домена в качестве SID домена, и она заново создает SID для компьютера, если он когда-либо будет понижен в должности. Windows выдает для нового домена SID-идентификаторы учетных записей, основанные на SID-идентификаторе домена с добавлением RID-идентификатора (который снова начинается с 1000 и повышается на 1 для каждого нового пользователя или группы). RID со значением 1028 показывает, что SID является двадцать девятым выданным в домене.

Windows выдает SID-идентификаторы, которые состоят из SID компьютера или домена с предопределенным RID-идентификатором для множества предопределенных учетных записей и групп. Например, RID для учетной записи администратора имеет значение 500, а RID для гостевой учетной записи имеет значение 501. Например, учетная запись локального администратора имеет в своей основе SID компьютера с добавленным к нему RID, который имеет значение 500:

```
S-1-5-21-13124455-12541255-61235125-500
```

Windows также определяет ряд встроенных локальных и доменных SID для представления широко известных групп. Например, SID, идентифицирующий любые учетные записи (за исключение анонимных пользователей), является всеобщим — Everyone SID: S-1-1-0. Другим примером группы, которая может быть представлена SID-идентификатором, является сетевая группа, т. е. группа, представляющая пользователей, зарегистрировавшихся на машине по сети. SID сетевой группы имеет значение S-1-5-2. В представленной ниже табл. 7.2 из документации по Windows SDK показываются некоторые основные, широко известные SID-идентификаторы, их числовые значения и использование. В отличие от пользовательских SID эти SID-идентификаторы являются предопределенными константами и имеют одинаковые значения на каждой системе Windows и домене во всем мире. Таким образом, файл, доступный членам группы Everyone на той системе, где он был создан, также будет доступен группе Everyone на любой другой системе или домене, на которую будет перемещен жесткий диск, на котором он размещается. Разумеется, пользователи на таких системах должны пройти аутентификацию учетной записи на этих системах, перед тем как стать членами группы Everyone.

ПРИМЕЧАНИЕ Список предопределенных SID-идентификаторов можно увидеть в базе знаний Microsoft Knowledge Base в статье 243330, которая находится по адресу *http://support.microsoft.com/kb/243330*.

Таблица 7.2. Широко известные SID-идентификаторы

SID	Группа	Использование
S-1-0-0	Nobody (Никто)	Используется, когда SID неизвестен
S-1-1-0	Everyone (Все)	Группа, включающая всех пользователей за исключением анонимных
S-1-2-0	Local (Локальная)	Пользователи, вошедшие в терминалы, которые локально (физически) подключены к системе
S-1-3-0	Creator Owner ID (ID владельца создателя)	Идентификатор безопасности, который будет заменен идентификатором безопасности того пользователя, который создал новый объект. Этот SID используется в наследуемых ACE-элементах
S-1-3-1	Creator Owner ID (ID владельца создателя)	Идентификатор безопасности, который будет заменен идентификатором безопасности основной группы того пользователя, который создал новый объект. Этот SID используется в наследуемых ACE-элементах
S-1-5-18	Учетная запись Local System	Используется службами
S-1-5-19	Учетная запись Local System	Используется службами
S-1-5-20	Учетная запись Network Service	Используется службами

И наконец, Winlogon создает уникальный SID входа в систему для каждого интерактивного сеанса входа. Обычно SID входа в систему применяется в элементе управления доступом — ACE (Access Control Entry), который разрешает доступ во время сеанса входа клиента в систему. Например, служба Windows для запуска нового сеанса входа в систему может использовать функцию `LogonUser`. Эта функция возвращает маркер доступа, из которого служба может извлечь SID входа в систему. Затем эта служба может использовать SID в ACE (описанная в разделе «Дескрипторы безопасности и управление доступом»), позволяющий сеансу входа клиента в систему получать доступ к интерактивной станции окна и к рабочему столу. SID для сеанса входа в систему имеет значение S-1-5-5-X-Y, где значения X и Y генерируются случайно.

ЭКСПЕРИМЕНТ: ИСПОЛЬЗОВАНИЕ PSGETSID И PROCESS EXPLORER ДЛЯ ПРОСМОТРА SID

SID-представление любой используемой вами учетной записи можно запросто посмотреть, запустив утилиту PsGetSid из набора Sysinternals. Параметры `PsGetSid` позволяют переводить имена машин и пользовательских учетных записей в соответствующие им SID-идентификаторы, и наоборот.

Если запустить `PsGetSid` без параметров, утилита выводит SID, назначенный локальному компьютеру. Пользуясь тем, что у учетной записи Administrator значение RID всегда равно 500, вы можете определить имя, назначенное учетной записи (в тех случаях, когда системный администратор переименовал учетную запись из соображений безопасности), просто передав в качестве аргумента

командной строки для запуска утилиты PsGetSid SID машины, к которому добавлена строка –500.

Для получения SID учетной записи домена введите имя пользователя, поставив в качестве префикса имя домена:

```
c:\>psgetsid redmond\johndoe
```

SID домена можно определить, указав имя домена в качестве аргумента при запуске утилиты PsGetSid:

```
c:\>psgetsid Redmond
```

И наконец, изучив RID своей собственной учетной записи, вы узнаете по крайней мере количество учетных записей безопасности (чтобы узнать его, вычтите 999 из вашего RID), созданных в вашем домене или на вашей локальной машине (в зависимости от того, что вами используется, учетная запись домена или учетная запись локальной машины). Чтобы определить, каким учетным записям были назначены RID-идентификаторы, можно передать SID с RID, который нужно запросить утилите PsGetSid. Если PsGetSid сообщит, что сопоставить SID с каким-нибудь именем учетной записи не удалось и значение RID меньше, чем у вашей учетной записи, вы будете знать, что учетная запись, которой был назначен такой RID, была удалена.

Например, чтобы найти имя, назначенное учетной записи с 28-м RID, передайте PsGetSid SID домена, дополненный суффиксом –1027:

```
c:\>psgetsid S-1-5-21-1787744166-3910675280-2727264193-1027
Account for S-1-5-21-1787744166-3910675280-2727264193-1027:
User: redmond\johndoe
```

Process Explorer в своей вкладке Security также может вывести информацию о SID-идентификаторах учетных записей и групп. В этой вкладке показывается информация о владельце данного процесса и принадлежности учетной записи к группам. Для просмотра этой информации нужно просто сделать двойной щелчок на любом процессе (например, на Explorer.exe) в списке процессов, а затем щелкнуть на вкладке Security. Результат выглядит примерно так:

Информация в поле User содержит легко читаемое имя учетной записи, владеющей этим процессом, а информация в поле SID содержит текущее значение SID-идентификатора. В список Group включена информация обо всех группах, в которые входит данная учетная запись. (Группы будут рассмотрены в данной главе чуть позже.)

Уровни целостности

Как уже ранее упоминалось, уровни целостности могут заменить разграничительный доступ, чтобы провести различия между процессом и объектами, запущенными от имени одного и того же пользователя и находящимися в его владении, предоставив возможность изоляции кода и данных в рамках учетной записи пользователя. Механизм MIC (Mandatory Integrity Control) позволяет SRM располагать более подробной информацией о природе вызывающего процесса путем его ассоциации с уровнем целостности. Он также предоставляет информацию о доверии, необходимую для доступа к объекту путем определения для него уровня целостности.

Для получения уровня целостности маркера можно воспользоваться API-функцией `GetTokenInformation` с перечисляемым значением `TokenIntegrityLevel`. Эти уровни целостности определяются с помощью SID. Хотя уровни целостности могут иметь произвольные значения, в системе используются пять основных уровней, описанных в табл. 7.3.

Таблица 7.3. SID-идентификаторы уровней целостности

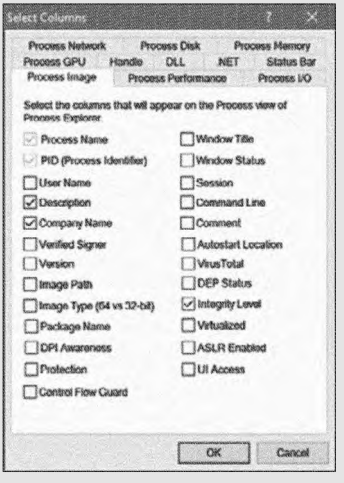
SID	Имя (уровень)	Использование
S-1-16-0x0	Ненадежный (0) (Untrusted)	Используется процессами, запущенными группой Anonymous. Блокирует большинство доступов по записи
S-1-16-0x1000	Низкий (1) (Low)	Используется процессами AppContainer (UWP) и защищенным режимом Internet Explorer. Блокирует доступ по записи к большинству объектов системы (таких, как файлы и разделы реестра)
S-1-16-0x2000	Средний (2) (Medium)	Используется обычными приложениями, запущенными при включенной системе UAC
S-1-16-0x3000	Высокий (3) (High)	Используется административными приложениями, запущенными через повышение уровня полномочий при включенной системе UAC, или обычными приложениями при выключенной системе UAC и при наличии у пользователя прав администратора
S-1-16-0x4000	Системный (4) (System)	Используется службами и другими приложениями системного уровня (например, Wininit, Winlogon, Smss и т. д.)
S-1-16-0x5000	Защищенный (5) (Protected)	В настоящее время не используется по умолчанию. Может устанавливаться только из режима ядра

Еще один уровень целостности, называемый *AppContainer*, используется приложениями UWP. Этот вроде бы отдельный уровень фактически эквивалентен низкому (Low). Маркеры процесса UWP содержат дополнительный атрибут, который показывает, что они выполняются внутри AppContainer (см. раздел «AppContainer»). Для получения этой информации используется API-функция `GetTokenInformation` с перечисляемым значением `TokenIsAppContainer`.

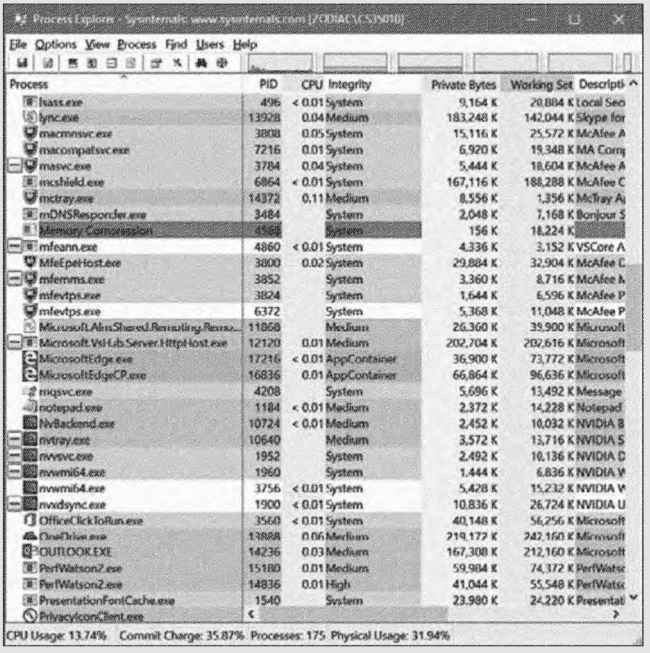
ЭКСПЕРИМЕНТ: ПРОСМОТР УРОВНЕЙ ЦЕЛОСТНОСТИ ПРОЦЕССОВ

Для вывода уровней целостности для процессов на вашей системе можно воспользоваться программой Process Explorer.

1. Запустите браузер Microsoft Edge и Calc.exe (в Windows 10).
2. Откройте окно командной строки в режиме повышенных привилегий.
3. Откройте в обычном режиме окно программы Блокнот (не пользуясь повышенными привилегиями).
4. Теперь откройте Process Explorer, щелкните правой кнопкой мыши на любом из столбцов в списке процессов, а затем выберите команду Select Columns.
5. Перейдите на вкладку Process Image и установите флажок Integrity Level. Диалоговое окно должно выглядеть примерно так:



6. Process Explorer выводит уровень целостности процессов на вашей системе. Для процесса Блокнота Notepad.exe должен быть показан средний уровень (Medium), для процесса Edge (MicrosoftEdge.exe) — уровень AppContainer, а для окна командной строки с повышенной привилегией — высокий уровень (High). Также следует заметить, что службы и системные процессы выполняются на еще более высоком системном уровне целостности (System).



У каждого процесса есть уровень целостности, который представлен в маркере процесса и распространяется в соответствии со следующими правилами.

- Процесс обычно наследует уровень целостности своего родителя (это означает, что окно командной строки с повышенными привилегиями будет порождать другие процессы с повышенными привилегиями).
- Если файловый объект исполняемого образа, которому принадлежит дочерний процесс, а также его родительский процесс имеют средний уровень целостности (Medium), дочерний процесс унаследует более низкий из этих двух уровней.
- Родительский процесс может создать дочерний процесс с явно заданным уровнем целостности со значением, которое ниже его собственного уровня. Для этого он использует функцию `DuplicateTokenEx`, чтобы продублировать его собственный маркер доступа, затем для изменения уровня целостности в новом маркере до нужного используется функция `SetTokenInformation`, после чего с этим новым маркером вызывается функция `CreateProcessAsUser`.

В табл. 7.3 перечислены уровни целостности, связанные с процессами, а как насчет объектов? Объекты также имеют уровень целостности, сохраненный как часть их дескриптора безопасности, в структуре под названием *обязательная метка* (mandatory label).

Для поддержки миграции из предыдущих версий Windows (тех, чьи разделы реестра и файлы не будут включать информацию об уровне целостности), а также для

упрощения работы разработчиков приложений все объекты имеют подразумеваемый уровень целостности, дабы избежать его конкретного указания. Этот подразумеваемый уровень целостности имеет среднее значение (Medium), означающее, что обязательная политика (которая вскоре будет рассмотрена) в отношении объекта будет выполняться на маркерах, связывающих этот объект с уровнем целостности, меньшим Medium.

Когда процесс создает объект без указания уровня целостности, система проверяет уровень целостности в маркере. Для маркеров с уровнем целостности Medium или выше подразумеваемое значение уровня целостности объекта остается средним (Medium). Но когда маркер содержит уровень целостности ниже Medium, объект создается с подразумеваемым уровнем целостности, соответствующим уровню в маркере.

Смысл того, что объекты, создаваемые процессами, имеющими уровни целостности High или System, сами по себе имеют уровень целостности Medium, заключается в том, что пользователи могут выключать и включать систему UAC. Если уровень целостности объекта всегда наследовал бы уровень целостности своего создателя, приложения администратора, выключившего UAC, а затем снова включившего эту систему, потенциально получали бы отказ, поскольку администратор не смог бы изменить какие-либо настройки реестра или файлов, созданных при выполнении на уровне целостности High. У объектов также может быть явно заданный уровень целостности, устанавливаемый системой или создателем объекта. Например, явный уровень целостности задается при создании процессам, потокам, маркерам и заданиям. Причиной присваивания уровня целостности этим объектам является предотвращение со стороны процесса того же пользователя, но запущенного на более низком уровне целостности доступа к таким объектам и изменения их контекста или поведения (например, внедрения DLL или изменения кода).

Таблица 7.4. Обязательные политики объекта

Политика	Присутствует по умолчанию	Описание
Отказ в записи (No-Write-Up)	Неявно на всех объектах	Используется для ограничения доступа к объекту по записи со стороны процессов, имеющих более низкий уровень целостности
Отказ в чтении (No-Read-Up)	Только на объектах процессов	Используется для ограничения доступа к объекту по чтению со стороны процессов, имеющих более низкий уровень целостности. Конкретное использование в отношении объектов процессов создает защиту от утечки информации путем блокирования чтения адресного пространства из внешнего процесса
Отказ в выполнении (No-Execute-Up)	Только на двоичных реализациях COM-классов	Используется для ограничения доступа к объекту по выполнению со стороны процессов, имеющих более низкий уровень целостности. В частности, COM-классы используются для ограничения прав на запуск и активацию COM-класса

Кроме уровней целостности у объектов есть также обязательная политика, определяющая действующий уровень защиты, применяемый на основе проверки уровня целостности. Три возможных типа такой политики показаны в табл. 7.4. Уровень целостности и обязательная политика хранятся вместе в одном и том же ACE.

ЭКСПЕРИМЕНТ: ПРОСМОТР УРОВНЯ ЦЕЛОСТНОСТИ ОБЪЕКТОВ

Для вывода уровня целостности объектов в системе (таких, как файлы, процессы и разделы реестра) можно воспользоваться утилитой AccessChk из пакета Sysinternals. Следующий эксперимент демонстрирует назначение каталога LocalLow в Windows.

1. Перейдите в окне командной строки в каталог C:\Users\<UserName>\, где <UserName> — имя пользователя.
2. Попробуйте запустить утилиту AccessChk из папки AppData:

```
C:\Users\имя_пользователя> accesschk -v appdata
```

3. Обратите внимание на разницу между Local и LocalLow в выводимой информации, которая должна быть похожа на следующую:

```
C:\Users\UserName\AppData\Local
  Medium Mandatory Level (Default) [No-Write-Up]
  [...]
C:\Users\UserName\AppData\LocalLow
  Low Mandatory Level [No-Write-Up]
  [...]
C:\Users\UserName\AppData\Roaming
  Medium Mandatory Level (Default) [No-Write-Up]
  [...]
```

4. Заметьте, что у каталога LocalLow имеется уровень целостности, установленный в Low, а у каталогов Local и Roaming уровень целостности имеет значение Medium (Default). Слово «default» означает, что система использует неявный уровень целостности.
5. Чтобы утилита AccessChk выводила только подразумеваемые уровни целостности, ей можно передать флаг -e. Если ее снова запустить в отношении папки AppData, вы заметите, что будет выведена только информация о папке LocalLow.

Ключи -o (объект), -k (раздел реестра) и -p (процесс) позволяют указывать объекты, отличные от файла или каталога.

Маркеры

Для идентификации контекста безопасности процесса или потока SRM использует объект под названием *маркер* (или *маркер доступа*). Контекст безопасности состоит из информации, описывающей учетную запись, группы и привилегии,

связанные с процессом или потоком. Маркеры также включают такую информацию, как ID сеанса, уровень целостности и состояние виртуализации UAC. (Привилегии и механизм виртуализации UAC будут рассмотрены в данной главе чуть позже.)

При выполнении процесса входа в систему (рассматриваемого в конце данной главы) LSASS создает исходный маркер для представления пользователя, входящего в систему. Затем он определяет, относится ли пользователь к группе, имеющей высокие полномочия, или обладает ли он высокими привилегиями. На данном этапе проверка принадлежности к группам происходит в следующем порядке:

- Built-In Administrators (Встроенная группа администраторов);
- Certificate Administrators (Администраторы сертификатов);
- Domain Administrators (Администраторы доменов);
- Enterprise Administrators (Администраторы предприятий);
- Policy Administrators (Администраторы политик);
- Schema Administrators (Администраторы схем);
- Domain Controllers (Контроллеры доменов);
- Enterprise Read-Only Domain Controllers (Контроллеры принадлежащих предприятию доменов, предназначенных только для чтения);
- Read-Only Domain Controllers (Контроллеры доменов, предназначенных только для чтения);
- Account Operators (Операторы учетных записей);
- Backup Operators (Операторы по созданию резервных копий);
- Cryptographic Operators (Операторы криптографических систем);
- Network Configuration Operators (Операторы сетевой конфигурации);
- Print Operators (Операторы вывода на печать);
- System Operators (Системные операторы);
- RAS Servers (Серверы служб удаленного доступа — RAS);
- Power Users (Пользователи с повышенными привилегиями);
- Pre-Windows 2000 Compatible Access (Доступ в режиме совместимости с системами, предшествовавшими Windows 2000).

Многие из перечисленных групп используются только на системах с объединенными доменами и не дают пользователям локальных административных прав напрямую. Вместо этого они позволяют пользователям изменять настройки, распространяющиеся на весь домен.

Проверяется также наличие следующих привилегий:

- `SeBackupPrivilege` (на создание резервной копии);
- `SeCreateTokenPrivilege` (на создание маркера);
- `SeDebugPrivilege` (на отладку);
- `SeImpersonatePrivilege` (на работу от имени других пользователей);
- `SeLabelPrivilege` (на создание меток);
- `SeLoadDriverPrivilege` (на загрузку драйверов);
- `SeRestorePrivilege` (на восстановление системы);
- `SeTakeOwnershipPrivilege` (на смену владельца);
- `SeTcbPrivilege` (на работу с блоком управления потоком).

Более подробно эти привилегии рассмотрены в одном из следующих разделов.

Если установлена принадлежность к одной и более групп или наличие одной и более привилегий, LSASS создает для пользователя маркер с ограничениями (также называемый *фильтрованным административным маркером*) и создает сеанс входа в систему. Обычный маркер пользователя прикрепляется к исходному процессу или процессам, запускаемым Winlogon (по умолчанию Userinit.exe).

ПРИМЕЧАНИЕ Если механизм UAC выключен, администраторы работают с маркером, включающим их членство в административных группах и привилегии.

Поскольку по умолчанию дочерние процессы наследуют копию маркера своего создателя, все процессы в сеансе пользователя запускаются под одним и тем же маркером. Маркер можно также сгенерировать, использовав Windows-функцию `LogonUser`. Затем этот маркер можно использовать для создания процесса, выполняющегося в контексте безопасности пользователя, вошедшего в систему посредством функции `LogonUser`, передав маркер Windows-функции `CreateProcessAsUser`. Функция `CreateProcessWithLogon` сочетает все это в одном вызове, именно так команда `Runas` запускает процессы под альтернативными маркерами.

Маркеры варьируются по длине, поскольку у разных учетных записей пользователей имеются разные наборы привилегий и связанные с ними учетные записи групп. Но все маркеры содержат одинаковые типы информации. Наиболее важное содержимое маркера представлено на рис. 7.7.

Для определения доступности объектов и вида защищаемых операций имеющиеся в Windows механизмы безопасности используют два компонента. Один компонент включает в себя принадлежащий маркеру SID учетной записи пользователя и поля SID групп. Монитор безопасности — SRM (Security Reference Monitor) использует SID-идентификаторы для определения, может ли процесс или поток получить запрашиваемый доступ к защищаемому объекту, например к NTFS-файлу.

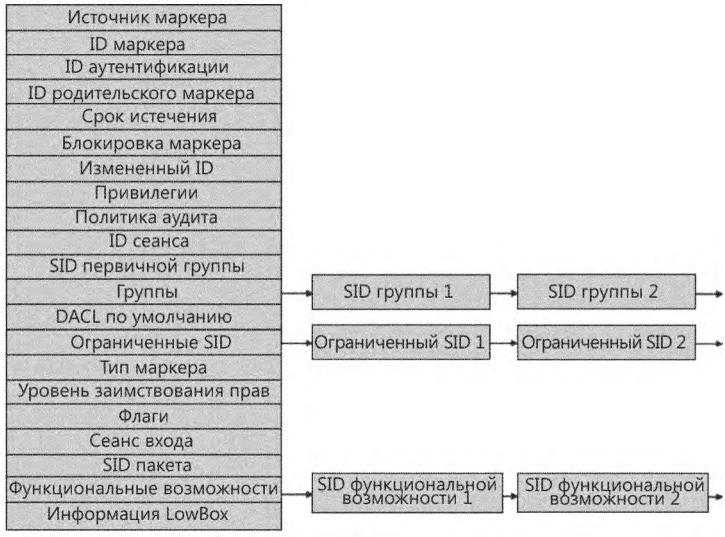


Рис. 7.7. Маркеры доступа

Присутствующие в маркере групповые SID-идентификаторы показывают, в какие группы входит учетная запись пользователя. Например, серверное приложение может запретить конкретные группы для ограничения полномочий маркера, когда серверное приложение выполняет действия, запрещенные клиентам. Запрет группы имеет примерно такой же эффект, как если бы группа не была представлена в маркере. (См. раздел «Ограниченные маркеры». Запрещенные SID используются в контексте проверок доступа безопасности, описанных в разделе «Определение уровня доступа» этой главы.) Групповые SID-идентификаторы могут также включать специальный SID, содержащий уровень целостности процесса или потока. SRM использует еще одно поле в маркере, дающее описание обязательной политики целостности, чтобы выполнять рассматриваемую далее обязательную проверку целостности.

Второй компонент маркера, который определяет, что может поток или процесс с этим маркером делать, называется *массивом привилегий*. Он представляет собой список прав, связанных с маркером. В качестве примера привилегии можно назвать право процесса или потока, связанного с маркером, выключать компьютер. Более подробное рассмотрение привилегий дается в этой главе чуть позже.

Присутствующее в маркере исходное поле основной группы и исходный список DACL (Discretionary Access Control List) являются теми атрибутами безопасности, которые Windows применяет к объектам, создаваемым процессом или потоком при использовании данного маркера. За счет включения в маркеры информации безопасности Windows упрощает для процесса или потока создание объектов со стандартными атрибутами безопасности, поскольку процесс или поток не нуждается в запросе отдельной информации безопасности для каждого создаваемого им объекта.

Каждый тип маркера устанавливает различие между основным маркером, идентифицирующим контекст безопасности процесса, и маркером заимствования (который является таким типом маркера, который потоки используют для временного заимствования другого контекста безопасности, обычно для другого пользователя). Маркеры заимствования содержат уровень заимствования, который обозначает тот тип заимствования, который активен в маркере (заимствование прав будет рассмотрено в этой главе чуть позже).

Маркер также включает обязательную политику для процесса или потока, которая определяет поведение обязательного контроля целостности (MIC) при обработке этого маркера. Есть две политики:

- `TOKEN_MANDATORY_NO_WRITE_UP`, которая включается по умолчанию, устанавливает для маркера политику No-Write-Up (отказ в записи), указывающую, что данный процесс или поток не сможет иметь допуск по записи к объектам с более высоким уровнем целостности;
- `TOKEN_MANDATORY_NEW_PROCESS_MIN`, которая также включается по умолчанию, указывает, что SRM будет искать уровень целостности исполняемого образа при запуске дочернего процесса и вычислять минимальный уровень целостности родительского процесса и уровень целостности файлового объекта в качестве уровня целостности дочернего процесса.

Флаги маркеров включают параметры, которые определяют поведение конкретных механизмов UAC и UIPI, например доступа к виртуализации и к пользовательскому интерфейсу. Эти механизмы будут рассмотрены в этой главе чуть позже.

Каждый маркер может также содержать атрибуты, назначаемые службой идентификации приложений — Application Identification (частью AppLocker), когда правила AppLocker уже определены. AppLocker и использование им атрибутов в маркере доступа рассматриваются в этой главе чуть позже.

Все остальные поля в маркере предоставляют информацию. Поле источника маркера (source) содержит краткое текстовое описание создателя маркера. Программы, желающие знать о происхождении маркера, используют источник маркера, чтобы отличить друг от друга такие источники, как диспетчер сеансов — Windows Session Manager, сетевой файловый сервер или сервер RPC (Remote Procedure Call). Идентификатор маркера является локально уникальным идентификатором — LUID (Locally Unique Identifier), — который SRM присваивает маркеру при его создании. Исполняющая система Windows обслуживает свой LUID (executive LUID), монотонно возрастающий счетчик, который используется ею для назначения уникального цифрового идентификатора каждому маркеру. LUID гарантирует уникальность только до тех пор, пока система не будет выключена.

Еще одной разновидностью LUID является ID аутентификации. Создатель маркера назначает ему ID аутентификации при вызове функции `LsaLogonUser`. Если создатель не указал LUID, LSASS получает LUID из LUID исполняющей системы. LSASS копирует ID аутентификации для всех маркеров, происходящих от ис-

ходного маркера входа в систему. Программа может получить ID аутентификации маркера, чтобы посмотреть, не принадлежит ли маркер к тому же сеансу входа в систему, что и другие маркеры, проанализированные программой.

При каждом изменении характеристик маркера ID модификации обновляется за счет LUID исполняющей системы. Приложение может проверять ID модификации для обнаружения изменений в контексте безопасности со времени последнего использования контекста.

Маркеры содержат поле истечения срока действия, которое может использоваться приложениями, выполняющими свои собственные мероприятия безопасности по отклонению маркера после определенного количества времени. Но сама система Windows не контролирует время истечения срока действия маркера.

ПРИМЕЧАНИЕ Для обеспечения системной безопасности поля маркера неизменяемы (потому что они находятся в памяти ядра). За исключением полей, которые могут изменяться конкретными системными вызовами для модификации некоторых атрибутов маркеров (в предположении, что вызывающая сторона обладает необходимыми правами доступа к объекту маркера), такие данные, как привилегии и SID в маркере, не могут изменяться из пользовательского режима.

ЭКСПЕРИМЕНТ: ПРОСМОТР МАРКЕРОВ ДОСТУПА

Команда отладчика ядра `dt _TOKEN` выводит формат внутреннего объекта маркера. Хотя эта структура отличается от структуры маркера пользовательского режима, возвращается функциями безопасности Windows API, их поля совпадают. Дополнительные сведения о маркерах можно найти в документации по Windows SDK.

Так выглядит структура маркера в Windows 10:

```
lkd> dt nt!_token
   +0x000 TokenSource      : _TOKEN_SOURCE
   +0x010 TokenId          : _LUID
   +0x018 AuthenticationId : _LUID
   +0x020 ParentTokenId    : _LUID
   +0x028 ExpirationTime   : _LARGE_INTEGER
   +0x030 TokenLock        : Ptr64 _ERESOURCE
   +0x038 ModifiedId       : _LUID
   +0x040 Privileges       : _SEP_TOKEN_PRIVILEGES
   +0x058 AuditPolicy      : _SEP_AUDIT_POLICY
   +0x078 SessionId        : Uint4B
   +0x07c UserAndGroupCount : Uint4B
   +0x080 RestrictedSidCount : Uint4B
   +0x084 VariableLength   : Uint4B
   +0x088 DynamicCharged   : Uint4B
   +0x08c DynamicAvailable : Uint4B
   +0x090 DefaultOwnerIndex : Uint4B
   +0x098 UserAndGroups    : Ptr64 _SID_AND_ATTRIBUTES
   +0x0a0 RestrictedSids   : Ptr64 _SID_AND_ATTRIBUTES
   +0x0a8 PrimaryGroup     : Ptr64 Void
```

```
   +0x0b0 DynamicPart      : Ptr64 Uint4B
   +0x0b8 DefaultDacl      : Ptr64 _ACL
   +0x0c0 TokenType        : _TOKEN_TYPE
   +0x0c4 ImpersonationLevel : _SECURITY_IMPERSONATION_LEVEL
   +0x0c8 TokenFlags       : Uint4B
   +0x0cc TokenInUse       : UChar
   +0x0d0 IntegrityLevelIndex : Uint4B
   +0x0d4 MandatoryPolicy  : Uint4B
   +0x0d8 LogonSession     : Ptr64 _SEP_LOGON_SESSION_REFERENCES
   +0x0e0 OriginatingLogonSession : _LUID
   +0x0e8 SidHash          : _SID_AND_ATTRIBUTES_HASH
   +0x1f8 RestrictedSidHash : _SID_AND_ATTRIBUTES_HASH
   +0x308 pSecurityAttributes : Ptr64 _AUTHZBASEP_SECURITY_ATTRIBUTES_INFORMATION
   +0x310 Package          : Ptr64 Void
   +0x318 Capabilities     : Ptr64 _SID_AND_ATTRIBUTES
   +0x320 CapabilityCount  : Uint4B
   +0x328 CapabilitiesHash : _SID_AND_ATTRIBUTES_HASH
   +0x438 LowboxNumberEntry : Ptr64 _SEP_LOWBOX_NUMBER_ENTRY
   +0x440 LowboxHandlesEntry : Ptr64 _SEP_LOWBOX_HANDLES_ENTRY
   +0x448 pClaimAttributes : Ptr64 _AUTHZBASEP_CLAIM_ATTRIBUTES_COLLECTION
   +0x450 TrustLevelSid    : Ptr64 Void
   +0x458 TrustLinkedToken : Ptr64 _TOKEN
   +0x460 IntegrityLevelSidValue : Ptr64 Void
   +0x468 TokenSidValues   : Ptr64 _SEP_SID_VALUES_BLOCK
   +0x470 IndexEntry       : Ptr64 _SEP_LUID_TO_INDEX_MAP_ENTRY
   +0x478 DiagnosticInfo   : Ptr64 _SEP_TOKEN_DIAG_TRACK_ENTRY
   +0x480 SessionObject    : Ptr64 Void
   +0x488 VariablePart     : Uint8B
```

Маркер процесса можно изучить с помощью команды !token. Адрес маркера можно найти в выводе команды !process. Пример выполнения команды для процесса explorer.exe:

```
lkd> !process 0 1 explorer.exe
PROCESS ffffe18304dfd780
    SessionId: 1  Cid: 23e4    Peb: 00c2a000  ParentCid: 2264
    DirBase: 2aa0f6000  ObjectTable: ffffcd82c72fcd80  HandleCount: <Data Not
Accessible>
    Image: explorer.exe
    VadRoot ffffe18303655840 Vads 705 Clone 0 Private 12264. Modified 376410. Locked
18.
    DeviceMap ffffcd82c39bc0d0
    Token                             ffffcd82c72fc060
    ...

PROCESS ffffe1830670a080
    SessionId: 1  Cid: 27b8    Peb: 00950000  ParentCid: 035c
    DirBase: 2cba97000  ObjectTable: ffffcd82c7ccc500  HandleCount: <Data Not
Accessible>
    Image: explorer.exe
    VadRoot ffffe183064e9f60 Vads 1991 Clone 0 Private 19576. Modified 87095. Locked
0.
    DeviceMap ffffcd82c39bc0d0
    Token                             ffffcd82c7cd9060
    ...

lkd> !token ffffcd82c72fc060
_TOKEN 0xffffcd82c72fc060
```

```
TS Session ID: 0x1
User: S-1-5-21-3537846094-3055369412-2967912182-1001
User Groups:
 00 S-1-16-8192
    Attributes - GroupIntegrity GroupIntegrityEnabled
 01 S-1-1-0
    Attributes - Mandatory Default Enabled
 02 S-1-5-114
    Attributes - DenyOnly
 03 S-1-5-21-3537846094-3055369412-2967912182-1004
    Attributes - Mandatory Default Enabled
 04 S-1-5-32-544
    Attributes - DenyOnly
 05 S-1-5-32-578
    Attributes - Mandatory Default Enabled
 06 S-1-5-32-559
    Attributes - Mandatory Default Enabled
 07 S-1-5-32-545
    Attributes - Mandatory Default Enabled
 08 S-1-5-4
    Attributes - Mandatory Default Enabled
 09 S-1-2-1
    Attributes - Mandatory Default Enabled
 10 S-1-5-11
    Attributes - Mandatory Default Enabled
11 S-1-5-15
    Attributes - Mandatory Default Enabled
 12 S-1-11-96-3623454863-58364-18864-2661722203-1597581903-1225312835-2511459453-
1556397606-2735945305-1404291241
    Attributes - Mandatory Default Enabled
 13 S-1-5-113
    Attributes - Mandatory Default Enabled
 14 S-1-5-5-0-1745560
    Attributes - Mandatory Default Enabled LogonId
 15 S-1-2-0
    Attributes - Mandatory Default Enabled
 16 S-1-5-64-36
    Attributes - Mandatory Default Enabled
Primary Group: S-1-5-21-3537846094-3055369412-2967912182-1001
Privs:
 19 0x000000013 SeShutdownPrivilege               Attributes -
 23 0x000000017 SeChangeNotifyPrivilege           Attributes - Enabled Default
 25 0x000000019 SeUndockPrivilege                 Attributes -
 33 0x000000021 SeIncreaseWorkingSetPrivilege     Attributes -
 34 0x000000022 SeTimeZonePrivilege               Attributes -
Authentication ID:         (0,1aa448)
Impersonation Level:       Anonymous
TokenType:                 Primary
Source: User32             TokenFlags: 0x2a00 ( Token in use )
Token ID: 1be803           ParentToken ID: 1aa44b
Modified ID:               (0, 43d9289)
RestrictedSidCount: 0      RestrictedSids: 0x0000000000000000
OriginatingLogonSession: 3e7
PackageSid: (null)
CapabilityCount: 0      Capabilities: 0x0000000000000000
LowboxNumberEntry: 0x0000000000000000
Security Attributes:
Unable to get the offset of nt!_AUTHZBASEP_SECURITY_ATTRIBUTE.ListLink
Process Token TrustLevelSid: (null)
```

Обратите внимание: SID пакета для Проводника не указан, потому что образ не выполняется в AppContainer.

Запустите в Windows 10 приложение calc.exe, которое запускает calculator.exe (теперь это приложение UWP), и проанализируйте его маркер:

```
lkd> !process 0 1 calculator.exe
PROCESS ffffe18309e874c0
    SessionId: 1  Cid: 3c18    Peb: cd0182c000  ParentCid: 035c
    DirBase: 7a15e4000  ObjectTable: ffffcd82ec9a37c0  HandleCount: <Data Not
Accessible>
    Image: Calculator.exe
    VadRoot ffffe1831cf197c0 Vads 181 Clone 0 Private 3800. Modified 3746. Locked 503.
    DeviceMap ffffcd82c39bc0d0
    Token                             ffffcd82e26168f0
...


lkd> !token ffffcd82e26168f0
_TOKEN 0xffffcd82e26168f0
TS Session ID: 0x1
User: S-1-5-21-3537846094-3055369412-2967912182-1001
User Groups:
 00 S-1-16-4096
    Attributes - GroupIntegrity GroupIntegrityEnabled
 01 S-1-1-0
    Attributes - Mandatory Default Enabled
 02 S-1-5-114
    Attributes - DenyOnly
 03 S-1-5-21-3537846094-3055369412-2967912182-1004
    Attributes - Mandatory Default Enabled
 04 S-1-5-32-544
    Attributes - DenyOnly
 05 S-1-5-32-578
    Attributes - Mandatory Default Enabled
 06 S-1-5-32-559
    Attributes - Mandatory Default Enabled
 07 S-1-5-32-545
    Attributes - Mandatory Default Enabled
 08 S-1-5-4
    Attributes - Mandatory Default Enabled
 09 S-1-2-1
    Attributes - Mandatory Default Enabled
 10 S-1-5-11
    Attributes - Mandatory Default Enabled
 11 S-1-5-15
    Attributes - Mandatory Default Enabled
 12 S-1-11-96-3623454863-58364-18864-2661722203-1597581903-1225312835-2511459453-
1556397606-2735945305-1404291241
    Attributes - Mandatory Default Enabled
 13 S-1-5-113
    Attributes - Mandatory Default Enabled
 14 S-1-5-5-0-1745560
    Attributes - Mandatory Default Enabled LogonId
 15 S-1-2-0
    Attributes - Mandatory Default Enabled
 16 S-1-5-64-36
    Attributes - Mandatory Default Enabled
```

```
Primary Group: S-1-5-21-3537846094-3055369412-2967912182-1001
Privs:
 19 0x000000013 SeShutdownPrivilege               Attributes -
 23 0x000000017 SeChangeNotifyPrivilege           Attributes - Enabled Default
 25 0x000000019 SeUndockPrivilege                 Attributes -
 33 0x000000021 SeIncreaseWorkingSetPrivilege     Attributes -
 34 0x000000022 SeTimeZonePrivilege               Attributes -
Authentication ID:         (0,1aa448)
Impersonation Level:       Anonymous
TokenType:                 Primary
Source: User32             TokenFlags: 0x4a00 ( Token in use )
Token ID: 4ddb8c0          ParentToken ID: 1aa44b
Modified ID:               (0, 4ddb8b2)
RestrictedSidCount: 0      RestrictedSids: 0x0000000000000000
OriginatingLogonSession: 3e7
PackageSid: S-1-15-2-466767348-3739614953-2700836392-1801644223-4227750657-
1087833535-2488631167
CapabilityCount: 1      Capabilities: 0xffffcd82e1bfccd0
Capabilities:
 00 S-1-15-3-466767348-3739614953-2700836392-1801644223-4227750657-1087833535-
2488631167
    Attributes - Enabled
LowboxNumberEntry: 0xffffcd82fa2c1670
LowboxNumber: 5
Security Attributes:
Unable to get the offset of nt!_AUTHZBASEP_SECURITY_ATTRIBUTE.ListLink
Process Token TrustLevelSid: (null)
```

Как видно из вывода, от Калькулятора требуется только одна возможность (она в действительности равна RID SID его контейнера AppContainer, см. раздел «AppContainer» далее в этой главе). При просмотре маркера процесса Cortana (searchui.exe) выводятся следующие возможности:

```
lkd> !process 0 1 searchui.exe
PROCESS ffffe1831307d080
    SessionId: 1  Cid: 29d8    Peb: fb407ec000  ParentCid: 035c
DeepFreeze
    DirBase: 38b635000  ObjectTable: ffffcd830059e580  HandleCount: <Data Not
Accessible>
    Image: SearchUI.exe
    VadRoot ffffe1831fe89130 Vads 420 Clone 0 Private 11029. Modified 2031. Locked 0.
    DeviceMap ffffcd82c39bc0d0
    Token                             ffffcd82d97d18f0
    ...

lkd> !token ffffcd82d97d18f0
_TOKEN 0xffffcd82d97d18f0
TS Session ID: 0x1
User: S-1-5-21-3537846094-3055369412-2967912182-1001
User Groups:
 ...
Primary Group: S-1-5-21-3537846094-3055369412-2967912182-1001
Privs:
 19 0x000000013 SeShutdownPrivilege               Attributes -
 23 0x000000017 SeChangeNotifyPrivilege           Attributes - Enabled Default
 25 0x000000019 SeUndockPrivilege                 Attributes -
 33 0x000000021 SeIncreaseWorkingSetPrivilege     Attributes -
```

```
 34 0x000000022 SeTimeZonePrivilege              Attributes -
Authentication ID:         (0,1aa448)
Impersonation Level:       Anonymous
TokenType:                 Primary
Source: User32             TokenFlags: 0x4a00 ( Token in use )
Token ID: 4483430          ParentToken ID: 1aa44b
Modified ID:               (0, 4481b11)
RestrictedSidCount: 0      RestrictedSids: 0x0000000000000000
OriginatingLogonSession: 3e7
PackageSid: S-1-15-2-1861897761-1695161497-2927542615-642690995-327840285-
2659745135-2630312742
CapabilityCount: 32     Capabilities: 0xffffcd82f78149b0
Capabilities:
 00 S-1-15-3-1024-1216833578-114521899-3977640588-1343180512-2505059295-473916851-
3379430393-3088591068
    Attributes - Enabled
 01 S-1-15-3-1024-3299255270-1847605585-2201808924-710406709-3613095291-873286183-
3101090833-2655911836
    Attributes - Enabled
 02 S-1-15-3-1024-34359262-2669769421-2130994847-3068338639-3284271446-2009814230-
2411358368-814686995
    Attributes - Enabled
 03 S-1-15-3-1
    Attributes - Enabled
...
 29 S-1-15-3-3633849274-1266774400-1199443125-2736873758
    Attributes - Enabled
 30 S-1-15-3-2569730672-1095266119-53537203-1209375796
    Attributes - Enabled
 31 S-1-15-3-2452736844-1257488215-2818397580-3305426111
    Attributes - Enabled
LowboxNumberEntry: 0xffffcd82c7539110
LowboxNumber: 2
Security Attributes:
Unable to get the offset of nt!_AUTHZBASEP_SECURITY_ATTRIBUTE.ListLink
Process Token TrustLevelSid: (null)
```

Всего Cortana требует наличия 32 возможностей. Это говорит лишь о том, что процесс богат функциональностью, которая должна проверяться системой.

Содержимое маркеров можно косвенно просматривать на вкладке Security процесса Process Explorer в диалоговом окне свойств процесса. В окне выводятся группы и привилегии, включенные в маркер просматриваемого процесса.

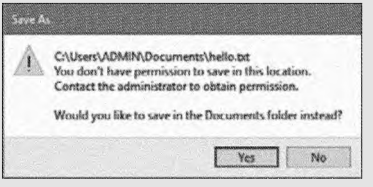
ЭКСПЕРИМЕНТ: ЗАПУСК ПРОГРАММЫ С НИЗКИМ УРОВНЕМ ЦЕЛОСТНОСТИ

Когда программа запускается с повышенными привилегиями — либо путем использования команды Запуск от имени администратора (Run As Administrator), либо потому, что программа этого требует, — программа явным образом запускается на высоком уровне целостности. При этом также есть возможность запустить программу на низком уровне целостности, использовав утилиту Psexec из пакета Sysinternals:

1. Запустите программу Блокнот (Notepad) на низком уровне целостности, использовав следующую команду:

```
c:\psexec -l notepad.exe
```

2. Попробуйте открыть файл (например, один из файлов с расширением .XML) в каталоге %SystemRoot%\System32. Обратите внимание на то, что вы можете просматривать каталог и открывать любой содержащийся в нем файл.
3. В программе Блокнот выполните команду Файл ▸ Создать (File ▸ New).
4. Введите в окне какой-нибудь текст и попробуйте сохранить его в каталоге %SystemRoot%\System32. Блокнот должен вывести окно сообщения об отсутствии прав и с рекомендацией сохранить файл в папке Документы (Documents).



5. Согласитесь с предложением программы Блокнот. Вы снова получите аналогичное окно — и так будет при каждой подобной попытке.
6. Теперь попробуйте сохранить файл в каталоге LocalLow вашего профиля пользователя в эксперименте, проделанном ранее в данной главе.

В предыдущем эксперименте сохранение файла в каталоге LocalLow сработает, поскольку программа Блокнот была запущена с низким уровнем целостности и только у каталога LocalLow также имеется низкий уровень целостности. Все остальные места, в которые вы пробовали записать файл, неявно имели уровень целостности Medium (средний). (Вы можете проверить этот факт с помощью утилиты Accesschk.) Но чтение из каталога %SystemRoot%\System32, равно как и открытие находящихся в нем файлов, работало даже при том, что каталог и его файл также имели неявный средний уровень целостности.

Заимствование прав

Заимствование прав (impersonation) является весьма эффективным свойством Windows, часто используемым в ее модели безопасности. Windows также использует заимствование прав в своей модели программирования «клиент-сервер». Например, серверное приложение может предоставить доступ к таким ресурсам, как файлы, принтеры или базы данных. Клиенты, желающие получить доступ к ресурсу, отправляют запрос на сервер. Когда сервер получает запрос, он должен убедиться, что у клиента есть разрешение на осуществление желаемой операции над ресурсом. Например, если пользователь на удаленной машине пробует удалить файл на общей NTFS-системе, сервер, экспортирующий общий ресурс, должен

определить, разрешено ли пользователю удалять файл. Наиболее очевидный способ проверки разрешения у пользователя — запрос SID-идентификаторов учетной записи и групп пользователя и сканирования атрибутов безопасности файла. Такой подход для программы слишком однообразен, при его реализации нетрудно наделать ошибок, и он не допускает прозрачной поддержки новых свойств системы безопасности. Поэтому для облегчения работы сервера Windows предоставляет службы заимствования прав.

Заимствование прав позволяет серверу уведомить SRM о том, что сервер временно заимствует профиль безопасности клиента, запрашивающего ресурс. Затем сервер может получить доступ к ресурсам от имени клиента, а проверку безопасности берет на себя SRM, но делает это на основе контекста безопасности того клиента, права которого были позаимствованы. Обычно у сервера имеется доступ к более широкому спектру ресурсов, чем у клиента, и в ходе заимствования прав он теряет некоторые из своих полномочий безопасности. Но может быть верным и обратное утверждение: в ходе заимствования прав сервер может получить полномочия безопасности.

Сервер заимствует права клиента только в рамках того потока, который делает запрос на заимствование прав. Структура данных управления потоком содержит дополнительную запись для маркера заимствования прав. Но основной маркер потока, представляющий его реальные полномочия безопасности, всегда доступен в управляющей структуре, принадлежащей процессу.

Windows обеспечивает заимствование прав посредством нескольких механизмов. Например, если сервер обменивается данными с клиентом через именованный канал, сервер может воспользоваться функцией `ImpersonateNamedPipeClient`, входящей в Windows API, чтобы сообщить SRM о том, что он хочет позаимствовать права пользователя на другом конце канала. Если сервер связан с клиентом через DDE (Dynamic Data Exchange) или через RPC, он межет сделать такие же запросы на заимствование прав с помощью функции `DdeImpersonateClient` и функции `RpcImpersonateClient`. С помощью функции `ImpersonateSelf` поток может создать маркер заимствования прав, являющийся простой копией маркера его процесса. Затем поток может изменить его маркер заимствования прав, возможно, для отключения SID-идентификаторов или привилегий. Пакет интерфейса поддержки безопасности поставщика — SSPI (Security Support Provider Interface) — может заимствовать права своих клиентов с помощью функции `ImpersonateSecurityContext`. SSPI-пакеты реализуют сетевой протокол аутентификации, например LAN Manager версии 2 или Kerberos. Другие интерфейсы, такие как COM, экспонируют заимствование прав через собственные API-функции, например через функцию `CoImpersonateClient`.

После завершения потоком сервера своей задачи он возвращается к своему основному контексту безопасности. Эти формы заимствования прав удобны для выполнения конкретных действий по запросу клиента и для обеспечения правильной проверки доступа к объекту. (Например, осуществляемая проверка дает идентичность клиента, чьи права заимствуются, а не идентичность серверного процесса.)

Недостатком таких форм заимствования прав является невозможность выполнения всей программы в контексте клиента. Кроме того, маркер заимствования не может дать доступ к файлам или принтерам на общих сетевых ресурсах, если права не позаимствованы на уровне делегирования (которое будет вскоре рассмотрено) и не имеются соответствующие полномочия на аутентификацию на удаленной машине или если общий файл или принтер не поддерживает нулевые сеансы. (Нулевой сеанс является результатом анонимного входа в систему.)

Если в контексте безопасности клиента должно выполняться все приложение или должен быть получен доступ к сетевым ресурсам без использования заимствования прав, клиент должен быть зарегистрирован в системе. Это действие позволяет выполнить функция `LogonUser` из состава Windows API. Функции `LogonUser` в качестве входных данных передается имя учетной записи, пароль, домен или имя компьютера, тип входа в систему (например, интерактивный, пакетный или служебный) и поставщик входа в систему, а она возвращает основной маркер доступа. Поток сервера может принять маркер в качестве маркера заимствования, или же сервер может запустить программу, имеющую полномочия клиента, в качестве основного маркера доступа. С точки зрения безопасности создаваемый процесс использует маркер, возвращенный из интерактивного входа в систему через функцию `LogonUser`, например, с API-функцией `CreateProcessAsUser`, подобно программе, запущенной пользователем, который зарегистрировался на машине в интерактивном режиме. Недостатком такого подхода является то, что сервер должен получить имя пользователя и пароль. Если серверу эта информация передается по сети, то она должна быть надежно зашифрована, чтобы злоумышленники, отслеживающие сетевой трафик, не могли ее перехватить.

Чтобы не допустить неправильного использования заимствования прав, Windows не позволяет серверам выполнять заимствование прав без согласия клиентов. Клиентский процесс может ограничить уровень заимствования прав, доступный для выполнения на серверном процессе, указав при подключении к серверу качество безопасности службы — SQOS (Security Quality of Service). Например, при открытии именованного канала процесс может указать для Windows-функции `CreateFile` флаги `SECURITY_ANONYMOUS`, `SECURITY_IDENTIFICATION`, `SECURITY_IMPERSONATION` или `SECURITY_DELEGATION`. Эти же возможности применимы к другим функциям, связанным с заимствованием прав, которые были перечислены выше. Каждый уровень позволяет серверу выполнять различные типы операций с учетом контекста безопасности клиента:

- `SecurityAnonymous` — самый ограниченный уровень заимствования прав; сервер не может заимствовать права или идентифицировать клиента;
- `SecurityIdentification` позволяет серверу получать информацию об идентичности (SID-идентификаторы) клиента и привилегии клиента, но сервер не может заимствовать его права;
- `SecurityImpersonation` позволяет серверу идентифицировать клиента и заимствовать его права на локальной системе;

- `SecurityDelegation` — наиболее свободный уровень заимствования прав. Он позволяет серверу заимствовать права клиента на локальной и удаленных системах.

Другие интерфейсы, такие как RPC, используют другие константы со сходными значениями (например, `RPC_C_IMP_LEVEL_IMPERSONATE`).

Если клиент не установил уровень заимствования прав, по умолчанию Windows выбирает уровень `SecurityImpersonation`. Функция `CreateFile` также принимает в качестве модификаторов настройки заимствования прав константы `SECURITY_EFFECTIVE_ONLY` и `SECURITY_CONTEXT_TRACKING`:

- `SECURITY_EFFECTIVE_ONLY` не позволяет серверу включать и отключать привилегии клиента или групп в ходе заимствования сервером их прав;
- `SECURITY_CONTEXT_TRACKING` указывает на то, что любые изменения, сделанные клиентом в отношении его контекста безопасности, воздействуют на сервер, позаимствовавший его права. Если эта настройка не указана, сервер заимствует контекст клиента на время заимствования прав и не принимает никаких изменений. Эта настройка принимается во внимание, только когда процессы клиента и сервера находятся на одной и той же системе.

Для предотвращения сценариев фальсификации, в ходе которых процесс с низким уровнем целостности может создать пользовательский интерфейс, получающий полномочия пользователя, а затем использующий функцию `LogonUser` для получения маркера этого пользователя, применяется специальная политика целостности для сценариев заимствования прав: поток не может заимствовать права на маркер с уровнем целостности более высоким, чем имеется у него самого. Например, приложение, имеющее низкий уровень целостности, не может имитировать диалоговое окно, запрашивающее административные полномочия с последующей попыткой запуска процесса с более высоким уровнем привилегий. Политика, использующая механизм целостности для маркеров доступа с заимствованием прав, заключается в том, что уровень целостности маркера доступа, возвращаемого функцией `LsaLogonUser`, не должен быть выше уровня целостности вызывавшего эту функцию процесса.

Ограниченные маркеры

Ограниченный маркер создается из основного маркера или маркера заимствования с использованием функции `CreateRestrictedToken`. Ограниченный маркер является копией маркера, из которого он происходит, со следующими возможными изменениями.

- Из массива привилегий может быть удален ряд привилегий.
- SID-идентификаторы в маркере могут быть помечены как предназначенные только для запрета (deny-only). Такие SID-идентификаторы удаляют доступ к любым ресурсам, для которых доступ по SID-идентификатору запрещен с помощью соответствующего запрещающего доступ ACE-элемента, который

в противном случае был бы отменен ACE-элементом, предоставляющим доступ к группе, содержащей SID, ранее указанный в дескрипторе безопасности.

- SID-идентификаторы в маркере могут быть помечены как ограниченные. Такие SID-идентификаторы подлежат повторному проходу в алгоритме проверки доступа, который будет подвергать разбору только ограниченные SID-идентификаторы маркера. В результате как первого, так и второго прохода должен быть предоставлен доступ к ресурсу; в противном случае он не предоставляется.

Ограниченные маркеры применяются в том случае, когда приложению нужно позаимствовать права клиента на усеченном уровне безопасности, главным образом из соображений предосторожности при запуске кода, не внушающего доверия. Например, ограниченный маркер может иметь удаленную из него привилегию по завершении работы системы, чтобы не дать коду, выполняемому в контексте безопасности ограниченного маркера, перезапустить систему.

Фильтрованный маркер администратора

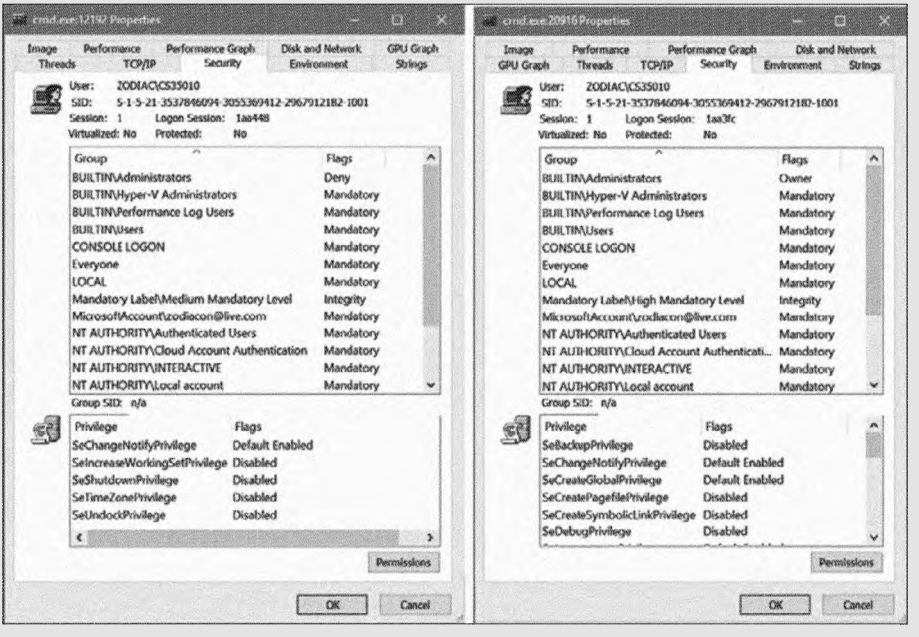
Как вы уже поняли, ограниченные маркеры также используются UAC для создания фильтрованного маркера администратора, который будет наследоваться всеми пользовательскими приложениями. Фильтрованный маркер администратора имеет следующие характеристики.

- Для уровня целостности устанавливается значение Medium (средний).
- Упомянутые ранее администраторские и подобные администраторским SID-идентификаторы помечаются только для запрета, чтобы предотвращать уязвимость в системе безопасности, возникающую при удалении целиком всей группы. Например, пусть у файла был доступ к списку ACL, который полностью отказывал группе «Администраторы» в доступе, но предоставлял некоторый доступ другой группе, к которой принадлежал пользователь. Тогда этот пользователь получил бы доступ, если группа «Администраторы» отсутствовала бы в маркере, что предоставило бы версии пользовательской идентификации обычного пользователя более широкий доступ, чем при административной идентификации пользователя.
- Удаляются все привилегии за исключением Change Notify, Shutdown, Undock, Increase Working Set и Time Zone.

ЭКСПЕРИМЕНТ: ПРОСМОТР ФИЛЬТРОВАННЫХ МАРКЕРОВ АДМИНИСТРАТОРА

Выполнив следующие действия на машине с включенным UAC-контролем, можно заставить Explorer запустить процесс либо с маркером обычного пользователя, либо с маркером администратора:

1. Войдите в систему под учетной записью, входящей в группу «Администраторы».
2. Откройте меню Пуск (Start), введите command, щелкните правой кнопкой мыши на открывшемся варианте Командная строка (Command Prompt) и выберите пункт Запуск от имени Администратора (Run As Administrator).
3. Запустите новый экземпляр cmd.exe, но на этот раз с обычным уровнем (т. е. без повышения привилегий).
4. Запустите утилиту Process Explorer с повышенными привилегиями, откройте диалоговые окна свойств для двух процессов окон командной строки и перейдите на вкладку Security. Обратите внимание: маркер обычного пользователя содержит SID только для запрета (deny-only) и обязательную метку Medium и у него совсем немного привилегий. Свойства, показанные в правой части копии экрана, относятся к окну командной строки, запущенному с маркером администратора, а свойства в левой части относятся к окну командной строки, запущенному с фильтрованным маркером администратора:



Виртуальные учетные записи служб

Windows предоставляет специализированный тип учетной записи, известный как *виртуальная учетная запись службы* (или просто *виртуальная учетная запись*), для улучшения защитной изоляции и управления доступом служб Windows с минимальными административными усилиями (см. главу 9). Без этого механизма

службам Windows пришлось бы запускаться либо под одной из учетных записей, определенных Windows для ее встроенных служб (например, Local Service или Network Service), либо под обычной учетной записью домена. Такие учетные записи, как Local Service, совместно используются многими существующими службами и поэтому предоставляют ограниченную детализацию привилегий и ограниченное управление доступом; более того, ими невозможно управлять в доменном пространстве. Доменные учетные записи требуют периодического изменения пароля для соблюдения мер безопасности, и цикл смены пароля может повлиять на доступность служб. Кроме того, для более качественной изоляции каждая служба должна выполняться под своей собственной учетной записью, но при использовании обычных учетных записей это в разы увеличивает усилия, затрачиваемые на управление.

При использовании виртуальных учетных записей служб каждая служба запускается под своей собственной учетной записью со своим собственным идентификатором безопасности. Имя учетной записи всегда начинается с префикса `NT SERVICE\`, за которым следует внутреннее имя службы. Виртуальные учетные записи служб могут появляться в списках управления доступом и могут быть связаны с привилегиями через групповую политику (Group Policy), как и всякое другое имя учетной записи. Но они не могут быть созданы или удалены с использованием обычных инструментов управления, а также не могут быть отнесены к каким-либо группам.

Windows автоматически устанавливает и периодически изменяет пароль виртуальной учетной записи службы. Как и в случае с учетной записью LocalSystem и другими учетными записями служб, пароль существует, но он неизвестен системным администраторам.

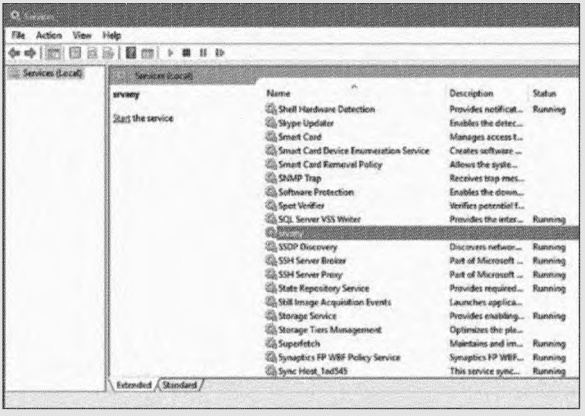
ЭКСПЕРИМЕНТ: ИСПОЛЬЗОВАНИЕ ВИРТУАЛЬНЫХ УЧЕТНЫХ ЗАПИСЕЙ СЛУЖБ

Создать службу, работающую под виртуальной учетной записью службы, можно с помощью средства Sc (service control, управление службой), выполнив следующие действия:

1. В окне командной строки, открытом с правами администратора, запустите утилиту командной строки Sc (service control) для создания службы и той виртуальной учетной записи, под которой она будет запущена, и введите команду create. В этом примере используется служба srvany из пакета Windows 2003 Resource Kit, который можно загрузить по адресу https://www.microsoft.com/en-us/download/details.aspx?id=17657

```
C:\Windows\system32>sc create srvany obj= "NT SERVICE\srvany" binPath=
"c:\temp\srvany.exe"
[SC] CreateService SUCCESS
```

2. Команда создает службу (в реестре, а также во внутреннем списке диспетчера контроллера служб) и виртуальную учетную запись службы. Теперь запустите MMC-оснастку Службы (Services) (services.msc), выберите новую службу и откройте ее диалоговое окно Свойства (Properties).

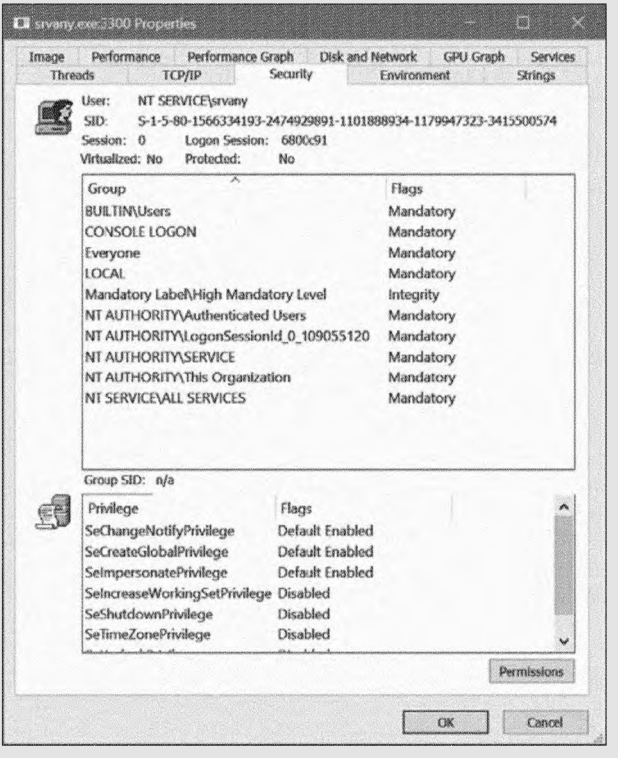


3. Перейдите на вкладку Вход в систему (Log On).

4. Диалоговое окно свойств службы можно также использовать для создания виртуальной учетной записи для уже существующей службы. Для этого измените имя учетной записи на «NT SERVICE\имя_службы» и очистите оба поля пароля. Но имейте в виду, что существующие службы под виртуальной учетной записью могут должным образом не запуститься, поскольку для этой учетной записи могут быть недоступны файлы или другие ресурсы, необходимые службе.

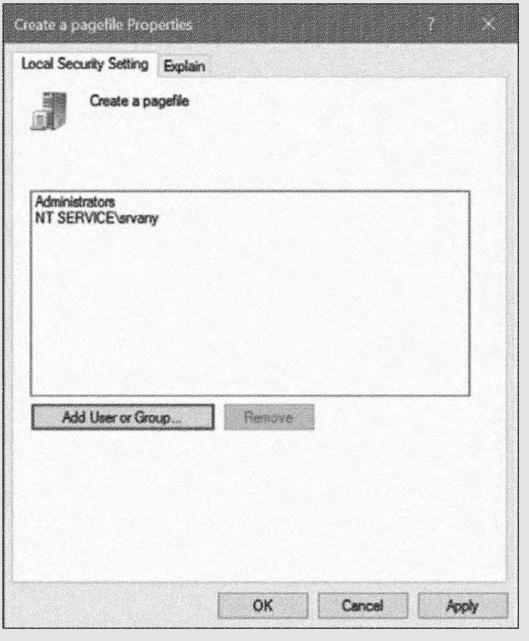
5. Если запустить Process Explorer и посмотреть на содержимое вкладки Security диалогового окна Свойства (Properties), открытого для службы с виртуальной

учетной записью, то можно увидеть имя виртуальной учетной записи и ее идентификатор безопасности (SID). Чтобы убедиться в этом, в диалоговом окне свойств службы srvany введите аргумент командной строки notepad.exe. (srvany может превращать обычные исполняемые образы в службы, поэтому в командной строке должен передаваться исполняемый файл). Запустите службу кнопкой Пуск (Start).



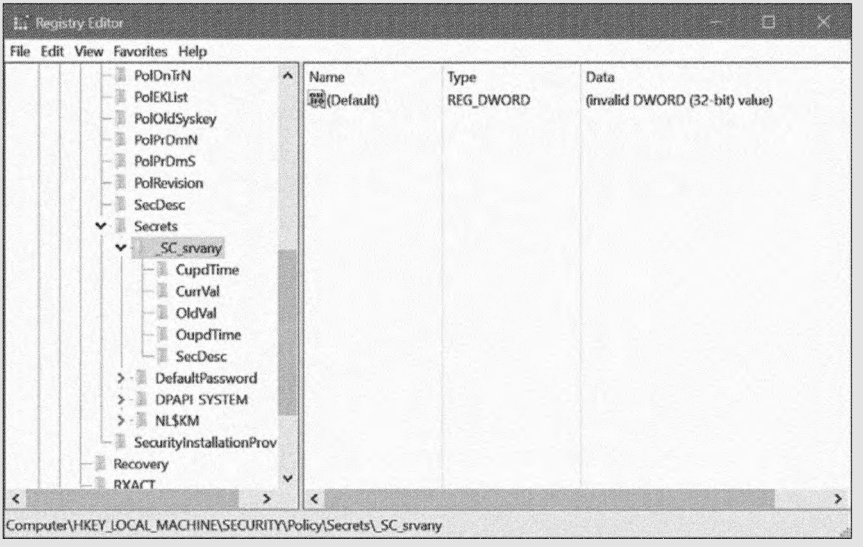
6. Виртуальная учетная запись службы может появиться в элементе управления доступом для любого объекта (например, файла), к которому служба должна иметь доступ. Если для файла открыть вкладку Безопасность (Security) в диалоговом окне Свойства (Properties) и создать ACL, ссылающийся на виртуальную учетную запись службы, вы увидите, что имя введенной вами учетной записи (например, NT SERVICE\srvany) было заменено просто именем службы (srvany) с помощью функции проверки имен, и оно появилось в списке управления доступом в укороченной форме.

7. Разрешения (или права пользователя) виртуальной учетной записи службы могут быть предоставлены через групповую политику. В данном примере виртуальной учетной записи службы srvany предоставлено право создания файла подкачки (с использованием редактора локальной политики безопасности secpol.msc).



8. В пользовательских средствах администрирования, таких как lusrmgr.msc, виртуальные учетные записи служб видны не будут, потому что они в кусте реестра SAM не сохраняются. Но если исследовать реестр в контексте встроенной учетной записи System (как описывалось ранее), вы увидите признаки учетной записи в разделе HKLM\Security\Policy\Secrets:

```
C:\>psexec -s -i -d regedit.exe
```



Дескрипторы безопасности и управление доступом

Маркеры, идентифицирующие учетные данные пользователя, являются лишь частью уравнения, определяющего безопасность объекта. Другой частью уравнения является информация безопасности, связанная с объектом, которая определяет, кто и какие действия с объектом может выполнять. Структура данных для этой информации называется *дескриптором безопасности*. В дескриптор безопасности входят следующие атрибуты.

- **Номер версии.** Версия модели безопасности SRM, использованная для создания дескриптора.
- **Флаги.** Дополнительные модификаторы, определяющие поведение или характеристики дескриптора. Эти флаги перечислены в табл. 7.5 (многие из них документированы в Windows SDK).
- **SID владельца.** Идентификатор безопасности владельца.
- **SID группы.** Идентификатор безопасности основной группы объекта (используется только подсистемой POSIX; в настоящее время не используется из-за прекращения поддержки POSIX).

- **Избирательный список управления доступом — DACL** (Discretionary Access Control list). Указывает, кто и какой доступ имеет к объекту.
- **Системный список управления доступом — SACL** (System Access Control List). Указывает, какие операции какими пользователями должны регистрироваться в журнале аудита безопасности и конкретный уровень целостности объекта.

Таблица 7.5. Флаги дескриптора безопасности

Флаг	Описание
SE_OWNER_DEFAULTED	Обозначает дескриптор безопасности с идентификатором безопасности (SID) владельца, используемым по умолчанию. Этот бит используется для поиска всех объектов, имеющих установленный по умолчанию набор полномочий владельца
SE_GROUP_DEFAULTED	Обозначает дескриптор безопасности с идентификатором безопасности (SID) группы, используемым по умолчанию. Этот бит используется для поиска всех объектов, имеющих установленный по умолчанию набор полномочий группы
SE_DACL_PRESENT	Обозначает дескриптор безопасности, имеющий DACL. Если этот флаг не установлен или если этот флаг установлен, а DACL имеет значение NULL, дескриптор безопасности разрешает полный доступ для всех
SE_DACL_DEFAULTED	Обозначает дескриптор безопасности с используемым по умолчанию DACL. Например, если создатель объекта не указал DACL, объект получает DACL, используемый по умолчанию, от маркера доступа создателя. Этот флаг может влиять на то, как система обрабатывает DACL с учетом наследования элемента управления доступом — ACE (Access Control Entry). Система игнорирует этот флаг, если не установлен флаг SE_DACL_PRESENT
SE_SACL_PRESENT	Обозначает дескриптор безопасности, у которого есть системный список управления доступом (SACL)
SE_SACL_DEFAULTED	Обозначает дескриптор безопасности с используемым по умолчанию SACL. Например, если создатель объекта не указал SACL, объект получает SACL по умолчанию от маркера доступа создателя. Этот флаг может повлиять на то, как система обрабатывает SACL с учетом наследования ACE. Система игнорирует этот флаг, если не установлен флаг SE_SACL_PRESENT
SE_DACL_UNTRUSTED	Обозначает, что ACL, указанный с помощью DACL дескриптора безопасности, был предоставлен ненадежным источником. Если этот флаг установлен и обнаружен составной ACE, система будет подставлять известные ей надежные SID-идентификаторы для серверных SID в ACE-элементах
SE_SERVER_SECURITY	Требует, чтобы поставщик объекта, защищенный дескриптором безопасности, был сервером ACL, основанным на входящем ACL, безотносительно его источника (явно указанного или взятого по умолчанию). Это делается путем замены всех GRANT ACE-элементов составными ACE-элементами, предоставляющими текущий серверный доступ. Этот флаг имеет смысл только при заимствовании прав

Флаг	Описание
SE_DACL_AUTO_INHERIT_REQ	Требует, чтобы поставщик объекта, защищенный дескриптором безопасности, автоматически распространял DACL на существующие дочерние объекты. Если поставщик поддерживает автоматическое наследование, DACL распространяется на любой существующий дочерний объект, а в дескрипторах безопасности родительского и дочернего объектов устанавливается бит SE_DACL_AUTO_INHERITED
SE_SACL_AUTO_INHERIT_REQ	Требует, чтобы поставщик объекта, защищенный дескриптором безопасности, автоматически распространял SACL на существующие дочерние объекты. Если поставщик поддерживает автоматическое наследование, SACL распространяется на любой существующий дочерний объект, а в дескрипторах безопасности родительского и дочернего объектов устанавливается бит SE_SACL_AUTO_INHERITED
SE_DACL_AUTO_INHERITED	Обозначает дескриптор безопасности, в котором DACL настроен на поддержку автоматического распространения наследования ACE-элементов на существующие дочерние объекты. Система устанавливает этот бит при выполнении алгоритма автоматического наследования для объекта и его существующих дочерних объектов
SE_SACL_AUTO_INHERITED	Обозначает дескриптор безопасности, в котором SACL настроен на поддержку автоматического распространения наследования ACE-элементов на существующие дочерние объекты. Система устанавливает этот бит при выполнении алгоритма автоматического наследования для объекта и его существующих дочерних объектов
SE_DACL_PROTECTED	Предотвращает модификацию DACL дескриптора безопасности наследуемыми ACE-элементами
SE_SACL_PROTECTED	Предотвращает модификацию SACL дескриптора безопасности наследуемыми ACE-элементами
SE_RM_CONTROL_VALID	Обозначает действительность битов диспетчера управления ресурсом, имеющихся в дескрипторе безопасности. Под биты диспетчера управления ресурсом в структуре дескриптора безопасности отводится 8 разрядов, в которых содержится информация, относящаяся к доступу к структуре диспетчера управления ресурсом
SE_SELF_RELATIVE	Обозначает дескриптор безопасности в самоопределяющемся относительном формате (self-relative format), при котором вся информация безопасности хранится в непрерывном блоке памяти. Если этот флаг не установлен, дескриптор безопасности хранится в абсолютном формате

Дескрипторы безопасности (SD, Security Descriptor) могут быть прочитаны на программном уровне функциями `GetSecurityInfo`, `GetKernelObjectSecurity`, `GetFileSecurity`, `GetNamedSecurityInfo` и другими, более экзотическими функциями. После чтения программа может выполнить различные манипуляции с SD, а затем вызвать соответствующую `Set`-функцию для внесения изменений. Кроме того,

дескриптор безопасности может быть построен в строковом виде на языке SDDL (Security Descriptor Definition Language), в котором дескриптор безопасности представляется компактной строкой. Строка преобразуется в дескриптор безопасности вызовом `ConvertStringSecurityDescriptorToSecurityDescriptor`. Как и следовало ожидать, существует обратная функция (`ConvertSecurityDescriptorToStringSecurityDescriptor`). За подробным описанием SDDL обращайтесь к Windows SDK.

Список управления доступом (ACL) состоит из заголовка и нескольких элементов управления доступом (ACE), которых может и не быть. Существует два типа ACL-списков: DACL и SACL. В DACL в каждом ACE-элементе содержится SID и маска доступа (а также набор флагов, которые будут вскоре рассмотрены), которая обычно определяет права доступа (чтение, запись, удаление и т. д.), предоставленные или запрещенные держателю SID. Существует девять типов ACE-элементов, которые могут появляться в DACL: доступ разрешен, доступ запрещен, объект разрешен, объект запрещен, обратный вызов разрешен, обратный вызов запрещен, обратный вызов объекта разрешен, обратный вызов объекта запрещен и условные требования. Как нетрудно догадаться, разрешающие доступ ACE предоставляют доступ пользователю, а запрещающие доступ ACE запрещают права доступа, указанные в маске доступа. ACE-элементы обратного вызова используются приложениями, применяющими API-интерфейс AuthZ (рассматриваемый чуть позже) для регистрации обратного вызова, который будет вызван AuthZ при выполнении проверки доступа с использованием этого ACE.

Разница между разрешенным объектом и разрешенным доступом и между запрещенным объектом и запрещенным доступом состоит в том, что типы объектов используются только в Active Directory. ACE-элементы этих типов имеют поле глобального уникального идентификатора — GUID, которое обозначает, что ACE применяется только к конкретным объектам или подчиненным объектам (к тем, у которых имеются GUID-идентификаторы). Кроме того, другой дополнительный GUID (128-разрядный идентификатор, гарантирующий его уникальность) обозначает тот тип дочернего объекта, который унаследует ACE при создании дочернего объекта в контейнере Active Directory, с примененным к нему ACE. ACE условных требований хранится в ACE-структуре типа `*-callback` и рассматривается в разделе, посвященном API-функциям AuthZ.

Аккумулирование прав доступа, предоставленных отдельными ACE-элементами, формирует набор прав доступа, предоставленных ACL-списком. Если в дескрипторе безопасности отсутствует DACL (указан нулевой DACL), то полный доступ к объекту предоставляется кому угодно. Если DACL пуст (т. е. в нем нуль ACE-элементов), никто не имеет доступа к объекту.

У ACE-элементов, используемых в DACL-списках, имеется набор флагов, управляющих характеристиками и определяющих характеристики ACE, связанные с наследованием. В некоторых пространствах имен объектов имеются контейнеры и объекты. Контейнер может содержать другие объекты-контейнеры и объекты-листья, являющиеся его дочерними объектами. Примерами контейнеров служат каталоги в пространстве имен файловой системы и разделы в пространстве имен

реестра. Некоторые флаги в ACE управляют тем, как ACE распространяется на дочерние объекты контейнера, связанные с ACE. В табл. 7.6, частично повторяющей Windows SDK, перечислены правила наследования для флагов ACE.

Таблица 7.6. Правила наследования для флагов ACE

Флаг	Правило наследования
`CONTAINER_INHERIT_ACE`	Дочерние объекты, являющиеся контейнерами, например каталоги, наследуют ACE в качестве действующего ACE-элемента. Унаследованный ACE является наследуемым при условии, что не установлен битовый флаг `NO_PROPAGATE_INHERIT_ACE`
`INHERIT_ONLY_ACE`	Этот флаг обозначает только наследуемый ACE, который не управляет доступом к объекту, к которому он прикреплен. Если этот флаг не установлен, ACE управляет доступом к тому объекту, к которому он прикреплен
`INHERITED_ACE`	Этот флаг обозначает, что ACE был унаследован. Система устанавливает этот бит при распространении наследуемого ACE на дочерний объект
`NO_PROPAGATE_INHERIT_ACE`	Если ACE унаследован дочерним объектом, система снимает флаги `OBJECT_INHERIT_ACE` и `CONTAINER_INHERIT_ACE` в унаследованном ACE. Это действие препятствует наследованию ACE следующими поколениями объектов
`OBJECT_INHERIT_ACE`	Не являющиеся контейнером дочерние объекты наследуют ACE в качестве действующего ACE. Для дочерних объектов, являющихся контейнерами, ACE наследуется в качестве только наследуемого ACE, если только не установлен флаг `NO_PROPAGATE_INHERIT_ACE`

В SACL содержатся два типа ACE-элементов, ACE-элементы системного аудита и ACE-элементы объекта системного аудита. Эти ACE-элементы определяют, какие операции выполняются в отношении объекта путем указания подвергаемых аудиту пользователей или групп. Информация об аудите хранится в системном журнале аудита — Audit Log. Аудиту могут быть подвержены как удачные, так и неудачные попытки. Как и в имеющихся в DACL двоюродных ACE, сориентированных на объекты, их системные родственники, ACE-элементы, сориентированные на объекты, указывают GUID, обозначающий типы объектов или подчиненных объектов, к которым применяется ACE, и дополнительный GUID, управляющий распространением ACE на конкретные типы дочерних объектов. Если SACL-список является нулевым, аудит в отношении объекта не проводится. Флаги наследования, применяемые к ACE-элементам DACL, также применяются к ACE-элементам системного аудита и к ACE-элементам объектов системного аудита.

На рис. 7.8 представлена упрощенная схема файлового объекта и его DACL. Как видно из иллюстрации, первый ACE разрешает пользователю USER1 запрашивать файл. Второй ACE запрещает участникам группы TEAM1 запись, а третий ACE предоставляет всем пользователям (Everyone) доступ к выполнению файла.

Рис. 7.8. Избирательный список управления доступом (DACL)

ЭКСПЕРИМЕНТ: ПРОСМОТР ДЕСКРИПТОРА БЕЗОПАСНОСТИ

Большинство подсистем исполняющей системы полагаются при управлении дескрипторами безопасности своих объектов на исходную функциональность безопасности диспетчера объектов. Исходные функции безопасности диспетчера объектов используют указатель дескриптора безопасности для хранения дескрипторов безопасности таких объектов. Например, диспетчер процессов использует исходные настройки безопасности, поэтому диспетчер объектов сохраняет дескрипторы безопасности процесса и потока в заголовках объектов процесса и потока соответственно. Указатель дескриптора безопасности событий, мьютексов и семафоров также хранит их дескрипторы безопасности. Для просмотра дескрипторов безопасности таких объектов при нахождении их заголовков можно воспользоваться живой отладкой ядра, выполняя указанные далее действия. (Следует заметить, что и Process Explorer, и AccessChk также могут показать дескрипторы безопасности для процессов.)

1. Запустите локальную отладку ядра.
2. Введите команду `!process 0 0 explorer.exe` для получения информации о процессе Explorer:

```
lkd> !process 0 0 explorer.exe
PROCESS ffffe18304dfd780
    SessionId: 1  Cid: 23e4    Peb: 00c2a000  ParentCid: 2264
    DirBase: 2aa0f6000  ObjectTable: ffffcd82c72fcd80  HandleCount:
<Data Not Accessible>
    Image: explorer.exe

PROCESS ffffe1830670a080
    SessionId: 1  Cid: 27b8    Peb: 00950000  ParentCid: 035c
    DirBase: 2cba97000  ObjectTable: ffffcd82c7ccc500  HandleCount:
<Data Not Accessible>
    Image: explorer.exe
```

3. Если указано более одного экземпляра Проводника, выберите один (неважно, какой именно). Введите команду `!object` с адресом, который следует за словом PROCESS в выводе предыдущей команды в качестве аргумента для вывода структуры данных объекта:

```
lkd> !object ffffe18304dfd780
Object: ffffe18304dfd780  Type: (ffffe182f7496690) Process
    ObjectHeader: ffffe18304dfd750 (new version)
    HandleCount: 15  PointerCount: 504639
```

4. Наберите команду `dt _OBJECT_HEADER` и адрес поля заголовка объекта из вывода предыдущей команды, чтобы показать структуру данных заголовка объекта, включая значение указателя дескриптора безопасности:

```
lkd> dt nt!_object_header ffffe18304dfd750
   +0x000 PointerCount     : 0n504448
   +0x008 HandleCount      : 0n15
   +0x008 NextToFree       : 0x00000000'0000000f Void
   +0x010 Lock             : _EX_PUSH_LOCK
   +0x018 TypeIndex        : 0xe5 ''
   +0x019 TraceFlags       : 0 ''
   +0x019 DbgRefTrace      : 0y0
   +0x019 DbgTracePermanent : 0y0
   +0x01a InfoMask         : 0x88 ''
   +0x01b Flags            : 0 ''
   +0x01b NewObject        : 0y0
   +0x01b KernelObject     : 0y0
   +0x01b KernelOnlyAccess : 0y0
   +0x01b ExclusiveObject  : 0y0
   +0x01b PermanentObject  : 0y0
   +0x01b DefaultSecurityQuota : 0y0
   +0x01b SingleHandleEntry : 0y0
   +0x01b DeletedInline    : 0y0
   +0x01c Reserved         : 0x30003100
   +0x020 ObjectCreateInfo : 0xffffe183'09e84ac0 _OBJECT_CREATE_INFORMATION
   +0x020 QuotaBlockCharged : 0xffffe183'09e84ac0 Void
   +0x028 SecurityDescriptor : 0xffffcd82'cd0e97ed Void
   +0x030 Body             : _QUAD
```

5. И наконец, воспользуйтесь командой отладчика `!sd` для вывода дампа дескриптора безопасности. Указатель дескриптора безопасности в заголовке объекта использует некоторые младшие разряды в качестве флагов, которые должны быть обнулены до следующего указателя. На 32-разрядных системах имеется три бита флагов, поэтому в следующем примере с адресом дескриптора безопасности, показанном в структуре заголовка объекта, используется аргумент `& -8`. На 64-разрядных системах имеется четыре бита флагов, поэтому вместо предыдущего аргумента нужно использовать аргумент `& -10`.

```
lkd> !sd 0xffffcd82'cd0e97ed & -10
->Revision: 0x1
->Sbz1    : 0x0
->Control : 0x8814
            SE_DACL_PRESENT
            SE_SACL_PRESENT
            SE_SACL_AUTO_INHERITED
            SE_SELF_RELATIVE
->Owner   : S-1-5-21-3537846094-3055369412-2967912182-1001
->Group   : S-1-5-21-3537846094-3055369412-2967912182-1001
->Dacl    :
->Dacl    : ->AclRevision: 0x2
->Dacl    : ->Sbz1       : 0x0
->Dacl    : ->AclSize    : 0x5c
```

```
->Dacl    : ->AceCount   : 0x3
->Dacl    : ->Sbz2       : 0x0
->Dacl    : ->Ace[0]: ->AceType: ACCESS_ALLOWED_ACE_TYPE
->Dacl    : ->Ace[0]: ->AceFlags: 0x0
->Dacl    : ->Ace[0]: ->AceSize: 0x24
->Dacl    : ->Ace[0]: ->Mask : 0x001fffff
->Dacl    : ->Ace[0]: ->SID: S-1-5-21-3537846094-3055369412-2967912182-1001

->Dacl    : ->Ace[1]: ->AceType: ACCESS_ALLOWED_ACE_TYPE
->Dacl    : ->Ace[1]: ->AceFlags: 0x0
->Dacl    : ->Ace[1]: ->AceSize: 0x14
->Dacl    : ->Ace[1]: ->Mask : 0x001fffff
->Dacl    : ->Ace[1]: ->SID: S-1-5-18

->Dacl    : ->Ace[2]: ->AceType: ACCESS_ALLOWED_ACE_TYPE
->Dacl    : ->Ace[2]: ->AceFlags: 0x0
->Dacl    : ->Ace[2]: ->AceSize: 0x1c
->Dacl    : ->Ace[2]: ->Mask : 0x00121411
->Dacl    : ->Ace[2]: ->SID: S-1-5-5-0-1745560

->Sacl    :
->Sacl    : ->AclRevision: 0x2
->Sacl    : ->Sbz1       : 0x0
->Sacl    : ->AclSize    : 0x1c
->Sacl    : ->AceCount   : 0x1
->Sacl    : ->Sbz2       : 0x0
->Sacl    : ->Ace[0]: ->AceType: SYSTEM_MANDATORY_LABEL_ACE_TYPE
->Sacl    : ->Ace[0]: ->AceFlags: 0x0
->Sacl    : ->Ace[0]: ->AceSize: 0x14
->Sacl    : ->Ace[0]: ->Mask : 0x00000003
->Sacl    : ->Ace[0]: ->SID: S-1-16-8192
```

Дескриптор безопасности содержит три разрешающих доступ ACE-элемента: один для текущего пользователя (S-1-5-21-3537846094-3055369412-2967912182-1001), один для учетной записи System (S-1-5-18) и последний для SID-входа в систему — Logon SID (S-1-5-5-0-1745560). Список управления системным доступом имеет один элемент (S-1-16-8192), который помечает процесс как имеющий уровень целостности Medium (средний).

Назначение ACL

Чтобы определить, какой DACL нужно назначить новому объекту, система безопасности использует первое подходящее правило из следующих четырех.

1. Если при создании объекта вызывающий процесс предоставляет дескриптор безопасности явным образом, система безопасности применяет его к объекту. Если у объекта есть имя и он находится в объекте-контейнере (например, именованный объект-событие в каталоге `\BaseNamedObjects` пространства имен диспетчера объектов), система объединяет любые наследуемые ACE-элементы (ACE-элементы, которые могут распространяться из контейнера объекта) в DACL, если только у дескриптора безопасности не установлен флаг `SE_DACL_PROTECTED`, мешающий наследованию.

2. Если вызывающий процесс не предоставляет дескриптор безопасности и у объекта есть имя, система безопасности смотрит на дескриптор безопасности контейнера, в котором хранится имя нового объекта. Некоторые ACE-элементы каталога объекта могут быть помечены наследуемыми, следовательно, они должны быть применены к новым объектам, созданным в каталоге объекта. Если имеются какие-нибудь из этих наследуемых ACE-элементов, система безопасности формирует из них ACL-список, который прикрепляется к новому объекту. (Отдельные флаги обозначают ACE-элементы, которые должны быть унаследованы только объектами-контейнерами, а не объектами, не являющимися контейнерами.)
3. Если дескриптор безопасности не указан и объект не наследует никаких ACE-элементов, система безопасности извлекает исходный DACL из маркера доступа вызывающего процесса и применяет его к новому объекту. Некоторые подсистемы в Windows имеют жестко заданные DACL-списки, которые они присваивают создаваемому объекту (например, объектам служб, LSA и SAM).
4. Если дескриптор не указан, наследуемые ACE-элементы отсутствуют и нет исходного DACL-списка, система создает объект без DACL, что позволяет всем группам и пользователям иметь полный доступ к объекту. Это правило похоже на третье правило, в котором маркер содержит нулевой исходный DACL.

Правила, используемые системой при назначении новому объекту списка SACL, аналогичны тем правилам, которые применяются для назначения списка DACL, за некоторыми исключениями.

- Наследуемые ACE-элементы системного аудита не распространяются на объекты, имеющие дескрипторы безопасности, помеченные флагом `SE_SACL_PROTECTED` (который похож на флаг `SE_DACL_PROTECTED`, защищающий списки DACL).
- Если нет указанных ACE-элементов системного аудита и нет наследуемых списков SACL, никакие SACL к объекту не применяются. Это поведение отличается от того, которое применялось к исходным спискам DACL, поскольку у маркеров нет исходных SACL.

Когда новый дескриптор безопасности, содержащий наследуемые ACE-элементы, применяется к контейнеру, система автоматически распространяет наследуемые ACE-элементы на дескрипторы безопасности дочерних объектов. (Следует иметь в виду, что список DACL дескриптора безопасности не принимает наследуемые ACE-элементы DACL, если установлен его флаг `SE_DACL_PROTECTED`, и его SACL не наследует ACE-элементы SACL, если у дескриптора установлен флаг `SE_SACL_PROTECTED`.) Порядок, в котором наследуемые ACE-элементы объединяются с существующим дескриптором безопасности дочернего объекта, таков, что любые ACE-элементы, которые были явным образом применены к ACL, хранятся перед теми ACE-элементами, которые были этим объектом унаследованы. Для распространения наследуемых ACE-элементов система применяет следующие правила.

- Если дочерний объект, не имеющий DACL, наследует ACE, это приводит к тому, что у этого дочернего объекта появляется DACL, содержащий только унаследованный ACE.
- Если дочерний объект с пустым DACL наследует ACE, это приводит к тому, что DACL у этого дочернего объекта содержит только унаследованный ACE.
- Применительно только к объектам Active Directory, если наследуемый ACE удаляется из родительского объекта, автоматически система наследования удаляет все копии ACE, унаследованные дочерними объектами.
- Применительно только к объектам Active Directory, если в результате автоматического наследования из DACL дочернего объекта удаляются все ACE-элементы, у дочернего элемента остается пустой, но все же имеющийся DACL.

Как вскоре обнаружится, порядок наследования ACE-элементов в ACL является весьма важным аспектом модели безопасности Windows.

ПРИМЕЧАНИЕ Вообще-то наследование для таких хранилищ объектов, как файловые системы, реестр или Active Directory, напрямую не поддерживается. API-интерфейсы Windows, поддерживающие наследование, включающие функцию SetEntriesInAcl, достигают своей цели путем вызова соответствующих функций в DLL-библиотеке поддержки наследования безопасности (%SystemRoot%\System32\Ntmarta.dll), которые знают, как проходить по таким хранилищам объектов.

Доверенные ACE-элементы

С появлением защищенных процессов и процессов PPL (Protected Processes Light; см. главу 3) возникла необходимость в ограничении доступа к объектам только со стороны защищенных процессов. Это важно для защиты некоторых ресурсов — таких, как раздел реестра `KnownDlls`, — от вмешательства даже со стороны кода административного уровня. Такие ACE-элементы задаются общеизвестными SID-идентификаторами, которые предоставляют уровень защиты и источник подписи, необходимые для получения доступа. В табл. 7.7 перечислены SID с указанием уровней и значений.

Таблица 7.7. Доверенные SID

SID	Уровень защиты	Источник защитной подписи
1-19-512-0	Облегченный защищенный (Protected Light)	Нет
1-19-512-4096	Облегченный защищенный (Protected Light)	Windows
1-19-512-8192	Облегченный защищенный (Protected Light)	WinTcb
1-19-1024-0	Защищенный (Protected)	Нет
1-19-1024-4096	Защищенный (Protected)	Windows
1-19-1024-8192	Защищенный (Protected)	WinTcb

Доверенный SID-идентификатор является частью объекта маркера, существующего для маркеров, присоединенных к защищенным или PPL-процессам. Чем выше SID, тем выше мощность маркера (вспомните, что уровень Protected выше, чем Protected Light).

ЭКСПЕРИМЕНТ: ПРОСМОТР ДОВЕРЕННЫХ SID-ИДЕНТИФИКАТОРОВ

В этом эксперименте рассматриваются доверенные SID-идентификаторы в маркерах защищенных процессов.

Запустите локальную отладку ядра.

Запросите данные о процессах Csrss.exe с базовой информацией:

```
lkd> !process 0 1 csrss.exe
PROCESS ffff8188e50b5780
    SessionId: 0  Cid: 0358    Peb: b3a9f5e000  ParentCid: 02ec
    DirBase: 1273a3000  ObjectTable: ffffbe0d829e2040  HandleCount:
<Data Not Accessible>
    Image: csrss.exe
    VadRoot ffff8188e6ccc8e0 Vads 159 Clone 0 Private 324. Modified 4470.
Locked 0.
    DeviceMap ffffbe0d70c15620
    Token                             ffffbe0d829e7060
    ...
PROCESS ffff8188e7a92080
    SessionId: 1  Cid: 03d4    Peb: d5b0de4000  ParentCid: 03bc
    DirBase: 162d93000  ObjectTable: ffffbe0d8362d7c0  HandleCount:
<Data Not Accessible>Modified 462372. Locked 0.
    DeviceMap ffffbe0d70c15620
    Token                             ffffbe0d8362d060
    ...
```

Выберите один из маркеров и выведите подробную информацию:

```
lkd> !token ffffbe0d829e7060
_TOKEN 0xffffbe0d829e7060
TS Session ID: 0
User: S-1-5-18
...
Process Token TrustLevelSid: S-1-19-512-8192
```

Как видите, это PPL-процесс с источником подписи WinTcb.

Определение возможности доступа

Для определения возможности доступа к объекту используются два метода:

- Обязательная (mandatory) проверка целостности, которая определяет, достаточно ли уровня целостности вызывающего процесса для доступа к ресурсу, на основе принадлежащего ресурсу уровня целостности и его обязательной политики.
- Проверка избирательного доступа, выявляющая, какой вид доступа к объекту имеется у конкретной учетной записи пользователя.

Когда процесс пытается открыть объект, проверка целостности осуществляется до стандартной Windows-проверки DACL в функции ядра `SeAccessCheck`, поскольку она выполняется быстрее и может быстро исключить надобность осуществления полной проверки избирательного доступа. Согласно исходным политикам целостности в его маркере доступа (`TOKEN_MANDATORY_NO_WRITE_UP` и `TOKEN_MANDATORY_NEW_PROCESS_MIN`, рассмотренным ранее), процесс может открыть объект для доступа по записи, если его уровень целостности выше уровня целостности объекта или равен ему, и DACL также предоставляет процессу требуемый доступ. Например, процесс с низким уровнем целостности не может открыть для записи объект, имеющий средний уровень целостности, даже если DACL предоставляет процессу право записи.

Применяя исходные политики целостности, процессы могут открывать любой объект, за исключением объектов процессов, потоков и маркеров, для доступа по чтению, если DACL этого объекта предоставляет им право на чтение. Это означает, что процесс, выполняемый с низким уровнем целостности, может открыть любые файлы, доступные пользовательской учетной записи, под которой он был запущен. Защищенный режим Internet Explorer — Protected Mode Internet Explorer — использует уровни целостности, чтобы помочь противодействовать вредоносным программам, ведущим заражение путем модификации настроек учетной записи пользователя, но он не помешает вредоносной программе прочитать документы пользователя.

Следует напомнить, что исключение составляют объекты процесса и потока, поскольку их политика целостности также включает запрет на чтение — No-Read-Up. Это означает, что уровень целостности процесса должен быть равен уровню целостности или выше уровня целостности того процесса или потока, который он хочет открыть, и что DACL должен предоставить ему требуемые доступы, чтобы попытка открытия завершилась успешно. В предположении, что списки DACL разрешают требуемый доступ, в табл. 7.8 показаны типы доступа, имеющиеся у процесса, выполняемого со средним или низким уровнем целостности к другим процессам и объектам.

Таблица 7.8. Управление доступом к объектам и процессам на основании уровня целостности

Процесс	Доступ к объектам	Доступ к другим процессам
Высокий уровень целостности (High)	Доступ для чтения/записи ко всем объектам с уровнем целостности High и ниже. Доступ для чтения к объектам с уровнем целостности System	Доступ для чтения/записи ко всем процессам с уровнем целостности High и ниже. Запрет доступа к процессам с уровнем целостности System
Средний уровень целостности (Medium)	Доступ для чтения/записи ко всем объектам с уровнем целостности Medium и Low. Доступ для чтения к объектам с уровнем целостности High или System	Доступ для чтения/записи ко всем процессам с уровнем целостности Medium и Low. Запрет доступа к объектам с уровнем целостности High или System

Процесс	Доступ к объектам	Доступ к другим процессам
Низкий уровень целостности (Low)	Доступ для чтения/записи ко всем объектам с уровнем целостности Low. Доступ для чтения к объектам с уровнем целостности Medium и выше	Доступ для чтения/записи ко всем процессам с уровнем целостности Low. Доступ для чтения к процессам с уровнем целостности Medium и выше

ПРИМЕЧАНИЕ Под доступом для чтения к процессу, упоминаемым в этом разделе, подразумевается полный доступ для чтения — например, чтение содержимого адресного пространства процесса. Уровень No-Read-Up не помешает открыть процесс с более высоким уровнем целостности из процесса с более низким уровнем для более ограниченного доступа (например, для запроса `PROCESS_QUERY_LIMITED_INFORMATION`, предоставляющего базовую информацию о процессе).

ИЗОЛЯЦИЯ ПРИВИЛЕГИЙ ПОЛЬЗОВАТЕЛЬСКОГО ИНТЕРФЕЙСА

Имеющаяся в Windows подсистема сообщений также соблюдает уровни целостности для реализации изоляции привилегий пользовательского интерфейса — User Interface Privilege Isolation (UIPI). Подсистема реализует эту изоляцию тем, что мешает процессу отправлять оконные сообщения окнам, которыми владеет процесс с более высоким уровнем целостности, за исключением следующих информационных сообщений:

- `WM_NULL`
- `WM_MOVE`
- `WM_SIZE`
- `WM_GETTEXT`
- `WM_GETTEXTLENGTH`
- `WM_GETHOTKEY`
- `WM_GETICON`
- `WM_RENDERFORMAT`
- `WM_DRAWCLIPBOARD`
- `WM_CHANGECBCHAIN`
- `WM_THEMECHANGED`

Такое использование уровней целостности не дает процессам обычного пользователя вводить информацию в окна процессов с повышенными привилегиями или не дает проводить так называемые подрывные атаки — shatter attack (например, отправку процессу некорректно сформированных сообщений, вызывающих переполнение внутреннего буфера, что может привести к выполнению кода на уровне процесса с повышенными привилегиями). UIPI также блокирует работу

перехватчиков событий окон (SetWindowsHookEx API) от влияния на процессы с более высоким уровнем целостности, чтобы процесс обычного пользователя не мог регистрировать нажатия клавиш, которые, к примеру, набираются пользователем в административном приложении. Аналогичным образом блокируются также и перехватчики журналов, чтобы не позволить процессам с более низким уровнем целостности отслеживать поведение процессов с более высоким уровнем целостности.

Процессы (только со средним и более высоким уровнем целостности) могут разрешить дополнительным сообщениям пройти защиту путем вызова API-функции ChangeWindowMessageFilterEx. Обычно эта функция используется для добавления сообщений, требуемых пользовательским элементам управления для обмена данными за пределами вне обычных общих элементов управления Windows. Более старая API-функция ChangeWindowMessageFilter выполняет те же действия, но в отношении процесса, а не окна. При использовании функции ChangeWindowMessageFilter два пользовательских элемента управления внутри одного и того же процесса могут быть использованы одними и теми же внутренними оконными сообщениями, что может привести к разрешению потенциально опасного оконного сообщения в одном из элементов управления, просто потому, что оно будет только лишь сообщением запроса для другого пользовательского элемента управления.

Поскольку под ограничения UIPI подпадают такие приложения для упрощения работы с компьютером, как экранная клавиатура (Osk.exe) (что потребует от приложений доступности исполняться для каждой разновидности видимых процессов рабочего стола, имеющих уровни целостности), эти процессы могут включить доступ к пользовательскому интерфейсу. Этот флаг может присутствовать в файле манифеста, который принадлежит образу, и благодаря ему процесс будет запущен с немного более высоким уровнем целостности по сравнению со средним (между 0x2000 и 0x3000) под учетной записью обычного пользователя, или с высоким уровнем целостности, если запускается под учетной записью администратора. Следует иметь в виду, что во втором случае запрос на повышение полномочий не будет отображен. Чтобы процесс установил этот флаг, его образ должен быть подписан и находиться в одном из нескольких безопасных мест, включая %SystemRoot% и %ProgramFiles%.

После завершения проверки целостности и в предположении, что обязательная политика разрешает доступ к объекту на основе уровня целостности вызывающего процесса, для проверки избирательного доступа к объекту используется один из двух алгоритмов, который определит окончательный результат проверки возможности доступа.

- Определение максимального разрешенного доступа к объекту, форма которого экспортируется в пользовательский режим с помощью AuthZ API (см. раздел «AuthZ API» далее в этой главе) или более старой Windows-функции GetEffectiveRightsFromAcl. Это определение также используется, когда программа указывает в качестве желаемого доступа MAXIMUM_ALLOWED, для устаревших API-функций, у которых не используется параметр желаемого доступа.

- Проверка возможности конкретного желаемого доступа, что может быть сделано с помощью Windows-функции `AccessCheck` или `AccessCheckByType`.

Первый алгоритм проводит следующую проверку элементов DACL.

1. Если у объекта нет DACL (нулевой DACL), объект не имеет защиты и система безопасности предоставляет полный доступ — если только доступ не осуществляется из процесса AppContainer (см. раздел «AppContainer» далее в этой главе), что означает отсутствие доступа.
2. Если у вызывающего процесса есть привилегия получения объекта во владение (take-ownership), система безопасности предоставляет перед исследованием DACL доступ по записи в качестве владельца (write-owner). (Привилегии получения владения и доступа к записи в качестве владельца будут вскоре рассмотрены.)
3. Если вызывающий процесс является владельцем объекта, система ищет SID с правами владельца `OWNER_RIGHTS` и использует его в качестве SID для следующих действий. В противном случае предоставляются права управления чтением (read-control) и записи DACL (write-DACL).
4. Для каждого ACE-элемента запрещения доступа (access-denied), который содержит SID, совпадающий с SID-идентификатором маркера доступа вызывающего процесса, маска доступа ACE-элемента удаляется из маски предоставляемого доступа (granted-access mask).
5. Для каждого ACE-элемента разрешения доступа, который содержит SID, совпадающий с SID-идентификатором маркера доступа вызывающего процесса, маска доступа ACE-элемента добавляется к вычисленной маске предоставляемого доступа, если только этот доступ уже не был запрещен.

Когда будут проверены все элементы DACL, вычисленная маска предоставляемого доступа возвращается вызывающему процессу в виде максимально разрешенного доступа к объекту. Эта маска представляет собой совокупный набор типов доступа, который вызывающий процесс может успешно запросить при открытии объекта.

Предыдущее описание применимо только к форме алгоритма режима ядра. Windows-версия, реализуемая функцией `GetEffectiveRightsFromAcl`, отличается тем, что она не выполняет действие 2 и рассматривает не маркер доступа, а SID отдельного пользователя или группы.

ПРАВА ВЛАДЕЛЬЦА

Поскольку владельцы объекта могут, как правило, отменить настройки безопасности объекта за счет безусловного получения прав на управление чтением и на запись DACL, специализированные методы управления этим поведением выражаются в Windows с помощью SID прав владельца — Owner Rights SID.

SID прав владельца создается по двум основным причинам:

- **Для усиления защиты служб в операционной системе.** Каждый раз, когда служба создает объект во время выполнения, с этим объектом связывается SID владельца той учетной записи, под которой выполняется служба (например, локальной системы или локальной службы), а не фактический SID службы. Это означает, что любая другая служба той же учетной записи будет иметь доступ к объекту. SID прав владельца предотвращает это нежелательное поведение.
- **Для достижения большей гибкости при определенных сценариях использования.** Предположим, например, что администратор хочет разрешить пользователям создавать файлы и папки без модификации ACL-списков таких объектов. (Пользователи могут случайно или преднамеренно предоставить доступ к этим файлам или папкам с нежелательных учетных записей.) Используя наследуемый SID прав владельца, пользователям можно помешать редактировать или даже просматривать ACL создаваемых ими объектов. Второй сценарий использования относится к групповым изменениям. Предположим, что работник, который входит в некую группу, имеющую доступ к закрытой информации, создал несколько файлов, будучи участником этой группы, а теперь он удален из этой группы по каким-то деловым соображениям. Поскольку этот работник все еще остается пользователем, он все еще может обращаться к конфиденциальным файлам.

Второй алгоритм используется для определения того, может ли быть предоставлен конкретный запрашиваемый доступ на основе маркера доступа вызывающего процесса. У каждой функции открытия в Windows API, работающей с защищенными объектами, имеется параметр, который указывает требуемую маску доступа — последний компонент уравнения безопасности. Чтобы определить, имеет ли вызывающий процесс доступ, выполняются следующие действия.

1. Если у объекта нет DACL (нулевой DACL), объект не имеет защиты и система безопасности предоставляет желаемый доступ.
2. Если у вызывающего процесса есть привилегия получения объекта во владение (take-ownership), система безопасности предоставляет перед исследованием DACL доступ по записи в качестве владельца (write-owner). Но если доступ по записи в качестве владельца был единственным запрошенным доступом со стороны вызывающего процесса, имеющего привилегии получения объекта во владение, система безопасности предоставляет этот доступ и не исследует DACL.
3. Если вызывающий процесс является владельцем объекта, система ищет SID с правами владельца — `OWNER_RIGHTS` — и использует его в качестве SID для следующих действий. В противном случае предоставляются права управления чтением (read-control) и записи DACL (write-DACL). Если эти права были единственными правами, запрошенными вызывающим процессом, доступ предоставляется без проверки DACL.

4. Проверяется каждый ACE-элемент DACL с первого до последнего. В случае удовлетворения одному из следующих условий ведется обработка ACE:
   - ACE является запрещающим доступ, и SID в ACE совпадает со включенным SID (SID-идентификаторы могут быть включенными и выключенными) или с SID только для запрета в маркере доступа вызывающего процесса.
   - ACE является разрешающим доступ, и SID в ACE совпадает со включенным SID в маркере доступа вызывающего процесса, который не относится к типу маркеров только для запрета.
   - Это второй проход по дескриптору для проверки на наличие ограниченных SID, и SID в ACE совпадает с ограниченным SID в маркере доступа вызывающего процесса.
   - ACE не помечен как предназначенный только для наследования.
5. Если ACE является разрешающим доступ, то в маске доступа в ACE предоставляются запрошенные права доступа; если были предоставлены все запрошенные права, то проверка доступа достигает своей цели. Если ACE является запрещающим доступ и какое-либо из запрошенных прав доступа относится к запрещенным правам доступа, доступ к объекту запрещается.
6. Если проверка дошла до конца DACL и некоторые из запрошенных прав доступа все еще не были предоставлены, доступ запрещается.
7. Если предоставляются все доступы, но в маркере доступа вызывающего процесса есть хотя бы один ограниченный SID, система повторно сканирует ACE-элементы DACL, проводя поиск таких элементов, у которых маска доступа совпадает с запрошенными пользователем правами доступа, и поиск совпадения SID ACE-элемента с любым ограниченным SID вызывающего процесса. Доступ к объекту предоставляется пользователю только в том случае, если запрошенные права доступа предоставляются при обоих сканированиях DACL.

Поведение обоих алгоритмов проверки прав доступа зависит от относительного порядка следования разрешающих и запрещающих ACE-элементов. Рассмотрим объект, у которого имеется только два ACE-элемента, где один ACE указывает, что конкретному пользователю разрешен полный доступ к объекту, а другой ACE запрещает пользовательский доступ. Если разрешающий ACE предшествует запрещающему, пользователь может получить полный доступ к объекту, но при обратном порядке пользователь не может получить доступ к объекту.

Некоторые функции Windows, например `SetSecurityInfo` и `SetNamedSecurityInfo`, применяют ACE-элементы в особом порядке: явно запрещающие ACE-элементы предшествуют явно разрешающим. Например, эти функции используются диалоговыми окнами редактирования безопасности, с помощью которых вы, к примеру, редактируете права доступа к NTFS-файлам и разделам реестра. Функции `SetSecurityInfo` и `SetNamedSecurityInfo` также применяют правила наследования ACE к дескриптору безопасности, в отношении которого они применяются.

На рис. 7.9 показан пример проверки доступа, показывающий важность порядка следования ACE. В этом примере запрещен доступ к желаемому пользователем открытию файла даже притом, что ACE-элемент в DACL объекта предоставляет доступ, поскольку ACE, запрещающий пользовательский доступ (в силу его принадлежности к группе Writers), предшествует ACE, предоставляющего доступ.

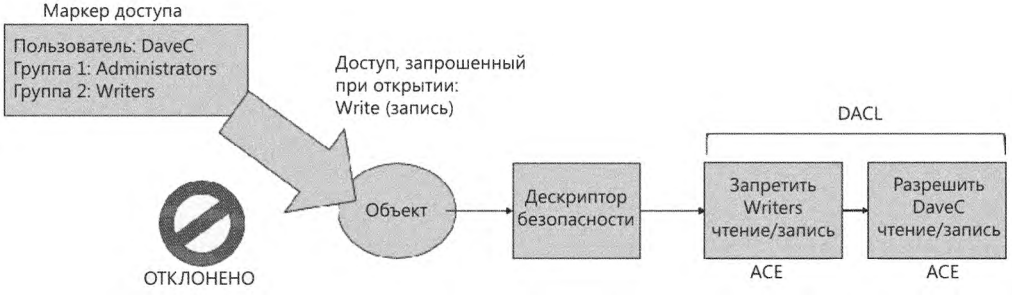


Рис. 7.9. Пример проверки прав доступа

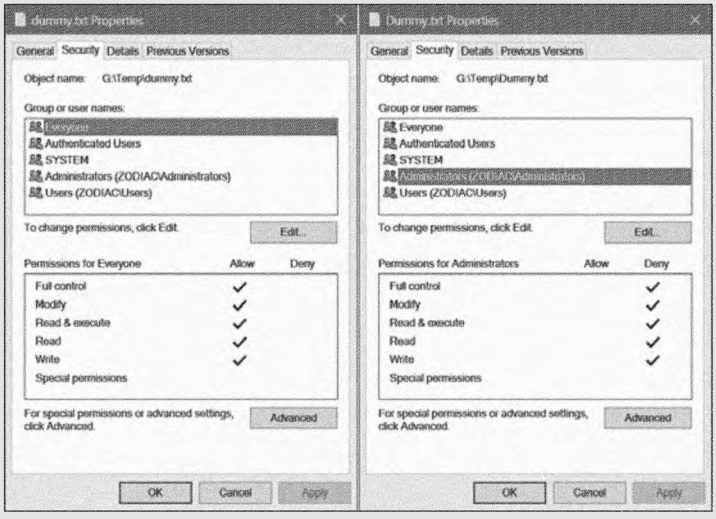
Как упоминалось выше, поскольку обрабатывать системе безопасности DACL при каждом использовании процессом дескриптора будет слишком неэффективно, SRM проводит эту проверку прав доступа только при открытии дескриптора, а не при каждом использовании. Следовательно, если процесс успешно открывает дескриптор, система безопасности не может отменить предоставленные права доступа, даже при изменении DACL объекта. Также следует иметь в виду, что, поскольку код режима ядра использует для доступа к объектам указатели, а не дескрипторы, при использовании объектов операционной системой проверка прав доступа не производится. Иными словами, исполняющая система Windows доверяет самой себе (и всем загруженным драйверам) в смысле безопасности.

Тот факт, что владельцу объекта всегда предоставляется доступ к записи DACL этого объекта, означает, что пользователи ни при каких условиях не могут быть отстранены от доступа к объектам, владельцами которых они являются. Если же по каким-то причинам у объекта есть пустой DACL (отсутствие доступа), владелец все равно может открыть объект с доступом к DACL по записи, а затем применить к нему новый DACL с нужными правами доступа.

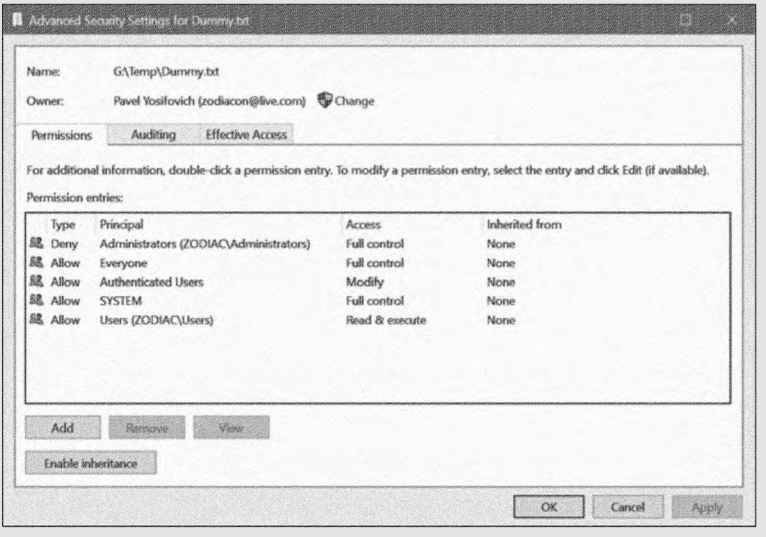
ПРЕДУПРЕЖДЕНИЕ, КАСАЮЩЕЕСЯ РЕДАКТОРОВ БЕЗОПАСНОСТИ С ГРАФИЧЕСКИМ ИНТЕРФЕЙСОМ

При использовании редакторов прав доступа с графическим интерфейсом для настройки файловых объектов, объектов реестра или Active Directory или какого-нибудь другого защищаемого объекта главное диалоговое окно настроек безопасности показывает вам представление настроек, примененных к объекту, и это представление может сбить с толку. Если вы разрешите Полный доступ (Full Control) для группы Все (Everyone) и запретите его для группы Админи-

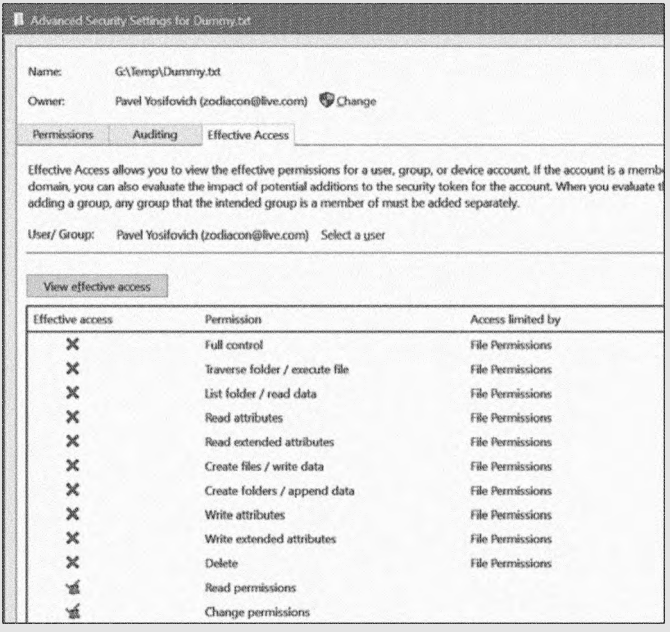
страторы (Administrator), список может заверить вас, что разрешающий доступ ACE-элемент группы Все (Everyone) предшествует запрещающему доступ ACE-элементу группы Администраторы (Administrator), поскольку это порядок, в котором они появляются. Но, как уже было сказано, редакторы применительно к ACL объекта помещают запрещающие доступ ACE до разрешающих доступ ACE.



Вкладка Разрешения (Permissions) диалогового окна Дополнительные параметры безопасности (Advanced Security Settings) показывает порядок следования ACE-элементов в DACL. Но даже это диалоговое окно может внести путаницу, поскольку в составных DACL запрещающие доступ ACE-элементы для различных доступов могут следовать за разрешающими ACE-элементами для других типов доступа.



Единственный надежный способ узнать о разрешениях доступов конкретного пользователя или группы к объекту (кроме того, чтобы попытаться получить от имени этого пользователя или группы доступ к объекту) заключается в использовании вкладки Действующие разрешения (Effective Permissions) диалогового окна, которая появляется после щелчка на кнопке Дополнительно (Advanced) в диалоговом окне Свойства (Properties). Введите имя проверяемого пользователя или группы, и в диалоговом окне будет показано, какие права доступа разрешены для них к объекту.



Динамическое управление доступом

Механизм избирательного управления доступом, описанный в предыдущих разделах, существовал еще с первой версии Windows NT и приносил пользу во многих обстоятельствах. Однако найдется немало ситуаций, в которых эта схема не обладает достаточной гибкостью. Например, представьте требование, согласно которому обращение пользователя к общему файлу должно быть разрешено с компьютера на рабочем месте, но не с его домашнего компьютера. Определить такое условие с использованием ACE не удастся.

В Windows 8 и Server 2012 появилось динамическое управление доступом, или *DAC* (Dynamic Access Control), — гибкий механизм, при помощи которого можно задавать правила на основании нестандартных атрибутов, определенных в Active Directory. DAC не заменяет существующий механизм, но дополняет его. Это означает, что для разрешения на выполнение операции должно быть получено как от

DAC, так и от классического механизма DACL. На рис. 7.10 представлены основные аспекты DAC.

Рис. 7.10. Компоненты динамического управления доступом

Требованием (claim) называется любой фрагмент информации о пользователе, устройстве (компьютере в домене) или ресурсе (обобщенный атрибут), опубликованный контроллером домена. Примеры допустимых требований — должность пользователя или отдел, к которому относится файл. Произвольная комбинация утверждений может использоваться в выражениях для построения правил. Такие правила в совокупности образуют *централизованную политику доступа*.

Конфигурация DAC определяется в Active Directory и распространяется через политику. Протокол билетов Kerberos был расширен для поддержки аутентифицированной передачи требований пользователей и устройств (это называется *защитой Kerberos*).

AuthZ API

Windows API-интерфейс AuthZ предоставляет функции авторизации и реализует такую же модель безопасности, что и монитор безопасности, но он реализует эту модель полностью в пользовательском режиме в библиотеке %SystemRoot%\System32\Authz.dll. Благодаря ему приложения, желающие защитить свои собственные закрытые объекты, например таблицы баз данных, получают возможность воспользоваться моделью безопасности Windows, не затрачиваясь на переходы между пользовательским режимом и режимом ядра, которые могли бы им понадобиться при расчете на монитор безопасности.

AuthZ API использует стандартные структуры данных дескриптора безопасности, SID-идентификаторы и привилегии. Вместо использования маркеров для представления клиентов, AuthZ использует `AUTHZ_CLIENT_CONTEXT`. AuthZ включает эквиваленты пользовательского режима для всех имеющихся в Windows функций проверок безопасности доступа, например функцию `AuthzAccessCheck` в версии AuthZ функции `AccessCheck` из Windows API, которая использует функцию `SeAccessCheck` монитора безопасности.

Другим преимуществом, доступным приложениям, использующим AuthZ, является то, что они могут заставить AuthZ кэшировать результаты проверок безопасности для упрощения последующих проверок, использующих тот же контекст и дескриптор безопасности клиента. Полная документация по AuthZ приведена в Windows SDK.

Этот тип проверки доступа, использующий идентификаторы безопасности (SID) и участие в группах безопасности в статической, контролируемой среде, известен как управление доступом, основанное на избирательном подходе — *IBAC* (Identity-Based Access Control), и он требует, чтобы система безопасности различала идентичность каждого возможного средства доступа, когда в дескриптор безопасности объекта включается DACL.

Windows включает поддержку управления доступом на основе заявок — *CBAC* (Claims Based Access Control), где доступ предоставляется не на основе идентичности или групповой принадлежности стороны, запрашивающей доступ, а на основе произвольных атрибутов, назначаемых стороне доступа и сохраняемых в ее маркере доступа. Атрибуты предоставляются поставщиком атрибутов, например AppLocker. Механизм CBAC обладает многими преимуществами, включая возможность создания DACL для пользователя, чья идентичность еще не известна, или динамически вычисляемые атрибуты пользователя.

CBAC ACE-элемент (известный также как условный ACE) хранится в ACE-структуре `*-callback`, которая полностью относится к AuthZ и игнорируется системной API-функцией `SeAccessCheck`. Процедура режима ядра `SeSrpAccessCheck` не понимает условных ACE-элементов, поэтому использовать CBAC могут только приложения, вызывающие API-функции AuthZ. Единственным системным компонентом, использующим CBAC для установки таких атрибутов, как путь или издатель, является AppLocker. Сторонние приложения используют CBAC, вызывая API-функции CBAC из интерфейса AuthZ.

Использование проверок безопасности CBAC позволяет применять эффективные политики управления, к числу которых относятся:

- запуск только тех приложений, которые утверждены IT-отделом;
- разрешение только одобренным приложениям получать доступ к контактам вашего Microsoft Outlook или календаря;
- разрешение только сотрудникам определенного этажа здания получать доступ к принтерам этого этажа;
- разрешение доступа к веб-сайтам внутренней сети только штатным работникам (в отличие от подрядчиков).

На атрибуты можно ссылаться в так называемом *условном ACE-элементе*, где проверяется присутствие, отсутствие или значение одного или нескольких атрибутов. Имя атрибута может состоять из любых алфавитно-цифровых символов Юникода (Unicode), а также из символов :, /, _. В качестве значения могут использоваться: 64-разрядное целое число, строка Юникода, строка байтов или массив.

Условные ACE-элементы

Формат строк языка определения дескрипторов безопасности — SDDL (Security Descriptor Definition Language) был расширен для поддержки ACE-элементов с условными выражениями. Новый формат строки SDDL имеет следующий вид: `AceType;AceFlags;Rights;ObjectGuid;InheritObjectGuid;Account Sid;(Conditional-Expression)`.

Типом ACE (`AceType`) для условного ACE является либо `XA` (для разрешающего `SDDL_CALLBACK_ACCESS_ALLOWED`), либо `XD` (для запрещающего `SDDL_CALLBACK_ACCESS_DENIED`). Следует заметить, что ACE-элементы с условными выражениями используются для авторизации по заявке (конкретно функциями AuthZ API и AppLocker) и не распознаются диспетчером объектов или файловыми системами.

Условное выражение может включать любые элементы, показанные в табл. 7.9.

Таблица 7.9. Допустимые элементы для условных выражений

Элемент выражения	Описание
`Имя_атрибута`	Проверяет наличие у конкретного атрибута ненулевого значения
`exists Имя_атрибута`	Проверяет существование указанного атрибута в контексте клиента
`Имя_атрибута оператор значение`	Возвращает результат указанной операции. Для тестирования значений атрибутов в условных выражениях используются следующие бинарные (в отличие от унарных) операторы, имеющие форму Имя_атрибута оператор значение: `Contains any_of , ==, !=, <, <=, >, >=`
`Условное_выражение // Условное_выражение`	Проверяет, не имеет ли любое из указанных условных выражений значение `true`
`Условное_выражение && Условное_выражение`	Проверяет, не имеют ли оба из указанных условных выражений значение `true`
`! (Условное_выражение)`	Превращает условное выражение в его противоположность
`Member_of {SidArray}`	Проверяет, содержит ли массив `SID_AND_ATTRIBUTES`, относящийся к контексту клиента, все идентификаторы безопасности (SID) в списке с запятыми в качестве разделителей, определенном с помощью `SidArray`

Условный ACE-элемент может содержать любое количество условий, и он либо игнорируется, если в результате вычисления условия будет получено значение `false`, либо применяется, если результат равен `true`. Условный ACE может быть добавлен к объекту с использованием API-функции `AddConditionalAce` и проверен с помощью API-функции `AuthzAccessCheck`.

Условный ACE может указать, какой доступ к конкретным записям данных внутри программы должен быть предоставлен только тому пользователю, который отвечает следующим критериям.

- У него имеется атрибут `Role` (роль) со значением `Architect` (создатель), `Program Manager` (Руководитель программы) или `Development Lead` (ведущий разработчик) и атрибут `Division` (подразделение) со значением Windows.
- Его атрибут `ManagementChain` (цепочка управления) содержит значение `John Smith`.
- Его атрибут `CommissionType` (тип полномочий) имеет значение `Officer` (служащий), а значение его атрибута `PayGrade` (разряд заработной платы) выше 6 (т. е. соответствует званию генерала — `General Officer` в вооруженных силах США).

В составе Windows нет средств для просмотра и редактирования условных ACE-элементов.

Права доступа и привилегии

Многие операции, выполняемые процессами в ходе их работы, не могут быть авторизованы через защиту доступа к объекту, поскольку они не касаются взаимодействия с конкретным объектом. Например, возможность обхода проверок безопасности при открытии файлов для резервного копирования является свойством учетной записи, а не конкретного объекта. Чтобы разрешить системному администратору управлять тем, под какими учетными записями можно выполнять связанные с безопасностью операции, в Windows используются как привилегии, так и права доступа.

Привилегии являются правом выполнять под той или иной учетной записью конкретную, связанную с системой операцию, например выключение компьютера или изменение системного времени. *Право учетной записи* разрешает или запрещает той учетной записи, которой оно назначено, выполнять конкретный тип входа в систему, например локальный или интерактивный.

Системный администратор назначает привилегии группам и учетным записям, используя такие инструментальные средства, как MMC-оснастка «Пользователи и группы Active Directory» — Active Directory Users and Groups для доменных учетных записей или редактор локальной политики безопасности — Local Security Policy Editor (%SystemRoot%\System32\secpol.msc). На рис. 7.11 показана конфигурация Назначение прав пользователя (User Rights Assignment) в редакторе локальной политики безопасности, где выведен полный список привилегий и прав учетных записей, доступных в Windows. Следует заметить, что это средство не делает различий между привилегиями и правами учетных записей. Тем не менее их можно отличить друг от друга, поскольку любые права пользователя, не содержащие слова «вход в» (log on), являются привилегиями учетных записей.

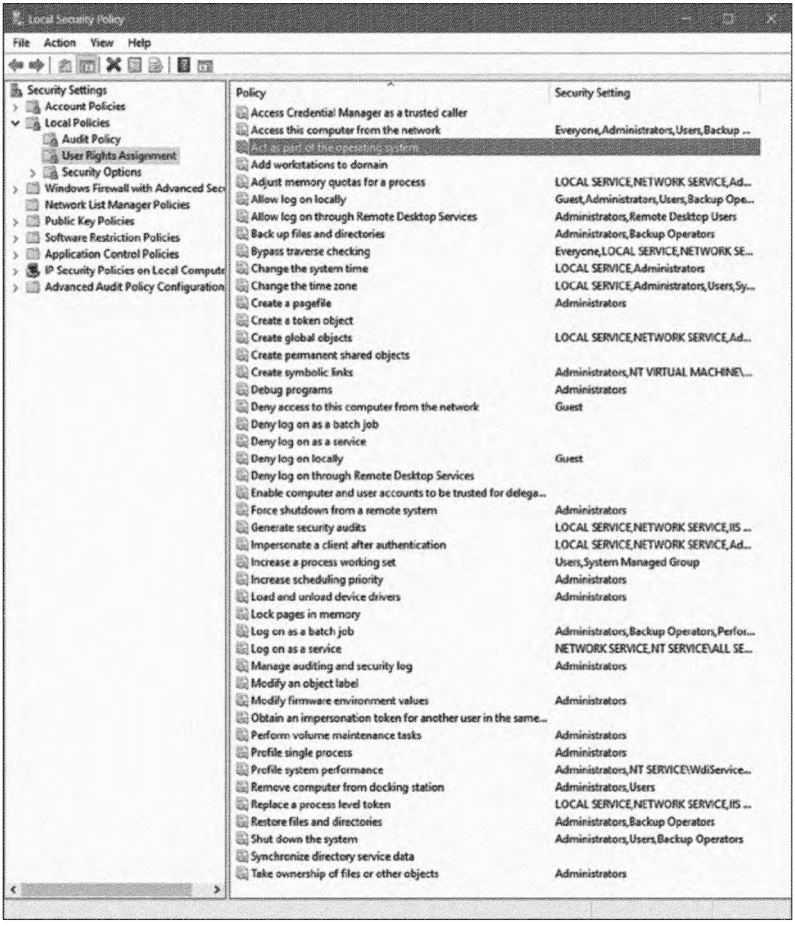


Рис. 7.11. Назначение прав пользователей через редактор локальной политики безопасности

Права учетной записи

Права учетной записи не обеспечиваются монитором безопасности и не хранятся в маркерах доступа. Функцией, отвечающей за вход в систему, является `LsaLogonUser`. Служба Winlogon, к примеру, вызывает API-функцию `LogonUser` при интерактивном входе пользователя на компьютер, а функция `LogonUser` вызывает функцию `LsaLogonUser`. Функции `LogonUser` передается параметр, включающий тип выполняемого входа в систему, который может быть интерактивным, сетевым, пакетным, служебным и в качестве клиента терминального сервера.

В ответ на запросы входа в систему механизм проверки подлинности локальной системы безопасности — LSA (Local Security Authority) извлекает права учетной

записи, назначенные пользователю, из базы данных политики LSA при попытке пользователя войти в систему. LSA сверяет тип входа в систему с правами учетной записи, назначенными той учетной записи, под которой происходит вход в систему, и отклоняет вход, если у учетной записи нет прав, разрешающих данный тип входа в систему. В табл. 7.10 перечислены права пользователя, определяемые системой Windows.

Таблица 7.10. Права учетной записи

Пользовательское право	Роль
Deny logon locally (запрет на локальный вход в систему), Allow logon locally (разрешение на локальный вход в систему)	Используется для интерактивных входов в систему, происходящих на локальной машине
Deny logon over the network (запрет на вход в систему по сети), Allow logon over the network (разрешение на вход в систему по сети)	Используется для входов в систему, происходящих на удаленной машине
Deny logon through Terminal Services (запрет на вход в систему через службу терминалов), Allow logon through Terminal Services (разрешение на вход в систему через службу терминалов)	Используется для входов в систему в качестве клиента службы терминалов
Deny logon as a service (запрет на вход в систему в качестве службы), Allow logon as a service (разрешение на вход в систему в качестве службы)	Используется диспетчером управления службами при запуске той или иной службы под конкретной учетной записью
Deny logon as a batch job (запрет на вход в систему в качестве пакетного задания), Allow logon as a batch job (разрешение на вход в систему в качестве пакетного задания)	Используется при выполнении входа в систему пакетного типа

Windows-приложения с помощью функций `LsaAddAccountRights` и `LsaRemove` AccountRights могут добавить или удалить права пользователя из учетной записи, а с помощью функции `LsaEnumerateAccountRights` они могут определить, какие права назначены учетной записи.

Привилегии

Количество привилегий, определяемых операционной системой, со временем существенно увеличилось. В отличие от прав пользователя, которые задаются LSA в одном месте, различные привилегии определяются разными компонентами, которые их же и применяют. Например, привилегия отладки, позволяющая процессу обходить проверки безопасности при открытии дескриптора другого процесса с помощью API-функции Windows `OpenProcess`, проверяется диспетчером процесса.

В табл. 7.11 представлен полный список привилегий и дано описание того, как и когда они проверяются системными компонентами. Для каждой привилегии

в заголовках SDK определяется макрос вида `SE_`*привилегия*`_NAME`, где *привилегия* — константа привилегии (например, `SE_DEBUG_NAME` для привилегии отладки). Эти макросы определяются в виде строк, которые начинаются с префикса `Se` и заканчиваются суффиксом `Privilege`, как в `SeDebugPrivilege`. Вроде бы это указывает на то, что привилегии идентифицируются строками, но на самом деле они идентифицируются LUID-кодами, которые, естественно, уникальны для текущей загрузки. Каждое обращение к привилегии требует поиска правильного LUID-кода с вызовом функции `LookupPrivilegeValue`. Однако следует учесть, что в Ntdll и коде ядра привилегии могут идентифицироваться целочисленными константами напрямую, без использования LUID.

Таблица 7.11. Привилегии

Привилегия	Право пользователя	Использование привилегии
`SeAssignPrimaryTokenPrivilege`	Замена маркера уровня процесса	Проверяется различными компонентами, задающими маркер процесса (например, `NtSetInformationJobObject`)
`SeAuditPrivilege`	Генерирование аудита безопасности	Необходима для генерирования событий журнала событий безопасности с использованием API `ReportEvent`
`SeBackupPrivilege`	Резервное копирование файлов и каталогов	Заставляет NTFS предоставлять следующий доступ к любому файлу или каталогу независимо от присутствующего дескриптора безопасности: `READ_CONTROL`, `ACCESS_SYSTEM_SECURITY`, `FILE_GENERIC_READ` и `FILE_TRAVERSE`. Учтите, что при открытии файла для резервного копирования вызывающая сторона должна указать флаг `FILE_FLAG_BACKUP_SEMANTICS`. Также обеспечивает соответствующий доступ к разделам реестра при использовании `RegSaveKey`
`SeChangeNotifyPrivilege`	Обход проверки при перемещениях	Используется NTFS для отказа от промежуточной проверки разрешений для промежуточных каталогов при многоуровневом просмотре каталогов. Также используется файловыми системами при регистрации приложений на уведомления об изменениях структуры файловой системы
`SeCreateGlobalPrivilege`	Создание глобальных объектов	Требуется для создания процессом объектов разделов и символических ссылок в каталогах пространства имен диспетчера объектов, назначенных сеансу, отличному от сеанса вызывающей стороны

Таблица 7.11 (*продолжение*)

Привилегия	Право пользователя	Использование привилегии
SeCreatePagefilePrivilege	Создание страничного файла	Проверяется NtCreatePagingFile — функцией, используемой для создания нового страничного файла
SeCreatePermanentPrivilege	Создание долгосрочных общих объектов	Проверяется диспетчером объектов при создании долгосрочного объекта (такого, который не уничтожается при отсутствии ссылок на него)
SeCreateSymbolicLinkPrivilege	Создание символических ссылок	Проверяется NTFS при создании символических ссылок в файловой системе с использованием API CreateSymbolicLink
SeCreateTokenPrivilege	Создание объекта маркера	Привилегия проверяется NtCreateToken — функцией, которая создает объект маркера
SeDebugPrivilege	Программы отладки	Если вызывающая сторона обладает этой привилегией, диспетчер процессов разрешает доступ к любому процессу или потоку с использованием NtOpenProcess или NtOpenThread независимо от дескриптора безопасности процесса или потока (кроме защищенных процессов)
SeEnableDelegationPrivilege	Обеспечение работы механизма доверия для компьютеров и учетных записей в отношении делегирования	Используется службами Active Directory для делегирования аутентифицированных учетных данных
SeImpersonatePrivilege	Заимствование прав клиента после аутентификации	Проверяется диспетчером процессов, когда поток хочет использовать маркер для заимствования прав, а маркер представляет пользователя, отличного от пользователя маркера процесса текущего потока
SeIncreaseBasePriorityPrivilege	Повышение приоритета планирования	Проверяется диспетчером процессов и требуется для повышения приоритета процесса
SeIncreaseQuotaPrivilege	Регулировка квот памяти для процесса	Проверяется при изменении порогов рабочих наборов процесса, квот выгружаемого и невыгружаемого пулов процесса, а также квоты загрузки процессоров

Привилегия	Право пользователя	Использование привилегии
SeIncreaseWorkingSetPrivilege	Расширение рабочего набора процесса	Требуется для вызова SetProcessWorkingSetSize с целью повышения минимального размера рабочего набора. Это косвенно позволяет процессу зафиксировать минимальный рабочий набор в памяти с использованием VirtualLock
SeLoadDriverPrivilege	Загрузка и выгрузка драйверов устройств	Проверяется функциями NtLoadDriver и NtUnloadDriver
SeLockMemoryPrivilege	Блокировка страниц в физической памяти	Проверяется NtLockVirtualMemory, реализацией функции VirtualLock из режима ядра
SeMachineAccountPrivilege	Добавление рабочих станций в домен	Проверяется диспетчером безопасности учетных записей для контроллера домена при создании учетной записи машины в домене
SeManageVolumePrivilege	Выполнение операций сопровождения с томом	Проверяется драйверами файловой системы во время операции открытия тома; это необходимо для выполнения операций проверки диска и дефрагментации
SeProfileSingleProcessPrivilege	Профилирование одного процесса	Проверяется механизмом супервыборки и предварительной выборки при запросе информации для отдельного процесса с использованием API-функции NtQuerySystemInformation
SeRelabelPrivilege	Изменение метки объекта	Проверяется SRM при поднятии уровня целостности объекта, принадлежащего другому пользователю, или при попытке поднять уровень целостности объекта выше уровня в маркере вызывающей стороны
SeRemoteShutdownPrivilege	Принудительное выключение из удаленной системы	Winlogon проверяет, что удаленная вызывающая сторона функции InitiateSystemShutdown обладает такой привилегией
SeRestorePrivilege	Восстановление файлов и каталогов	Привилегия заставляет NTFS предоставить следующий доступ к любому файлу или каталогу независимо от действующего дескриптора безопасности: WRITE_DAC, WRITE_OWNER, ACCESS_SYSTEM_SECURITY, FILE_GENERIC_WRITE, FILE_ADD_FILE, FILE_ADD_SUBDIRECTORY и DELETE. Учтите, что при открытии

Таблица 7.11 (*продолжение*)

Привилегия	Право пользователя	Использование привилегии
		файла для восстановления вызывающая сторона должна указать флаг FILE_FLAG_BACKUP_SEMANTICS. Разрешает соответствующий доступ к разделам реестра при использовании RegSaveKey
SeSecurityPrivilege	Управление аудитом и журналом безопасности	Необходимо для обращения к списку SACL дескриптора безопасности, чтения и очистки журнала событий безопасности
SeShutdownPrivilege	Выключение системы	Проверяется NtShutdownSystem и функцией NtRaiseHardError, выводящей диалоговое окно системной ошибки на интерактивной консоли
SeSyncAgentPrivilege	Синхронизация данных службы каталогов	Необходимо для использования службы синхронизации каталогов LDAP. Позволяет держателю читать все объекты и свойства в каталоге независимо от защиты объектов и свойств
SeSystemEnvironmentPrivilege	Изменение переменных окружения прошивки	Требуется функциям NtSetSystemEnvironmentValue и NtQuerySystemEnvironmentValue для модификации и чтения переменных окружения прошивки с использованием уровня аппаратных абстракций (HAL)
SeSystemProfilePrivilege	Профилирование производительности системы	Проверяется NtCreateProfile — функцией, используемой для профилирования системы. Например, эта функция используется утилитой Kernprof
SeSystemtimePrivilege	Изменение системного времени	Требуется для изменения даты или времени
SeTakeOwnershipPrivilege	Получение прав владения для файлов и других объектов	Требуется для получения прав владения объекта без предоставления избирательного доступа
SeTcbPrivilege	Функционирование в качестве части операционной системы	Проверяется SRM при назначении идентификатора сеанса в маркере, диспетчером PnP — для создания и управления событиями PnP, функцией BroadcastSystemMessageEx — при вызове с флагом BSM_ALLDESKTOPS, функцией LsaRegisterLogonProcess и при назначении приложения в качестве VDM функцией NtSetInformationProcess

Привилегия	Право пользователя	Использование привилегии
SeTimeZonePrivilege	Изменение часового пояса	Требуется для изменения часового пояса
SeTrustedCredManAccessPrivilege	Обращение к диспетчеру учетных данных в качестве доверенной вызывающей стороны	Проверяется диспетчером учетных данных для проверки того, что вызывающая сторона заслуживает доверия с учетными данными, которые могут запрашиваться в простом текстовом виде. По умолчанию предоставляется только Winlogon
SeUndockPrivilege	Отсоединение компьютера от док-станции	Проверяется диспетчером PnP пользовательского режима при отстыковке компьютера или при запросе на извлечение устройства
SeUnsolicitedInputPrivilege	Получение непредусмотренных данных от терминального устройства	В настоящее время эта привилегия не используется Windows

Когда компоненту нужно проверить маркер с целью определения наличия привилегии, им используются API-функции `PrivilegeCheck` или `LsaEnumerateAccountRights` при работе в пользовательском режиме и `SeSinglePrivilegeCheck` или `SePrivilegeCheck` при работе в режиме ядра. API-функции, связанные с привилегиями, ничего не знают о существовании прав учетных записей, а вот API-функции, связанные с правами учетных записей, о наличии привилегий осведомлены.

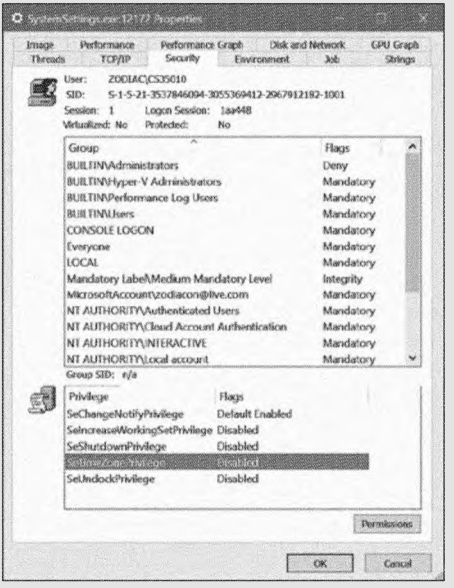
В отличие от прав учетных записей, привилегии могут быть включены или выключены. Чтобы проверка привилегии прошла успешно, она должна присутствовать в указанном маркере и быть включена. В основу этой схемы положена идея о том, что привилегии должны быть включены только при их востребованности, чтобы процесс не мог по недосмотру выполнить привилегированную небезопасную операцию. Включение и выключение привилегий осуществляется функцией `AdjustTokenPrivileges`.

ЭКСПЕРИМЕНТ: НАБЛЮДЕНИЕ ЗА ВКЛЮЧЕНИЕМ ПРИВИЛЕГИИ

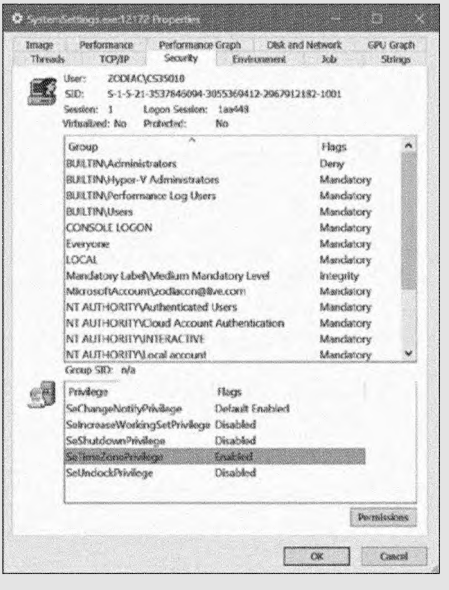
Выполняя следующие действия, можно видеть, что приложение Панели управления Дата и время (Date and Time) включает привилегию SeTimeZonePrivilege в ответ на ваше использование ее интерфейса для изменения часового пояса компьютера (Windows 10).

1. Запустите Process Explorer с повышенными привилегиями.
2. Щелкните правой кнопкой мыши на часах в области уведомлений и выберите команду Настройка даты и времени (Adjust Date/Time).

3. Щелкните правой кнопкой мыши на процессе SystemSettings.exe в Process Explorer и выберите команду Properties. Перейдите на вкладку Security в диалоговом окне Properties. Вы увидите, что привилегия `SeTimeZonePrivilege` отключена.



4. Измените часовой пояс, закройте диалоговое окно Properties и откройте его снова. На вкладке Security вы увидите, что привилегия `SeTimeZonePrivilege` включена.



ЭКСПЕРИМЕНТ: ПРИВИЛЕГИЯ ОБХОДА ПРОМЕЖУТОЧНЫХ ПРОВЕРОК

Системные администраторы должны знать о привилегии обхода промежуточных проверок — Bypass Traverse Checking (которая внутри системы называется SeNotifyPrivilege) и последствиях ее использования. В данном эксперименте демонстрируется, как непонимание ее поведения может привести к нарушению режима безопасности.

1. Создайте папку, а внутри этой папки создайте новый текстовый файл с каким-нибудь текстом.
2. Перейдите в Проводнике к новому файлу и откройте в его диалоговом окне Свойства (Properties) вкладку Безопасность (Security).
3. Щелкните на кнопке Дополнительно (Advanced).
4. Снимите флажок Наследование (Inheritance).
5. Когда появится предложение удалить или скопировать права наследования, выберите Копировать (Copy).
6. Теперь измените безопасность новой папки, чтобы у вашей учетной записи не было никакого доступа к папке. Для этого выберите свою учетную запись, а затем выберите все флажки Запретить (Deny) в списке разрешений.
7. Запустите Блокнот. В меню Файл (File) выберите команду Открыть (Open) и перейдите к новому каталогу в открывшемся диалоговом окне. Доступ к новому каталогу должен быть запрещен.
8. В поле Имя файла (File Name) диалогового окна Открыть (Open) наберите полный путь к новому файлу. Файл должен открыться.

Если у вашей учетной записи нет привилегии обхода промежуточных проверок, NTFS выполняет проверку доступа к каждому каталогу пути при попытке открыть файл, что в данном примере приведет к запрещению доступа к файлу.

Суперпривилегии

Некоторые привилегии настолько влиятельны, что пользователь, которому они назначены, фактически является «суперпользователем», имеющим полный контроль над компьютером. Эти привилегии могут использоваться несчетным количеством способов для получения несанкционированного доступа к неограниченным в иной ситуации ресурсам и к выполнению несанкционированных операций. Но мы сосредоточимся на использовании привилегии выполнения кода, предоставляющего пользователю не назначенные ему привилегии, имея в виду, что этой возможностью можно воспользоваться для выполнения на локальной машине любой нужной пользователю операции.

В этом разделе дается перечень привилегий и рассматриваются способы их возможных применений. Другие привилегии, например блокировка страниц в физической памяти (`SeLockMemoryPrivilege`), могут применяться в системе для атак

«отказа в обслуживании» (denial-of-service attacks), но такие привилегии здесь не рассматриваются. Следует заметить, что системы с включенным контролем учетных записей (UAC) такие привилегии будут предоставлять только приложениям, запущенным на высоком уровне целостности (high) или выше, даже если учетная запись ими обладает.

- **Проводить отладку программ** (`SeDebugPrivilege`). Пользователь с этой привилегией может открывать любой процесс в системе (кроме защищенного), не обращая внимания на имеющийся у процесса дескриптор безопасности. Например, пользователь может выполнять программу, которая открывает LSASS-процесс, копирует исполняемый код в его адресное пространство, а затем с помощью Windows API-функции `CreateRemoteThread` внедряет поток для выполнения внедренного кода в более привилегированном контексте безопасности. Код может предоставлять пользователю дополнительные привилегии и членство в группах.

- **Приобретать права владения** (`SeTakeOwnershipPrivilege`). Эта привилегия позволяет ее владельцу приобретать права владения любым защищаемым объектом (даже защищенными потоками и процессами) путем записи своего собственного SID в поле владельца (owner) дескриптора безопасности объекта. Следует напомнить, что владельцу всегда предоставляются права на чтение и внесение изменений в DACL дескриптора безопасности, поэтому процесс с этой привилегией может вносить изменения в DACL, предоставляя самому себе полный доступ к объекту, а затем закрывать и снова открывать объект с полным доступом. Это позволит владельцу просматривать конфиденциальные данные и даже заменять системные файлы, выполняющие часть обычной системной операции, например LSASS, своими собственными программами, предоставляющими повышенные пользовательские привилегии.

- **Восстанавливать файлы и каталоги** (`SeRestorePrivilege`). Пользователь, получивший эту привилегию, может заменять любые файлы в системе своими собственными файлами. Он может воспользоваться этой властью путем замены системных файлов, как описывалось в предыдущем пункте.

- **Загружать и выгружать драйверы устройств** (`SeLoadDriverPrivilege`). Злоумышленник может воспользоваться этой привилегией для загрузки в систему драйверов устройств. Эти драйверы рассматриваются в качестве доверенной части операционной системы, которые могут выполняться внутри этой системы с правами учетной записи System, поэтому драйвер может запускать привилегированные программы, назначающие пользователю дополнительные права.

- **Создавать объект маркера** (`SeCreateTokenPrivilege`). Эта привилегия может использоваться в соответствии с предназначением для создания маркеров, представляющих произвольные учетные записи пользователей с произвольной принадлежностью к той или иной группе и назначенными привилегиями.

- **Действовать в качестве части операционной системы** (`SeTcbPrivilege`). Эта привилегия проверяется функцией `LsaRegisterLogonProcess`, которую процесс вызывает для установки доверенного подключения к LSASS. Злоумышленник,

обладающий данной привилегией, может установить доверенное подключение к LSASS, а затем выполнить функцию `LsaLogonUser`, которая используется для создания новых сеансов входа в систему. Функции `LsaLogonUser` необходима действительная комбинация имени пользователя и пароля, а в дополнительном параметре функция получает список SID-идентификаторов, которые добавляются этой функцией к исходному маркеру, создаваемому для нового сеанса входа в систему. Следовательно, пользователь может воспользоваться своим собственным именем и паролем для создания нового сеанса входа в систему, включающего SID-идентификаторы с большим количеством привилегированных групп или пользователей в получающемся в результате этого маркере.

ПРИМЕЧАНИЕ Использование повышенных привилегий за границы машины не выходит и на сеть не распространяется, поскольку любое взаимодействие с другим компьютером требует аутентификации в контроллере домена и проверки доменных паролей, которые не хранятся на компьютере ни в виде простого текста, ни в зашифрованном виде, поэтому вредоносному коду они не доступны.

Маркеры доступа процессов и потоков

Концепции, рассмотренные в данной главе, сведены воедино на рис. 7.12, где показаны основные структуры безопасности процесса и потока. На рисунке показано, что у объекта процесса и у объектов потоков имеются ACL-списки, которые также имеются и у самих маркеров доступа. Также на этом рисунке поток 2 и поток 3 имеют маркеры заимствования прав, а поток 1 использует исходный маркер доступа процесса.

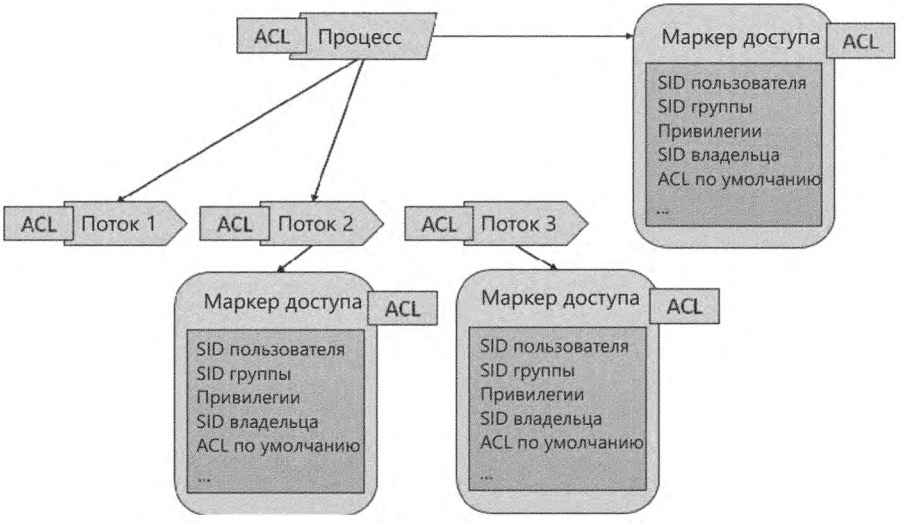


Рис. 7.12. Структуры безопасности процесса и потоков

Аудит безопасности

Диспетчер объектов может генерировать события аудита в результате проверки доступа, а Windows-функции, доступные пользовательским приложениям, могут генерировать эти события напрямую. Код режима ядра всегда позволяет генерировать событие аудита. С аудитом связаны две привилегии: `SeSecurityPrivilege` и `SeAuditPrivilege`. Для управления журналом событий безопасности и для просмотра или настройки SACL объекта у процесса должна быть привилегия `SeSecurityPrivilege`. Но процессы, вызывающие системные службы аудита для успешного генерирования записи аудита, должны обладать привилегией `SeAuditPrivilege`.

Политика аудита локальной системы управляет решением на аудит конкретного типа события безопасности. Политика аудита, также называемая локальной политикой безопасности, является одной частью политики безопасности LSASS, обеспечиваемой на локальной системе, и настраивается с помощью редактора локальной политики безопасности, показанного на рис. 7.13. Конфигурация политики аудита (как основные настройки в разделе локальных политик, так и расширенная конфигурация политики аудита, рассматриваемая далее) хранится в реестре в качестве битового значения в разделе HKEY_LOCAL_MACHINE\SECURITY\Policy\PolAdtEv.

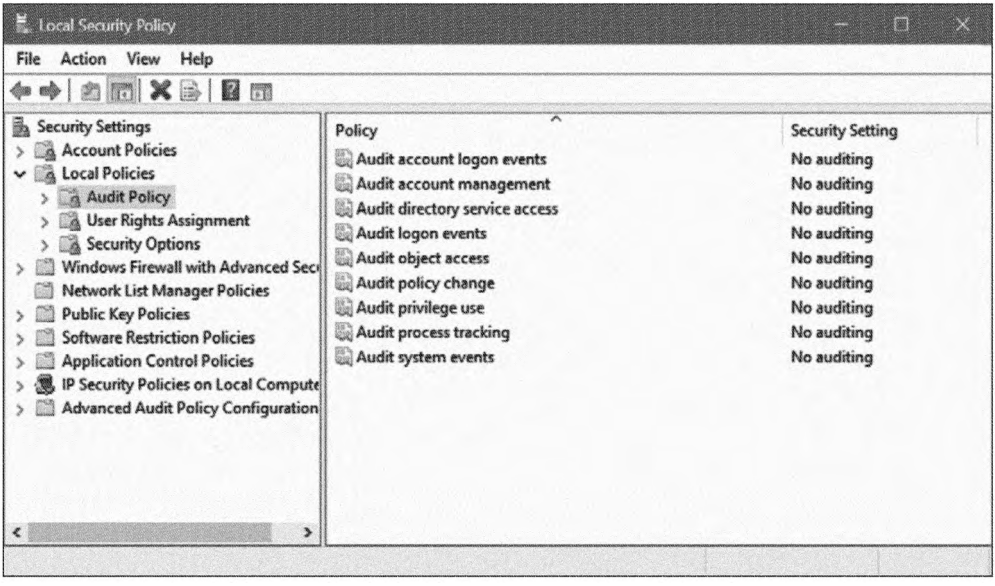


Рис. 7.13. Конфигурация редактора политики локальной безопасности

LSASS отправляет сообщения SRM, чтобы проинформировать его о политике аудита в ходе инициализации системы, а потом при изменениях политики. LSASS отвечает за получение записей аудита, сгенерированных на основе событий аудита от SRM, редактирует записи и отправляет их регистратору событий (Event Logger). LSASS (вместо SRM) отправляет эти записи потому, что добавляет к ним вполне

уместные подробности, например информацию, необходимую для более полной идентификации подвергаемого аудиту процесса.

SRM отправляет записи аудита через свое подключение ALPC к LSASS. Затем регистратор событий помещает запись аудита в журнал регистрации событий безопасности. Кроме записей аудита, передаваемых SRM, и LSASS, и SAM генерируют записи аудита, которые LSASS отправляет непосредственно регистратору событий, а API-функции AuthZ позволяют приложениям генерировать аудиты, определяемые самими приложениями. Эти общие потоки показаны на рис. 7.14.

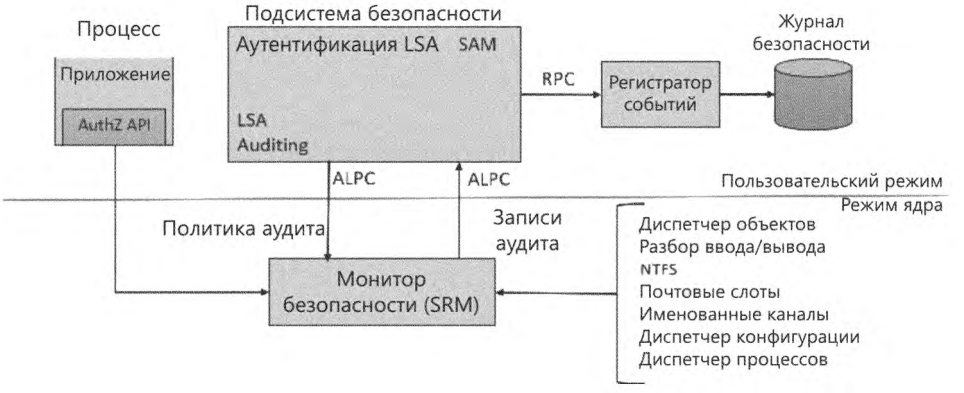


Рис. 7.14. Поток записей аудита безопасности

По мере получения записи аудита ставятся в очередь на отправку LSA, они не передаются пакетами. Записи аудита передаются от SRM к подсистеме безопасности одним из двух возможных способов. Если запись аудита невелика по размеру (меньше чем максимальный размер сообщения ALPC), она отправляется как ALPC-сообщение. Записи аудита копируются из адресного пространства SRM в адресное пространство процесса LSASS. Если запись аудита имеет большой размер, SRM использует общую память, чтобы сообщение было доступно LSASS, и просто передает указатель в ALPC-сообщении.

Аудит доступа к объекту

Важной областью применения механизма аудита во многих средах окружения является ведение журнала доступов к защищенным объектам, в частности к файлам. Для этого должна быть включена политика **Аудит доступа к объектам** (Audit Object Access) и в системных списках управления доступом должны быть ACE-элементы аудита, разрешающие проведение аудита в интересующих объектах.

Когда средство доступа пробует открыть дескриптор объекта, монитор безопасности сначала определяет, разрешена или запрещена подобная попытка. Если включен аудит доступа к объекту, SRM затем сканирует системный ACL объекта. Есть два типа ACE-элементов аудита: доступ разрешен и доступ запрещен. ACE аудита должен

соответствовать любому идентификатору безопасности, имеющемуся у средства доступа, он должен соответствовать любому из запрошенных методов доступа, и его тип (доступ разрешен или доступ запрещен) должен соответствовать результату проверки доступа, чтобы была сгенерирована запись аудита доступа к объекту.

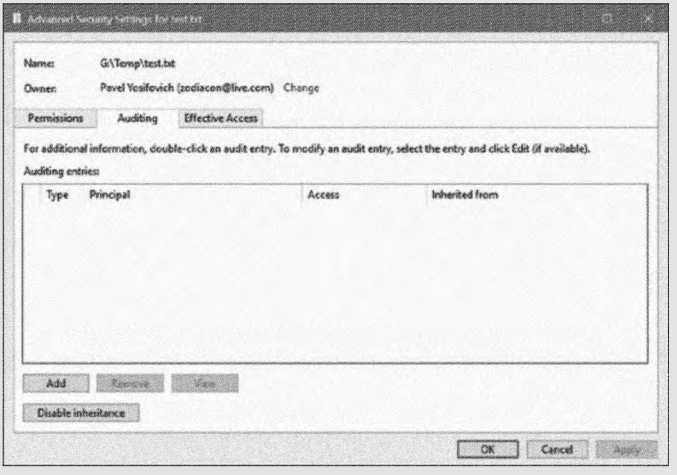
Записи аудита доступа к объекту включают не только сам факт разрешенного или запрещенного доступа, но также и причину успеха или отказа. Эта «причина для доступа», дающая отчет в общем виде, принимает в записи аудита форму записи управления доступом, указанной в языке определения дескриптора безопасности — SDDL (Security Descriptor Definition Language). Это позволяет проводить диагностику сценариев, в которых объекту, к которому по вашему убеждению доступ был запрещен, этот доступ был разрешен, или наоборот, путем идентификации определенных записей управления доступом, ставших причиной того, что попытка доступа удалась или провалилась.

Как видно из рис. 7.13, проведение аудита доступа к объекту по умолчанию выключено (что касается и всех остальных политик аудита).

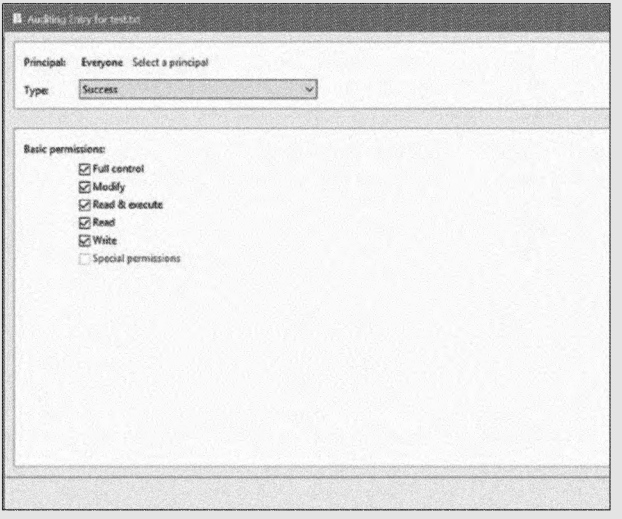
ЭКСПЕРИМЕНТ: АУДИТ ДОСТУПА К ОБЪЕКТУ

Чтобы понаблюдать за тем, как работает аудит доступа к объекту, выполните следующие действия.

1. В Explorer перейдите к файлу, к которому в обычных условиях доступ разрешен (например, к текстовому файлу). В диалоговом окне Свойства (Properties) этого файла щелкните на вкладке Безопасность (Security), а затем щелкните на кнопке Дополнительно (Advanced).
2. Щелкните на вкладке Аудит (Auditing) и пройдите через предупреждение о необходимости административных привилегий. Появившееся в результате этого диалоговое окно позволяет вам добавлять в системный список управления доступом — System Access Control List — этого файла записи об аудите управления доступом.

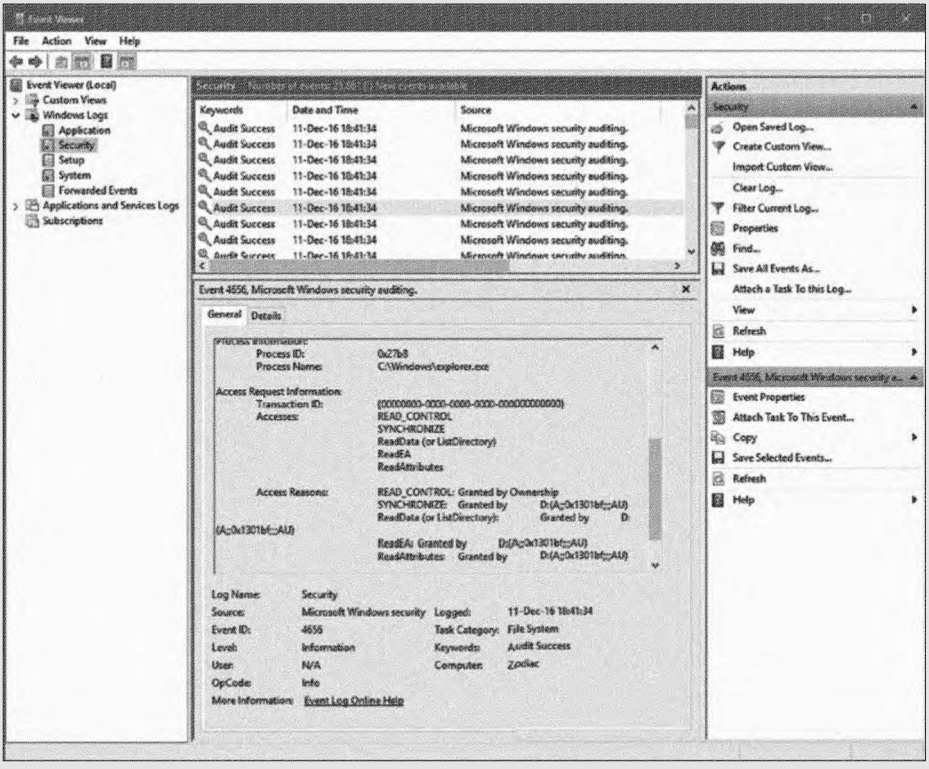


3. Щелкните на кнопке Добавить (Add) и выберите вариант Select a Principal.
4. В появившемся диалоговом окне Выбор «Пользователь» или «Группа» (Select User Or Group) введите свое собственное имя пользователя и пароль или название группы, в которой состоите, например Все (Everyone). Щелкните на кнопке Проверить имена (Check Names), а затем на кнопке OK. В результате появится диалоговое окно для создания элемента аудита (Auditing Access Control Entry) этого файла при доступе к нему со стороны пользователя или группы.



5. Щелкните три раза на кнопке OK, чтобы закрыть диалоговое окно Свойства (Properties).
6. Находясь в Проводнике, дважды щелкните на значке файла, чтобы открыть его в связанной с ним программе (например, в Блокноте для текстового файла).
7. В меню Пуск (Start) введите строку «event» и выберите программу просмотра событий (Event Viewer).
8. Перейдите к журналу безопасности (Security log). Обратите внимание на то, что там нет записи для доступа к файлу. Причина в том, что политика аудита для доступа к объекту еще не настроена.
9. В редакторе локальной политики безопасности перейдите к разделу Локальные политики (Local Policies) и выберите Политика аудита (Audit Policy).
10. Дважды щелкните на записи Аудит доступа к объектам (Audit Object Access), а затем установите флажок Успех (Success), чтобы включить аудит успешного доступа к файлу.
11. В утилите Event Viewer щелкните на пунктах Action (Действие) и Refresh (Обновить). Обратите внимание на то, как изменения в политике аудита повлияли на записи аудита.

12. В Проводнике дважды щелкните на значке файла, чтобы открыть его еще раз.
13. В утилите Event Viewer щелкните на пунктах Action (Действие), Refresh (Обновить). Обратите внимание на то, что теперь появилось несколько записей аудита доступа к файлу.
14. Найдите одну из записей аудита доступа к файлу, у которой идентификатор события Event ID имеет значение 4656, он проявляется в качестве «дескриптора объекта, который был запрошен». (Вы можете воспользоваться средствами поиска для нахождения имени открытого файла.)
15. Прокрутите вниз содержимое текстового поля и найдите раздел Access Reasons. В следующем примере показано, что были запрошены два метода доступа: READ_CONTROL, SYNCHRONIZE и `ReadAttributes`, `ReadEA` (расширенные атрибуты) и `ReadData`. Первый из них был предоставлен потому, что запрашивающий доступ был владельцем файла, а последний был предоставлен по причине показанного элемента управления доступом — Access Control Entry.



Глобальная политика аудита

В дополнение к ACE-элементам доступа к объектам, применяемым в отношении отдельных объектов, в отношении системы может быть определена глобальная политика аудита, позволяющая проводить аудит доступа к объекту для всех объектов файловой системы, для всех разделов реестра или и к тем и к другим. Поэтому аудитор безопасности может быть уверен, что нужный аудит будет выполнен без необходимости установки или изучения списков SACL у всех отдельных интересующих объектов.

Администратор может установить или запросить глобальную политику аудита командой `AuditPol` с ключом `/resourceSACL`. То же самое можно сделать на программном уровне вызовом API-функций `AuditSetGlobalSacl` и `AuditQueryGlobalSacl`. Как и в случае изменений SACL-списков объектов, изменение этих глобальных SACL-списков требует привилегии `SeSecurityPrivilege`.

ЭКСПЕРИМЕНТ: УСТАНОВКА ПОЛИТИКИ ГЛОБАЛЬНОГО АУДИТА

Для включения политики глобального аудита можно воспользоваться командой AuditPol.

1. Если это еще не сделано в предыдущем эксперименте, в редакторе локальной политики безопасности перейдите к настройкам политики аудита (как показано на рис. 7.13), дважды щелкните на пункте Аудит доступа к объектам (Audit Object Access) и включите аудит, как для успеха, так и для отказа. Учтите, что на большинстве систем SACL-списки, определяющие аудит доступа к объектам, встречаются довольно редко, поэтому на данный момент времени записей будет не так уж и много (или не будет вовсе).
2. В окне командной строки с повышенными привилегиями введите следующую команду:

   ```
   C:\> auditpol /resourceSACL
   ```

   Она выдаст краткую информацию о командах для настройки и запроса политики глобального аудита.
3. В том же самом окне командной строки с повышенными привилегиями введите следующие команды:

   ```
   C:\> auditpol /resourceSACL /type:File /view
   C:\> auditpol /resourceSACL /type:Key /view
   ```

   На обычной системе каждая из этих команд сообщит об отсутствии глобального списка Global SACL для соответствующего типа ресурсов. (Учтите, что ключевые слова «File» и «Key» чувствительны к регистру букв.)
4. В том же самом окне командной строки с повышенными привилегиями введите следующую команду:

   ```
   C:\> auditpol /resourceSACL /set /type:File /user:yourusername /success /
   failure /access:FW
   ```

В результате ее выполнения глобальная политика аудита будет настроена таким образом, что все попытки открытия указанным пользователем файлов с доступом по записи (FW) будут приводить к появлению записей аудита как при удачной, так и при неудавшейся попытке открытия файла. Именем пользователя должно быть конкретное имя пользователя в системе, группа (например, Все (Everyone)), определенное доменом имя пользователя вида имя_домена\имя_пользователя или SID.

5. Работая под указанным именем пользователя, откройте файл в Проводнике или в другой программе. Затем загляните в журнал безопасности в системном журнале событий и найдите в нем записи аудита.
6. В завершение эксперимента воспользуйтесь командой auditpol для удаления глобального списка SACL, созданного при выполнении пункта 4:

```
C:\> auditpol /resourceSACL /remove /type:File /user:yourusername
```

Глобальная политика аудита хранится в реестре в виде пары системных списков управления доступом в разделах HKEY_LOCAL_MACHINE\SECURITY\Policy\GlobalSaclNameFile и в HKEY_LOCAL_MACHINE\SECURITY\Policy\GlobalSaclNameKey. Чтобы проанализировать содержимое этих разделов, запустите программу Regedit.exe под учетной записью System (см. раздел «Системные компоненты безопасности» этой главы). Эти разделы не будут присутствовать в реестре, пока хотя бы однажды не будут установлены соответствующие глобальные SACL-списки.

Глобальная политика аудита не может быть отменена путем применения SACL-списков объектов, а вот SACL-списки объектов могут позволить проводить дополнительный аудит. Например, глобальная политика аудита может потребовать проведение аудита доступа по чтению всех пользователей ко всем файлам, а SACL-списки отдельных файлов могут добавить аудит доступа по записи к таким файлам со стороны определенных пользователей или более конкретных групп пользователей.

Глобальная политика аудита может также быть сконфигурирована через редактор локальной политики безопасности через Конфигурацию расширенной политики аудита, рассматриваемую в следующем подразделе.

Конфигурация расширенной политики аудита

В дополнение к ранее рассмотренным настройкам политики аудита редактор локальной политики безопасности предлагает еще более тонко настраиваемый набор средств управления аудитом, показанный на рис. 7.15 под заголовком Конфигурация расширенной политики аудита (Advanced Audit Policy Configuration).

Каждая из девяти настроек политики аудита в разделе Локальные политики, показанных на рис. 7.13, соответствует группе настроек, обеспечивающей более точное

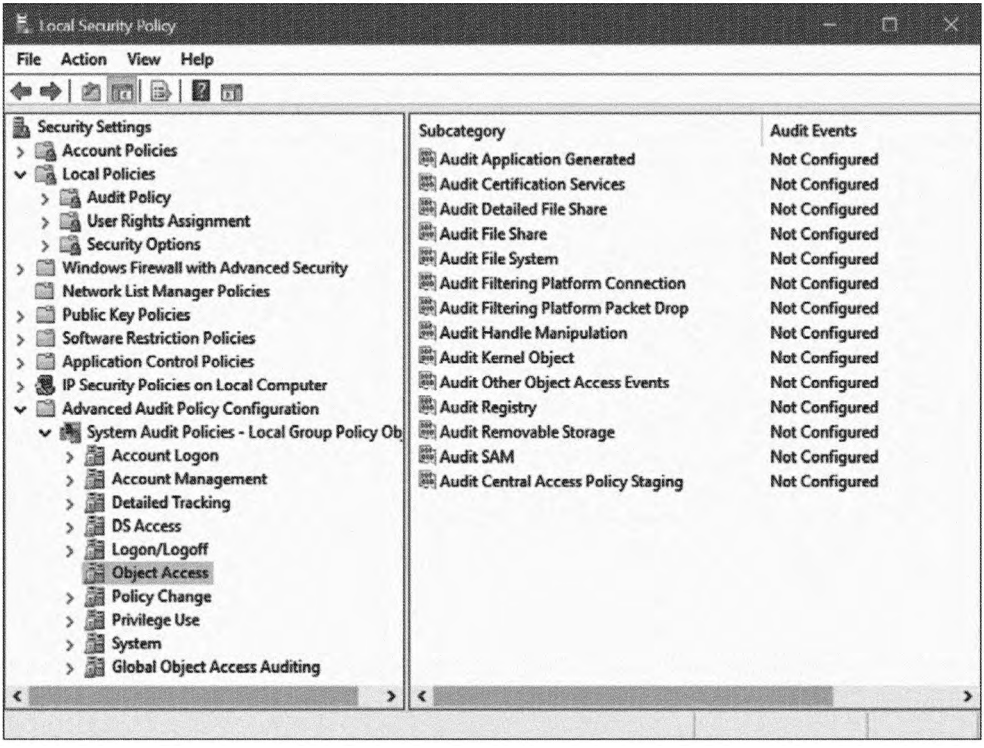


Рис. 7.15. Конфигурация расширенной политики аудита в редакторе локальной политики безопасности

управление. Например, Аудит доступа к объектам в разделе Локальные политики разрешает доступ ко всем подвергаемым аудиту объектам, а здесь можно настроить аудит доступа к объектам различного типа так, чтобы он управлялся индивидуально. Включение одной из настроек политики аудита в разделе Локальные политики косвенным образом включает все соответствующие расширенные события политики аудита, но если требуется более тонкое управление над контентами журнала аудита, расширенные настройки могут быть установлены индивидуальным образом. Затем стандартные настройки становятся продуктом расширенных настроек, но это нельзя увидеть в редакторе локальной политики безопасности. Попытки указать настройки аудита с использованием как основных, так и расширенных настроек могут привести к неожиданным результатам.

Подраздел Аудит доступа к глобальным объектам (Global Object Access Auditing) в разделе Конфигурация расширенной политики аудита (Advanced Audit Policy Configuration) может использоваться для настройки глобальных системных списков управления доступом (Global SACL), с использованием графического интерфейса, идентичного тому, который можно увидеть в Проводнике или в редакторе реестра для дескрипторов безопасности в файловой системе или в реестре.

AppContainer

В Windows 8 появилось новое средство безопасности — контейнеры *AppContainer*. Хотя изначально они создавались для управления процессами UWP, контейнеры AppContainer могут использоваться и для «нормальных» процессов (хотя встроенных инструментов для этого не существует). В этом разделе в основном рассматриваются атрибуты *пакетных* контейнеров AppContainer — этим термином обозначаются контейнеры AppContainer, связанные с процессами UWP и итоговым форматом .Appx. Полное описание приложений UWP выходит за рамки этой главы.

Дополнительную информацию по теме можно найти в главе 3, а также в главах 8 и 9 части 2. А здесь мы сосредоточимся на аспектах безопасности контейнеров AppContainer и их типичном использовании в качестве хостов UWP.

ПРИМЕЧАНИЕ «UWP» (*Universal Windows Platform*) — самый новый термин для описания процессов, управляющих исполнительной средой Windows Runtime. Различные старые названия — «*иммерсивное приложение*», «*современное приложение*», «*приложение с метро-интерфейсом*», а иногда и просто «приложение Windows». Прилагательное «универсальный» (Universal) обозначает способность приложения развертываться и выполняться в различных выпусках Windows 10 и форм-факторах, от IoT до мобильных и настольных систем, от Xbox до HoloLens. Тем не менее по сути это те же приложения, которые появились в Windows 8. Следовательно, концепция контейнеров AppContainer, рассматриваемая в этом разделе, актуальна для Windows 8 и более поздних версий Windows. Учтите, что вместо термина UWP иногда используется термин UAP (Universal Application Platform); это одно и то же.

ПРИМЕЧАНИЕ Изначально для контейнеров AppContainer использовалось кодовое имя *LowBox*. Этот термин все еще встречается во многих именах API-функций и структур данных. Он обозначает ту же концепцию.

Общие сведения о приложениях UWP

Революция мобильных устройств открыла новые возможности распространения и запуска программных продуктов. Мобильные устройства обычно получают приложения из централизованного магазина, с автоматической установкой и обновлением при минимальном вмешательстве пользователя. После того как пользователь выберет приложение в магазине, он видит разрешения, необходимые для правильного функционирования приложения. Эти разрешения, называемые *возможностями* (capabilities), объявляются как часть пакета при отправке в магазин. Таким образом пользователь может решить, приемлем ли для него этот список возможностей.

На рис. 7.16 изображен пример списка возможностей для UWP-игры (*Minecraft*, бета-версия Windows 10). Игре необходим доступ к интернету в качестве клиента и сервера, а также доступ к локальной домашней или рабочей сети. После того как пользователь загрузит игру, он неявно соглашается с тем, что игра может пользоваться этими возможностями. И наоборот, пользователь может быть уверен в том,

что игра использует *только* эти возможности. Иначе говоря, игра ни при каких условиях не может использовать другие неподтвержденные возможности (например, доступ к камере на устройстве).

Рис. 7.16. Страница приложения в магазине; среди прочего приведен перечень возможностей приложения

Чтобы получить представление о различиях между приложениями UWP и настольными (классическими) приложениями на высоком уровне, обратитесь к табл. 7.12. С точки зрения разработчика платформа Windows может рассматриваться так, как показано на рис. 7.17.

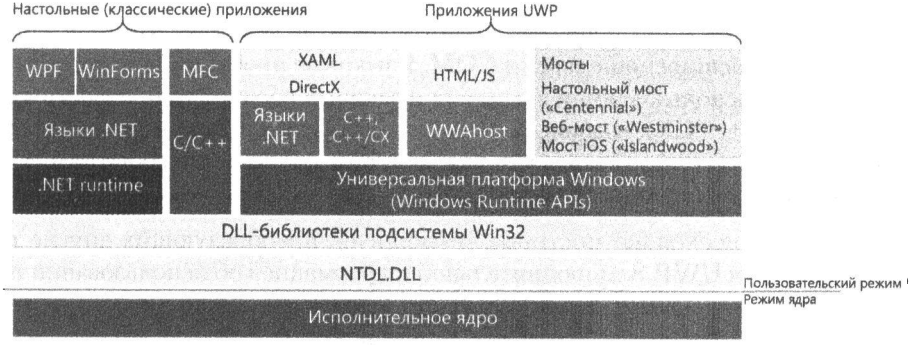


Рис. 7.17. Структура платформы Windows

Таблица 7.12. Высокоуровневое сравнение приложений UWP и настольных приложений

	Приложение UWP	**Настольное (классическое) приложение**
Поддержка устройств	Работает во всех семействах Windows	Работает только на PC
API	Может обращаться к WinRT, подмножеству COM и подмножеству Win32 API	Может обращаться к COM, Win32 и подмножеству WinRT API
Идентификация	Сильная идентификация приложения (статическая и динамическая)	Низкоуровневые EXE-образы и процессы
Информация	Декларативный манифест APPX	Непрозрачные двоичные файлы
Установка	Автономные пакеты APPX	Свободные файлы или MSI
Данные приложения	Изолированное хранилище уровня пользователя/приложения (локальное и перемещаемое)	Совместный пользовательский профиль
Жизненный цикл	Участвует в управлении ресурсами приложения и PLM	Жизненный цикл уровня процесса
Создание экземпляров	Только один экземпляр	Любое количество экземпляров

Некоторые блоки на рис. 7.17 заслуживают подробного упоминания.

- На базе приложений UWP могут создаваться нормальные исполняемые файлы, как и для настольных приложений. В качестве хоста для UWP-приложений на базе HTML/JavaScript используется Wwahost.exe (%SystemRoot%\System32\wwahost.exe), так как для них создается DLL-библиотека, а не исполняемый файл.
- Технология UWP реализуется API-интерфейсом Windows Runtime, базирующемся на расширенной версии COM. Языковые проекции предоставляются для C++ (с использованием закрытых языковых расширений, известных под названием C++/CX), языков .NET и JavaScript. Эти проекции позволяют относительно легко обращаться к типам, методам, свойствам и событиям WinRT из сред, знакомых разработчикам.
- Существует несколько мостовых технологий, преобразующих другие типы приложений в UWP. За дополнительной информацией об использовании таких технологий обращайтесь к документации MSDN.
- Среда Windows Runtime построена на базе DLL-библиотек подсистем Windows, как и .NET Framework. Она не содержит компонентов ядра и не является ча-

стью другой подсистемы, потому что она использует те же функции Win32 API, предоставляемые системой. Впрочем, некоторые политики реализуются в ядре, как и общая поддержка AppContainer.

- API-функции Windows Runtime реализуются в DLL-библиотеках, находящихся в каталоге %SystemRoot%\System32. Имена этих библиотек строятся по схеме *Windows.Xxx.Yyy*...Dll, при этом имя файла обычно обозначает реализованное пространство имен Windows Runtime API. Например, библиотека Windows.Globalization.Dll реализует классы, находящиеся в пространстве имен `Windows.Globalization`. (За полной информацией о WinRT API обращайтесь к документации MSDN.)

AppContainer

Мы уже описали действия, необходимые для создания процессов, в главе 3; также были представлены некоторые дополнительные действия, необходимые для создания процессов UWP. Процесс создания инициируется службой `DCOMLaunch`, потому что пакеты UWP поддерживают набор протоколов, одним из которых является протокол Launch. Полученный процесс выполняется в контейнере AppContainer. Ниже перечислены некоторые характеристики пакетных процессов, выполняемых в AppContainer.

- Уровню целостности в маркере процесса присвоено значение Low, что автоматически ограничивает доступ ко многим объектам и к некоторым API или видам функциональности для процесса, как упоминалось ранее в этой главе.
- Процессы UWP всегда создаются внутри задания (одно задание на каждое приложение UWP). Задание управляет процессом UWP и всеми фоновыми процессами, выполняемыми от его имени (через вложенные задания). Задания позволяют диспетчеру состояний процесса (PSM, Process State Manager) быстро приостанавливать/возобновлять приложение или фоновую обработку.
- Маркер процессов UWP содержит SID-идентификатор AppContainer, представляющий отдельную сущность на базе хеша SHA-2 имени пакета UWP. Как вы увидите, этот SID-идентификатор используется системой и приложениями для того, чтобы явно разрешать доступ к файлам и другим объектам ядра. SID является частью иерархии `APPLICATION PACKAGE AUTHORITY` вместо `NT AUTHORITY`, которая в основном встречается в этой главе. Иначе говоря, в строковом формате он начинается с S-1-15-2, что соответствует `SECURITY_APP_PACKAGE_BASE_RID` (15) и `SECURITY_APP_PACKAGE_BASE_RID` (2). Поскольку хеш SHA-2 имеет длину 32 байта, оставшаяся часть SID состоит из 8 RID (напомним, что RID — размер 4-байтового типа `ULONG`).
- Маркер может содержать набор возможностей, каждая из которых представлена SID-идентификатором. Эти возможности объявляются в манифесте приложения и показываются на странице приложения в магазине. Они хранятся

в разделе возможностей манифеста и преобразуются в формат SID по правилам, которые будут представлены ниже, и принадлежат к иерархии SID из предыдущего пункта, но с использованием значения `SECURITY_CAPABILITY_BASE_RID` (3). Различные компоненты Windows Runtime, классы обращения к устройствам пользовательского режима и ядро могут обращаться к информации возможностей для разрешения или запрета некоторых операций.

- Маркер может содержать только следующие привилегии: `SeChangeNotifyPrivilege`, `SeIncreaseWorkingSetPrivilege`, `SeShutdownPrivilege`, `SeTimeZonePrivilege` и `SeUndockPrivilege`. Это набор привилегий по умолчанию, связываемый со стандартными учетными записями пользователей. Кроме того, функция `AppContainerPrivilegesEnabledExt`, являющаяся частью расширения ms-win-ntos-ksecurity, может присутствовать на некоторых устройствах для дальнейшего ограничения привилегий, включенных по умолчанию.

Маркер содержит до четырех атрибутов безопасности, идентифицирующих этот маркер как связанный с пакетным приложением UWP. Эти атрибуты добавляются упоминавшейся выше службой DcomLaunch, отвечающей за активацию приложений UWP:

- `WIN://PKG` — идентифицирует маркер как принадлежащий упакованному приложению UWP. Содержит целочисленное значение, в котором хранится источник приложения и флаги (см. табл. 7.13 и 7.14).
- `WIN://SYSAPPID` — содержит идентификаторы приложения, называемые *моникерами* (monickers) или строковыми именами, в форме массива строк в кодировке Юникод.
- `WIN://PKGHOSTID` — идентифицирует хост пакета UWP для пакетов с явно задаваемым хостом (целочисленное значение).
- `WIN://BGKD` — используется только для фоновых хостов (например, хоста обобщенных фоновых задач BackgroundTaskHost.exe), которые могут содержать пакетные службы UWP, работающие в режиме провайдера COM. Атрибут содержит целочисленное значение — идентификатор явно заданного хоста.

Флаг `TOKEN_LOWBOX` (0x4000) устанавливается в поле флагов маркера, для чтения которого используются различные API-функции Windows и ядра (например, `GetTokenInformation`). Это позволяет компонентам работать по-другому в присутствии маркера AppContainer.

ПРИМЕЧАНИЕ Существует другой тип контейнеров AppContainer: дочерние контейнеры AppContainer. Он используется в тех случаях, когда UWP-контейнер AppContainer (родительский) хочет создать собственный вложенный контейнер AppContainer для дальнейшего укрепления безопасности приложения. Вместо восьми RID дочерний контейнер AppContainer содержит четыре дополнительных RID (первые восемь совпадают с родительскими значениями) для его однозначной идентификации.

Таблица 7.13. Источники пакетов

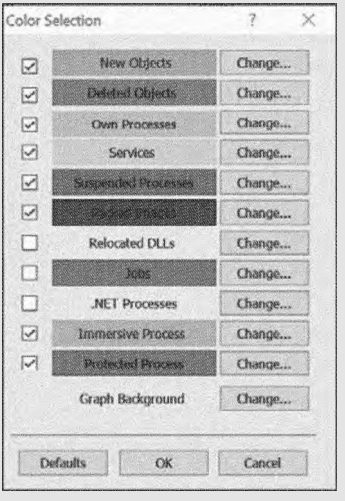
Источник	Описание
Неизвестен (0)	Источник пакета неизвестен
Без подписи (1)	Пакет не подписан
Встроенный (2)	Пакет связан со встроенным приложением Windows
Магазин (3)	Пакет связан с приложением UWP, загруженным из магазина. Для подтверждения источника система проверяет, что список DACL файла, связанный с основным исполняемым файлом приложения UWP, содержит доверенный элемент ACE
Разработчик без подписи (4)	Пакет связан с ключом разработчика без подписи
Разработчик с подписью (5)	Пакет связан с ключом разработчика с подписью
`Line-of-Business` (6)	Пакет связан с неопубликованным (side-loaded) LOB-приложением (Line-of-Business)

Таблица 7.14. Флаги пакетов

Флаг	Описание
`PSM_ACTIVATION_TOKEN_PACKAGED_APPLICATION` (0x1)	Означает, что UWP-приложение в AppContainer хранится в пакетном формате AppX
`PSM_ACTIVATION_TOKEN_SHARED_ENTITY` (0x2)	Означает, что маркер используется для нескольких исполняемых образов, которые все являются частью одного пакетного UWP-приложения AppX
`PSM_ACTIVATION_TOKEN_FULL_TRUST` (0x4)	Означает, что маркер AppContainer является хостом для преобразованного Win32-приложения Project Centennial (Windows Bridge for Desktop)
`PSM_ACTIVATION_TOKEN_NATIVE_SERVICE` (0x8)	Означает, что маркер AppContainer является хостом для пакетной службы, созданной диспетчером ресурсов SCM (Service Control Manager). За дополнительной информацией о службах обращайтесь главе 9 части 2
`PSM_ACTIVATION_TOKEN_DEVELOPMENT_APP` (0x10)	Означает, что приложение относится к внутренней разработке и не может использоваться в системах продажи
`BREAKAWAY_INHIBITED` (0x20)	Пакет не может создать процесс, который сам не является пакетным. Устанавливается при помощи атрибута процесса `PROC_THREAD_ATTRIBUTE_DESKTOP_APP_POLICY` (за дополнительной информацией обращайтесь к главе 3)

ЭКСПЕРИМЕНТ: ПРОСМОТР ИНФОРМАЦИИ О ПРОЦЕССАХ UWP

Есть несколько более или менее очевидных способов получения информации о процессах UWP. Process Explorer может выделять процессы, использующие Windows Runtime, цветом (по умолчанию ярко-голубым). Чтобы увидеть эту возможность в действии, откройте Process Explorer, откройте меню Options и выберите команду Configure Colors. Убедитесь в том, что флажок Immersive Processes установлен.



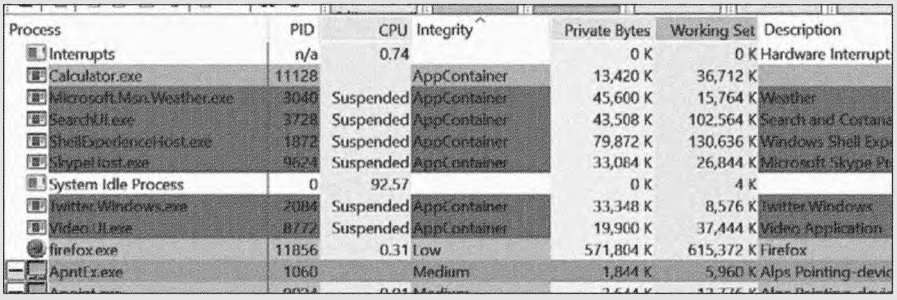
Термин «*иммерсивный процесс*» использовался для описания приложений WinRT (теперь UWP) в Windows 8. Для получения этой информации используется вызов API-функции `IsImmersiveProcess`.

Запустите Calc.exe и переключитесь на Process Explorer. Некоторые процессы в списке, в том числе и Calculator.exe, должны быть выделены ярко-голубым цветом. Сверните приложение Калькулятор; вы увидите, что ярко-голубой цвет заменился серым. Это произошло из-за того, что приложение Калькулятор было приостановлено. Восстановите окно Калькулятора, и приложение снова вернется к ярко-голубому цвету.

Аналогичное поведение наблюдается и с другими приложениями — например, поисковым интерфейсом Cortana (SearchUI.exe). Щелкните на значке Cortana на панели задач, а затем закройте приложение; вы увидите, как серый цвет превращается в ярко-голубой, который затем снова возвращается к серому. Также можно щелкнуть на кнопке Пуск (Start); процесс ShellExperienceHost.exe выделяется аналогичным образом.

Присутствие в списке некоторых процессов, выделенных ярко-голубым цветом, может вас удивить — например, Explorer.exe, TaskMgr.Exe и RuntimeBroker.exe. Они не являются приложениями, но используют API-функции Windows Runtime и поэтому классифицируются как иммерсивные (роль RuntimeBroker будет рассмотрена ниже).

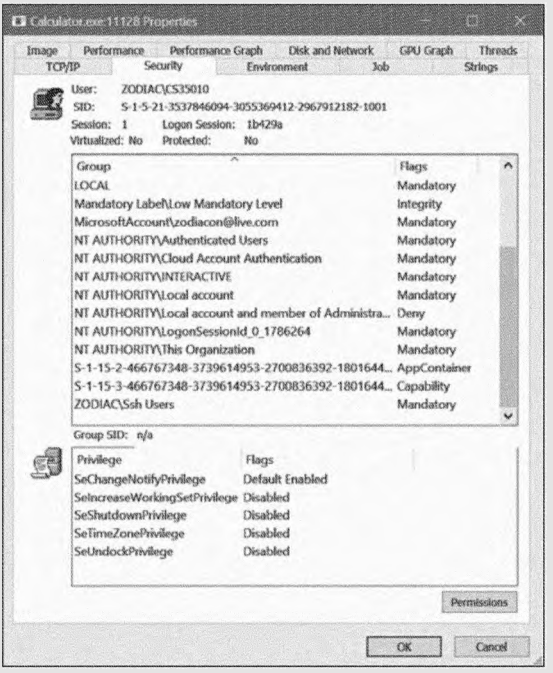
Наконец, убедитесь в том, что в Process Explorer отображается столбец Integrity, и проведите сортировку списка по этому столбцу. Вы увидите, что такие процессы, как Calculator.exe и SearchUI.exe, имеют уровень целостности AppContainer. Обратите внимание: Проводник и TaskMgr в этом списке отсутствуют; это наглядно показывает, что они не являются процессами UWP и существуют по другим правилам.



Process	PID	CPU	Integrity	Private Bytes	Working Set	Description
Interrupts	n/a	0.74		0 K	0 K	Hardware Interrupt
Calculator.exe	11128		AppContainer	13,420 K	36,712 K	
Microsoft.Msn.Weather.exe	3040	Suspended	AppContainer	45,600 K	15,764 K	Weather
SearchUI.exe	3728	Suspended	AppContainer	43,508 K	102,564 K	Search and Cortana
ShellExperienceHost.exe	1872	Suspended	AppContainer	79,872 K	130,636 K	Windows Shell Exp
SkypeHost.exe	9624	Suspended	AppContainer	33,084 K	26,844 K	Microsoft Skype Pr
System Idle Process	0	92.57		0 K	4 K	
Twitter.Windows.exe	7084	Suspended	AppContainer	33,348 K	8,576 K	Twitter.Windows
Video.UI.exe	8772	Suspended	AppContainer	19,900 K	37,444 K	Video Application
firefox.exe	11856	0.31	Low	571,804 K	615,372 K	Firefox
ApntEx.exe	1060		Medium	1,844 K	5,960 K	Alps Pointing-devic

ЭКСПЕРИМЕНТ: ПРОСМОТР МАРКЕРА APPCONTAINER

Существует ряд инструментов для просмотра свойств процессов, выполняемых в контейнерах AppContainer. В Process Explorer на вкладке Security выводятся возможности, связанные с маркером. Вот как выглядит вкладка Security для Calculator.exe:



Обратите внимание на два интересных значения: AppContainer SID (в столбце Flags отображается в виде AppContainer) и одна возможность (Capability) прямо под AppContainer SID. Не считая базового RID (SECURITY_APP_PACKAGE_BASE_RID вместо SECURITY_CAPABILITY_BASE_RID), восемь оставшихся RID идентичны; в обоих случаях используется имя пакета в формате SHA-2, как упоминалось ранее. Таким образом, в списке всегда присутствует одна неявная возможность «быть пакетом», это на самом деле означает, что Калькулятор вообще не требует никаких возможностей. В разделе, посвященном возможностям, рассматривается намного более сложный пример.

ЭКСПЕРИМЕНТ: ПРОСМОТР АТРИБУТОВ МАРКЕРА APPCONTAINER

Аналогичную информацию можно вывести и в командной строке при помощи программы AccessChk из пакета Sysinternals, которая также выводит полный список всех атрибутов маркера. Например, при запуске AccessChk с ключами -p, -f и идентификатором процесса SearchUI.exe (хост Cortana) выводится следующая информация:

```
C:\ >accesschk -p -f 3728

Accesschk v6.10 - Reports effective permissions for securable objects
Copyright (C) 2006-2016 Mark Russinovich
Sysinternals - www.sysinternals.com

[7416] SearchUI.exe
  RW DESKTOP-DD6KTPM\aione
  RW NT AUTHORITY\SYSTEM
  RW Package
\S-1-15-2-1861897761-1695161497-2927542615-642690995-327840285-2659745135-2630312742
  Token security:
  RW DESKTOP-DD6KTPM\aione
  RW NT AUTHORITY\SYSTEM
  RW DESKTOP-DD6KTPM\aione-S-1-5-5-0-459087
  RW Package
\S-1-15-2-1861897761-1695161497-2927542615-642690995-327840285-2659745135-2630312742
  R  BUILTIN\Administrators
  Token contents:
    User:
      DESKTOP-DD6KTPM\aione
    AppContainer:
      Package
\S-1-15-2-1861897761-1695161497-2927542615-642690995-327840285-2659745135-2630312742
    Groups:
      Mandatory Label\Low Mandatory Level              INTEGRITY
      Everyone                                         MANDATORY
      NT AUTHORITY\Local account and member of Administrators group DENY
      …
       Security Attributes:
      WIN://PKGHOSTID
          TOKEN_SECURITY_ATTRIBUTE_TYPE_UINT64
          [0] 1794402976530433
      WIN://SYSAPPID
          TOKEN_SECURITY_ATTRIBUTE_TYPE_STRING
          [0] Microsoft.Windows.Cortana_1.8.3.14986_neutral_neutral_cw5n1h2txyewy
```

```
        [1] CortanaUI
        [2] Microsoft.Windows.Cortana_cw5n1h2txyewy
    WIN://PKG
        TOKEN_SECURITY_ATTRIBUTE_TYPE_UINT64
        [0] 131073
    TSA://ProcUnique
        [TOKEN_SECURITY_ATTRIBUTE_NON_INHERITABLE]
        [TOKEN_SECURITY_ATTRIBUTE_COMPARE_IGNORE]
        TOKEN_SECURITY_ATTRIBUTE_TYPE_UINT64
        [0] 204
        [1] 24566825
```

Сначала идет идентификатор хоста пакета, преобразованный в шестнадцатеричную систему: 0x6600000000001. Так как все идентификаторы хостов пакетов начинаются с 0x66, это означает, что Cortana использует первый доступный идентификатор хоста: 1. Далее идут идентификаторы системных приложений, которые содержат три строки: сильный моникер пакета, понятное имя приложения и упрощенное имя приложения. В завершение следует утверждение пакета — 0x20001 в шестнадцатеричном виде. Как следует из полей табл. 7.13 и 7.14, приведенных выше, это означает встроенный источник (2) и флаги `PSM_ACTIVATION_TOKEN_PACKAGED_APPLICATION`, подтверждающие, что Cortana является частью пакета AppX.

Среда безопасности AppContainer

Один из самых значительных побочных эффектов, обусловленных присутствием AppContainer SID и связанных с ним флагов, заключается в том, что алгоритм проверки прав доступа (см. раздел «Проверки прав доступа» ранее в этой главе) изменен так, что он фактически игнорирует все SID-идентификаторы обычных пользователей и групп, которые может содержать маркер; по сути, они интерпретируются как SID только для запрета. Это означает, что хотя Калькулятор может быть запущен пользователем Джоном, принадлежащим к группам Пользователи и Остальные, он не пройдет проверки доступа, предоставляющие доступ SID Джона, SID группы Пользователи или SID группы Остальные. Фактически алгоритм проверки прав доступа на уровне пользователей будет совпадать с алгоритмом для SID AppContainer, а за ним следует алгоритм проверки возможностей, проверяющий все SID возможностей из соответствующей части маркера.

Однако простой интерпретацией SID уровня пользователей как предназначенных только для запрета дело не ограничивается. Маркеры AppContainer влияют на одно критическое изменение безопасности в алгоритме проверки прав доступа: NULL-список DACL, который обычно рассматривается как ситуация с полным разрешением доступа из-за отсутствия какой-либо информации (напомним: это отличается от *пустого* списка DACL, который интерпретируется как ситуация с полным запретом из-за правил явного разрешения доступа), игнорируется и интерпретируется как запрет. Для упрощения ситуации единственными типами защищаемых объектов, к которым может обращаться AppContainer, являются те, у которых присутствует

разрешающий ACE-элемент для SID AppContainer или одной из его возможностей. Даже незащищенные (NULL DACL) объекты исключаются из рассмотрения.

Такая ситуация создает проблемы с совместимостью. Как вообще может функционировать приложение, не имеющее доступа даже к базовой файловой системе, реестру и ресурсам диспетчера объектов? Для таких ситуаций Windows создает отдельную среду исполнения, или «тюрьму» (jail), для каждого контейнера AppContainer:

ПРИМЕЧАНИЕ До сих пор подразумевалось, что каждое пакетное приложение UWP соответствует одному маркеру AppContainer. Тем не менее это не означает, что с AppContainer может быть связан только один исполняемый файл. Пакеты UWP могут содержать несколько исполняемых файлов, которые все относятся к одному контейнеру AppContainer. Это позволяет им совместно использовать один SID-идентификатор и возможности и обмениваться данными друг с другом (например, исполняемый файл подсистемы баз данных, работающий в режиме микрослужбы, и исполняемый файл интерфейсной части, работающий на переднем плане).

- Строковое представление SID AppContainer используется для создания подкаталога в пространстве имен диспетчера объектов в иерархии \Sessions\x\AppContainerNamedObjects. Этот каталог становится приватным каталогом для именованных объектов ядра. Указанный объект подкаталога затем связывается через ACL с SID-идентификатором, связанным с контейнером AppContainer, который обладает маской полностью разрешенного доступа. В этом ситуация отличается от настольных приложений, которые все используют подкаталог \Sessions\x\BaseNamedObjects (в пределах одного сеанса x). Последствия будут описаны ниже, как и требования относительно хранения дескрипторов в маркерах.

- Маркер будет содержать номер LowBox — уникальный идентификатор в массиве структуры LowBox Number Entry, который сохраняется ядром в глобальной переменной `g_SessionLowboxArray`. Каждая структура соответствует структуре `SEP_LOWBOX_NUMBER_ENTRY`, которая, что самое важное, содержит таблицу атомов, уникальную для данного контейнера AppContainer, потому что драйвер режима ядра подсистемы Windows (Win32k.sys) не позволяет контейнерам AppContainer обращаться к глобальным таблицам атомов.

- В файловой системе присутствует подкаталог %LOCALAPPDATA% с именем Packages. В нем хранятся моникеры пакетов (строковая версия AppContainer SID — т. е. имя пакета) для всех установленных приложений UWP. Каждый из этих каталогов приложений содержит каталоги, специфические для приложений (например, TempState, RoamingState, Settings, LocalCache и т. д.), которым назначаются ACL-списки с AppContainer SID, соответствующим приложению, с маской доступа с полным разрешением.

- В каталоге Settings находится Settings.dat — файл куста реестра, который загружается как куст приложения. (О кустах приложений более подробно рассказано в главе 9 части 2.) Куст выполняет функции локального реестра приложения, в котором API-функции WinRT хранят различные долгосрочные данные состояния приложения. И снова ACL-список разделов реестра явно предоставляет полный доступ для соответствующего SID-идентификатора AppContainer.

Эти четыре «тюрьмы» позволяют AppContainer безопасно и локально хранить свою файловую систему, реестр и таблицу атомов без необходимости доступа к критичным областям (пользовательским и системным) в системе. Но как насчет возможности обращаться (хотя бы только для чтения) к критическим системным файлам (таким, как Ntdll.dll и Kernel32.dll), или разделам реестра (включая те, которые необходимы этим библиотекам), или даже именованным объектам (например, к ALPC-порту \RPC Control\DNSResolver, используемому для DNS-поиска)? Было бы неэффективно для каждого приложения UWP и удаления приложений создавать заново ACL для целых каталогов, разделов реестра и пространств имен объектов с добавлением или удалением различных SID.

Для решения этой проблемы подсистема безопасности поддерживает специальный SID-идентификатор группы `ALL APPLICATION PACKAGES`, который автоматически связывается с любым маркером AppContainer. Многие критические системные области (например, %SystemRoot%\System32 и HKLM\Software\Microsoft\Windows\CurrentVersion) содержат этот SID-идентификатор как часть своих списков DACL, обычно с маской доступа для чтения или чтения с выполнением. Некоторые объекты в пространстве имен диспетчера объектов также включают его — как, например, APLC-порт DNSResolver в каталоге диспетчера объектов \RPC Control. Среди других примеров можно отметить некоторые объекты COM, предоставляющие право выполнения. И хотя такая возможность официально не документирована, сторонние разработчики при создании приложений, не являющихся приложениями UWP, также могут разрешить взаимодействия с UWP-приложениями, применяя этот SID-идентификатор к собственным ресурсам.

К сожалению, так как UWP-приложения формально могут загрузить любую DLL-библиотеку Win32 в составе своих потребностей WinRT (поскольку WinRT строится на базе Win32, как было показано ранее), а поскольку трудно предсказать, что может понадобиться конкретному приложению UWP, у многих системных ресурсов в качестве меры предосторожности с их DACL связывается SID `ALL APPLICATION PACKAGES`. Это означает, например, что разработчик UWP-приложения не сможет запретить DNS-поиск из своих приложений. Этот доступ, превышающий пределы необходимого, может принести пользу авторам эксплойтов, которые воспользуются им для выхода из изоляции AppContainer. В новых версиях Windows 10, начиная с версии 1607 (Anniversary Update), появился новый элемент безопасности для противодействия этому риску: ограниченные контейнеры AppContainers.

Используя атрибут процесса `PROC_THREAD_ATTRIBUTE_ALL_APPLICATION_PACKAGES_POLICY` и присваивая ему значение `PROCESS_CREATION_ALL_APPLICATION_PACKAGES_OPT_OUT` при создании процесса (за дополнительной информацией об атрибутах процессов обращайтесь к главе 3), маркер не будет связываться ни с каким ACE-элементом, задающим SID ALL APPLICATION PACKAGES, блокируя доступ ко многим системным ресурсам, которые в противном случае были бы доступными. Такие маркеры можно узнать по присутствию четвертого атрибута маркера с именем `WIN://NOALLAPPPKG`, которому присвоено целочисленное значение 1.

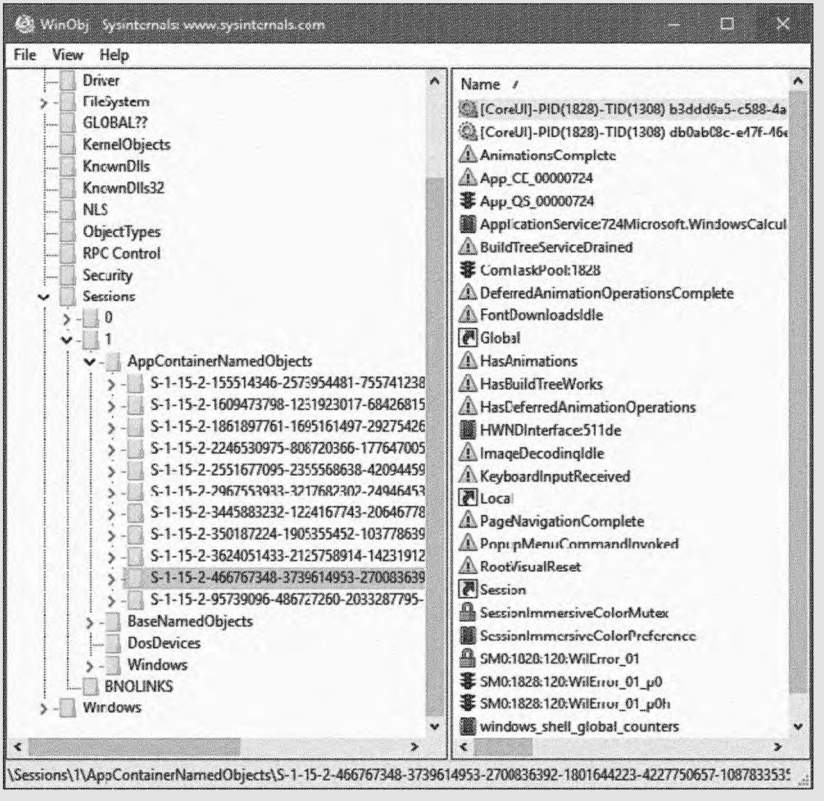
Конечно, при этом мы возвращаемся к той же проблеме: как приложение сможет загрузить даже библиотеку Ntdll.dll, которая играет ключевую роль в инициализа-

ции любого процесса? В Windows 10 версии 1607 появилась новая группа с именем `ALL RESTRICTED APPLICATION PACKAGES`, которая решает эту проблему. Например, каталог System32 теперь также содержит этот SID-идентификатор, также настроенный для установки разрешений чтения и выполнения, поскольку загрузка DLL в этом каталоге играет ключевую роль даже во многих изолированных процессах. Однако у ALPC-порта `DNSResolver` этот SID-идентификатор отсутствует, так что AppContainer потеряет доступ к DNS.

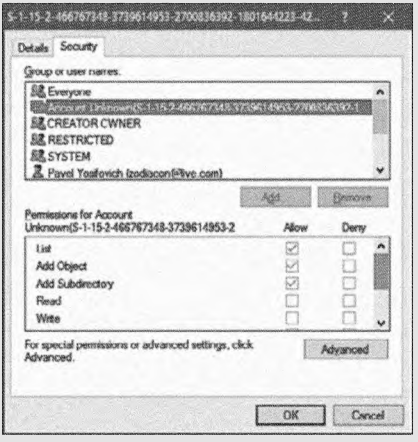
ЭКСПЕРИМЕНТ: ПРОСМОТР АТРИБУТОВ БЕЗОПАСНОСТИ APPCONTAINER

В этом эксперименте рассматриваются атрибуты безопасности некоторых каталогов, упомянутых в предыдущем разделе.

1. Убедитесь в том, что приложение Калькулятор выполняется в системе.
2. Откройте приложение WinObj из Sysinternals с повышенными привилегиями и перейдите в каталог объектов, соответствующий SID контейнера AppContainer для Калькулятора (см. предыдущий эксперимент).



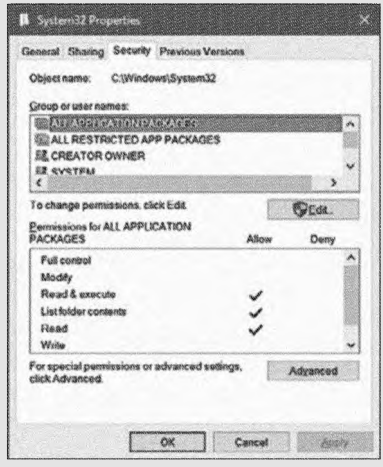
3. Щелкните правой кнопкой мыши на каталоге, выберите команду Properties и щелкните на вкладке Security. Примерный вид экрана показан ниже. SID контейнера AppContainer для Калькулятора обладает разрешениями для перечисления, добавления объектов и добавления подкаталогов (среди других, невидимых из-за прокрутки); по сути это означает, что Калькулятор может создавать объекты ядра в этом каталоге.



4. Откройте локальную папку Калькулятора (%LOCALAPPDATA%\Packages\Microsoft.WindowsCalculator_8wekyb3d8bbwe). Затем щелкните правой кнопкой мыши на подкаталоге Settings, выберите команду Properties и перейдите на вкладку Security. Вы увидите, что SID контейнера AppContainer для Калькулятора имеет полный набор разрешений для папки:

5. В Проводнике откройте каталог %SystemRoot% (например, C:\Windows), щелкните правой кнопкой мыши на каталоге System32, выберите команду Properties и щелкните на вкладке Security. Вы увидете разрешения чтения

и выполнения для всех пакетов приложений и всех ограниченных пакетов приложений (если вы используете Windows 10 версии 1607 и выше):



Ту же информацию можно просмотреть при помощи программы командной строки AccessChk из пакета Sysinternals.

ЭКСПЕРИМЕНТ: ПРОСМОТР ТАБЛИЦЫ АТОМОВ APPCONTAINER

Таблица атомов представляет собой хеш-таблицу, связывающую целые числа со строками, которая используется оконной системой для различных целей идентификации, как, например, для регистрации классов Windows (RegisterClassEx) и нестандартных сообщений. Для просмотра приватной таблицы атомов AppContainer можно воспользоваться отладчиком ядра:

1. Запустите Калькулятор, откройте WinDbg и запустите сеанс локальной отладки ядра.
2. Найдите процесс Калькулятора.

```
lkd> !process 0 1 calculator.exe
PROCESS ffff828cc9ed1080
    SessionId: 1  Cid: 4bd8    Peb: d040bbc000  ParentCid: 03a4
DeepFreeze
    DirBase: 5fccaa000  ObjectTable: ffff950ad9fa2800  HandleCount:
<Data Not Accessible>
    Image: Calculator.exe
    VadRoot ffff828cd2c9b6a0 Vads 168 Clone 0 Private 2938. Modified 3332.
Locked 0.
    DeviceMap ffff950aad2cd2f0
    Token                             ffff950adb313060
   ...
```

Используйте значение маркера в следующих выражениях:

```
lkd> r? @$t1 = @$t0->NumberOfBuckets
lkd> r? @$t0 = (nt!_RTL_ATOM_TABLE*)((nt!_token*)0xffff950adb313060)-
>LowboxNumberEntry->AtomTable
lkd> .for (r @$t3 = 0; @$t3 < @$t1; r @$t3 = @$t3 + 1) {  ?? (wchar_t*)@$t0-
>Buckets[@$t3]->Name }
wchar_t * 0xffff950a'ac39b78a
 "Protocols"
wchar_t * 0xffff950a'ac17b7aa
 "Topics"
wchar_t * 0xffff950a'b2fd282a
 "TaskbarDPI_Deskband"
wchar_t * 0xffff950a'b3e2b47a
 "Static"
wchar_t * 0xffff950a'b3c9458a
 "SysTreeView32"
wchar_t * 0xffff950a'ac34143a
 "UxSubclassInfo"
wchar_t * 0xffff950a'ac5520fa
 "StdShowItem"
wchar_t * 0xffff950a'abc6762a
 "SysSetRedraw"
wchar_t * 0xffff950a'b4a5340a
 "UIA_WindowVisibilityOverridden"
wchar_t * 0xffff950a'ab2c536a
 "True"
...
wchar_t * 0xffff950a'b492c3ea
 "tooltips_class"
wchar_t * 0xffff950a'ac23f46a
 "Save"
wchar_t * 0xffff950a'ac29568a
 "MSDraw"
wchar_t * 0xffff950a'ac54f32a
 "StdNewDocument"
wchar_t * 0xffff950a'b546127a
 "{FB2E3E59-B442-4B5B-9128-2319BF8DE3B0}"
wchar_t * 0xffff950a'ac2e6f4a
 "Status"
wchar_t * 0xffff950a'ad9426da
 "ThemePropScrollBarCtl"
wchar_t * 0xffff950a'b3edf5ba
 "Edit"
wchar_t * 0xffff950a'ab02e32a
 "System"
wchar_t * 0xffff950a'b3e6c53a
 "MDIClient"
wchar_t * 0xffff950a'ac17a6ca
 "StdDocumentName"
wchar_t * 0xffff950a'ac6cbeea
 "StdExit"
wchar_t * 0xffff950a'b033c70a
 "{C56C5799-4BB3-7FAE-7FAD-4DB2F6A53EFF}"
wchar_t * 0xffff950a'ab0360fa
 "MicrosoftTabletPenServiceProperty"
wchar_t * 0xffff950a'ac2f8fea
 "OLEsystem"
```

Возможности AppContainer

Как вы только что убедились, приложения UWP имеют крайне ограниченные права доступа. Как же тогда приложение Microsoft Edge, например, может разбирать пути локальной файловой системы и открывать файлы PDF в папке Документы пользователя? Или как приложение-проигрыватель может воспроизводить MP3-файлы из каталога Музыка? Как бы это ни делалось — напрямую через проверки прав доступа режима ядра или через брокеров (см. следующий раздел) — ключевую роль играют SID возможностей. Давайте посмотрим, откуда они берутся, как создаются и когда используются.

Сначала разработчик UWP-приложения создает манифест приложения, который задает многие подробности о приложении (имя пакета, логотип, ресурсы, поддерживаемые устройства и т. д.). Одним из ключевых элементов управления возможностями является список возможностей в манифесте. Для примера рассмотрим манифест приложения Cortana, находящийся в файле %SystemRoot%\SystemApps\Microsoft.Windows.Cortana_cw5n1h2txywey\AppxManifest.xml:

```
<Capabilities>
        <wincap:Capability Name="packageContents"/>
        <!-- Needed for resolving MRT strings -->
        <wincap:Capability Name="cortanaSettings"/>
        <wincap:Capability Name="cloudStore"/>
        <wincap:Capability Name="visualElementsSystem"/>
        <wincap:Capability Name="perceptionSystem"/>
        <Capability Name="internetClient"/>
        <Capability Name="internetClientServer"/>
        <Capability Name="privateNetworkClientServer"/>
        <uap:Capability Name="enterpriseAuthentication"/>
        <uap:Capability Name="musicLibrary"/>
        <uap:Capability Name="phoneCall"/>
        <uap:Capability Name="picturesLibrary"/>
        <uap:Capability Name="sharedUserCertificates"/>
        <rescap:Capability Name="locationHistory"/>
        <rescap:Capability Name="userDataSystem"/>
        <rescap:Capability Name="contactsSystem"/>
        <rescap:Capability Name="phoneCallHistorySystem"/>
        <rescap:Capability Name="appointmentsSystem"/>
        <rescap:Capability Name="chatSystem"/>
        <rescap:Capability Name="smsSend"/>
        <rescap:Capability Name="emailSystem"/>
        <rescap:Capability Name="packageQuery"/>
        <rescap:Capability Name="slapiQueryLicenseValue"/>
        <rescap:Capability Name="secondaryAuthenticationFactor"/>
        <DeviceCapability Name="microphone"/>
        <DeviceCapability Name="location"/>
        <DeviceCapability Name="wiFiControl"/>
</Capabilities>
```

В списке присутствует множество записей. Например, среди записей присутствуют стандартные SID, связанные с исходным набором возможностей, реализованным

в Windows 8. Они начинаются с SECURITY_CAPABILITY_ — как, например, возможность SECURITY_CAPABILITY_INTERNET_CLIENT, которая является частью RID возможности для APPLICATION PACKAGE AUTHORITY. В результате мы получаем SID S-1-15-3-1 в строковом формате.

Другие записи имеют префикс uap, rescap или wincap. Один из этих префиксов (rescap) относится к ограниченным возможностям. Это возможности, требующие специального внедрения от Microsoft и специального заявления для размещения в магазине. В случае Cortana к их числу относятся такие возможности, как обращение к текстовым сообщениям SMS, электронной почте, контактам, позиционным и пользовательским данным. С другой стороны, возможности Windows (wincap) зарезервированы для Windows и системных приложений. Ни одно приложение не может ими пользоваться. Наконец, к возможностям UAP относятся стандартные возможности, которые могут быть запрошены кем угодно. (Напомним, что UAP — старое название UWP.)

В отличие от первого набора возможностей, связанных с жестко закодированными RID, эти возможности реализуются иначе. Тем самым снимается необходимость постоянного ведения списка хорошо известных RID. Вместо этого в этом режиме возможности могут полностью настраиваться и обновляться «на ходу». Для этого они просто берут строку возможности, преобразуют ее к верхнему регистру и вычисляют хеш SHA-2 полученной строки — по аналогии с тем, как пакетный SID-идентификатор AppContainer является хешем SHA-2 моникера пакета. И снова, поскольку хеши SHA-2 имеют длину 32 байта, для каждой возможности остаются 8 RID, за которыми следует хорошо известное значение SECURITY_CAPABILITY_BASE_RID (3).

Наконец, в списке можно заметить несколько записей DeviceCapability. Они относятся к классам устройств, к которым должны обращаться приложения UWP; их можно опознать по хорошо известным строкам (таким, как показанные выше) или напрямую по GUID, идентифицирующим класс устройства. Вместо одного из двух методов создания SID, описанных выше, в этом случае используется третий! Для таких возможностей GUID преобразуется в двоичный формат и затем отображается на 4 RID (потому что GUID занимает 16 байт). С другой стороны, если вместо него будет задано хорошо известное имя, оно сначала должно быть преобразовано в GUID. Для этого проверяется раздел реестра HKLM\Software\Microsoft\Windows\CurrentVersion\DeviceAccess\CapabilityMappings; он содержит список разделов реестра, связанных с возможностями устройств, и список GUID-идентификаторов, соответствующих этим возможностям. Далее GUID преобразуется в SID, как было показано выше.

ПРИМЕЧАНИЕ Обновленный список поддерживаемых возможностей доступен по адресу *https://msdn.microsoft.com/en-us/windows/uwp/packaging/app-capability-declarations.*

При кодировании всех этих возможностей в маркере применяются два дополнительных правила.

- Как было показано в предыдущем эксперименте, каждый маркер AppContainer содержит свой SID пакета, закодированный в качестве возможности. Он может использоваться системой возможностей для ограничения доступа к конкретному приложению через стандартную проверку безопасности вместо отдельного получения и проверки SID.
- Каждая возможность заново кодируется как SID группы с использованием RID `SECURITY_CAPABILITY_APP_RID` (1024) как дополнительного подчиненного защитного кода (sub-authority).

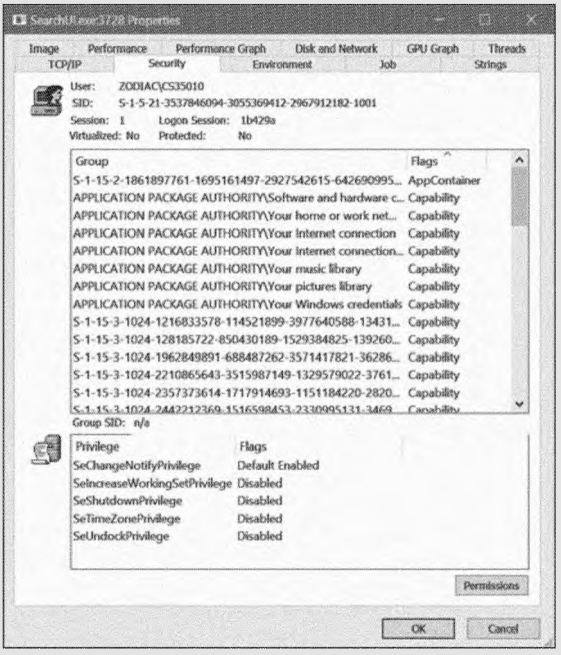
После того как возможности будут закодированы в маркере, различные компоненты системы читают их для определения того, разрешено ли выполнение операции, выполняемой AppContainer. Следует заметить, что большая часть API не документирована, так как взаимодействие с приложениями UWP не поддерживается официально, и его лучше оставить для брокерских служб, встроенных драйверов или компонентов ядра. Например, ядро и драйверы могут использовать API-функцию `RtlCapabilityCheck` для аутентификации доступа к некоторым аппаратным интерфейсам или API-функциям.

Например, диспетчер электропитания проверяет возможность `ID_CAP_SCREENOFF` перед тем, как предоставить разрешение запросу на отключение экрана из AppContainer. Драйвер порта Bluetooth проверяет возможность `bluetoothDiagnostics`, тогда как драйвер идентификации приложения проверяет поддержку EDP (Enterprise Data Protection) через возможность `enterpriseDataPolicy`. В пользовательском режиме можно использовать документированную API-функцию `CheckTokenCapability`, хотя она должна знать SID возможности вместо имени (впрочем, нужное значение можно сгенерировать недокументированной функцией `RtlDeriveCapability SidFromName`). Другой возможный вариант — недокументированная API-функция `CapabilityCheck`, получающая строку.

Наконец, многие службы RPC используют `RpcClientCapabilityCheck` — вспомогательную API-функцию, которая берет на себя чтение маркера и требует только строки возможности. Эта функция очень часто используется многими брокерами и службами с поддержкой WinRT, использующими RPC для взаимодействия с клиентскими приложениями UWP.

ЭКСПЕРИМЕНТ: ПРОСМОТР ВОЗМОЖНОСТЕЙ APPCONTAINER

Чтобы наглядно продемонстрировать все комбинации возможностей и их заполнение в маркере, рассмотрим возможности такого сложного приложения, как Cortana. Вы уже видели его манифест, что позволяет сравнить выходные данные с пользовательским интерфейсом. Для начала вкладка Security процесса SearchUI.exe выглядит так (с сортировкой по столбцу Flags):



Очевидно, что приложение Cortana получило множество возможностей в своем манифесте. Некоторые из них первоначально были в Windows 8 и функционируют как `IsWellKnownSid`, для которых Process Explorer показывает понятное имя. Другие возможности видны только с помощью SID, поскольку они представляют либо хеши, либо GUID, как пояснялось ранее.

Чтобы получить подробную информацию о пакете, на базе которого был создан процесс UWP, вы можете воспользоваться инструментом `UWPList` из загружаемых ресурсов этой книги. Программа может вывести информацию обо всех иммерсивных процессах в системе или об одном процессе по его идентификатору:

```
C:\WindowsInternals>UwpList.exe 3728
List UWP Processes - version 1.1 (C)2016 by Pavel Yosifovich

Building capabilities map... done.

Process ID:   3728
------------------
Image name: C:\Windows\SystemApps\Microsoft.Windows.Cortana_cw5n1h2txyewy\
SearchUI.exe
Package name: Microsoft.Windows.Cortana
Publisher: CN=Microsoft Windows, O=Microsoft Corporation, L=Redmond,
S=Washington, C=US
Published ID: cw5n1h2txyewy
Architecture: Neutral
Version: 1.7.0.14393
AppContainer SID: S-1-15-2-1861897761-1695161497-2927542615-642690995-
327840285-2659745135-2630312742
Lowbox Number: 3
```

```
Capabilities: 32
cortanaSettings (S-1-15-3-1024-1216833578-114521899-3977640588-1343180512-
2505059295-473916851-3379430393-3088591068) (ENABLED)
visualElementsSystem (S-1-15-3-1024-3299255270-1847605585-2201808924-
710406709-3613095291-873286183-3101090833-2655911836) (ENABLED)
perceptionSystem (S-1-15-3-1024-34359262-2669769421-2130994847-3068338639-
3284271446-2009814230-2411358368-814686995) (ENABLED)
internetClient (S-1-15-3-1) (ENABLED)
internetClientServer (S-1-15-3-2) (ENABLED)
privateNetworkClientServer (S-1-15-3-3) (ENABLED)
enterpriseAuthentication (S-1-15-3-8) (ENABLED)
musicLibrary (S-1-15-3-6) (ENABLED)
phoneCall (S-1-15-3-1024-383293015-3350740429-1839969850-1819881064-1569454686-
4198502490-78857879-1413643331) (ENABLED)
picturesLibrary (S-1-15-3-4) (ENABLED)
sharedUserCertificates (S-1-15-3-9) (ENABLED)
locationHistory (S-1-15-3-1024-3029335854-3332959268-2610968494-1944663922-
1108717379-267808753-1292335239-2860040626) (ENABLED)
userDataSystem (S-1-15-3-1024-3324773698-3647103388-1207114580-2173246572-
4287945184-2279574858-157813651-603457015) (ENABLED)
contactsSystem (S-1-15-3-1024-2897291008-3029319760-3330334796-465641623-
3782203132-742823505-3649274736-3650177846) (ENABLED)
phoneCallHistorySystem (S-1-15-3-1024-2442212369-1516598453-2330995131-
3469896071-605735848-2536580394-3691267241-2105387825) (ENABLED)
appointmentsSystem (S-1-15-3-1024-2643354558-482754284-283940418-2629559125-
2595130947-547758827-818480453-1102480765) (ENABLED)
chatSystem (S-1-15-3-1024-2210865643-3515987149-1329579022-3761842879-
3142652231-371911945-4180581417-4284864962) (ENABLED)
smsSend (S-1-15-3-1024-128185722-850430189-1529384825-139260854-329499951-
1660931883-3499805589-3019957964) (ENABLED)
emailSystem (S-1-15-3-1024-2357373614-1717914693-1151184220-2820539834-
3900626439-4045196508-2174624583-3459390060) (ENABLED)
packageQuery (S-1-15-3-1024-1962849891-688487262-3571417821-3628679630-
802580238-1922556387-206211640-3335523193) (ENABLED)
slapiQueryLicenseValue (S-1-15-3-1024-3578703928-3742718786-7859573-
1930844942-2949799617-2910175080-1780299064-4145191454) (ENABLED)
S-1-15-3-1861897761-1695161497-2927542615-642690995-327840285-2659745135-
2630312742 (ENABLED)
S-1-15-3-787448254-1207972858-3558633622-1059886964 (ENABLED)
S-1-15-3-3215430884-1339816292-89257616-1145831019 (ENABLED)
S-1-15-3-3071617654-1314403908-1117750160-3581451107 (ENABLED)
S-1-15-3-593192589-1214558892-284007604-3553228420 (ENABLED)
S-1-15-3-3870101518-1154309966-1696731070-4111764952 (ENABLED)
S-1-15-3-2105443330-1210154068-4021178019-2481794518 (ENABLED)
S-1-15-3-2345035983-1170044712-735049875-2883010875 (ENABLED)
S-1-15-3-3633849274-1266774400-1199443125-2736873758 (ENABLED)
S-1-15-3-2569730672-1095266119-53537203-1209375796 (ENABLED)
S-1-15-3-2452736844-1257488215-2818397580-3305426111 (ENABLED)
```

В выводе указано полное имя пакета, каталог с исполняемым файлом, AppContainer SID, информация об издателе, версия и список возможностей. Также указан номер LowBox — локальный индекс приложения.

Наконец, эти свойства можно просмотреть в отладчике ядра при помощи команды `!token`.

Некоторые приложения UWP называются *доверенными*; хотя они используют платформу Windows Runtime, как и другие приложения UWP, они не выполняются в контейнере AppContainer, а их уровень целостности выше Low. Классический пример — приложение System Settings (%SystemRoot%\ImmersiveControlPanel\SystemSettings.exe); это выглядит разумно, так как приложение должно иметь возможность внести изменения в систему, которые невозможно было бы внести из процесса, работающего под управлением AppContainer. При просмотре маркера вы увидите те же три атрибута — `PKG`, `SYSAPPID` и `PKGHOSTID`, которые подтверждают, что приложение является пакетным, несмотря на отсутствие маркера AppContainer.

AppContainer и пространство имен объектов

Настольные приложения могут легко совместно использовать объекты ядра по имени. Например, предположим, что процесс A создает объект события вызовом `CreateEvent(Ex)` с именем `MyEvent`. Он получает дескриптор, который может в дальнейшем использоваться для работы с событием. Процесс B, выполняемый в том же сеансе, может вызвать `CreateEvent(Ex)` или `OpenEvent` с тем же именем и (при наличии необходимых разрешений — но обычно это условие выполняется при выполнении в одном сеансе) получает другой дескриптор для того же объекта события. Если теперь процесс A вызовет `SetEvent` для объекта события, в то время как процесс B был заблокирован вызовом `WaitForSingleObject` для этого дескриптора события, ожидающий поток процесса B будет освобожден, потому что это тот же объект события. Совместный доступ работает, потому что именованные объекты создаются в каталоге диспетчера объектов \Sessions\x\BaseNamedObjects, как показано в окне программы WinObj из пакета Sysinternals на рис. 7.18.

Кроме того, настольные приложения могут совместно использовать объекты между сеансами, для чего имя снабжается префиксом Global\. При этом объект создается в каталоге объектов сеанса 0 в \BaseNamedObjects (см. рис. 7.18).

У процессов на базе AppContainer корневое пространство имен находится в иерархии \Sessions\x\AppContainerNamedObjects\<AppContainerSID>. Так как каждый контейнер AppContainer имеет свой SID-идентификатор AppContainer, два приложения UWP никак не смогут совместно использовать объекты ядра. Способность создания именованных объектов ядра в пространстве имен объектов сеанса 0 для процессов AppContainer запрещена. На рис. 7.19 показан каталог объектов ядра для UWP-приложения Калькулятор.

Приложения UWP, желающие организовать совместное использование данных, могут воспользоваться четко определенными контрактами под управлением среды Windows Runtime. (За дополнительной информацией обращайтесь к документации MSDN.)

Совместное использование объектов ядра между настольными приложениями и UWP-приложениями возможно; более того, оно часто осуществляется брокерскими службами. Например, при запросе доступа к файлу в папке Документы (и про-

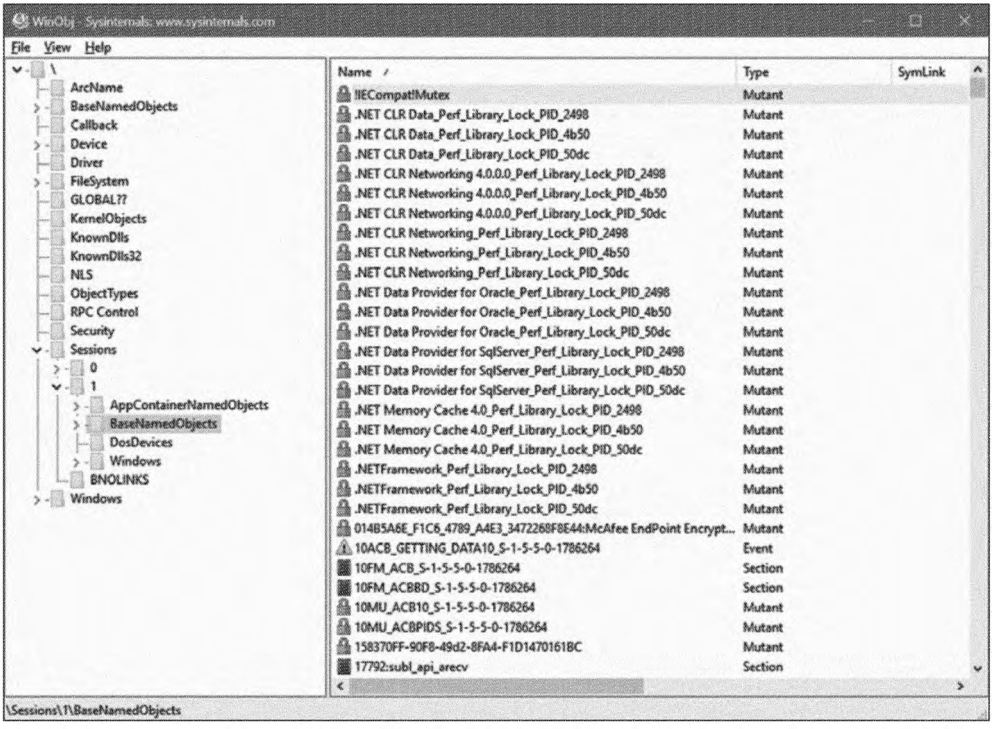


Рис. 7.18. Каталог диспетчера объектов для именованных объектов

хождении проверки правильной возможности) из брокера выбора файлов при ложение UWP получит дескриптор файла, который оно сможет использовать для прямого чтения или записи без лишних затрат на маршалинг запросов. Для этого брокер дублирует полученный им дескриптор файла прямо в таблице дескрипторов приложения UWP. (Дополнительная информация о дублировании дескрипторов приводится в главе 8 части 2.) Чтобы дополнительно упростить ситуацию, подсистема ALPC (также описанная в главе 8) допускает автоматическую передачу дескрипторов через атрибуты дескриптора ALPC, а службы RPC (Remote Procedure Call), использующие ALPC в качестве базового протокола, могут задействовать эту функциональность как часть своих интерфейсов. Пригодные для упаковки дескрипторы в файле IDL будут автоматически передаваться в подсистеме ALPC.

Помимо официальных брокерских служб RPC, настольное приложение может создать именованный (и даже неименованный) объект обычным способом, а затем воспользоваться функцией `DuplicateHandle` для ручного внедрения дескриптора того же объекта в процесс UWP. Такая схема работает, потому что настольные приложения обычно выполняются на среднем уровне целостности и ничто не мешает им дублировать дескрипторы в процессы UWP — ограничения действуют только в обратном направлении.

Рис. 7.19. Каталог диспетчера объектов для Калькулятора

ПРИМЕЧАНИЕ Взаимодействие между настольным приложением и UWP обычно не обязательно, потому что приложение в магазине не может зависеть от наличия на устройстве парного приложения-«компаньона». Возможность внедрения дескрипторов в приложение UWP может пригодиться в специальных ситуациях — например, при использовании настольного моста (Centennial) для преобразования настольного приложения в приложение UWP и взаимодействия с другим, заведомо существующим настольным приложением.

Дескрипторы AppContainer

В типичном приложении Win32 присутствие локального для сеанса и глобального каталога BaseNamedObjects гарантируется подсистемой Windows, так как они создаются при загрузке и создании сеанса. К сожалению, каталог AppContainerBaseNamedObjects создается самим запускающим приложением. В случае активации UWP это доверенная служба DComLaunch, но стоит напомнить, что не все контейнеры AppContainer обязательно связаны с UWP. Их также можно создавать вручную с нужными атрибутами создания процесса. (За дополнительной информацией об этих атрибутах обращайтесь к главе 3.) В этом случае недоверенное приложение может создать каталог объектов (и необходимые символические ссылки внутри него), в результате чего приложение сможет закрывать дескрипторы из приложения AppContainer. Даже без злого умысла исходное запускающее приложение может завершить работу, стереть дескрипторы и уничтожить каталог объектов, относящийся к AppContainer. Чтобы избежать подобных ситуаций, маркеры AppContainer могут хранить массив дескрипторов, которые гарантиро-

ванно существуют на протяжении всего жизненного цикла любого приложения, использующего маркер. Эти дескрипторы обычно передаются при создании маркера AppContainer (с `NtCreateLowBoxToken`) и дублируются как дескрипторы ядра.

По аналогии с таблицей атомов, существующих на уровне контейнера AppContainer, используется специальная структура `SEP_CACHED_HANDLES_ENTRY`; она создается на основе хеш-таблицы, хранящейся в структуре сеанса входа для данного пользователя. (За дополнительной информацией о сеансах входа обращайтесь к разделу «Вход в систему» этой главы.) Структура содержит массив дескрипторов ядра, продублированных при создании маркера AppContainer. Они будут закрыты либо при уничтожении маркера (потому что приложение завершает работу), либо при выходе пользователя из системы (что приведет к уничтожению сеанса входа).

ЭКСПЕРИМЕНТ: ПРОСМОТР ДЕСКРИПТОРОВ, ХРАНИМЫХ В МАРКЕРЕ

Чтобы просмотреть дескрипторы, хранящиеся в маркере, выполните следующие действия.

1. Запустите Калькулятор и откройте сеанс локальной отладки ядра.
2. Найдите сеанс Калькулятора:

```
lkd> !process 0 1 calculator.exe
PROCESS ffff828cc9ed1080
    SessionId: 1  Cid: 4bd8    Peb: d040bbc000  ParentCid: 03a4
DeepFreeze
    DirBase: 5fccaa000  ObjectTable: ffff950ad9fa2800  HandleCount:
<Data Not Accessible>
    Image: Calculator.exe
    VadRoot ffff828cd2c9b6a0 Vads 168 Clone 0 Private 2938. Modified 3332.
Locked 0.
    DeviceMap ffff950aad2cd2f0
    Token                             ffff950adb313060
    ElapsedTime                       1 Day 08:01:47.018
    UserTime                          00:00:00.015
    KernelTime                        00:00:00.031
    QuotaPoolUsage[PagedPool]         465880
    QuotaPoolUsage[NonPagedPool]      23288
    Working Set Sizes (now,min,max)  (7434, 50, 345) (29736KB, 200KB, 1380KB)
    PeakWorkingSetSize                11097
    VirtualSize                       303 Mb
    PeakVirtualSize                   314 Mb
    PageFaultCount                    21281
    MemoryPriority                    BACKGROUND
    BasePriority                      8
    CommitCharge                      4925
    Job                               ffff828cd4914060
```

3. Выведите маркер командой `dt`. (Не забудьте замаскировать младшие 3 или 4 бита, если они отличны от нуля.)

```
lkd> dt nt!_token ffff950adb313060
   +0x000 TokenSource      : _TOKEN_SOURCE
   +0x010 TokenId          : _LUID
   +0x018 AuthenticationId : _LUID
```

```
   +0x020 ParentTokenId    : _LUID
   ...
   +0x0c8 TokenFlags       : 0x4a00
   +0x0cc TokenInUse       : 0x1 ''
   +0x0d0 IntegrityLevelIndex : 1
   +0x0d4 MandatoryPolicy  : 1
   +0x0d8 LogonSession     : 0xffff950a'b4bb35c0 _SEP_LOGON_SESSION_
                             REFERENCES
   +0x0e0 OriginatingLogonSession : _LUID
   +0x0e8 SidHash          : _SID_AND_ATTRIBUTES_HASH
   +0x1f8 RestrictedSidHash : _SID_AND_ATTRIBUTES_HASH
   +0x308 pSecurityAttributes : 0xffff950a'e4ff57f0 _AUTHZBASEP_SECURITY_
ATTRIBUTES_INFORMATION
   +0x310 Package          : 0xffff950a'e00ed6d0 Void
   +0x318 Capabilities     : 0xffff950a'e8e8fbc0 _SID_AND_ATTRIBUTES
   +0x320 CapabilityCount  : 1
   +0x328 CapabilitiesHash : _SID_AND_ATTRIBUTES_HASH
   +0x438 LowboxNumberEntry : 0xffff950a'b3fd55d0 _SEP_LOWBOX_NUMBER_ENTRY
+0x440 LowboxHandlesEntry : 0xffff950a'e6ff91d0 _SEP_LOWBOX_HANDLES_ENTRY
   +0x448 pClaimAttributes : (null)
   ...
```

4. Выведите поле LowboxHandlesEntry:

```
lkd> dt nt!_sep_lowbox_handles_entry 0xffff950a'e6ff91d0
   +0x000 HashEntry        : _RTL_DYNAMIC_HASH_TABLE_ENTRY
   +0x018 ReferenceCount   : 0n10
   +0x020 PackageSid       : 0xffff950a'e6ff9208 Void
   +0x028 HandleCount      : 6
   +0x030 Handles          : 0xffff950a'e91d8490  -> 0xffffffff'800023cc Void
```

5. Количество маркеров равно 6. Выведем их значения:

```
lkd> dq 0xffff950ae91d8490 L6
ffff950a'e91d8490  ffffffff'800023cc ffffffff'80001e80
ffff950a'e91d84a0  ffffffff'80004214 ffffffff'8000425c
ffff950a'e91d84b0  ffffffff'800028c8 ffffffff'80001834
```

6. Как нетрудно увидеть, эти дескрипторы являются дескрипторами ядра, поскольку их значения начинаются с 0xffffffff (64 бита). Теперь можно воспользоваться командой !handle для просмотра отдельных дескрипторов. Два примера для шести дескрипторов из предыдущего примера:

```
lkd> !handle ffffffff'80001e80

PROCESS ffff828cd71b3600
    SessionId: 1  Cid: 27c4    Peb: 3fdfb2f000  ParentCid: 2324
DirBase: 80bb85000  ObjectTable: ffff950addabf7c0  HandleCount:
<Data Not Accessible>
    Image: windbg.exe

Kernel handle Error reading handle count.

80001e80: Object: ffff950ada206ea0  GrantedAccess: 0000000f (Protected)
(Inherit) (Audit) Entry: ffff950ab5406a00
Object: ffff950ada206ea0  Type: (ffff828cb66b33b0) Directory
    ObjectHeader: ffff950ada206e70 (new version)
        HandleCount: 1  PointerCount: 32770
```

```
        Directory Object: ffff950ad9a62950  Name: RPC Control

        Hash Address          Type                      Name
        ---- -------          ----                      ----
         23  ffff828cb6ce6950 ALPC Port
OLE376512B99BCCA5DE4208534E7732
lkd> !handle ffffffff'800028c8

PROCESS ffff828cd71b3600
    SessionId: 1  Cid: 27c4    Peb: 3fdfb2f000  ParentCid: 2324
    DirBase: 80bb85000  ObjectTable: ffff950addabf7c0  HandleCount: <Data
Not Accessible>
    Image: windbg.exe

Kernel handle Error reading handle count.

800028c8: Object: ffff950ae7a8fa70  GrantedAccess: 000f0001 (Audit) Entry:
ffff950acc426320
Object: ffff950ae7a8fa70  Type: (ffff828cb66296f0) SymbolicLink
    ObjectHeader: ffff950ae7a8fa40 (new version)
        HandleCount: 1  PointerCount: 32769
        Directory Object: ffff950ad9a62950  Name: Session
        Flags: 00000000 ( Local )
        Target String is '\Sessions\1\AppContainerNamedObjects
\S-1-15-2-466767348-3739614953-2700836392-1801644223-4227750657
-1087833535-2488631167'
```

Наконец, поскольку возможность ограничения именованных объектов конкретным каталогом пространства имен объектов является важным средством безопасности для изоляции доступа к именованным объектам, в будущем (на момент написания книги) обновлении Windows 10 Creators Update появилась дополнительная возможность маркера, называемая *изоляцией BNO* (BNO означает «Base Named Objects»). При использовании той же структуры `SEP_CACHE_HANDLES_ENTRY` в структуру `TOKEN` добавляется новое поле `BnoIsolationHandlesEntry` с типом `SepCachedHandlesEntryBnoIsolation` вместо `SepCachedHandlesEntryLowbox`. Для использования этой функциональности должен использоваться специальный атрибут процесса (за дополнительной информацией обращайтесь к главе 3) с префиксом изоляции и списком дескрипторов. На этой стадии используется тот же механизм LowBox, но вместо каталога объектов SID AppContainer используется каталог с префиксом, обозначенным в атрибуте.

Брокеры

Так как процессы AppContainer не имеют практически никаких разрешений, кроме тех, которые неявно предоставляются возможностями, некоторые типичные операции не могут выполняться AppContainer напрямую и требуют помощи. (Возможностей для них не существует, так как эти операции относятся к слишком низкому уровню, чтобы быть видимыми для пользователей в магазине, и ими трудно управлять.) Примеры — выбор файлов в стандартном окне открытия файлов или печать

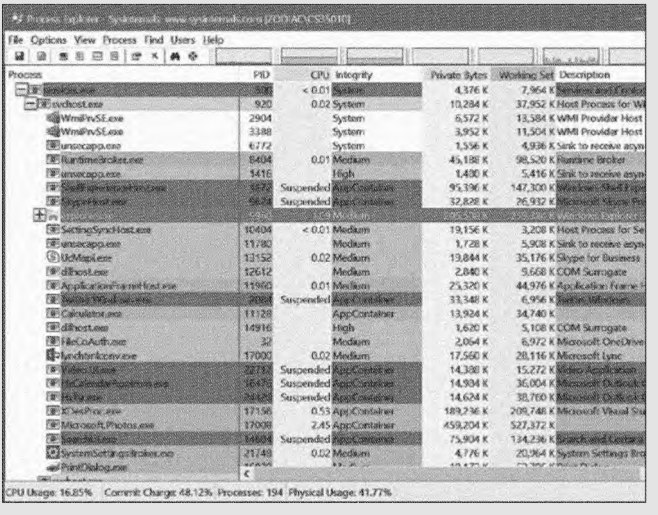
из диалогового окна печати. Для таких и других подобных операций Windows предоставляет вспомогательные процессы, которые называются *брокерами* (brokers). Для управления ими используется системный процесс RuntimeBroker.exe.

Процесс AppContainer, которому требуется эта функциональность, взаимодействует с RuntimeBroker через защищенный ALPC-канал, а Runtime Broker инициирует создание запрошенного процесса-брокера. Примеры — %SystemRoot%\PrintDialog\PrintDialog.exe и %SystemRoot%\System32\PickerHost.exe.

ЭКСПЕРИМЕНТ: БРОКЕРЫ

Этот эксперимент показывает, как запускаются и завершаются процессы-брокеры.

1. Щелкните на кнопке Пуск (Start), введите текст Фото и выберите пункт Фотографии для запуска встроенного приложения Windows 10 Фотографии (Photos).
2. Откройте Process Explorer, переключите список процессов в иерархическое представление и найдите процесс Microsoft.Photos.exe. Разместите два окна рядом друг с другом.
3. В приложении Фотографии (Photos) выберите файл с графическим изображением и выберите команду Печать (Print) из верхнего меню (или щелкните правой кнопкой мыши на изображении и выберите команду Печать (Print) в открывшемся меню. Откроется диалоговое окно Печать (Print), и в Process Explorer должен появиться вновь созданный брокер (PrintDialog.exe). Обратите внимание: процессы являются потомками одного процесса Svchost. (Все процессы UWP запускаются службой DCOMLaunch, хостом которой является этот процесс.)



4. Закройте диалоговое окно Печать (Print). Процесс PrintDialog.exe должен завершиться.

Вход в систему

Интерактивный вход в систему (в отличие от сетевого входа) осуществляется следующими способами:

- процесс входа в систему (Winlogon);
- процесс интерфейса входа пользователя в систему (LogonUI) и его поставщики учетных данных;
- Lsass.exe;
- один или несколько пакетов аутентификации;
- SAM или Active Directory.

Пакеты аутентификации являются DLL-библиотеками, выполняющими проверки аутентификации. Kerberos — пакет аутентификации Windows для интерактивного входа в домен, а MSV1_0 — пакет аутентификации Windows для интерактивного входа на локальный компьютер, для доменных входов на доверенные домены под управлением версий Windows, предшествующих Windows 2000, и для тех случаев, когда недоступен контроллер доменов.

Winlogon — доверенный процесс, отвечающий за управление взаимодействиями с пользователем, связанными с безопасностью. Им координируются вход в систему, запуск первого процесса пользователя при входе в систему, обработка выхода из системы и управление рядом других операций, относящихся к безопасности, включая запуск LogonUI для ввода паролей при входе в систему, изменении паролей и блокировке и разблокировке рабочей станции. Процесс Winlogon должен гарантировать, что операции, связанные с безопасностью, невидимы любым другим активным процессам. Например, Winlogon гарантирует, что не пользующийся доверием процесс не может получить управление рабочим столом в ходе одной из таких операций, получив тем самым доступ к паролю.

Winlogon при получении имени и пароля учетной записи пользователя зависит от установленных в системе поставщиков учетных записей. Эти поставщики являются COM-объектами, которые находятся внутри DLL-библиотек. Исходными поставщиками являются authui.dll, SmartcardCredentialProvider.dll и FaceCredentialProvider.dll, и они поддерживают пароль, PIN аутентификации смарткарты и аутентификацию с распознаванием лица. Разрешение установки других поставщиков учетных данных позволяет Windows использовать различные механизмы идентификации пользователей. Например, сторонний разработчик может предоставить поставщика учетных записей, который использует для идентификации пользователей устройство распознавания отпечатка пальца и извлекает пароли этих пользователей из зашифрованной базы данных. Поставщики учетных данных перечислены в HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Authentication\Credential Providers, где каждый подраздел идентифицирует класс поставщика учетных данных по идентификатору COM CLSID. (Сам идентификатор CLSID должен быть зарегистрирован

в HKCR\CLSID, как и любой другой класс COM.) Вы можете воспользоваться утилитой CPlist.exe, включенной в состав загружаемых ресурсов книги, для вывода списка поставщиков учетных данных с CLSID, удобным именем и DLL-библиотекой реализации.

Чтобы защитить адресное пространство Winlogon от ошибок поставщиков учетных данных, которые могут привести к сбою процесса Winlogon (что, в свою очередь, приведет к системному сбою, поскольку Winlogon считается критическим системным процессом), для фактической загрузки поставщиков учетных данных и демонстрации пользователю Windows-интерфейса входа в систему используется отдельный процесс LogonUI.exe. Этот процесс запускается по запросу, как только Winlogon требуется присутствие пользовательского интерфейса, а выход из него осуществляется после завершения нужного действия. Это также позволяет Winlogon просто перезапустить новый процесс LogonUI в случае сбоя, возникшего по какой-либо причине.

Winlogon является единственным процессом, перехватывающим запросы на вход в систему с клавиатуры, которые отправляются через RPC-сообщения из Win32k.sys. Winlogon тут же запускает приложение LogonUI, чтобы вывести пользователю интерфейс входа в систему. После получения от поставщика учетных данных имени пользователя и пароля Winlogon вызывает LSASS, чтобы аутентифицировать попытку пользователя войти в систему. Если пользователь аутентифицирован, процесс входа в систему активирует оболочку входа в систему от имени этого пользователя. Взаимодействие между компонентами, участвующими во входе в систему, показано на рис. 7.20.

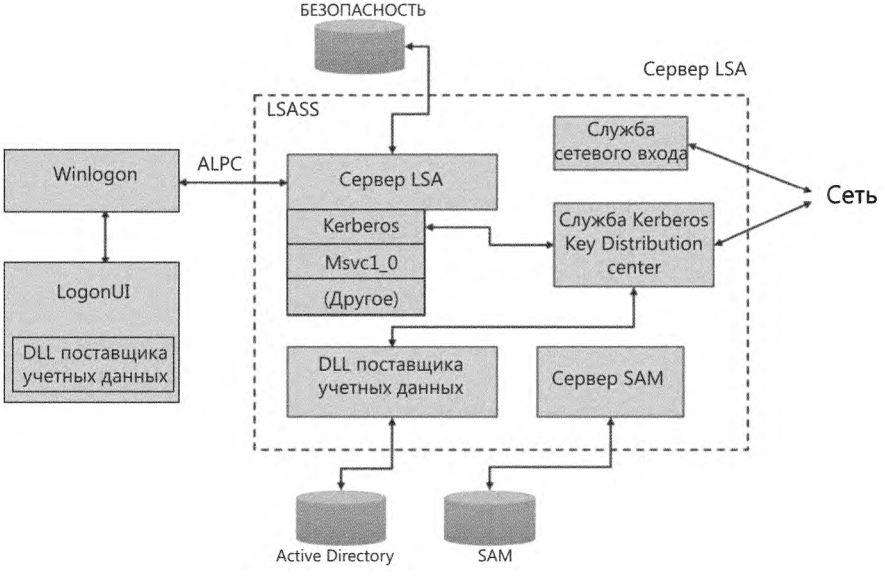


Рис. 7.20. Компоненты, участвующие во входе в систему

Кроме поддержки альтернативных поставщиков учетных данных LogonUI может загрузить дополнительные DLL-библиотеки сетевого поставщика, необходимые для проведения вторичной аутентификации. Эта возможность позволяет нескольким сетевым поставщикам собрать всю информацию, касающуюся идентификации и аутентификации в ходе обычного входа в систему. Пользователь, входящий в систему Windows, может одновременно быть аутентифицирован на сервере Linux. Затем этот пользователь сможет получить доступ к ресурсам сервера UNIX с Windows-машины без необходимости прохождения дополнительной аутентификации. Такая возможность известна как одна из форм единого входа.

Инициализация Winlogon

В ходе инициализации системы, перед тем как станет активным любое пользовательское приложение, процесс Winlogon выполняет следующие действия, чтобы убедиться в том, что он управляет рабочей станцией, в то время как система готова к взаимодействию с пользователем.

1. Создает и открывает интерактивную станцию окна (например, \Sessions\1\Windows\WindowStations\WinSta0 в пространстве имен диспетчера объектов) для представления клавиатуры, мыши и монитора. Winlogon создает дескриптор безопасности для станции, у которого имеется только один ACE-элемент, содержащий только один системный SID. Этот уникальный дескриптор безопасности гарантирует, что никакой другой процесс не сможет получить доступ к рабочей станции, пока это не будет явным образом разрешено процессом Winlogon.

2. Создает и открывает два рабочих стола: рабочий стол приложения (\Sessions\1\Windows\WinSta0\Default, известный также как интерактивный рабочий стол) и рабочий стол Winlogon (\Sessions\1\Windows\WinSta0\Winlogon, известный также как защищенный рабочий стол). Защита рабочего стола Winlogon устроена таким образом, что доступ к этому рабочему столу может получить только процесс Winlogon. Другой рабочий стол позволяет иметь доступ к нему как процессу Winlogon, так и пользователям. Такой порядок означает, что в любое время, когда активен рабочий стол Winlogon, никакой другой процесс не имеет доступа ни к какому активному коду или данным, связанным с рабочим столом. Windows использует это свойство для защиты операций безопасности, использующих пароли и блокирование и разблокирование рабочего стола.

3. Перед тем как кто-нибудь зарегистрируется на компьютере, видимым будет рабочий стол Winlogon. После того как пользователь вошел в систему, нажатие комбинации клавиш (по умолчанию Ctrl+Alt+Delete) переключает рабочий стол с исходного (Default) на Winlogon и запускает LogonUI. (Этим объясняется вопрос, почему все окна на вашем интерактивном рабочем столе кажутся исчезнувшими, когда вы нажимаете комбинацию Ctrl+Alt+Delete, а затем возвра-

щаются назад, когда вы освобождаете диалоговое окно безопасности Windows.) Таким образом, SAS всегда выдает рабочий стол безопасности, управляемый процессом Winlogon.

4. Устанавливает ALPC-подключение к LSASS. Это подключение будет использоваться для обмена информацией в процессе входа в систему, выхода из системы и операций с паролями, который будет осуществляться с помощью вызова функции `LsaRegisterLogonProcess`.

5. Регистрирует Winlogon сервер RPC-сообщений, который прослушивает уведомления SAS, выхода из системы и блокировки рабочей станции от Win32k. Эта мера не дает троянским программам получать контроль над экраном при вводе комбинации SAS.

ПРИМЕЧАНИЕ Процесс Wininit (см. главу 3) выполняет действия, подобные рассмотренным в пунктах 1 и 2, позволяя устаревшим интерактивным службам выполняться в сеансе 0, чтобы отображать окна, но он не выполняет никаких других действий, поскольку сеанс 0 не доступен для входа пользователя в систему.

КАК РЕАЛИЗОВАНА SAS

Безопасность SAS обеспечивается тем, что ни одно приложение не может перехватить комбинацию клавиш Ctrl+Alt+Delete или помешать Winlogon получить эту комбинацию. Win32k.sys резервирует комбинацию клавиш Ctrl+Alt+Delete таким образом, что, как только система ввода Windows (реализованная как простой поток ввода в Win32k) видит эту комбинацию, она отправляет RPC-сообщение серверу сообщений Winlogon, который прослушивает подобные уведомления. Нажатия клавиш, соответствующие зарегистрированной комбинации, не отправляются никакому другому процессу, кроме того, который ее зарегистрировал, и только поток, зарегистрировавший комбинацию клавиш, может отменить регистрацию, поэтому троянские приложения не могут снять регистрацию принадлежности SAS процессу Winlogon.

Windows-функция `SetWindowsHook` позволяет приложению установить процедуру перехвата, которая вызывается при каждом нажатии комбинации клавиш, даже перед обработкой клавиш быстрого вызова, и она позволяет перехватчику подавлять действие комбинации клавиш. Но код Windows, используемый для обработки клавиш быстрого вызова, содержит особый вариант обработки для Ctrl+Alt+Delete, который отключает перехват, поэтому данная комбинация клавиш не перехватывается. Кроме того, если интерактивный рабочий стол заблокирован, обрабатывается только та комбинация клавиш, которой владеет процесс Winlogon.

Рабочий стол Winlogon, созданный в ходе инициализации, становится активным рабочим столом. Когда рабочий стол Winlogon активен, он всегда заблокирован. Winlogon разблокирует свой рабочий стол только для переключения на рабочий стол приложения или на рабочий стол заставки. (Заблокировать рабочий стол или снять его блокировку может только процесс Winlogon.)

Этапы входа пользователя в систему

Вход в систему начинается, когда пользователь нажимает комбинацию SAS (Ctrl+Alt+Delete). После нажатия SAS процесс Winlogon запускает процесс LogonUI, который вызывает поставщика учетных данных для получения имени пользователя и пароля. Winlogon также создает для этого пользователя уникальный локальный SID входа в систему, который назначается данному экземпляру рабочего стола (куда входят клавиатура, экран и мышь). Winlogon передает этот SID процессу LSASS в качестве части вызова функции LsaLogonUser. Если вход пользователя в систему пройдет успешно, этот SID будет включен в маркер процесса входа в систему, на этом этапе создается защита доступа к рабочему столу. Например, еще один вход под той же учетной записью, но в другую систему не сможет вести запись на рабочий стол первой машины, поскольку эта вторая регистрация не попадет в маркер рабочего стола первой регистрации.

После ввода имени пользователя и пароля Winlogon извлекает дескриптор пакета путем вызова LSASS-функции `LsaLookupAuthenticationPackage`. Пакеты аутентификации перечислены в разделе реестра HKLM\SYSTEM\CurrentControlSet\Control\Lsa. Winlogon передает информацию входа в систему пакету аутентификации через функцию `LsaLogonUser`. Как только пакет проведет аутентификацию пользователя, Winlogon продолжает процесс входа в систему для этого пользователя. Если ни один из пакетов аутентификации не покажет успешного входа в систему, процесс входа прекращается.

Для интерактивного входа в систему Windows использует два стандартных пакета аутентификации:

- **MSV1_0.** Исходным пакетом аутентификации на автономной Windows-системе является MSV1_0 (Msv1_0.dll), который реализует протокол LAN Manager 2. LSASS также использует MSV1_0 на компьютерах, являющихся частью домена для аутентификации на доменах и компьютерах с версиями Windows, предшествующими версии Windows 2000, которые не могут определить с целью аутентификации местонахождение доменного контроллера. (Компьютеры, отключенные от сети, относятся к этой последней категории.)
- **Kerberos.** Пакет аутентификации Kerberos (Kerberos.dll) используется на компьютерах, входящих в домены Windows. Пакет Windows Kerberos во взаимодействии со службами Kerberos, выполняемыми на контроллере домена, поддерживает протокол Kerberos. Этот протокол основан на Internet RFC 1510. (Подробную информацию о стандарте Kerberos можно найти на веб-сайте Internet Engineering Task Force [IETF] по адресу *www.ietf.org*.)

MSV1_0

Пакет аутентификации MSV1_0 принимает имя пользователя и хешированную версию пароля и отправляет запрос локальному SAM-администратору для извлече-

ния информации об учетной записи, включающей хешированный пароль, группы, в которые входит пользователь, и любые ограничения учетной записи. MSV1_0 сначала проверяет ограничения учетной записи, например период времени или тип разрешенного доступа. Если пользователь не может войти в систему по причине ограничений в базе данных SAM, вызов входа в систему завершается отказом, и MSV1_0 возвращает LSA статус ошибки.

Затем MSV1_0 сравнивает хешированный пароль и имя пользователя с той информацией, которая была получена из SAM. В случае входа в кэшированный домен, MSV1_0 обращается к кэшированной информации с помощью LSASS-функций, которые сохраняют и извлекают «секреты» из базы данных LSA (куст реестра SECURITY). Если информация совпадает, MSV1_0 генерирует LUID для сеанса входа в систему и создает сеанс входа в систему путем вызова LSASS, связывая этот уникальный идентификатор с сеансом и передавая информацию, необходимую для создания в итоге маркера доступа для пользователя. (Следует напомнить, что маркер доступа включает SID пользователя, SID-идентификаторы групп и назначенные привилегии.)

ПРИМЕЧАНИЕ Учтите, что MSV1_0 не кэширует целый хеш пароля пользователя в реестре, поскольку это позволит кому-нибудь, имеющему физический доступ к системе, без особого труда несанкционированно вскрыть доменную учетную запись пользователя и получить доступ к зашифрованным файлам и к сетевым ресурсам, к которым имеет доступ авторизованный пользователь. Вместо этого кэшируется половина хеша. Этой кэшированной половины хеша вполне достаточно для проведения проверки правильности пользовательского пароля, но недостаточно для получения доступа к EFS-ключам и для аутентификации в качестве пользователя домена, поскольку эти действия требуют полного хеша.

Если пакету MSV1_0 нужна аутентификация с использованием удаленной системы, как при входе пользователя в заслуживающий доверия домен под управлением версии Windows, предшествующей версии Windows 2000, MSV1_0 использует службу Netlogon для обмена данными с экземпляром Netlogon на удаленной системе. Netlogon на удаленной системе взаимодействует с пакетом аутентификации MSV1_0 на этой системе, возвращая результаты аутентификации той системе, на которой выполнялся вход.

Kerberos

Основной поток операций для аутентификации Kerberos аналогичен потоку для MSV1_0. Но в большинстве случаев входы в домены выполняются с входящих с них рабочих станций или серверов (а не с контроллера домена), поэтому пакет аутентификации должен вести обмен данными по сети в качестве части процесса аутентификации. Пакет справляется с этим путем обмена данными через порт Kerberos TCP/IP (порт 88) при наличии на контроллере домена службы Kerberos. Служба Kerberos Key Distribution Center (Kdcsvc.dll), которая реализует протокол аутентификации Kerberos, выполняется в LSASS-процессе на контроллерах домена.

После проверки приемлемости хешированной информации об имени пользователя и пароле с помощью принадлежащих Active Directory объектов учетной записи пользователя (с использованием Ntdsa.dll), Kdcsvc возвращает LSASS учетные данные домена, в которых по сети той системе, с которой осуществляется вход, возвращаются результаты аутентификации и учетные данные входа пользователя в домен (если вход был выполнен успешно).

ПРИМЕЧАНИЕ Описание аутентификации с использованием Kerberos сильно упрощено, но оно подчеркивает роли задействованных компонентов. Хотя аутентификация с использованием Kerberos играет ключевую роль в распределенной доменной безопасности в Windows, подробности этого процесса выходят за рамки данной книги.

После того как вход в систему был аутентифицирован, LSASS обращается к базе данных локальной политики за разрешенными пользователю доступами, включая интерактивные, сетевые, пакетные или служебные процессы. Если требуемый вход в систему не соответствует разрешенным доступам, попытка входа в систему будет прекращена. LSASS удаляет только что созданный сеанс входа в систему, очищая любые его структуры данных, а затем возвращая отказ процессу Winlogon, который, в свою очередь, выводит пользователю соответствующее сообщение. Если запрошенный доступ разрешен, LSASS добавляет соответствующие дополнительные идентификаторы безопасности (например, Everyone, Interactive и т. п.). Затем LSASS проверяет свою базу данных политики на предмет наличия любых предоставленных привилегий для всех SID-идентификаторов для этого пользователя и добавляет эти привилегии к принадлежащему этому пользователю маркеру доступа.

Когда подсистема LSASS аккумулирует всю нужную информацию, она вызывает исполняющую систему для создания маркера доступа. Исполняющая система создает первичный маркер доступа для интерактивного или служебного входа в систему и маркер заимствования для сетевого входа в систему. После успешного создания маркера доступа LSASS дублирует маркер, создавая дескриптор, который может быть передан Winlogon, и закрывает свой собственный дескриптор. Если нужно, операция входа в систему подвергается аудиту. В этот момент LSASS возвращает сообщение об успехе процессу Winlogon наряду с дескриптором маркера доступа, LUID для сеанса входа в систему и профильной информацией, если таковая имеется, которая была возвращена пакетом аутентификации.

ЭКСПЕРИМЕНТ: ВЫВОД СПИСКА АКТИВНЫХ СЕАНСОВ ВХОДА В СИСТЕМУ

Если существует хотя бы один маркер для заданного LUID сеанса входа в систему, Windows рассматривает сеанс входа в систему в качестве активного. Для вывода списка активных сеансов входа в систему можно воспользоваться инструментальным средством LogonSessions из набора Sysinternals, которое использует функцию `LsaEnumerateLogonSessions` (документированную в Windows SDK):

```
C:\WINDOWS\system32>logonsessions

LogonSessions v1.4 - Lists logon session information
Copyright (C) 2004-2016 Mark Russinovich
Sysinternals - www.sysinternals.com

[0] Logon session 00000000:000003e7:
    User name:    WORKGROUP\ZODIAC$
    Auth package: NTLM
    Logon type:   (none)
    Session:      0
    Sid:          S-1-5-18
    Logon time:   09-Dec-16 15:22:31
    Logon server:
    DNS Domain:
    UPN:

[1] Logon session 00000000:0000cdce:
    User name:
    Auth package: NTLM
    Logon type:   (none)
    Session:      0
    Sid:          (none)
    Logon time:   09-Dec-16 15:22:31
    Logon server:
    DNS Domain:
    UPN:

[2] Logon session 00000000:000003e4:
    User name:    WORKGROUP\ZODIAC$
    Auth package: Negotiate
    Logon type:   Service
    Session:      0
    Sid:          S-1-5-20
    Logon time:   09-Dec-16 15:22:31
    Logon server:
    DNS Domain:
    UPN:

[3] Logon session 00000000:00016239:
    User name:    Window Manager\DWM-1
    Auth package: Negotiate
    Logon type:   Interactive
    Session:      1
    Sid:          S-1-5-90-0-1
    Logon time:   09-Dec-16 15:22:32
    Logon server:
    DNS Domain:
    UPN:

[4] Logon session 00000000:00016265:
    User name:    Window Manager\DWM-1
    Auth package: Negotiate
    Logon type:   Interactive
    Session:      1
```

```
    Sid:          S-1-5-90-0-1
    Logon time:   09-Dec-16 15:22:32
    Logon server:
    DNS Domain:
    UPN:

[5] Logon session 00000000:000003e5:
    User name:    NT AUTHORITY\LOCAL SERVICE
    Auth package: Negotiate
    Logon type:   Service
    Session:      0
    Sid:          S-1-5-19
    Logon time:   09-Dec-16 15:22:32
    Logon server:
    DNS Domain:
    UPN:

...
[8] Logon session 00000000:0005c203:
    User name:    NT VIRTUAL MACHINE\AC9081B6-1E96-4BC8-8B3B-C609D4F85F7D
    Auth package: Negotiate
    Logon type:   Service
    Session:      0
    Sid:          S-1-5-83-1-2895151542-1271406230-163986315-2103441620
    Logon time:   09-Dec-16 15:22:35
    Logon server:
    DNS Domain:
    UPN:

[9] Logon session 00000000:0005d524:
    User name:    NT VIRTUAL MACHINE\B37F4A3A-21EF-422D-8B37-AB6B0A016ED8
    Auth package: Negotiate
    Logon type:   Service
    Session:      0
    Sid:          S-1-5-83-1-3011463738-1110254063-1806382987-3631087882
    Logon time:   09-Dec-16 15:22:35
    Logon server:
    DNS Domain:
    UPN:

...
[12] Logon session 00000000:0429ab2c:
    User name:    IIS APPPOOL\DefaultAppPool
    Auth package: Negotiate
    Logon type:   Service
    Session:      0
    Sid:          S-1-5-82-3006700770-424185619-1745488364-794895919-4004696415
    Logon time:   09-Dec-16 22:33:03
    Logon server:
    DNS Domain:
    UPN:
```

Информация, представленная для сеанса, включает SID и имя пользователя, связанного с сеансом, а также пакет аутентификации сеанса и время входа в систему. Учтите, что пакет аутентификации Negotiate, показанный в сеансе

входа в систему 2 в предыдущем выводе, будет пытаться провести аутентификацию с использованием Kerberos или NTLM, в зависимости от того, какой из них окажется наиболее подходящим для запроса на аутентификацию.

LUID сеанса выводится в строке «Logon Session» каждого блока сеанса, и с помощью утилиты Handle.exe (также из пакета Sysinternals) вы можете найти маркеры, представляющие конкретный сеанс входа в систему. Например, чтобы найти маркеры для сеанса входа в систему 8 в только что показанном примере вывода, вы можете ввести следующую команду:

```
C:\WINDOWS\system32>handle -a 5c203

Nthandle v4.1 - Handle viewer
Copyright (C) 1997-2016 Mark Russinovich
Sysinternals - www.sysinternals.com

System             pid: 4      type: Directory     1274: \Sessions\0\
DosDevices\00000000-0005c203
lsass.exe          pid: 496    type: Token          D7C: NT VIRTUAL MACHINE\
AC9081B6-1E96-4BC8-8B3B-C609D4F85F7D:5c203
lsass.exe          pid: 496    type: Token         2350: NT VIRTUAL MACHINE\
AC9081B6-1E96-4BC8-8B3B-C609D4F85F7D:5c203
lsass.exe          pid: 496    type: Token         2390: NT VIRTUAL MACHINE\
AC9081B6-1E96-4BC8-8B3B-C609D4F85F7D:5c203
svchost.exe        pid: 900    type: Token          804: NT VIRTUAL MACHINE\
AC9081B6-1E96-4BC8-8B3B-C609D4F85F7D:5c203
svchost.exe        pid: 1468   type: Token         10EC: NT VIRTUAL MACHINE\
AC9081B6-1E96-4BC8-8B3B-C609D4F85F7D:5c203
vmms.exe           pid: 4380   type: Token          A34: NT VIRTUAL MACHINE\
AC9081B6-1E96-4BC8-8B3B-C609D4F85F7D:5c203
vmcompute.exe      pid: 6592   type: Token          200: NT VIRTUAL MACHINE\
AC9081B6-1E96-4BC8-8B3B-C609D4F85F7D:5c203
vmwp.exe           pid: 7136   type: WindowStation  168: \Windows\
WindowStations\Service-0x0-5c203$
vmwp.exe           pid: 7136   type: WindowStation  170: \Windows\
WindowStations\Service-0x0-5c203$
```

Затем Winlogon ищет в реестре значение параметра HKLM\SOFTWARE\Microsoft\Windows NT\Current Version\Winlogon\Userinit и создает процесс для запуска в зависимости от содержимого хранящегося там строкового значения. (Это значение может включать несколько имен файлов с расширениями .EXE, отделенных друг от друга запятыми.) По умолчанию используется процесс Userinit.exe, загружающий профиль пользователя, а затем создающий процесс для запуска того, что указано в значении параметра HKCU\SOFTWARE\Microsoft\Windows NT\Current Version\Winlogon\Shell, если этот параметр существует. Изначально он отсутствует. Если он отсутствует, Userinit.exe делает то же самое для параметра HKLM\SOFTWARE\Microsoft\Windows NT\Current Version\Winlogon\Shell, в значении которого изначально находится Explorer.exe. Затем происходит выход из Userinit (именно поэтому для Explorer.exe не показывается родительский процесс при его просмотре в Process Explorer).

Гарантированная аутентификация

Основная проблема, связанная с аутентификацией на основе пароля, заключается в том, что пароль может быть вскрыт или украден и использован третьими лицами для нанесения вреда. В Windows появился механизм, отслеживающий ту защищенность, с которой пользователь прошел аутентификацию пользователя. Это позволяет объектам быть защищенными от обращений, если пользователь не прошел аутентификацию безопасным образом. (Аутентификация с помощью смарткарт считается более защищенной формой аутентификации по сравнению с парольной аутентификацией.)

На системах, присоединенных к домену, администратор домена может указать отображение между идентификатором объекта — Object Identifier (OID), являющимся уникальной цифровой последовательностью, представляющей определенный тип объекта, сертификатом, использующимся для аутентификации пользователя (например, на смарткарте или на аппаратном маркере безопасности), и идентификатором безопасности (SID), помещаемым в пользовательский маркер доступа, при успешной аутентификации пользователя системой. ACE-элемент в DACL объекта может указать такой SID в качестве части пользовательского маркера с целью получения пользователем доступа к объекту. С технической точки зрения это называется *групповым требованием* (group claim). Иными словами, пользователь требует участия в конкретной группе, что позволяет получить конкретные права доступа к конкретным объектам, с требованием, основанным на механизме аутентификации. Это свойство изначально не включено, и оно должно быть настроено администратором домена в том домене, в котором используется аутентификация на основе сертификата.

Гарантированная аутентификация расширяет существующие в Windows средства безопасности таким образом, чтобы предоставить IT-администраторам и всем заинтересованным в IT-безопасности предприятия максимальную гибкость. Предприятия решают, какие OID-идентификаторы встраивать в сертификаты, используемые ими для аутентификации и для отображения конкретных OID на универсальные группы Active Directory (SID-идентификаторы). Принадлежность пользователя к группе может использоваться для идентификации того, был ли использован сертификат в ходе операции входа в систему. Различные сертификаты могут иметь различные политики изданий и, таким образом, различные уровни безопасности, которые могут использоваться для защиты особо важных объектов (например, файлов или чего-нибудь еще, что может иметь дескриптор безопасности).

Протоколы аутентификации — authentication protocols (AP) — получают OID-идентификаторы от сертификатов в ходе аутентификации, основанной на использовании сертификатов. Эти OID-идентификаторы должны отображаться на SID-идентификаторы, которые, в свою очередь, обрабатываются в ходе расширения участия в группе и помещаются в маркер доступа. Отображение OID на универсальную группу указывается в Active Directory.

В качестве примера: у организации может быть несколько политик выпуска сертификатов с именами Contractor (подрядчик), Full Time Employee (штатный работ-

ник) и Senior Management (представитель высшего руководства), что отображается на универсальные группы Contractor-Users, FTE-Users и SM-Users соответственно. У пользователя Abby есть смарткарта с сертификатом, выпущенным с использованием политики выпуска Senior Management, и когда она входит в систему, используя свою смарткарту, она получает дополнительное членство в группе (что представлено с помощью SID в ее маркере доступа), показывающее, что она входит в группу SM-Users. Права доступа могут быть установлены (с использованием ACL) на такие объекты, доступ к которым предоставляется только представителям групп FTE-Users или SM-Users (идентифицируемым по их SID внутри ACE). Если Abby входит в систему с помощью своей смарткарты, она может получать доступ к таким объектам, но если она входит в систему, используя только свое имя пользователя и пароль (не пользуясь смарткартой), она не может получить доступ к таким объектам, поскольку в ее маркере доступа у нее не будет членства ни в группе FTE-Users, ни в группе SM-Users. Пользователь Toby, вошедший в систему с помощью смарткарты, имеющей сертификат, выпущенный с использованием политики выпуска Contractor, не сможет получить доступ к объекту, имеющему ACE, который требует членства в группе FTE-Users или в группе SM-Users.

Биометрическая среда для аутентификации пользователей

Windows предоставляет стандартизированный механизм для поддержки определенных типов биометрических устройств, в частности сканеров отпечатка пальца, чтобы обеспечивать идентификацию пользователей: Windows Biometric Framework (WBF). Подобно многим другим таким средам, Windows Biometric Framework была разработана для изолирования различных функций, используемых в поддержке подобных устройств таким образом, чтобы минимизировать код, необходимый для реализации нового устройства.

Основные компоненты среды Windows Biometric Framework показаны на рис. 7.21. Кроме особо оговоренных в следующем списке, все эти компоненты поставляются вместе с Windows.

- **Биометрическая служба Windows** — Windows Biometric Service (%SystemRoot%\System32\Wbiosrvc.dll). Предоставляет среду выполнения процесса, в которой может выполняться один или несколько поставщиков биометрических служб.
- **Интерфейс биометрического драйвера Windows** — Windows Biometric Driver Interface (WBDI). Это набор интерфейсных определений (основные функциональные коды IRP, коды `DeviceIoControl` и т. д.), которым должен соответствовать любой драйвер для устройства биометрического сканирования, чтобы быть совместимым с биометрической службой Windows. Драйверы WBDI могут разрабатываться с использованием любых стандартных фреймворков драйверов (UMDF, KMDF и WDM). Впрочем, рекомендуется использовать UMDF

для сокращения объема кода и повышения надежности. Описание WBDI дано в документации по Windows Driver Kit.

- **Биометрический API Windows** — Windows Biometric API. Позволяет существующим компонентам Windows, например WinLogon и LoginUI, получать доступ к биометрической службе. Приложения сторонних разработчиков имеют доступ к биометрическим API-функциям и могут использовать биометрический сканер для функций, отличающихся от входа в систему Windows. Примером функции в этом API может послужить `WinBioEnumServiceProviders`. Биометрический API-интерфейс предоставляется библиотекой %SystemRoot%\System32\Winbio.dll.

- **Поставщик биометрической службы отпечатков пальцев** — Fingerprint Biometric Service Provider. Служит оболочкой для функций биометрических адаптеров того или иного типа с целью присутствия общего интерфейса, независимого от типа биометрии, с биометрической службой Windows Biometric Service. Возможно, в будущем с помощью дополнительных поставщиков биометрической службы будут поддерживаться дополнительные типы биометрии, например сканирование сетчатки глаза или анализ индивидуальных особенностей голоса. Поставщик биометрической службы, в свою очередь, использует три адаптера, реализованных в форме DLL-библиотек пользовательского режима.

  - **Адаптер сенсора**, предоставляющий функциональные возможности сканера, связанные с получением данных. Адаптер сенсора будет, как правило, использовать для доступа к оборудованию сканера Windows-вызовы ввода/вывода. Windows предоставляет адаптер сенсора, который может использоваться с простыми сенсорами, для которых имеется драйвер Windows Biometric Device Interface (WBDI). Для более сложных сенсоров адаптеры пишутся их поставщиками.

  - **Адаптер механизма**, предоставляющий функциональность по обработке и сравнению, характерной для формата исходных данных сканера и других свойств. Фактическая обработка и сравнение могут проводиться внутри DLL-библиотеки адаптера механизма, или DLL может обмениваться данными с каким-нибудь другим модулем. Адаптер механизма всегда предоставляется поставщиком сенсора.

  - **Адаптер хранилища** предоставляет набор функций безопасного хранения. Они используются для хранения и извлечения шаблонов, которые используются адаптером механизма для сравнения с данными сканера биометрических данных. Windows предоставляет адаптер хранилища, использующий криптографические службы Windows, и стандартное дисковое хранилище файлов. Поставщик сенсора может предоставлять другой адаптер хранилища.

- **Функциональный драйвер для реального устройства биометрического сканирования.** Предоставляет самый последний WBDI и обычно для доступа к устройству сканирования использует службы низкоуровневого драйвера

шины, например драйвера шины USB. Этот драйвер всегда предоставляется поставщиком сенсора.

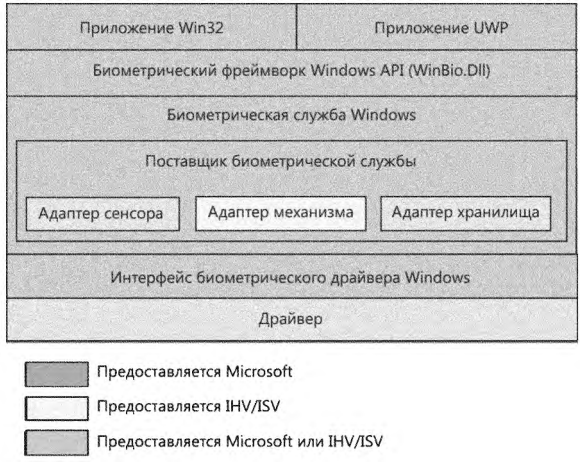


Рис. 7.21. Компоненты и архитектура среды Windows Biometric Framework

Типовая последовательность операций поддержки входа в систему посредством сканирования отпечатка пальца может быть следующей.

1. После инициализации адаптер сенсора получает от поставщика службы запрос на получение данных. Адаптер сенсора, в свою очередь, отправляет запрос `DeviceIoControl` с кодом управления `IOCTL_BIOMETRIC_CAPTURE_DATA` драйверу WBDI для устройства сканирования отпечатка пальца.

2. Драйвер WBDI переводит сканер в режим захвата и ставит в очередь запрос `IOCTL_BIOMETRIC_CAPTURE_DATA` до тех пор, пока не произойдет сканирование отпечатка.

3. Потенциальный пользователь проводит пальцем по сканеру. Драйвер WBDI получает об этом уведомление, получает необработанные данные сканирования от сенсора и возвращает эти данные драйверу сенсора в буфере, связанном с запросом `IOCTL_BIOMETRIC_CAPTURE_DATA`.

4. Адаптер сенсора предоставляет данные поставщику биометрической службы отпечатков пальцев — Fingerprint Biometric Service Provider, — который, в свою очередь, передает данные адаптеру механизма.

5. Адаптер механизма обрабатывает исходные данные, придавая им форму, совместимую с его хранилищем образцов.

6. Поставщик биометрической службы отпечатков пальцев использует адаптер хранилища для получения образцов и соответствующих идентификаторов безопасности от хранилища обеспечения безопасности. Он вызывает адаптер

механизма для сравнения каждого образца с обработанными данными сканирования. Адаптер механизма возвращает статус, показывающий, найдено соответствие или нет.

7. Если соответствие найдено, биометрическая служба уведомляет процесс WinLogon через DLL поставщика учетных данных об успешном входе и передает ему идентификатор безопасности распознанного пользователя. Это уведомление отправляется через сообщение ALPC с передачей пути, который не может быть фальсифицирован.

Windows Hello

Технология Windows Hello, появившаяся в Windows 10, предоставляет новые средства аутентификации пользователей на основании биометрической информации. С этой технологией пользователь может выполнить вход с минимальными усилиями — для этого достаточно заглянуть в камеру устройства или провести пальцем.

На момент написания книги технология Windows Hello поддерживает три типа биометрической идентификации:

- отпечаток пальца;
- лицо;
- сетчатка глаза.

Сначала нужно рассмотреть аспект биометрики, относящийся к безопасности. Какова вероятность того, что другой человек будет идентифицирован как вы? Какова вероятность того, что вы не будете идентифицированы как вы? Эти вопросы параметризуются по двум факторам:

- **Доля ложных позитивных срабатываний** (уникальность) — вероятность того, что другой пользователь обладает такими же биометрическими данными, как вы. Алгоритм Microsoft обеспечивает вероятность ложных позитивных срабатываний на уровне 1 к 100 000.
- **Доля ложных негативных срабатываний** (надежность) — вероятность того, что вы не будете правильно опознаны (например, из-за аномального освещения при распознавании лица или сетчатки глаза). С реализацией Microsoft вероятность такого события ниже 1 %. Если же это произойдет, пользователь может повторить попытку или ввести PIN-код.

PIN-код может показаться менее надежным, чем полноценный пароль (PIN-код может быть простым числом из четырех цифр). Однако на самом деле PIN-код надежнее пароля по двум основным причинам:

- PIN-код является локальным для устройства и никогда не передается по сети. Это означает, что даже если посторонний захватит PIN-код, он не сможет воспользоваться им для входа в качестве пользователя на другом устройстве.

С другой стороны, пароли передаются контроллеру домена. Если посторонний захватит пароль, он сможет войти в домен с другой машины.

- PIN-код хранится на TPM (Trusted Platform Module) — аппаратном блоке, который играет роль в Secure Boot (подробно рассматривается в главе 11 части 2), поэтому добраться до него достаточно сложно. В любом случае для этого необходим физический доступ к устройству, что заметно поднимает планку надежности для потенциальных нарушений безопасности.

Технология Windows Hello строится на базе WBF (Windows Biometric Framework — см. предыдущий раздел). Современные портативные компьютеры поддерживают биометрическую идентификацию по отпечатку пальца и распознаванию лица, тогда как распознавание сетчатки глаза поддерживается только телефонами Microsoft Lumia 950 и 950 XL. (Вероятно, ситуация изменится с выходом новых устройств.) Учтите, что для распознавания лица требуется как инфракрасная (IR), так и обычная (RGB) камера, и оно поддерживается на таких устройствах, как Microsoft Surface Pro 4 и Surface Book.

Управление учетными записями пользователей и виртуализация

Система UAC предназначена для того, чтобы позволить пользователям работать с правами обычного пользователя, в отличие от работы с правами администратора. Без административных прав пользователи не могут случайно (или преднамеренно) изменить настройки системы, вредоносные программы не смогут обычным образом изменить настройки безопасности системы или отключить антивирусное программное обеспечение, и пользователи не смогут получить несанкционированный доступ к конфиденциальной информации других пользователей на общих компьютерах. Работа с правами обычного пользователя может таким образом подавить воздействие вредоносных программ и защитить важные данные на общих компьютерах.

Чтобы работа под обычной учетной записью была практичной, при разработке UAC пришлось решить ряд проблем. Во-первых, поскольку модель использования Windows предполагала наделение административными правами, разработчики программного обеспечения предполагали, что их программы будут запускаться с этими правами и смогут тем самым получать доступ к любому файлу, к разделу реестра или к настройкам операционной системы и вносить в них изменения. Во-вторых, пользователям иногда нужны административные права для выполнения таких операций, как установка программного обеспечения, изменение системного времени и открытие портов в брандмауэре.

В UAC эти проблемы были решены таким образом, что большинство приложений запускалось с правами обычного пользователя, даже если пользователь вошел в систему под учетной записью с административными правами. Но в то же время

UAC позволяла обычным пользователям по мере надобности получать доступ к административным правам, когда они требовались устаревшим приложениям или когда нужно было изменить конкретные настройки системы. Как уже ранее упоминалось, UAC выполняет эти задачи с помощью создания фильтрованных маркеров администратора, а также с помощью обычных маркеров администратора, когда пользователь вошел в систему под административной учетной записью. Ко всем процессам, создаваемым в ходе сеанса пользователя, будут, как правило, применяться фильтрованные маркеры администратора, чтобы те приложения, которые могут выполняться с правами обычного пользователя, именно так и выполнялись. Но пользователи с правами администратора могут запустить программу или выполнить другие функции, требующие полных административных прав, путем повышения привилегий в системе UAC.

Windows также позволяет обычным пользователям выполнять определенные задачи, которые ранее считались предназначенными для администраторов, повышая тем самым удобства использования обычной среды пользователя. Например, существуют такие настройки групповой политики, которые могут позволить обычным пользователям устанавливать драйверы принтеров или других устройств, одобренные IT-администраторами, и устанавливать элементы управления ActiveX с одобренных администраторами сайтов.

И наконец, когда разработчики программного обеспечения тестируют его в среде UAC, это стимулирует их к созданию приложений, которые могут работать без административных прав. По сути, неадминистративным программам не требуется работа с привилегиями администратора, а программы, которым часто требуются привилегии администратора, являются, как правило, устаревшими, использующими старые API-функции или технологии, и такие программы должны быть обновлены.

Вместе взятые все эти изменения устраняют для пользователей необходимость постоянной работы с административными правами.

Файловая система и виртуализация реестра

Хотя некоторые устаревшие программы требуют административных прав, многим программам совершенно не нужно хранить пользовательские данные в глобальных для системы местах. При выполнении приложения оно может запускаться под разными учетными записями пользователя, поэтому данные конкретного пользователя должны храниться в каталоге %AppData% уровня пользователя, а настройки каждого пользователя — в находящемся в реестре профиле пользователя в разделе HKEY_CURRENT_USER\Software. Учетные записи обычных пользователей не имеют прав доступа по записи к каталогу %ProgramFiles% или к разделу HKEY_LOCAL_MACHINE\Software, но поскольку большинство систем Windows являются однопользовательскими и большинство пользователей до реализации UAC были администраторами, приложения, которые некорректно сохраняли пользовательские данные и настройки в этих местах, все равно работали.

Windows позволяет таким устаревшим приложениям работать под учетными записями обычных пользователей благодаря виртуализации файловой системы и пространства имен реестра. Когда приложение изменяет системное глобальное местоположение в файловой системе или в реестре и попытка выполнения операции завершается неудачей по причине запрета доступа, Windows перенаправляет операцию в область, предназначенную для конкретного пользователя. Когда приложение производит чтение из системного глобального места, Windows сначала проверяет наличие данных в области конкретного пользователя, если они не будут найдены, позволяет предпринять попытку чтения из глобального места.

Windows всегда будет разрешать такой тип виртуализации, за исключением следующих ситуаций:

- **Приложение является 64-разрядным.** Поскольку виртуализация является исключительно технологией обеспечения совместимости приложений, предназначенной для содействия работе устаревших программ, ее применение возможно только на 32-разрядных приложениях. Мир 64-разрядных приложений является относительно новым, и разработчики должны следовать рекомендациям по созданию приложений, ориентированных на стандартного пользователя.
- **Приложение уже работает с административными правами.** В таком случае необходимость в виртуализации отпадает.
- **Операция инициирована вызывающей процедурой режима ядра.**
- **Операция выполняется во время заимствования прав вызывающим процессом.** Например, любые операции, не инициированные процессом, классифицированным в соответствии с его определением как «устаревший», включая сетевые доступы к общим файлам, виртуализации не подвергаются.
- **Исполняемый образ для процесса имеет совместимый с UAC манифест** (определенный настройкой `requestedExecutionLevel`, рассматриваемой в следующем разделе).
- **Администратор не имеет прав доступа по записи к файлу или к разделу реестра.** Своим существованием это исключение обязано навязыванию принципа обратной совместимости, поскольку устаревшие приложения получили бы отказ в выполнении еще до реализации UAC, даже если приложение было запущено с правами администратора.
- **Виртуализация никогда не применяется в отношении служб.**

Статус виртуализации процесса (как уже ранее упоминалось, этот статус хранится в маркере в виде флага) можно увидеть путем добавления к странице процессов диспетчера задач столбца Виртуализация UAC (UAC Virtualization), как показано на рис. 7.22. Многие компоненты Windows, включая диспетчер окон рабочего стола — Desktop Window Manager (Dwm.exe), клиент-серверную подсистему времени выполнения — Client Server Run-Time Subsystem (Csrss.exe) и Проводник, работают с отключенной виртуализацией, поскольку у них имеется совместимый с UAC манифест или они запущены с административными правами и поэтому не

Рис. 7.22. Просмотр статуса виртуализации в диспетчере задач

допускают виртуализацию. У 32-разрядной версии Internet Explorer (iexplore.exe) виртуализация включена, поскольку он может владеть несколькими элементами управления ActiveX и сценариями и должен допускать, что они не создавались для корректной работы с правами обычных пользователей. Следует заметить, что при необходимости виртуализацию можно полностью отключить для всей системы с помощью настроек локальной политики безопасности.

Кроме виртуализации файловой системы и реестра, некоторым приложениям требуется дополнительная помощь для нормальной работы с правами обычных пользователей. Например, приложение, тестирующее учетную запись, под которой оно было запущено на членство в группе Администраторы, могло бы работать, но не будет, если учетная запись не входит в эту группу. В Windows определен ряд про-

кладок обеспечения совместимости приложений, чтобы все равно позволить этим приложениям работать. Прокладки чаще всего применяются к устаревшим приложениям для работы с правами обычных пользователей, как показано в табл. 7.15.

Таблица 7.15. Оболочки совместимости виртуализации UAC

Флаг	Описание
`ElevateCreateProcess`	Изменяет CreateProcess для обработки ошибок запрошенного повышения привилегий — `ERROR_ELEVATION_REQUIRED` путем вызова службы информации приложений (application information service) для вывода приглашения на запрос повышения привилегий
`ForceAdminAccess`	Имитирует запрос принадлежности к группе Администраторы
`VirtualizeDeleteFile`	Имитирует успешное удаление глобальных файлов и каталогов
`LocalMappedObject`	Заставляет объекты глобальных разделов отображаться на пользовательское пространство имен
`VirtualizeHKCRLite`	Перенаправляет глобальную регистрацию COM-объектов в место, определяемое для каждого пользователя
`VirtualizeRegisterTypeLib`	Превращает регистрацию typelib уровня машины в регистрацию уровня пользователя

Файловая виртуализация

К местам файловой системы, виртуализированным для устаревших процессов, относятся %ProgramFiles%, %ProgramData% и %SystemRoot%, за исключением некоторых конкретных подкаталогов. Но любой файл с расширением, указывающим на возможность его исполнения, включая .exe, .bat, .scr, .vbs и др., из виртуализации исключается. Это означает, что программы, обновляющие сами себя, будут под обычной учетной записью получать отказ в выполнении вместо создания закрытых версий своих исполняемых файлов, невидимых администратору, работающему в режиме глобального обновления.

ПРИМЕЧАНИЕ Чтобы добавить дополнительные расширения к списку исключений, введите их в параметр реестра HKLM\System\CurrentControlSet\ Services\Luafv\Parameters\ ExcludedExtensionsAdd и перезагрузите систему. Для отделения нескольких расширений друг от друга воспользуйтесь мультистроковым типом параметра и не включайте в расширение начальную точку.

Изменения виртуализированных каталогов, вносимые устаревшими процессами, перенаправляются в виртуальный корневой каталог пользователя %LocalAppData%\ VirtualStore. Компонент Local в этом пути подчеркивает тот факт, что виртуализированные файлы не перемещаются со всем остальным профилем, когда у учетной записи есть перемещаемый профиль (roaming profile).

Виртуализация файловой системы реализуется драйвером фильтра виртуализации UAC — UAC File Virtualization Filter Driver (%SystemRoot%\System32\Drivers\Luafv.sys). Поскольку это драйвер фильтра файловой системы, он видит все локальные операции файловой системы, но реализует свою функциональность только для операций с устаревших процессов. Как показано на рис. 7.23, драйвер фильтра изменяет путь целевого файла для устаревшего процесса, который создает файл в глобальном системном месте, но не изменяет его для невиртуализированных процессов с правами обычных пользователей. Исходные полномочия в отношении каталога \Windows запрещают доступ к приложениям, написанным с поддержкой UAC, но устаревшие процессы действуют таким образом, будто операция прошла успешно, когда на самом деле файл создается в том месте, которое полностью доступно пользователю.

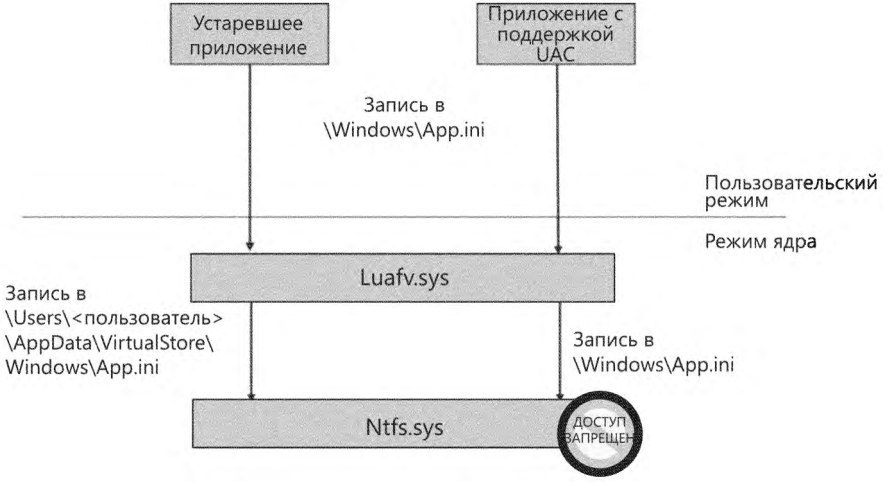


Рис. 7.23. Действие драйвера фильтра виртуализации файлов системы UAC

ЭКСПЕРИМЕНТ: ПОВЕДЕНИЕ, СВЯЗАННОЕ С ВИРТУАЛИЗАЦИЕЙ ФАЙЛОВ

В данном эксперименте мы будет включать и отключать виртуализацию окна командной строки и наблюдать за примерами поведения для демонстрации виртуализации файлов системой UAC.

1. Откройте окно командной строки без повышенных привилегий (для этой работы у вас должна быть включена система UAC) и включите для него виртуализацию. Чтобы изменить статус виртуализации процесса, выберите команду Виртуализация UAC (UAC Virtualization) в меню быстрого вызова, появляющегося при щелчке правой кнопкой мыши на имени процесса в диспетчере задач.

2. Перейдите в каталог C:\Windows и воспользуйтесь для записи файла следующей командой:

```
echo hello-1 > test.txt
```

3. Теперь выведите список содержимого каталога:

```
dir test.txt
```

Вы увидите, что файл появился.

4. Теперь отключите виртуализацию, щелкнув правой кнопкой на имени процесса на странице Процессы (Processes) в диспетчере задач, и отмените выбор Виртуализация UAC (UAC Virtualization), а затем снова выведите список содержимого каталога (см. п. 3). Обратите внимание на то, что файл исчез. Но вывод содержимого каталога VirtualStore покажет наличие файла:

```
dir  %LOCALAPPDATA%\VirtualStore\Windows\test.txt
```

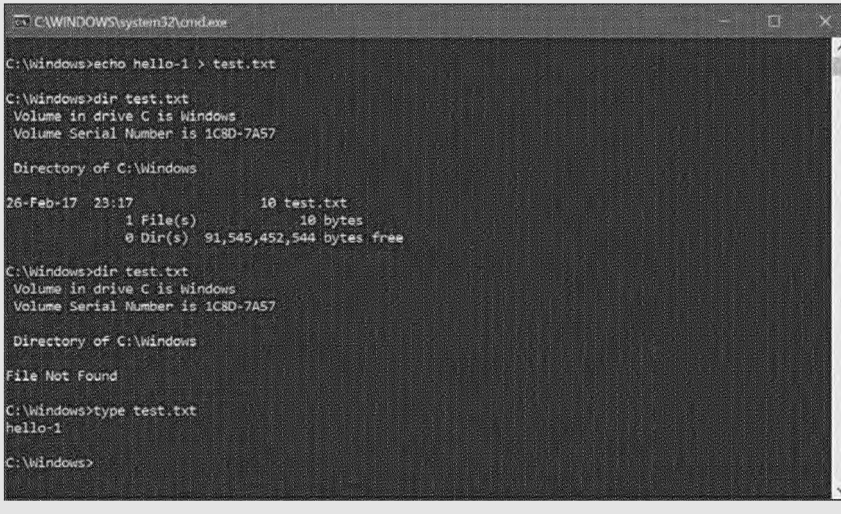
5. Опять включите виртуализацию для этого процесса.

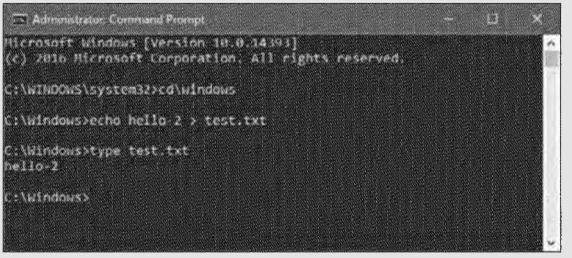
6. Чтобы рассмотреть более сложный сценарий, создайте новое окно командной строки, но теперь с повышенными привилегиями, а затем повторите действия пунктов 2 и 3, использовав строку «hello-2».

7. Изучите текст внутри этих файлов, использовав следующую команду в обоих окнах командной строки:

```
type test.txt
```

Ожидаемый вывод показан в следующих двух иллюстрациях.





8. И наконец, из своего окна командной строки с повышенными привилегиями удалите файл test.txt:

   ```
   del test.txt
   ```

9. Повторите действие из пункта 6 эксперимента. Обратите внимание на то, что в окне командной строки с повышенными привилегиями найти файл уже не удается, а вот в окне командной строки обычного пользователя опять показано старое содержимое файла. В этом эксперименте демонстрируется рассмотренный ранее механизм преодоления отказов — операции чтения сначала будут смотреть в виртуальное место хранения, создаваемое для каждого пользователя, но если файл отсутствует, будет предоставлен доступ по чтению к системному месту хранения.

Виртуализация реестра

Виртуализация реестра реализована несколько иначе, чем виртуализация файловой системы. Виртуализированные разделы реестра включают большую часть ветви HKEY_LOCAL_MACHINE\Software, но есть и ряд исключений:

- HKLM\Software\Microsoft\Windows;
- HKLM\Software\Microsoft\Windows NT;
- HKLM\Software\Classes.

Виртуализации подвергаются только те разделы, которые обычно изменяются устаревшими приложениями, но при этом не создают проблем совместимости или взаимодействия. Windows перенаправляет изменения виртуализированных разделов, вносимые устаревшими приложениями в создаваемый в реестре виртуальный корневой раздел пользователя HKEY_CURRENT_USER\Software\Classes\VirtualStore. Раздел находится в пользовательском кусте Classes, %LocalAppData%\Microsoft\Windows\UsrClass.dat, который, подобно любым другим виртуализированным файловым данным, не перемещается вместе с перемещаемым профилем пользователя. Вместо ведения постоянного списка виртуализированных мест, как это делается Windows для файловой системы, статус виртуализации раздела сохраняется в виде комбинации флагов из табл. 7.16. В отличие от файловой виртуализации, использующей драйвер фильтра, реестровая виртуализация реализована в диспетчере конфигурации (см. главу 9 части 2). Как и в виртуализации файловой системы, устаревший процесс, создающий подраздел виртуализированного раздела, перенаправляется на пользовательский виртуальный корневой раздел реестра, но UAC-совместимый процесс получить доступ с исходными полномочиями не сможет (рис. 7.25).

Таблица 7.16. Флаги виртуализации реестра

Флаг	Описание
`REG_KEY_DONT_VIRTUALIZE`	Указывает, разрешена ли виртуализация для этого раздела. Если флаг установлен, виртуализация отключена
`REG_KEY_DONT_SILENT_FAIL`	Если флаг `REG_KEY_DONT_VIRTUALIZE` установлен (виртуализация отключена), этот флаг указывает, что устаревшее приложение, которое не сможет получить доступ при выполнении операции над разделом, вместо этого взамен прав, запрошенных приложением, получает в отношении раздела права максимального разрешения `MAXIMUM_ALLOWED` (любого доступа, предоставленного учетной записи). Если этот флаг установлен, он также косвенно отключает виртуализацию
`REG_KEY_RECURSE_FLAG`	Определяет, будут ли флаги виртуализации распространяться на дочерние разделы (подразделы) этого раздела

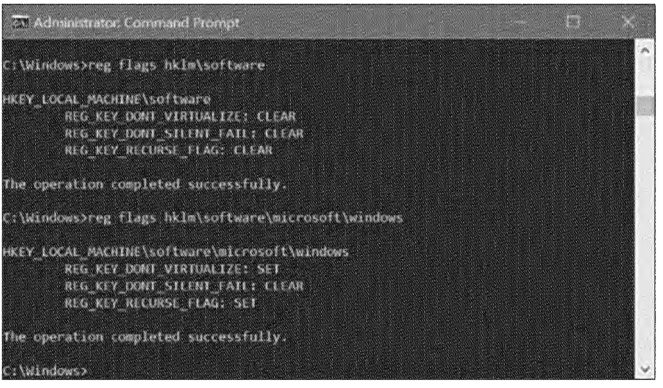


Рис. 7.24. Используемые UAC флаги виртуализации разделов Software и Windows

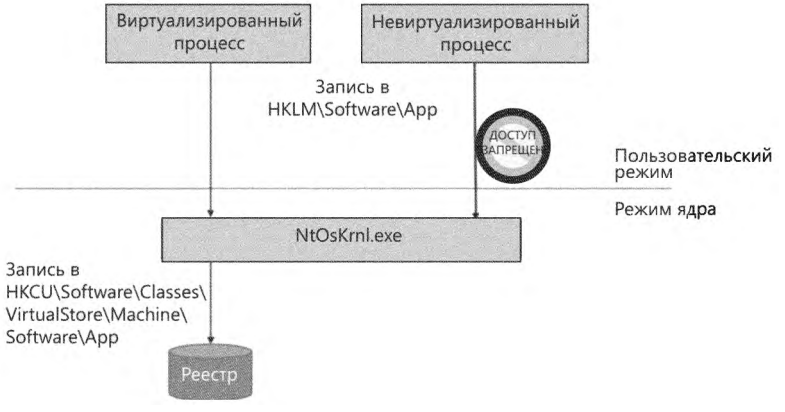


Рис. 7.25. Виртуализация реестра для UAC

Повышение привилегий

Даже если пользователи запускают только программы, совместимые с правами обычного пользователя, некоторые операции все же требуют административных прав. Например, подавляющее большинство установок программного обеспечения требуют административных прав для создания каталогов и разделов реестра в глобальных местах системы или установки служб или драйверов устройств. Изменение системных глобальных настроек Windows и приложений также требует административных прав, то же самое относится и к функции родительского контроля. Можно было бы выполнить большинство этих операций путем переключения на специальную административную учетную запись, но это настолько неудобно, что, скорее всего, большинство пользователей для выполнения своих повседневных задач, основная часть которых не требует административных прав, предпочтет остаться в административной учетной записи.

Важно знать, что повышение привилегий в системе UAC — удобство, а не граница безопасности. Граница безопасности требует, чтобы политика безопасности однозначно определяла, что именно может проходить через границу. Примером границы безопасности в Windows являются учетные записи пользователей, поскольку один пользователь не может получить доступ к данным, принадлежащим другому пользователю, не имея разрешения этого пользователя.

Поскольку повышение привилегий не является границей безопасности, нельзя дать гарантий, что вредоносная программа, выполняемая в системе с правами обычного пользователя, не может скомпрометировать процесс с повышенными привилегиями для получения административных прав. Например, диалоговые окна получения повышенных привилегий всего лишь идентифицируют исполняемый код, чьи привилегии будут повышены, и ничего не говорят о том, чем он будет заниматься при своем выполнении.

Выполнение с административными правами

В Windows включена усовершенствованная функциональная возможность «run as» (выполнение от имени), чтобы обычные пользователи могли, не испытывая неудобств, запускать процессы с административными правами. Эта функциональная возможность требует предоставления приложениям способа идентификации операций, для которых система, если это необходимо, может получить административные права от имени приложения. (К этой теме мы еще вскоре вернемся.)

Чтобы позволить пользователям временно выполнять обязанности системного администратора с правами обычного пользователя и чтобы при этом не приходилось вводить пользовательские имена и пароли при каждой необходимости получения административных прав, в Windows применяется механизм, который называется режимом, одобренным администратором, — *Admin Approval Mode* (AAM). Это свойство при входе в систему создает две идентичности: одну с правами обычного пользователя и другую с административными правами. Поскольку каждый пользо-

ватель в системе Windows либо является обычным пользователем, либо действует большей частью как обычный пользователь в AAM, разработчики должны предполагать, что все пользователи Windows являются обычными пользователями, что приведет к увеличению количества программ, работающих с правами обычного пользователя без виртуализации или оболочек совместимости.

Предоставление административных прав процессу называется *повышением привилегий*. Когда повышение привилегий выполняется под учетной записью обычного пользователя (или пользователя, входящего в административную группу, но не в группу Администраторы), это называется запросом на повышение прав — OTS (over-the-shoulder), потому что оно требует ввода учетных данных для учетной записи, входящей в группу Администраторы, что напоминает заполнение данных пользователем, набирающим текст через плечо обычного пользователя. Повышение привилегий, выполняемое AAM-пользователем, называется *согласованным повышением*, потому что пользователь просто должен одобрить назначение его административных прав.

Автономные системы, которые обычно представлены домашними компьютерами и объединенными в домен системами, рассматривают AAM-доступ со стороны удаленных пользователей по-другому, потому что подключенные к домену компьютеры в своих полномочиях на ресурсы могут использовать доменные административные группы. Когда пользователь обращается к общему файлу автономного компьютера, Windows запрашивает принадлежащую удаленному пользователю идентичность обычного пользователя, но на системах, объединенных в домен, Windows принимает во внимание все пользовательские участия в доменной группе, запрашивая административную идентичность пользователя. Исполнение образа, запрашивающего административные права, вызывает службу сведений о приложении — application information service (AIS, содержится в файле %SystemRoot%\System32\Appinfo.dll), которая запускает внутри службы хост-процесс (%SystemRoot%\System32\Svchost.exe) для запуска Consent.exe (%SystemRoot%\System32\Consent.exe). Consent захватывает растровое изображение экрана, применяет к нему эффект затемнения, переключается на рабочий стол, доступный только учетной записи локальной системы (защищенный рабочий стол), вырисовывает растровое изображение в качестве фона и отображает диалоговое окно повышения привилегий, содержащее сведения об исполняемом файле. Вывод этого диалогового окна в отдельном рабочем столе не дает любому приложению, запущенному под учетной записью пользователя, изменять внешний вид диалогового окна.

Если образ является компонентом Windows, имеющим цифровую подпись Microsoft, и этот образ находится в системном каталоге Windows, в верхней части диалогового окна отображается синяя полоска, показанная в верхнем фрагменте рис. 7.26, с синим и золотистым щитом, находящимся слева на этой полосе (различия между образами, подписанными компанией Microsoft, и другими источниками в Windows 10 были устранены). Если образ не подписан, то и поле щита и полоса становятся оранжевыми, и на щите появляется восклицательный знак, и в сообщении подчеркивается, что издатель образа неизвестен (справа на рис. 7.26).

В диалоговом окне повышения привилегий показывается значок образа, описание и издатель для образов, имеющих цифровую подпись, но для образов без подписи показывается только имя файла и надпись «Издатель: Неизвестно» («Unknown publisher»). Такие различия затрудняют маскировку вредоносных программ под законное программное обеспечение. Кнопка Показать подробности (Show details) в нижней части диалогового окна при щелчке на ней расширяет окно, чтобы показать командную строку, которая будет передана исполняющей системе при запуске образа на выполнение.

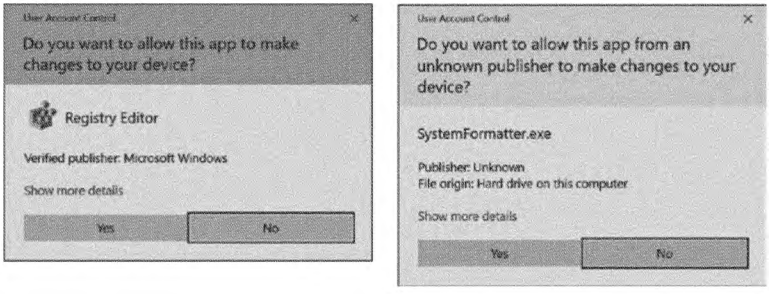


Рис. 7.26. Диалоговые окна повышения привилегий ААС UAC, выводимые в зависимости от состояния цифровой подписи образа

Диалоговое окно OTS-согласия, показанное на рис. 7.27, имеет похожий вид, но содержит приглашение на ввод учетных данных администратора. В нем будет выводиться список любых учетных записей с административными правами.

Рис. 7.27. Диалоговое окно OTS-согласия

Если пользователь отклоняет повышение привилегий, Windows возвращает процессу, инициировавшему запуск, ошибку отказа в доступе. Когда пользователь соглашается с повышением привилегий либо путем ввода учетных данных администратора, либо путем щелчка на кнопке Да (Yes), AIS вызывает функцию `CreateProcessAsUser` для запуска процесса с соответствующей административной

идентичностью. Хотя AIS с технической точки зрения является родителем процесса с повышенными привилегиями, AIS использует новую поддержку в API-функции `CreateProcessAsUser`, которая устанавливает идентификатор для родительского процесса на тот процесс, который изначально запускал эту функцию. Именно поэтому в таких утилитах, как Process Explorer, показывающих древовидную структуру процессов, процесс с повышенными привилегиями не показывается в качестве дочернего для хост-процесса службы AIS. На рис. 7.28 показаны операции, задействованные в запуске процесса с повышенными привилегиями под учетной записью обычного пользователя.

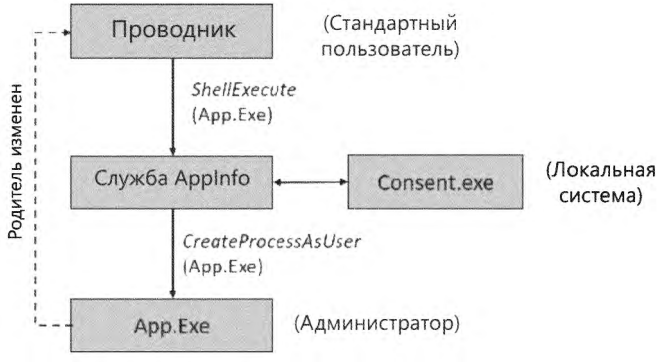


Рис. 7.28. Запуск административного приложения от имени обычного пользователя

Запрос административных прав

Существует несколько способов идентификации потребности системы и приложений в административных правах. Один из них, который проявляется в пользовательском интерфейсе Проводника, является командой контекстного меню Запуск от имени администратора (Run As Administrator) и ее вариантом использования в ярлыке. Эти элементы также имеют в своем составе значок сине-золотистого щита, который должен размещаться рядом с любой кнопкой или пунктом меню, приводящим к повышению привилегий. Выбор команды Запуск от имени администратора (Run As Administrator) заставляет Проводника вызвать API-функцию `ShellExecute` с командой «runas».

Основная масса программ установки требует административных прав, то же самое относится и к загрузчику образов, который инициирует запуск исполняющей системы, включает код обнаружения установщика для определения вероятности наличия устаревших установщиков. Некоторые из используемых им эвристических правил выполняют сравнительно несложные задачи обнаружения внутренней информации о версии или обнаружения в имени файла образа таких слов, как *setup* (настройка), *install* (установка) или *update* (обновление). Более сложные средства обнаружения включают сканирование байтовых последовательностей в исполняемой части кода, которые являются общими последовательностями

для оболочек установки, разработанных сторонними производителями. Загрузчик образа также вызывает библиотеку совместимости приложений, чтобы посмотреть, не требует ли целевой исполняемый файл прав администратора. Библиотека просматривает базы данных совместимости приложений, чтобы установить, есть ли у исполняемого образа связанный с ним флаг совместимости `RequireAdministrator` или `RunAsInvoker`.

Наиболее распространенным способом востребования исполняемым образом прав администратора является включение тега `requestedExecutionLevel` в файл манифеста приложения. Принадлежащий элементу атрибут уровня может иметь одно из трех значений, показанных в табл. 7.17.

Таблица 7.17. Запрашиваемые уровни повышения привилегий

Уровень повышения	Значение	Использование
As Invoker (с правами процесса-родителя)	Права администратора не нужны; просьба на повышение никогда не высказывается	Обычные пользовательские приложения, которым не требуются административные привилегии, например Блокнот
Highest available (наивысший)	Доступно согласие на допустимый запрос — Available Request для наивысших прав. Если пользователь вошел в систему как обычный пользователь, процесс будет запущен с правами процесса-родителя; в противном случае появится AAM-приглашение на повышение привилегий, и процесс будет запущен с полными административными правами	Приложения, которые могут работать без полных административных прав, но предполагают, что пользователям нужен полный доступ, если он дается без особого труда. Например, этот уровень используют Редактор реестра, Консоль управления Microsoft Management Console и просмотрщик событий Event Viewer
Require Administrator (с требованием прав администратора)	Всегда запрашивает административные права — для обычных пользователей будет показано диалоговое окно приглашения OTS, в противном случае будет показано диалоговое окно AAM	Приложения, требующие для работы административных прав, например Редактор настроек брандмауэра, который влияет на безопасность всей системы

Присутствие в манифесте элемента `trustInfo` (который можно увидеть в извлечении из рассматриваемого далее дампа eventvwr.exe) свидетельствует о том, что исполняемый образ был написан с поддержкой UAC и в него вложен элемент `requestedExecutionLevel`. Атрибут `uiAccess` находится там, где доступные приложения могут воспользоваться обходом упомянутой ранее изоляции привилегий пользовательского интерфейса (UIPI):

```
C:\>sigcheck -m c:\Windows\System32\eventvwr.exe
...
<trustInfo xmlns="urn:schemas-microsoft-com:asm.v3">
```

```
    <security>
        <requestedPrivileges>
            <requestedExecutionLevel
                level="highestAvailable"
                uiAccess="false"
            />
        </requestedPrivileges>
    </security>
</trustInfo>
<asmv3:application>
   <asmv3:windowsSettings xmlns="http://schemas.microsoft.com/SMI/2005/
WindowsSettings">
        <autoElevate>true</autoElevate>
   </asmv3:windowsSettings>
</asmv3:application>
...
```

Автоповышение привилегий

В исходной конфигурации (о том, как ее изменить, рассказано в следующем разделе) многие исполняемые файлы Windows и апплеты панели управления не выдают приглашение на повышение привилегий для пользователей-администраторов, даже если им для работы нужны административные права. Так происходит благодаря механизму под названием *автоповышение привилегий*, который предназначен для исключения пользователей-администраторов из числа тех, кому показываются приглашения на повышение привилегий для большинства случаев их работы; программы будут автоматически запускаться под полным административным маркером пользователя.

У автоповышения есть несколько требований. Интересующий исполняемый файл должен рассматриваться в качестве исполняемого файла Windows. Это означает, что он должен быть подписан издателем Windows (не просто Microsoft; как ни странно, они не эквивалентны — подпись Windows считается более привилегированной, чем подпись Microsoft) и он должен находиться в одном из нескольких считающихся безопасными каталогов: %SystemRoot%\System32 и в большинстве его подкаталогов, %Systemroot%\Ehome и в небольшом количестве каталогов, находящихся в каталоге %ProgramFiles%, например в тех, в которых содержится Защитник Windows и журнал Windows (Windows Journal).

Есть еще и дополнительные требования, которые зависят от типа исполняемого файла. Файлы с расширением .exe (за исключением Mmc.exe) получают автоповышение привилегий, если таковое ими запрашивается через элемент `autoElevate` в своих манифестах. Это проиллюстрировано в предыдущем разделе в строковом дампе файла eventVwr.exe.

Файл Mmc.exe — особый случай, поскольку решение вопроса об автоповышении для него привилегий зависит от того, какую оснастку управления системой он должен загрузить. Обычно Mmc.exe вызывается с командной строкой, указанной в файле с расширением .msc, которая, в свою очередь, указывает на загружаемую

оснастку. Когда консоль Mmc.exe запускается из защищенной учетной записи администратора (работающей с ограниченным маркером администратора), она запрашивает у Windows административные права. Windows проверяет тот факт, что Mmc.exe является исполняемым файлом Windows, а затем проверяет файл с расширением .msc. Этот файл также должен пройти тесты для исполняемых файлов Windows executable, и, более того, он должен быть внесен во внутренний список файлов с расширением .msc, для которых применяется автоповышение привилегий. В этот список включены почти все файлы .msc в Windows.

И наконец, административные права могут запрашиваться COM-объектами в разделах реестра. Для этого требуется подраздел `Elevation` с REG_DWORD-параметром под названием `Enabled`, имеющим значение 1. Как COM-объект, так и создающий его экземпляр исполняемый файл должны отвечать требованиям, предъявляемым Windows к исполняемым файлам, хотя исполняемый файл не нуждался в запросе автоповышения привилегий.

Управление поведением UAC

Изменить поведение UAC можно с помощью диалогового окна, показанного на рис. 7.29. Это диалоговое окно вызывается по ссылке Изменение параметров контроля учетных записей (Change User Account Control Settings). Элемент управления в исходной позиции показан на рис. 7.29.

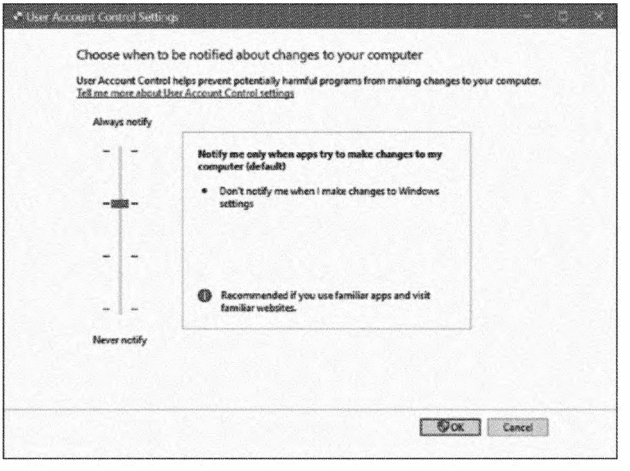


Рис. 7.29. Настройки управления учетными записями пользователей

Описание четырех возможных значений приведено в табл. 7.18.

Использовать третью позицию не рекомендуется, потому что UAC-приглашение на повышение привилегий появляется не на защищенном рабочем столе, а на обычном рабочем столе пользователя. Это может позволить вредоносным программам,

запущенным в том же сеансе, изменить появление приглашения. Эта настройка предназначена для использования только на тех системах, где видеоподсистема затрачивает много времени на затенение рабочего стола или по другим причинам не подходит для обычного вывода UAC.

Таблица 7.18. Параметры управления учетными записями

Позиция ползунка	Когда пользователь-администратор не работает с административными правами...		Примечания
	попытки изменить настройки Windows, например, используя конкретные апплеты Панели управления, приводят к тому, что	**...попытки установки программного обеспечения или запуска программы, чей манифест требует повышения привилегий, или с использованием команды Запустить от имени Администратора (Run As Administrator) приводят к тому, что**	
Верхняя позиция (Всегда уведомлять)	UAC-приглашение на повышение привилегий появляется на защищенном рабочем столе	UAC-приглашение на повышение привилегий появляется на защищенном рабочем столе	Такое поведение использовалось в Windows Vista
Вторая позиция	Повышение привилегий UAC происходит автоматически без приглашения или уведомления	UAC-приглашение на повышение привилегий появляется на защищенном рабочем столе	Исходная установка Windows
Третья позиция	Повышение привилегий UAC происходит автоматически без приглашения или уведомления	UAC-приглашение на повышение привилегий появляется на обычном рабочем столе пользователя	Не рекомендуется
Нижняя позиция (Никогда не уведомлять)	UAC для пользователей-администраторов выключен	UAC для пользователей-администраторов выключен	Не рекомендуется

Использовать самую низкую позицию крайне нежелательно, поскольку при ее установке система UAC с точки зрения административных учетных записей полностью отключается. Все процессы, запускаемые пользователем под учетной записью администратора, будут выполняться с полными правами пользователя-администратора, без фильтрованных административных маркеров. Для этих учетных записей также отключена виртуализация реестра и файловой системы и отключен защищенный режим Internet Explorer. Но виртуализация по-прежнему действует для неадминистративных учетных записей, и для таких учетных записей будут по-прежнему выводиться OTS-приглашения на повышение привилегий в случае, если будут предприняты попытки изменения настроек Windows, запуска программы, требующей повышения привилегий или использования в Проводнике пункта контекстного меню Запуск от имени Администратора (Run As Administrator).

Как показано в табл. 7.19, настройки UAC хранятся в разделе HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Policies\System в четырех параметрах реестра. Параметр ConsentPromptBehaviorAdmin управляет UAC-приглашением на повышение привилегий для администраторов, работающих с фильтрованным административным маркером, а параметр `ConsentPromptBehaviorUser` управляет UAC-приглашением для пользователей, не являющихся администраторами.

Таблица 7.19. Параметры реестра, связанные с управлением учетными записями пользователя

Позиция ползунка	ConsentPrompt BehaviorAdmin	ConsentPrompt BehaviorUser	EnableIUA	PromptonSecure Desktop
Верхняя позиция (Всегда уведомлять)	2 (показывать ААС UAC-приглашение на повышение привилегий)	3 (показывать OTS UAC-приглашение на повышение привилегий)	1 (включено)	1 (включено)
Вторая позиция	5 (показывать ААС UAC-приглашение на повышение привилегий, за исключением попыток изменения настроек Windows)	3	1	1
Третья позиция	5	3	1	0 (отключено; UAC-приглашение появляется на обычном рабочем столе пользователя)
Нижняя позиция (Никогда не уведомлять)	0	3	0 (отключено. Вход в систему под административными учетными записями не приводит к созданию ограниченных административных маркеров)	0

Снижение риска атак

В этой главе были рассмотрены некоторые технологии, которые помогают защитить пользователя, гарантировать свойства подписи исполняемого кода и заблокировать доступ к ресурсам посредством формирования изолированных сред. Однако в конечном итоге в любой защищенной системе существует точка отказа,

в любом коде присутствуют ошибки, а атакующие применяют все более изощренные атаки. Модель безопасности, в которой весь код считается свободным от ошибок или разработчик считает, что все ошибки кода со временем будут обнаружены и исправлены, обречена на провал. Кроме того, многие средства безопасности, предоставляющие «гарантии» выполнения кода, делают это за счет снижения производительности или совместимости, которое может оказаться неприемлемым в таких ситуациях.

Другой, куда более эффективный подход — выявление самых распространенных приемов, применяемых при атаках, а также создание внутренней «красной команды» (внутренней команды, пытающейся проводить атаки на собственный программный продукт) для обнаружения новых методов атаки до того, как они будут обнаружены атакующими, и реализация защитных мер против них. (Эти защитные меры могут быть как очень простыми, вроде перемещения данных, так и очень сложными — например, применение методов CFI [Control Flow Integrity].) В такой сложной кодовой базе, как Windows, могут содержаться тысячи уязвимостей, но количество методов их использования ограничено. Идея заключается в том, чтобы очень сильно затруднить (а иногда и сделать невозможным) использование больших классов уязвимостей без выявления всех ошибок.

Защитные меры уровня процессов

Хотя отдельные приложения могут реализовать защитные меры самостоятельно (например, в Microsoft Edge используется технология *MemGC*, предотвращающая многие классы атак с нарушением целостности памяти), в этом разделе будут рассмотрены защитные меры, которые предоставляются операционной системой всем приложениям или используются самой системой для сокращения возможностей использования уязвимостей. В табл. 7.20 перечислены все защитные меры в новейшей версии Windows 10 Creators Update, классы ошибок, от которых они защищают, и механизмы их активизации.

Таблица 7.20. Средства защиты от атак уровня процессов

Название	Пример использования	Механизм включения
Восходящая рандомизация ASLR	К вызовам `VirtualAlloc` применяется ASLR с 8-разрядной энтропией, включая рандомизацию базы стека	Включается флагом `PROCESS_CREATION_MITIGATION_POLICY_BOTTOM_UP_ASLR_ALWAYS_ON` в атрибутах создания процесса
Принудительное перемещение образов ASLR	Активизирует ASLR даже для двоичных образов, не имеющих флага компоновки `/DYNAMICBASE`	Включается вызовом `SetProcessMitigationPolicy` или флагом `PROCESS_CREATION_MITIGATION_POLICY_FORCE_RELOCATE_IMAGES_ALWAYS_ON` в атрибутах создания процесса

Таблица 7.20 (*продолжение*)

Название	Пример использования	Механизм включения
ASLR с высокой энтропией (HEASLR)	Значительно увеличивает энтропию ASLR для 64-разрядных образов, в результате чего вариативность восходящей рандомизации повышается до 1 Тбайт (т. е. восходящие операции выделения памяти могут начинаться с любого адреса в диапазоне от 64 Кбайт до 1 Тбайт адресного пространства, а размер энтропии составляет 24 бита)	Включается флагом `/HIGHENTROPYVA` на стадии компоновки или флагом `PROCESS_CREATION_MITIGATION_POLICY_HIGH_ENTROPY_ASLR_ALWAYS_ON` в атрибутах создания процесса
Блокировка неперемещаемых образов ASLR	Блокирует загрузку любой неперемещаемой библиотеки (флаг компоновки `/FIXED`) в сочетании с принудительным перемещением образов ASLR	Включается вызовом `SetProcessMitigationPolicy` или флагом `PROCESS_CREATION_MITIGATION_POLICY_FORCE_RELOCATE_IMAGES_ALWAYS_ON_REQ_RELOCS` в атрибутах создания процесса
DEP: постоянное включение	Запрещает процессу отключать DEP по своей инициативе; актуально только для x86 и только для 32-разрядных приложений (и/или с WoW64)	Включается вызовом `SetProcessMitigationPolicy`, атрибутом создания процесса или вызовом `SetProcessDEPPolicy`
DEP: запрет эмуляции преобразователей ATL	Запрещает устаревшему коду библиотеки ATL выполнять преобразователи (thunks) ATL даже при наличии известных проблем совместимости; актуально только для x86 и только для 32-разрядных приложений (и/или с WoW64)	Включается вызовом `SetProcessMitigationPolicy`, атрибутом создания процесса или вызовом `SetProcessDEPPolicy`
SEH: защита от перезаписи (SEHOP)	Не позволяет заменить структурные обработчики исключений другими, даже если образ не был скомпонован с флагом `/SAFESEH`; актуально только для x86 и только для 32-разрядных приложений (и/или с WoW64)	Включается вызовом `SetProcessDEPPolicy` или флагом `PROCESS_CREATION_MITIGATION_POLICY_SEHOP_ENABLE` в атрибутах создания процесса
Инициирование исключения по некорректному дескриптору	Помогает выявлять атаки повторного использования дескрипторов (использование после закрытия дескриптора). При таких атаках процесс использует дескриптор, отличный от ожидаемого (например, `SetEvent` для мьютекса); процесс завершается	Включается вызовом `SetProcessMitigationPolicy` или флагом `PROCESS_CREATION_MITIGATION_POLICY_STRICT_HANDLE_CHECKS_ALWAYS_ON` в атрибутах создания процесса

Название	Пример использования	Механизм включения
	аварийно вместо того, чтобы вернуть ошибку, которую процесс может проигнорировать	
Инициирование исключения при некорректном закрытии дескриптора	Помогает выявлять атаки повторного использования дескрипторов, при которых процесс пытается закрыть уже закрытый дескриптор	Возможность не документирована; может включаться только через недокументированный API
Запрет системных вызовов Win32k	Запрещает доступ к драйверу подсистемы режима ядра Win32, реализующему Window Manager (GUI), интерфейс графического устройства (GDI) и DirectX. Любые системные вызовы к этому компоненту становятся невозможными	Включается вызовом `SetProcessMitigationPolicy` или флагом `PROCESS_CREATION_MITIGATION_POLICY_WIN32K_SYSTEM_CALL_DISABLE_ALWAYS_ON` в атрибутах создания процесса
Фильтрация системных вызовов Win32k	Фильтрует доступ к драйверу подсистемы режима ядра Win32 и разрешает только простые операции GUI и DirectX. Таким образом подавляются многие возможные атаки без полной блокировки доступности сервисов GUI/GDI	Включается внутренним флагом в атрибутах создания процесса, который может определять один из трех возможных наборов фильтров Win32k. Но поскольку наборы фильтров жестко фиксированы, эта защитная мера зарезервирована для внутреннего использования Microsoft
Блокировка точек расширения	Запрещает процессу загрузку редакторов метода ввода (IME), подключаемых DLL-библиотек Windows (`SetWindowsHookEx`), DLL-библиотеки инициализации приложения (параметр `AppInitDlls` в реестре) или многоуровневого поставщика услуг (LSP) Winsock	Включается вызовом `SetProcessMitigationPolicy` или флагом `PROCESS_CREATION_MITIGATION_POLICY_EXTENSION_POINT_DISABLE_ALWAYS_ON` в атрибутах создания процесса
Защита от выполнения произвольного кода (CFG)	Не позволяет процессу выделить память для исполняемого кода или изменить разрешения существующего исполняемого кода, чтобы разрешить запись в него. Может быть настроено таким образом, чтобы конкретный поток внутри процесса мог запросить эту возможность или мог позволить удаленному процессу отключить эту возможность снижения риска, которая не поддерживается с точки зрения безопасности	Включается вызовом `SetProcessMitigationPolicy` или флагами `PROCESS_CREATION_MITIGATION_POLICY_PROHIBIT_DYNAMIC_CODE_ALWAYS_ON` и `PROCESS_CREATION_MITIGATION_POLICY_PROHIBIT_DYNAMIC_CODE_ALWAYS_ON_ALLOW_OPT_OUT` в атрибутах создания процесса

Таблица 7.20 (*продолжение*)

Название	Пример использования	Механизм включения
Control Flow Guard (CFG)	Предотвращает использование уязвимостей порчи памяти для перехвата управления посредством проверки цели любой косвенной инструкции CALL или JMP по списку допустимых ожидаемых целевых функций. Некоторые механизмы CFI (Control Flow Integrity) описаны в следующем разделе	Образ должен компилироваться и компоноваться с ключом `/guard:cf`. Включается флагом `PROCESS_CREATION_MITIGATION_POLICY_CONTROL_FLOW_GUARD_ALWAYS_ON` в атрибутах создания процесса в том случае, если эта возможность не поддерживается образом, но защитные меры CFG все равно желательны для других образов, загружаемых в процессе
Подавление экспорта CFG	Укрепляет защиту CFG за счет блокировки косвенных вызовов таблицы API образа	Образ должен быть откомпилирован с ключом `/guard:exportsuppress`; также настройка может осуществляться вызовом SetProcessMitigationPolicy или флагом `PROCESS_CREATION_MITIGATION_POLICY_CONTROL_FLOW_GUARD_EXPORT_SUPPRESSION` в атрибутах создания процесса
Жесткий режим CFG	Блокирует загрузку в текущем процессе образов библиотек, которые не были скомпонованы с ключом `/guard:cf`	Включается вызовом `SetProcessMitigationPolicy` или флагом `PROCESS_CREATION_MITIGATION_POLICY2_STRICT_CONTROL_FLOW_GUARD_ALWAYS_ON` в атрибутах создания процесса
Отключение несистемных шрифтов	Блокирует загрузку файлов шрифтов, которые не были зарегистрированы Winlogon на момент входа, после установки в каталог C:\windows\fonts	Включается вызовом `SetProcessMitigationPolicy` или флагом `PROCESS_CREATION_MITIGATION_POLICY_FONT_DISABLE_ALWAYS_ON` в атрибутах создания процесса
Двоичные образы только с подписью Microsoft	Блокирует загрузку в текущем процессе образов библиотек, которые не были подписаны сертификатом, выданным Microsoft CA	Включается вызовом `PROCESS_CREATION_MITIGATION_POLICY_BLOCK_NON_MICROSOFT_BINARIES_ALWAYS_ON` в атрибутах создания процесса
Двоичные образы только с подписью магазина	Блокирует загрузку в текущем процессе образов библиотек, которые не были подписаны сертификатом, выданным Microsoft Store CA	Включается флагом `PROCESS_CREATION_MITIGATION_POLICY_BLOCK_NON_MICROSOFT_BINARIES_ALLOW_STORE` в атрибутах создания процесса при запуске
Запрет удаленных образов	Блокирует загрузку в текущем процессе образов библиотек, размещенных на нелокальных путях (UNC или WebDAV)	Включается вызовом `SetProcessMitigationPolicy` или флагом `PROCESS_CREATION_MITIGATION_POLICY_IMAGE_LOAD_NO_REMOTE_ALWAYS_ON` в атрибутах создания процесса

Название	Пример использования	Механизм включения
Запрет образов с низким IL	Блокирует загрузку в текущем процессе образов библиотек с обязательной меткой ниже средней (0x2000)	Включается вызовом `SetProcessMitigationPolicy` или флагом `PROCESS_CREATION_MITIGATION_POLICY_IMAGE_LOAD_NO_LOW_LABEL_ALWAYS_ON` в атрибутах создания процесса. Также может включаться ресурсной заявкой ACE с именем `IMAGELOAD` для файла загружаемого процесса
Предпочтительное использование образов из каталога System32	Изменяет путь поиска загрузчика так, чтобы поиск загружаемого образа (по относительному имени) всегда осуществлялся в каталоге %SystemRoot%\System32 независимо от текущего пути поиска	Включается вызовом `SetProcessMitigationPolicy` или флагом `PROCESS_CREATION_MITIGATION_POLICY_IMAGE_LOAD_PREFER_SYSTEM32_ALWAYS_ON` в атрибутах создания процесса
RFG (Return Flow Guard)	Способствует предотвращению дополнительных классов уязвимостей повреждения памяти, влияющих на передачу управления в программе. Для этого перед выполнением инструкции RET проверяется, что функция не была вызвана через эксплойт ROP (Return-Oriented Programming). Эта защитная мера является частью механизмов CFI (Control Flow Integrity)	В настоящее время разработка надежной и высокоэффективной реализации продолжается, так что эта защитная мера еще недоступна, но она включена в список для полноты
Ограничение модификации контекста потока	Ограничивает модификацию контекста текущего потока	В настоящее время недоступно до завершения реализации RFG, которая сделает эту защитную меру более надежной, но она может появиться в будущей версии Windows. Здесь она включена для полноты списка
Соответствие целостности при загрузке	Не позволяет процессу динамически загружать любые DLL-библиотеки, уровень целостности которых отличается от уровня целостности процесса, в тех случаях, когда защитные меры политики подписи не могут быть активизированы при запуске из-за проблем совместимости. Данная мера предназначена для предотвращения атак внедрения DLL	Включается вызовом `SetProcessMitigationPolicy` или флагом `PROCESS_CREATION_MITIGATION_POLICY2_LOADER_INTEGRITY_CONTINUITY_ALWAYS_ON` в атрибутах создания процесса

Таблица 7.20 (*окончание*)

Название	Пример использования	Механизм включения
Завершение процесса при нарушении целостности кучи	Блокирует механизм FTH (Fault Tolerant Heap) и инициирование исключения с возможностью продолжения работы в случае повреждения кучи; вместо этого процесс будет завершаться. Таким образом предотвращается возможность повреждения кучи для выполнения обработчика исключения, находящегося под контролем атакующего, или ситуации с игнорированием исключений кучи программой, или те случаи, в которых эксплойт не всегда вызывает повреждение кучи (что ограничивает универсальность и надежность решения)	Включается вызовом HeapSetInformation или флагом PROCESS_CREATION_MITIGATION_POLICY_HEAP_TERMINATE_ALWAYS_ON в атрибутах создания процесса
Запрет создания дочерних процессов	Блокирует создание дочерних процессов, для чего маркер помечается специальным ограничением, запрещающим любому другому компоненту создавать процесс при заимствовании прав маркера этого процесса (например, при создании процессов WMI или компонентов ядра)	Включается флагом PROCESS_CREATION_CHILD_PROCESS_RESTRICTED в атрибутах создания процесса. Может переопределяться для того, чтобы предоставить разрешение пакетным (UWP) приложениям, флагом PROCESS_CREATION_DESKTOP_APPX_OVERRIDE
Политика пакетов всех приложений	Приложение, выполняемое в AppContainer, не может обращаться к ресурсам с присутствием SID ALL APPLICATION PACKAGES SID (См. раздел «Контейнеры AppContainer» этой главы). Вместо него необходим SID ALL RESTRICTED APPLICATION PACKAGES. Иногда такие контейнеры называются LPAC (Less Privileged App Container)	Включается флагом PROC_THREAD_ATTRIBUTE_ALL_APPLICATION_PACKAGES_POLICY в атрибутах создания процесса

Обратите внимание: некоторые из этих защитных мер могут включаться на уровне приложения или системы без содействия со стороны разработчика. Для этого откройте редактор локальной групповой политики, раскройте узел **Конфигурация компьютера** (Computer Configuration), а затем узлы **Административные шаблоны** (Administrative Templates), **Система** (System) и, наконец, **Mitigation Options** (рис. 7.30). В диалоговом окне **Process Mitigation Options** введите значение, соответствующее включаемым защитным мерам: `1` для включения, `0` для отключения или `?` для сохранения значения по умолчанию или значения, запрошенного про-

цессом (также см. рис. 7.30). Номера битов берутся из перечисления `PROCESS_MITIGATION_POLICY`, определенного в заголовочном файле Winnt.h. В результате соответствующий параметр реестра записывается в раздел IFEO (Image File Execution Options) для указанного имени образа. К сожалению, текущая версия Windows 10 Creators Update и более ранние версии отсекают многие новые защитные меры. Чтобы этого не происходило, вы можете вручную установить параметр реестра `MitigationOptions` типа `REG_DWORD`.

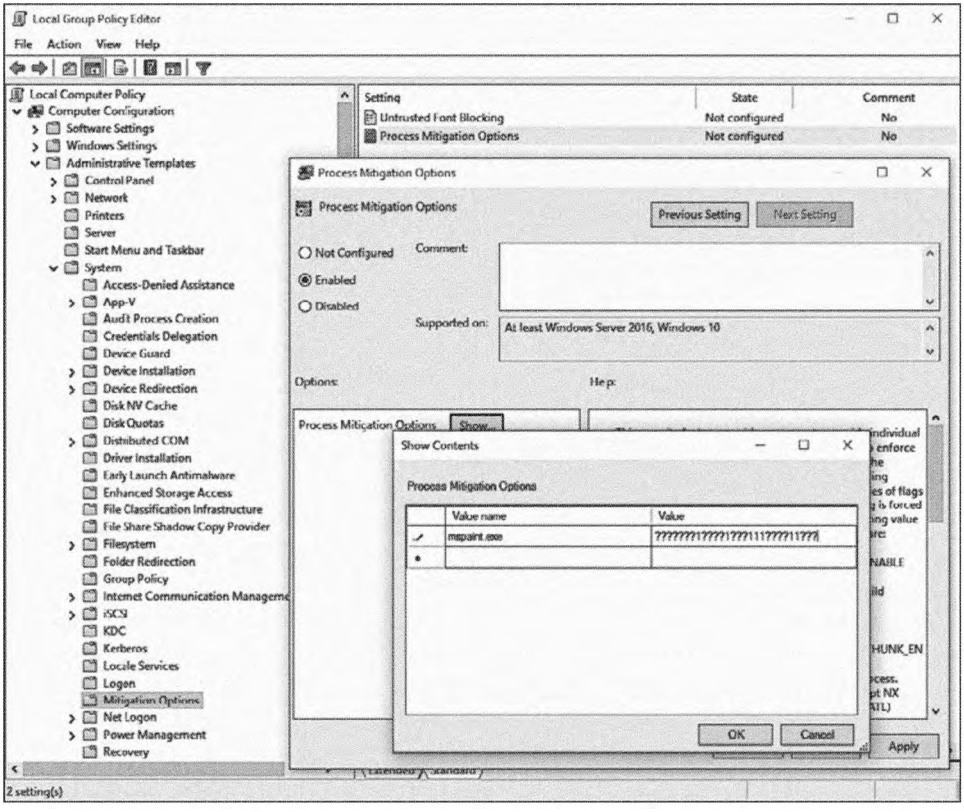


Рис. 7.30. Настройка конфигурации защитных мер на уровне процессов

CFI

Механизмы DEP (Data Execution Prevention) и ACG (Arbitrary Code Guard) усложняют размещение эксплойтами исполняемого кода в куче или в стеке с целью создания нового исполняемого кода или для модификации существующего исполняемого кода. В результате атаки против данных в памяти становятся более интересными. Такие атаки позволяют изменять отдельные части памяти для перенаправления потока управления (например, посредством модификации адресов

возврата в памяти или косвенных указателей на функции, хранящихся в памяти). Такие приемы, как ROP (Return-Oriented-Programming) и JOP (Jump-Oriented-Programming), часто используются для нарушения обычного пути передачи управления в программе и передачи его специальным фрагментам кода.

Поскольку такие фрагменты часто встречаются в середине или в конце различных функций, управление должно передаваться в середину или в конец нормальной функции. Применяя технологии CFI (Control Flow Integrity) — которые, например, могут проверить, что косвенная инструкция JMP или CALL передает управление в начало реальной функции, или что инструкция RET указывает на ожидаемый адрес, или что инструкция RET выполняется после входа в функцию с ее начального адреса, — операционная система и компилятор могут обнаруживать и предотвращать многие классы таких эксплойтов.

CFG

CFG (Control Flow Guard) — механизм сокращения риска эксплойтов, впервые появившийся в Windows 8.1 Update 3, — существует в расширенной версии в Windows 10 и Server 2016, а его дальнейшие усовершенствования выходили в различных обновлениях (включая последнее обновление Creators Update). Изначально реализованный только для кода пользовательского режима, механизм CFG теперь также существует в форме KFCG (Kernel CFG) в Creators Update. CFG ориентируется на часть CFI, связанную с косвенными переходами CALL/JMP, и проверяет, что целевой адрес косвенного вызова находится в начале известной функции (подробнее об этом будет рассказано позднее). Если целевой адрес не находится в начале известной функции, процесс просто завершается. На рис. 7.31 представлена концептуальная схема работы CFG.

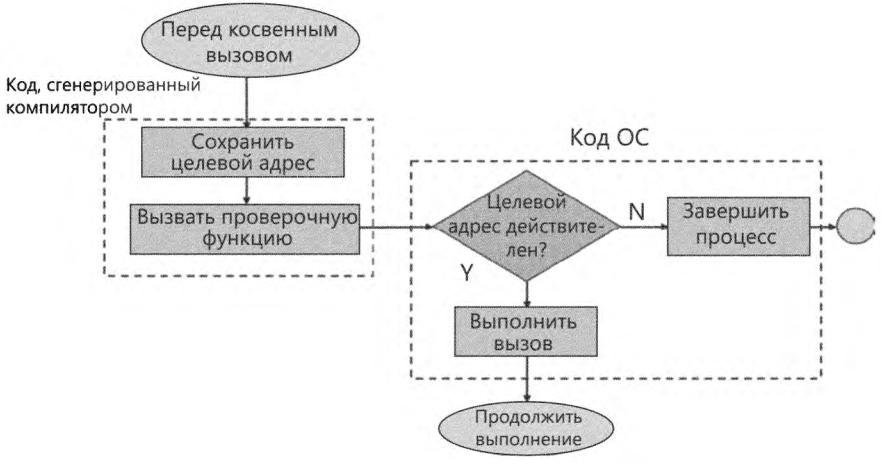


Рис. 7.31. Концептуальная схема CFG

CFG требует содействия со стороны поддерживаемого компилятора, который добавляет вызов кода проверки перед косвенной передачей управления в потоке команд. У компилятора Visual C++ имеется параметр `/guard:cf`, который должен устанавливаться для образов, которые должны строиться с поддержкой CFG (эта возможность также доступна в графическом интерфейсе Visual Studio в виде параметра `C/C++/Code Generation/Control Flow Guard` в свойствах проекта). Этот параметр также должен быть включен в настройках компилятора, так как для поддержки CFG требуется содействие обоих компонентов Visual Studio.

При наличии этих параметров образы (EXE и DLL), откомпилированные с поддержкой CFG, включают соответствующую информацию в заголовок PE. Кроме того, в секции PE-заголовка `gfids` у них хранится список функций, которые считаются действительными целями для косвенной передачи управления (по умолчанию объединяется компоновщиком с секцией `.rdata`). Этот список строится компоновщиком и содержит относительные виртуальные адреса (RVA) всех функций в образе. В этот список включаются функции, которые могут вызываться косвенными инструкциями кодом, присутствующим в образе, потому что невозможно заранее спрогнозировать, что внешний код каким-то законным образом узнает адрес функции и попробует вызвать ее. Это особенно справедливо для экспортируемых функций, которые могут вызываться после получения указателя вызовом `GetProcAddress`.

Тем не менее программисты могут использовать *блокировку CFG*, которая обеспечивается аннотацией `DECLSPEC_GUARD_SUPRESS`; функция в таблице действительных функций помечается специальным флагом, который означает, что программист не ожидает, что эта функция будет вызываться в результате косвенного вызова или перехода.

При наличии таблицы действительных целевых функций простой проверочной функции будет достаточно сравнить цель инструкции CALL или JMP с одной из функций в таблице. С алгоритмической точки зрения такой алгоритм будет иметь сложность $O(n)$, где количество функций, необходимых для проверки, будет в худшем случае эквивалентно количеству функций в таблице. Очевидно, линейное сканирование всего массива при каждой косвенной передаче управления приведет к неприемлемому замедлению программы, а следовательно, для эффективного выполнения проверок CFG необходима поддержка операционной системы. В следующем разделе вы увидите, как в Windows решается эта задача.

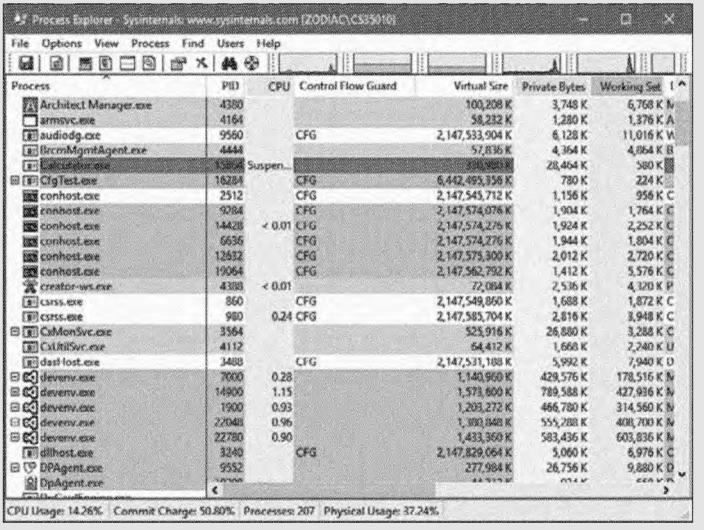
ЭКСПЕРИМЕНТ: ИНФОРМАЦИЯ CFG

Утилита DumpBin из Visual Studio может вывести базовую информацию CFG. В следующем примере выводятся данные конфигурации загрузчика и заголовка для Smss:

```
c:\> dumpbin /headers /loadconfig c:\windows\system32\smss.exe
Microsoft (R) COFF/PE Dumper Version 14.00.24215.1
Copyright (C) Microsoft Corporation.  All rights reserved.
Dump of file c:\windows\system32\smss.exe
   PE signature found
   File Type: EXECUTABLE IMAGE
   FILE HEADER VALUES
               8664 machine (x64)
                  6 number of sections
           57899A7D time date stamp Sat Jul 16 05:22:53 2016
                  0 file pointer to symbol table
                  0 number of symbols
                 F0 size of optional header
                 22 characteristics
                      Executable
                      Application can handle large (>2GB) addresses

   OPTIONAL HEADER VALUES
                20B magic # (PE32+)
              14.00 linker version
              12800 size of code
               EC00 size of initialized data
                  0 size of uninitialized data
               1080 entry point (0000000140001080) NtProcessStartupW
               1000 base of code
         140000000 image base (0000000140000000 to 0000000140024FFF)
               1000 section alignment
                200 file alignment
              10.00 operating system version
              10.00 image version
              10.00 subsystem version
                  0 Win32 version
              25000 size of image
                400 size of headers
              270FD checksum
                  1 subsystem (Native)
               4160 DLL characteristics
   High Entropy Virtual Addresses
                      Dynamic base
                      NX compatible
                      Control Flow Guard
   ...
   Section contains the following load config:

               000000D0 size
                      0 time date stamp
                   0.00 Version
                      0 GlobalFlags Clear
                      0 GlobalFlags Set
                      0 Critical Section Default Timeout
```

```
                  0 Decommit Free Block Threshold
                  0 Decommit Total Free Threshold
    0000000000000000 Lock Prefix Table
                  0 Maximum Allocation Size
                  0 Virtual Memory Threshold
                  0 Process Heap Flags
                  0 Process Affinity Mask
                  0 CSD Version
               0800 Dependent Load Flag
    0000000000000000 Edit List
    0000000140020660 Security Cookie
00000001400151C0 Guard CF address of check-function pointer
00000001400151C8 Guard CF address of dispatch-function pointer
00000001400151D0 Guard CF function table
             2A Guard CF function count
        00010500 Guard Flags
                      CF Instrumented
                      FID table present
                      Long jump target table present
               0000 Code Integrity Flags
               0000 Code Integrity Catalog
           00000000 Code Integrity Catalog Offset
           00000000 Code Integrity Reserved
0000000000000000 Guard CF address taken IAT entry table
              0 Guard CF address taken IAT entry count
0000000000000000 Guard CF long jump target table
              0 Guard CF long jump target count
    0000000000000000 Dynamic value relocation table

    Guard CF Function Table

          Address
          --------
          0000000140001010  _TlgEnableCallback
          0000000140001070  SmpSessionComplete
          0000000140001080  NtProcessStartupW
          0000000140001B30  SmscpLoadSubSystemsForMuSession
          0000000140001D10  SmscpExecuteInitialCommand
          0000000140002FB0  SmpExecPgm
          0000000140003620  SmpStartCsr
          00000001400039F0  SmpApiCallback
          0000000140004E90  SmpStopCsr
...
```

Информация, относящаяся к CFG, выделена **жирным шрифтом**. Вскоре мы рассмотрим ее более подробно. А пока откройте Process Explorer, щелкните правой кнопкой мыши на заголовке столбца Process и выберите команду Select Columns. Затем на вкладке Process Image установите флажок Control Flow Guard. Также выберите Virtual Size на вкладке Process Memory. Результат должен выглядеть примерно так:



Вы увидите, что большинство процессов, предоставляемых Microsoft (включая Smss, Csrss, Audiodg, Notepad и многие другие), было построено с поддержкой CFG. Виртуальный размер процессов, построенных с поддержкой CFG, на удивление высок. Напомним, что виртуальный размер обозначает общее адресное пространство, используемое в процессе, — как для выделенной, так и для зарезервированной памяти. С другой стороны, в столбце Private Bytes выводится приватная выделенная память, которая даже отдаленно не приближается к виртуальному размеру (хотя виртуальный размер также включает неприватную память). Для 64-разрядных процессов виртуальный размер составляет не менее 2 Тбайт; вскоре мы обоснуем этот факт.

Битовая карта CFG

Как было показано ранее, заставлять программу перебирать список функций через каждые несколько инструкций было бы нереально. Следовательно, вместо алгоритма с линейным временем *O(n)* требования к производительности требуют применения алгоритма *O(1)* — с постоянным временем поиска независимо от количества функций в таблице. Постоянное время поиска должно быть как можно ниже. Очевидным фаворитом в свете такого требования будет массив, индексируемый по адресу целевой функции; элемент массива указывает, действителен этот адрес или нет (например, простым значением `BOOL`). Однако с 128 Тбайт возможных адресов такой массив будет занимать 128 Тбайт `* sizeof(BOOL)`, что явно неприемлемо — это больше, чем само адресное пространство. Можно ли придумать что-то лучше?

Для начала можно воспользоваться тем фактом, что компилятор должен генерировать код функций x64 по границе 16 байт. Размер необходимого массива уменьшается до 8 Тбайт `* sizeof(BOOL)`. Однако использовать тип `BOOL` (4 байта в худшем случае, 1 байт в лучшем) было бы в высшей степени неэффективно. Значение представляет

одно состояние: действительное или недействительное, а для этого достаточно 1 бита. Получаем 8 Тбайт / 8, или просто 1 Тбайт. К сожалению, все не так просто. Нет гарантий того, что компилятор сгенерирует все функции на границе 16 байт. Написанный вручную код на языке ассемблера и некоторые оптимизации могут привести к нарушению этого правила. Одно из возможных решений — просто добавить еще один бит для обозначения того, что функция начинается *где-то* среди следующих 15 байт вместо 16-байтовой границы. Таким образом, возможны следующие варианты:

- **{0, 0}** Ни одна действительная функция не начинается внутри этой 16-байтовой границы.
- **{1, 0}** Действительная функция точно выровнена по 16-байтовой границе.
- **{1, 1}** Действительная функция начинается где-то внутри 16-байтового блока.

В такой конфигурации при попытке атакующего осуществить внутренний вызов функции, помеченной компоновщиком как выровненная по 16-байтовой границе, мы получим 2-битовое состояние {1, 0}, тогда как необходимые биты (т. е. биты 3 и 4) в адресе будут содержать {1, 1}, так как адрес не будет выравниваться по 16-бавтовой границе. Следовательно, атакующий сможет вызвать произвольную инструкцию в первых 16 байтах функции, если компоновщик не сгенерировал функцию как выровненную (биты содержат {1, 1}, как в приведенном примере). Впрочем, даже в этом случае инструкция должна принести какую-то пользу атакующему без сбоя функции (как правило, некий фрагмент, завершающийся инструкцией RET).

С учетом всего сказанного для вычисления размера битовой карты CFG можно воспользоваться следующими формулами:

- **32-разрядное приложение на платформе x86 или x64** 2 Гбайт / 16 * 2 = = 32 Мбайт.
- **32-разрядное приложение с ключом `/LARGEADDRESSAWARE`, загруженное в 3-гигабайтном режиме на платформе x86** 3 Гбайт / 16 * 2 = 48 Мбайт.
- **64-разрядное приложение** 128 Тбайт / 16 * 2 = 2 Тбайт.
- **32-разрядное приложение с ключом `/LARGEADDRESSAWARE` на платформе x64** 4 Гбайт / 16 * 2 = 64 Мбайт + размер 64-разрядной битовой карты для защиты 64-разрядных компонентов Ntdll.dll и WoW64, итого 2 Тбайт + 64 Мбайт.

Впрочем, выделение и заполнение 2 Тбайт для каждого процесса все равно обходится слишком дорого. Хотя мы имеем фиксированную сложность выполнения самого косвенного вызова, запуск процесса не должен занимать столько времени, а 2 Тбайт выделенной памяти мгновенно приведут к исчерпанию лимита выделения. По этой причине используются два приема для экономии памяти и повышения производительности.

Во-первых, диспетчер памяти только резервирует битовую карту, основываясь на предположении о том, что проверочная функция CFG будет рассматривать исключение при обращении к битовой карте CFG как признак того, что биты находятся в состоянии {0,0}. Таким образом, если блок содержит 4 Кбайт битов состояний,

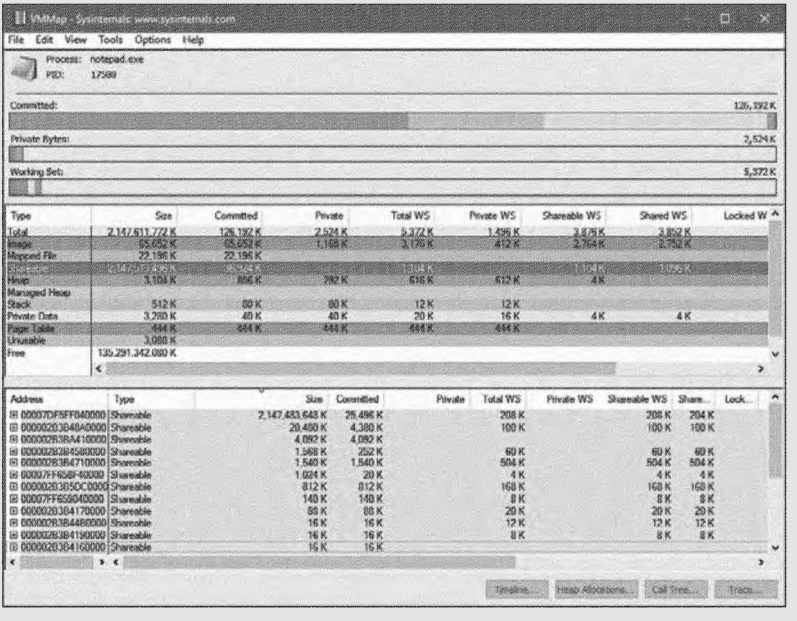
все из которых равны {0, 0}, его можно оставить зарезервированным, а выделять только те страницы, у которых установлен хотя бы один бит {1, X}.

Кроме того, как описано в разделе, посвященном ASLR, главы 5 «Управление памятью», система обычно выполняет рандомизацию/перемещение библиотек только один раз во время загрузки, чтобы избежать повторных перемещений. А значит, после того как библиотека с поддержкой ASLR будет загружена по некоторому адресу, она всегда будет загружаться по этому адресу. Также это означает, что после вычисления состояний битовой карты для функций в библиотеке они будут идентичны во всех остальных процессах, также загружающих тот же двоичный образ. Диспетчер памяти рассматривает карту CFG как область общей памяти на основе страничного файла, а физические страницы, соответствующие общим битам, существуют в памяти только в одном экземпляре.

Такое решение сокращает затраты на хранение выделенных страниц в памяти и означает, что вычисляться будут только биты, соответствующие приватной памяти. В обычных приложениях содержимое приватной памяти не может выполняться (кроме ситуации с копированием при записи и модификацией библиотеки — но это не произойдет при загрузке образа), так что затраты на загрузку приложения, совместно использующего те же библиотеки, что и ранее загруженные приложения, близки к нулю. Следующий эксперимент демонстрирует этот факт.

ЭКСПЕРИМЕНТ: БИТОВАЯ КАРТА CFG

Запустите утилиту VMMap и выберите процесс Notepad. В разделе Shareable должен отображаться большой зарезервированный блок:



Отсортируйте нижнюю панель по размеру — это поможет вам быстро найти большой блок, используемый для CFGBitmap. Кроме того, для просмотра битовой карты CFG можно воспользоваться WinDBG — присоединитесь к процессу и введите команду !address:

```
+     7df5'ff530000 7df6'0118a000 0'01c5a000   MEM_MAPPED MEM_RESERVE
Other  [CFG Bitmap]
      7df6'0118a000 7df6'011fb000 0'00071000   MEM_MAPPED MEM_COMMIT PAGE_NOACCESS
Other  [CFG Bitmap]
      7df6'011fb000 7ff5'df530000 1ff'de335000 MEM_MAPPED MEM_RESERVE
Other  [CFG Bitmap]
      7ff5'df530000 7ff5'df532000 0'00002000   MEM_MAPPED MEM_COMMIT PAGE_READONLY
Other  [CFG Bitmap]
```

Обратите внимание: большие блоки имеют пометку `MEM_RESERVE`, а промежуточные — пометку `MEM_COMMIT`, которая говорит о том, что установлен по крайней мере один действительный бит состояния `{1, X}`. Кроме того, все (или почти все) области снабжены пометкой `MEM_MAPPED`, потому что они принадлежат общей битовой карте.

Построение битовой карты CFG

После завершения инициализации системы вызывается функция `MiInitializeCfg`, инициализирующая поддержку CFG. Функция создает один или два объекта разделов (`MmCreateSection`) как зарезервированную память с размером, соответствующим платформе (см. выше). Для 32-разрядных платформ достаточно одной битовой карты. Для платформ x64 необходимы две битовые карты: для 64-разрядных процессов и для процессов Wow64 (32-разрядных приложений). Указатели на объекты разделов хранятся в подструктуре, на которую указывает глобальная переменная `MiState`.

После того как процесс будет создан, происходит безопасное отображение соответствующего раздела в адресное пространство процесса. «*Безопасное*» в данном случае означает то, что код, выполняемый в процессе, не может отменить отображение раздела или изменить его уровень защиты. (В противном случае вредоносный код мог бы просто отменить отображение, выделить память заново и заполнить ее битами 1, что приведет к фактическому отключению CFG, или же просто изменить любые биты, помечая область как доступную для чтения/записи.)

Битовая карта (или карты) CFG пользовательского режима заполняется в следующих ситуациях.

- Во время отображения в память у тех образов, которые были динамически перемещены из-за ASLR (за дополнительной информацией об ASLR обращайтесь к главе 5), извлекаются метаданные целевых адресов косвенных вызовов. Если у образа такие метаданные отсутствуют (это означает, что он не был откомпилирован с поддержкой CFG), предполагается, что по любому адресу в образе возможна косвенная передача управления. Как объяснялось ранее, поскольку

динамически перемещаемые образы должны загружаться по одному адресу во всех процессах, их метаданные используются для заполнения общего раздела, используемого для битовой карты CFG.

- В ходе отображения в память необходимо принять особые меры для образов, не перемещаемых динамически и не отображаемых по предпочтительному базовому адресу. Для таких отображений соответствующие страницы битовой карты CFG делаются приватными и заполняются метаданными CFG из образа. Для образов, у которых биты CFG присутствуют в общей битовой карте CFG, выполняется проверка того, что все актуальные страницы битовой карты CFG остаются общими. Если это не так, биты приватных страниц битовой карты CFG заполняются по метаданным CFG из образа.
- Когда виртуальная память выделяется как исполняемая (или происходит изменение атрибутов ее защиты), соответствующие страницы битовой карты CFG объявляются приватными и по умолчанию инициализируются единицами. Это необходимо в таких ситуациях, как JIT-компиляция, когда код генерируется «на ходу» и затем выполняется (например, в .NET или Java).

Укрепление защиты CFG

Хотя механизм CFG хорошо справляется с предотвращением типов эксплойтов, основанных на косвенных вызовах или переходах, его можно обойти следующими способами:

- Если процесс удастся «обмануть» или воспользоваться слабостью существующего JIT-ядра для выделения исполняемой памяти, все соответствующие биты будут заполнены {1, 1}; это означает, что все адреса памяти считаются действительными целями для перехода.
- Для 32-разрядных приложений, если ожидаемая цель перехода снабжена пометкой `__stdcall`, но атакующему удастся изменить ее на `__cdecl` (конвенция вызова C), это приведет к повреждению стека, так как функция с конвенцией вызова C не будет выполнять зачистку аргументов вызывающей стороны, в отличие от функций со стандартной конвенцией вызова. Так как CFG не может различать разные конвенции вызова, это приведет к повреждению стека — возможно, с адресом возврата под контролем атакующего и обходом защитного механизма CFG.
- Также сгенерированные компилятором цели перехода `setjmp/longjmp` по своему поведению отличаются от полноценных косвенных вызовов. CFG не умеет их различать.
- От некоторых косвенных вызовов защититься сложнее — таких, как IAT (Import Address Table) или Delay-Load Address Table, обычно расположенных в секции исполняемого образа, доступной только для чтения.
- Экспортированные функции не могут быть желательными целями для косвенного вызова.

В Windows 10 были введены усовершенствования CFG, направленные на решение этих проблем. Во-первых, у функции `VirtialAlloc` появился новый флаг `PAGE_TARGETS_INVALID`, а у функции `VirtualProtect` — флаг `PAGE_TARGETS_NO_UPDATE`. При установке этих флагов JIT-механизмы, выделяющие исполняемую память, не увидят установку всех выделяемых битов в состояние {1, 1}. Вместо этого они должны вручную вызвать функцию `SetProcessValidCallTargets` (которая вызывает внутреннюю функцию `NtSetInformationVirtualMemory`), чтобы задать фактические начальные адреса функций в JIT-коде. Кроме того, функция помечается флагом `DECLSPEC_GUARD_SUPPRESS`, чтобы атакующие не могли воспользоваться косвенной инструкцией CALL или JMP для передачи управления даже в начало функции. (Так как эта функция небезопасна по своей природе, вызов ее с управляемым стеком или регистрами может привести к обходу CFG.)

Кроме того, улучшенный механизм CFG изменяет поток передачи управления, описанный в начале раздела. В этом потоке вместо простой функции «проверить цель, вернуть управление» реализуется функция «проверить цель перехода, вызвать, проверить стек, вернуть управление», которая используется в ряде мест в 32-разрядных приложениях (и/или приложениях, работающих в WoW64). Усовершенствованный поток выполнения показан на рис. 7.32.

Также улучшенный механизм CFG добавляет в исполняемый образ дополнительные таблицы — такие, как таблица IAT и таблица адресов дальних переходов. Когда таблицы дальних переходов и защиты IAT CFG включены в компиляторе, они используются для хранения целевых адресов для этих конкретных типов косвенных вызовов, а соответствующие функции *не* помещаются в обычную таблицу функций, а следовательно, не отражаются в битовой карте. Это означает, что если код пытается осуществить косвенный переход/вызов к одной из этих функций, такой переход будет рассматриваться как некорректный. Вместо этого исполнительная среда C и компоновщик проверят целевые адреса, скажем, `longjmp`-функции посредством ручной проверки этой таблицы. И хотя такое решение уступает битовой карте по эффективности, таблица должна содержать совсем небольшое количество функций, вследствие чего эти затраты могут оказаться приемлемыми.

Наконец, в усовершенствованный механизм CFG реализована функция, называемая *подавлением экспорта*, которая должна поддерживаться компилятором и политикой защитных мер на уровне процессов. (О защитных мерах уровня процессов подробнее рассказано в разделе «Защитные меры уровня процессов» этой главы.) При включении этой возможности реализуется новое состояние битов (вспомните, что в списке состояние {0, 1} обозначено как неопределенное). Это состояние означает, что данная функция считается действительной, но к ней применено подавление экспорта, и она будет особым образом обработана загрузчиком.

Чтобы определить, какие возможности присутствуют в заданном двоичном образе, обратитесь к флагам защиты из каталога Image Load Configuration Directory, который может декодироваться приложением DumpBin, использованным ранее. Для удобства эти флаги перечислены в табл. 7.21.

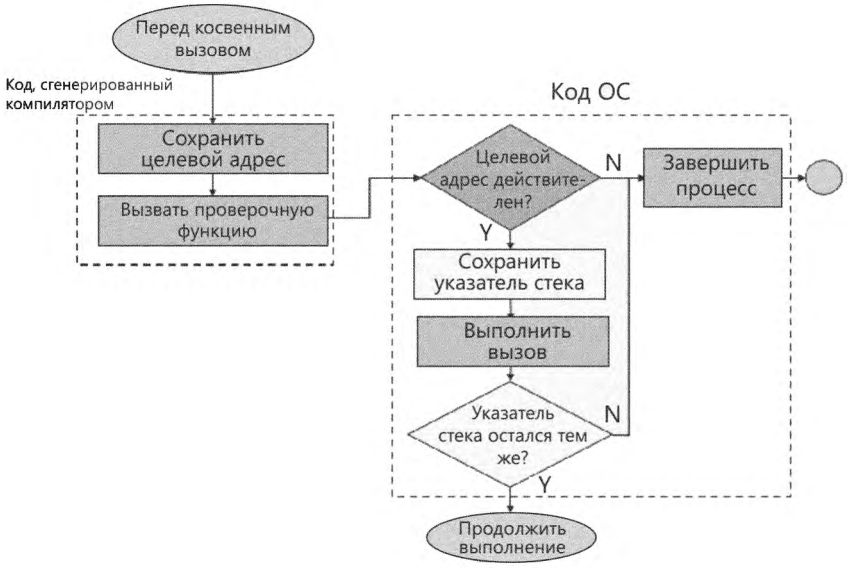


Рис. 7.32. Усовершенствованный механизм CFG

Таблица 7.21. Флаги CFG

Флаг	Значение	Описание
IMAGE_GUARD_CF_INSTRUMENTED	0x100	Означает, что для модуля присутствует поддержка CFG
IMAGE_GUARD_CFW_INSTRUMENTED	0x200	Модуль выполняет проверки CFG и целостности записи
IMAGE_GUARD_CF_FUNCTION_TABLE_PRESENT	0x400	Модуль содержит списки функций для CFG
IMAGE_GUARD_SEURITY_COOKIE_UNUSED	0x800	Модуль не использует значение cookie безопасности, внедренное флагом компилятора /GS
IMAGE_GUARD_PROTECT_DELAYLOAD_IAT	0x1000	Модуль поддерживает таблицы адресов импорта (IAT) с задержкой, доступные только для чтения
IMAGE_GUARD_DELAYLOAD_IAT_IN_ITS_OWN_SECTION	0x2000	IAT с задержкой находится в собственном разделе, поэтому при желании для нее можно переопределить уровень защиты

Флаг	Значение	Описание
IMAGE_GUARD_CF_EXPORT_SUPPRESSION_INFO_PRESENT	0x4000	Модуль содержит информацию подавления экспорта
IMAGE_GUARD_CF_ENABLE_EXPORT_SUPPRESSION	0x8000	Модуль включает подавление экспорта
IMAGE_GUARD_CF_LONGJUMP_TABLE_PRESENT	0x10000	Модуль содержит информацию о целях longjmp

Взаимодействие загрузчика с CFG

Хотя битовая карта CFG строится диспетчером памяти, загрузчик пользовательского режима (см. главу 3) решает две задачи. Во-первых, он динамически активизирует поддержку CFG только в том случае, если она включена (например, вызывающая сторона могла запретить CFG для дочернего процесса или сам процесс не имеет поддержки CFG). Для этого используется функция загрузчика `LdrpCfgProcessLoadConfig`, которая вызывается для инициализации CFG для каждого загруженного модуля. Если среди флагов `DllCharacteristics` в дополнительном заголовке PE не установлен флаг CFG (`IMAGE_DLLCHARACTERISTICS_GUARD_CF`), в поле `GuardFlags` структуры `IMAGE_LOAD_CONFIG_DIRECTORY` не установлен флаг `IMAGE_GUARD_CF_INSTRUMENTED` или ядро принудительно отключило CFG для этого модуля, то и делать ничего не нужно.

Во-вторых, если модуль использует CFG, `LdrpCfgProcessLoadConfig` получает от образа указатель на функцию проверки косвенных переходов (поле `GuardCFCheckFunctionPointer` структуры `IMAGE_LOAD_CONFIG_DIRECTORY`) и присваивает ему `LdrpValidateUserCallTarget` или `LdrpValidateUserCallTargetES` в `Ntdll` (в зависимости от того, включено ли подавление экспорта). Кроме того, функция сначала убеждается в том, что косвенный указатель не был модифицирован так, чтобы он указывал за пределы самого модуля.

Кроме того, если усовершенствованный механизм CFG использовался для компиляции этого двоичного образа, доступна вторая косвенная функция, называемая *функцией диспетчеризации CFG*. Она используется для реализации расширенной передачи управления, описанной ранее. Если образ включает такой указатель на функцию (в поле `GuardCFDispatchFunctionPointer` упомянутой структуры), он инициализируется `LdrpDispatchUserCallTarget` или `LdrpDispatchUserCallTargetES`, если включено подавление экспорта.

ПРИМЕЧАНИЕ В некоторых случаях само ядро может эмулировать или выполнять косвенные переходы или вызовы от имени пользовательского режима. В ситуациях, в которых это возможно, ядро реализует собственную функцию `MmValidateUserCallTarget`, которая выполняет ту же работу, что и `LdrpValidateUserCallTarget`.

Код, генерируемый компилятором при включенной поддержке CFG, выдает косвенный вызов, который передает управление функции `LdrpValidateCallTarget(ES)` или `LdrpDispatchUserCallTarget(ES)` в Ntdll. Эта функция использует целевой адрес перехода и проверяет биты состояния для функции:

- Если состояние равно {0, 0}, передача управления потенциально недействительна.
- Если состояние равно {1, 0}, а адрес выровнен по 16-байтовой границе, передача управление допустима. В противном случае она потенциально недействительна.
- Если состояние равно {1, 1}, а адрес не выровнен по 16-байтовой границе, передача управления допустима. В противном случае она потенциально недействительна.
- Если состояние равно {0, 1}, передача управление потенциально недействительна.

Если передача управления потенциально недействительна, выполняется функция `RtlpHandleInvalidUserCallTarget` для определения подходящего действия. Сначала она проверяет, разрешено ли подавление вызовов в процессе — необычный режим совместимости, который может быть установлен при включении Application Verifier или при помощи реестра. В таком случае проверяется, включено ли подавление для данного адреса — возможно, он не был вставлен в битовую карту именно по этой причине (вспомните, что на это указывает специальный флаг в записи защитной таблицы функций). В таком случае вызов считается допустимым. Если же функция недействительна, то передача управления отменяется, а процесс завершается.

Затем проверяется, разрешено ли подавление экспорта. Если оно разрешено, то целевой адрес проверяется по списку адресов подавления экспорта (на это также указывает другой флаг, включенный в запись защитной таблицы функций). В таком случае загрузчик проверяет, что целевой адрес является ссылкой на таблицу экспорта другой DLL-библиотеки — единственный разрешенный случай косвенного вызова к образу с подавленными экспортами. При этом используется сложная проверка, которая убеждается в том, что целевой адрес принадлежит другому образу, в его каталоге загрузки образа включено подавление экспорта, а адрес находится в каталоге импорта этого образа. Если все проверки проходят, вызывается функция `NtSetInformationVirtualMemory` для изменения состояния на {1, 0}. Если хотя бы одна из проверок завершается неудачей или подавление экспорта не разрешено, процесс завершается.

Для 32-разрядных приложений выполняется дополнительная проверка, если для процесса включена поддержка DEP (за информацией о DEP обращайтесь к главе 5). В противном случае некорректный вызов разрешается, так как нельзя исключить, что старое приложение обращается с вызовом к куче или стеку по законным причинам.

Наконец, поскольку большие блоки битов состояния {0, 0} не выделяются для экономии памяти, если проверка битовой карты CFG попадает на зарезервирован-

ную страницу, происходит исключение нарушения доступа. На платформе x86, где настройка обработки исключений обходится дорого, вместо обработки в составе проверочного кода исключение распространяется обычным образом. (За дополнительной информацией о диспетчеризации исключений обращайтесь к главе 8 части 2.) Обработчик пользовательского режима `KiUserExceptionDispatcher` содержит проверки для распознавания исключений нарушения доступа к битовой карте CFG и автоматически возобновляет выполнение, если код исключения был равен `STATUS_IN_PAGE_ERROR`. Это упрощает код `LdrpValidateUserCallTarget(ES)` и `LdrpDispatchUserCallTarget(ES)`, в который не нужно включать код обработки исключений. На платформе x64, где обработчики исключений просто регистрируются в таблицах, вместо этого выполняется обработчик `LdrpICallHandler` с такой же логикой.

CFG в режиме ядра

Хотя драйверы, откомпилированные в Visual Studio с параметром `/guard:cf`, включали те же двоичные свойства, что и образы пользовательского режима, первые версии Windows 10 никак не использовали эти данные. В отличие от битовой карты CFG пользовательского режима, защищаемой более надежной сущностью (ядром), нет ничего, что могло бы полноценно «защитить» битовую карту CFG ядра, если бы она создавалась. Вредоносный эксплойт мог бы просто отредактировать PTE-запись для страницы, содержащей изменяемые биты, пометить ее как доступную для чтения/записи и продолжить косвенный вызов или переход. Таким образом, затраты на настройку защитных мер, которые так тривиально обходятся, попросту не оправданны.

Сейчас все больше пользователей включают функциональность VBS, что позволяет использовать более высокую границу безопасности, предоставляемую VTL 1. На помощь приходят записи таблицы страниц SLAT: они предоставляют вторую границу защиты против изменений в защите страниц PTE. Хотя битовая карта доступна для чтения из VTL 0, потому что элементы SLAT помечены как доступные только для чтения, атака режима ядра, пытающаяся изменить PTE-записи для пометки их доступными для чтения/записи, не сможет сделать то же с элементами SLAT. Ситуация будет опознана как недействительное обращение к битовой карте KCFG, по которому может принять меры HyperGuard (исключительно по соображениям телеметрии — биты изменить все равно не удастся).

Реализация KCFG практически идентична обычной реализации CFG, если не считать того, что подавление экспорта отключено, как и поддержка `longjmp` и возможность динамического запроса дополнительных битов для целей JIT. Драйверы ядра ничего такого делать не должны. Вместо этого биты устанавливаются в битовой карте на основании записей полученных адресов IAT-таблицы, если они установлены; обычными записями функций в защитной таблице при каждой загрузке образа драйвера; и `MiInitializeKernelCfg` для HAL и ядра в процессе загрузки. Если гипервизор не включен, а поддержка SLAT отсутствует, ничего из этого инициализировано не будет, и поддержка Kernel CFG останется отключенной.

Как и в случае с пользовательским режимом, динамический указатель в каталоге данных конфигурации загрузки обновляется; в случае включения он указывает на `__guard_check_icall` для проверочной функции и на `__guard_dispatch_icall` для функции диспетчеризации в расширенном режиме CFG. Кроме того, в переменной с именем `guard_icall_bitmap` будет храниться виртуальный адрес битовой карты.

И последняя подробность по поводу Kernel CFG: к сожалению, динамические настройки Driver Verifier настроить не удастся (за дополнительной информацией о Driver Verifier обращайтесь к главе 6), так как это требует добавления динамических обработчиков ядра и передачи управления функциям, которые могут отсутствовать в битовой карте. В таком случае возвращается код `STATUS_VRF_CFG_ENABLED` (0xC000049F), и потребуется перезагрузка (при которой битовая карта может быть построена с обработчиками Driver Verifier).

Заявления безопасности

Ранее мы уже описали, как Control Flow Guard завершает процесс. Также мы объяснили, как некоторые другие защитные меры или средства безопасности инициируют исключение для уничтожения процесса. Важно точно представлять, что *именно* происходит во время этих нарушений безопасности, потому что оба описания скрывают важные подробности о механизме.

На самом деле при возникновении дефекта, относящегося к безопасности (например, когда CFG обнаруживает некорректный косвенный вызов или переход), завершение процесса с использованием стандартного механизма `TerminateProcess` не подойдет. При этом не будет сгенерирована ошибка, а в Microsoft не будут отправлены телеметрические данные — это важные средства, при помощи которых администратор узнает о выполнении потенциального эксплойта или наличии проблем с совместимостью приложения, а компания Microsoft может отслеживать свежие уязвимости в реальной обстановке.

С другой стороны, хотя выдача исключения приведет к желаемому результату, исключения являются обратными вызовами, которые могут:

- перехватываться атакующими при отключенных защитных мерах `/SAFESEH` и `SEHOP`, в результате чего сама проверка безопасности передаст управление атакующему — или же атакующий может просто «поглотить» исключение;
- перехватываться законными частями продукта через фильтр необработанных исключений или векторный обработчик исключения; в обоих случаях возможно непреднамеренное поглощение исключения;
- то же самое, но с перехватом управления сторонним продуктом, который внедрил свою библиотеку в процесс. В этой ситуации, типичной для многих средств безопасности, исключение не будет корректно доставлено WER (Windows Error Reporting);

- у процесса может быть определен обратный вызов восстановления, зарегистрированный в WER. Это может привести к выдаче невразумительного пользовательского интерфейса и перезапуску процесса в его текущем уязвимом состоянии с любыми последствиями, от рекурсивного цикла сбоев/перезапусков до поглощения исключения;
- в продуктах на базе C++ исключение, инициированное самой программой, может быть перехвачено внешним обработчиком, который также может поглотить исключение или продолжить выполнение в небезопасном режиме.

Для решения этих проблем необходим механизм, способный инициировать исключения, которые не могут перехватываться компонентами процесса за пределами службы WER, тогда как последняя должна гарантированно получить исключение. И здесь на помощь приходят *заявления системы безопасности*.

Поддержка компилятора и ОС

Когда программы, компоненты ядра или библиотеки Microsoft сталкиваются с необычными ситуациями безопасности или когда защитные механизмы распознают опасные нарушения состояния безопасности, они теперь используют специальную внутреннюю (intrinsic) функцию компилятора `__fastfail` с одним входным параметром, поддерживаемую Visual Studio. В некоторых случаях WDK или SDK содержит встроенные (inline) функции, которые вызывают эту внутреннюю функцию — например, при использовании функций `LIST_ENTRY InsertTailList` и `RemoveEntryList`. Также uCRT (Universal CRT) включает эту внутреннюю функцию в своих функциях. Кроме того, эта внутренняя функция может использоваться для выполнения проверок некоторыми API-функциями, вызываемыми из приложений.

В любом случае, встречая эту внутреннюю функцию, компилятор генерирует код на языке ассемблера, который получает входной параметр, перемещает его в регистр RCX (x64) или ECX (x86), а затем выдает программное прерывание с номером 0x29. (За дополнительной информацией о прерываниях обращайтесь к главе 8 части 2.)

В Windows 8 и выше это программное прерывание регистрируется в таблице IDT (Interrupt Dispatch Table) с обработчиком `KiRaiseSecurityCheckFailure`; чтобы самостоятельно убедиться в этом, введите команду `!idt 29` в отладчике. Это приведет (по причинам совместимости) к вызову `KiFastFailDispatch` с кодом статуса `STATUS_STACK_BUFFER_OVERRUN` (0xC0000409). Затем происходит обычная диспетчеризация прерывания через `KiDispatchException`, но исключение при этом рассматривается как повторное (second-chance), это означает, что отладчик и процесс о нем оповещены не будут.

Эта конкретная ситуация распознается, и на ALPC-порт ошибок WER отправляется сообщение, как обычно. WER помечает исключение как не допускающее продолжение, из-за чего ядро завершит процесс обычным системным вызовом

`ZwTerminateProcess`. Все это гарантирует, что после использования прерывания в этом процессе не произойдет возврат в пользовательский режим, что WER получит уведомление, а процесс будет завершен (причем код ошибки будет соответствовать коду исключения). При генерировании записи исключения первым аргументом исключения будет входной параметр `__fastfail`.

Коды быстрого отказа/заявлений системы безопасности

Поскольку внутренняя функция `__fastfail` содержит входной аргумент, передаваемый записи исключения или экрану фатальной ошибки, это позволяет определить, какая часть системы работает неверно или столкнулась с нарушением системы безопасности. В табл. 7.22 перечислены различные условия ошибок, их смысл и критичность.

Таблица 7.22. Коды ошибок __fastfail

Код	Описание
Нарушение наследных ОС (0x0)	Старая проверка безопасности буфера в старом двоичном образе завершилась неудачей и была преобразована в заявление системы безопасности
Ошибка защиты V-Table (0x1)	Защита виртуальных таблиц в Internet Explorer 10 и выше обнаружила повреждение указателя на таблицу виртуальных функций
Ошибка проверки cookie стека (0x2)	Защитное значение cookie стека, сгенерированное ключом компилятора `/GS`, было повреждено
Повреждение элемента списка (0x3)	Один из макросов для манипуляции со структурами `LIST_ENTRY` обнаружил нарушение целостности связанного списка: родитель или потомок второго уровня не указывает на родителя или потомка обрабатываемого элемента
Нарушение структуры стека (0x4)	Была вызвана API-функция пользовательского режима или режима ядра, которая часто вызывается из ROP-эксплойтов со стеком, находящимся под контролем атакующего, и структура стека отличается от ожидаемой
Недопустимые аргументы (0x5)	API-функция CRT пользовательского режима (обычно) или другая критичная функция была вызвана с недействительным аргументом, что может служить признаком потенциальных атак на базе ROP или повреждения стека по иным причинам
Ошибка инициализации cookie стека (0x6)	Инициализация cookie стека завершилась неудачей, что может служить признаком модификации или нарушения целостности образа
Фатальная ошибка приложения (0x7)	Приложение использовало API-функцию пользовательского режима `FatalAppExit`, которая была преобразована в заявление безопасности с соответствующими преимуществами
Ошибка проверки диапазона (0x8)	Дополнительные проверки в некоторых фиксированных буферах проверяют, что индекс элемента массива лежит в ожидаемых границах

Код	Описание
Небезопасное обращение к реестру (0x9)	Драйвер режима ядра пытается обратиться к данным реестра из куста, находящегося под контролем пользователя (например, куст приложения или пользовательского профиля), не используя флаг `RTL_QUERY_REGISTRY_TYPECHECK` для защиты
Ошибка косвенного вызова CFG (0xA)	Механизм Control Flow Guard обнаружил косвенную инструкцию `CALL` или `JMP` с целевым адресом, недопустимым в соответствии с битовой картой CFG
Ошибка проверки записи CFG (0xB)	Механизм Control Flow Guard с защитой записи обнаружил недействительную запись в защищенные данные. Данная возможность (`/guard:cfw`) поддерживается только при тестировании в компании Microsoft
Недопустимое переключение волокна (0xC)	API-функция SwitchToFiber была использована для недействительного волокна или из потока, который не был преобразован в волокно
Недопустимое назначение контекста (0xD)	При попытке восстановления была обнаружена недействительная структура записи контекста (из-за исключения или API-функции `SetThreadContext`) с недействительным указателем стека. Проверяется только для процессов с активным CFG
Недопустимый счетчик ссылок (0xE)	У объекта со счетчиком ссылок (как, например, `OBJECT_HEADER` в режиме ядра или объект GDI Win32k.sys) счетчик ссылок упал ниже `0` или вернулся через максимум обратно к `0`
Недопустимый буфер перехода (0x12)	Совершается попытка перехода `longjmp` с буфером, содержащим недействительный адрес стека или недействительный указатель инструкций. Проверяется только для процессов с активным `CFG`
Изменение MRDATA (0x13)	Была изменена куча данных/секция загрузчика, доступная только для чтения. Проверяется только для процессов с активным `CFG`
Ошибка сертификации (0x14)	Одна из API-функций криптографической подсистемы столкнулась с проблемами при разборе сертификата или недействительным потоком ASN.1
Недействительная цепочка исключений (0x15)	Образ, скомпонованный с параметром `/SAFESEH` или с защитой `SEHOP`, столкнулся с недействительной диспетчеризацией обработчиков исключений
Криптографическая библиотека (0x16)	CNG.SYS, KSECDD.SYS или их эквивалентные API-функции пользовательского режима столкнулись с критической ошибкой
Недействительный вызов при исходящем вызове DLL (0x17)	Попытка вызова опасных функций из обратного вызова загрузчик пользовательского режима
Недействительная база образа (0x18)	Загрузчик образа пользовательского режима обнаружил недействительное значение `__ImageBase` (структура `IMAGE_DOS_HEADER`)
Ошибка защиты отложенной загрузки (0x19)	Обнаружено повреждение целостности таблицы IAT при загрузке импортированной функции с задержкой. Проверяется только для процессов с активным `CFG` и включенной защитой IAT с задержкой
Вызов небезопасного расширения (0x1A)	Инициируется при вызове некоторых API-функций расширения режима ядра, если состояние вызывающей стороны некорректно

Таблица 7.22 (*продолжение*)

Код	Описание
Вызов устаревшей службы (0x1B)	Инициируется при вызове некоторых неподдерживаемых и недокументированных системных функций
Недопустимое обращение к буферу (0x1C)	Инициируется функциями библиотеки времени выполнения в Ntdll при обнаружении каких-либо повреждений обобщенной структуры буфера
Недействительное сбалансированное дерево (0x1D)	Инициируется функциями библиотеки времени выполнения в Ntdll и ядре, когда структура `RTL_RB_TREE` или `RTL_AVL_TABLE` содержит недействительные узлы (у которых одноуровневые и/или родительские узлы не соответствуют предкам/потомкам по аналогии с проверками `LIST_ENTRY`)
Недействительный следующий поток (0x1E)	Инициируется планировщиком ядра, если следующий поток, планируемый в `KPRCB`, недействителен в каком-либо отношении
Подавленный вызов CFG (0x1F)	Инициируется при разрешении подавленного вызова CFG по соображениям совместимости. В такой ситуации WER помечает ошибку как обработанную, а ядро не завершает процесс, но телеметрические данные будут переданы Microsoft
Отключение APC (0x20)	Инициируется ядром при возврате в пользовательский режим, если APC ядра остаются заблокированными
Недопустимое состояние бездействия (0x21)	Инициируется диспетчером электропитания ядра, когда процессор пытается войти в недействительное состояние
Ошибка защиты MRDATA (0x22)	Инициируется загрузчиком пользовательского режима, если с изменяемой секции кучи, доступной только для чтения, уже снята защита за пределами ожидаемого пути выполнения кода
Непредвиденное исключение кучи (0x23)	Инициируется диспетчером кучи при повреждении кучи, которое может свидетельствовать о потенциальных эксплойтах
Недействительное состояние блокировки (0x24)	Инициируется ядром, когда некоторые блокировки не находятся в ожидаемом состоянии — например, если полученная блокировка уже находится в освобожденном состоянии
Недопустимая инструкция longjmp (0x25)	Инициируется `longjmp` при вызове, и для процесса включен механизм `CFG` с защитой `Longjmp`, но таблица `Longjmp` повреждена или отсутствует
Недопустимая цель longjmp (0x26)	Условия те же, что выше, но таблица `Longjmp` показывает, что функция не является допустимой целевой функцией `longjmp`
Недопустимый контекст диспетчеризации (0x27)	Инициируется обработчиком исключения в режиме ядра при попытке диспетчеризации исключения с неверной записью CONTEXT
Недействительный поток (0x28)	Инициируется планировщиком в режиме ядра при повреждении структуры `KTHREAD` в некоторых операциях планирования
Недопустимый номер системного вызова (0x29)	Аналогично «Вызов устаревшей службы», но WER помечает исключение как обработанное, в результате чего процесс продолжается; таким образом, используется только для сбора телеметрических данных

Код	Описание
Недопустимая операция с файлом (0x2A)	Используется диспетчером ввода/вывода и некоторыми файловыми системами для сбора телеметрических данных, как и в предыдущем пункте
Отказано в доступе к LPAC (0x2B)	Используется функцией проверки доступа SRM, когда AppContainer с более низкими привилегиями пытается обратиться к объекту, который не обладает SID ALL RESTRICTED APPLICATION PACKAGES, и отслеживание таких ошибок включено. В этом случае также происходит только сбор телеметрических данных без аварийного завершения процесса
Ошибка стека RFG (0x2C)	Используется RFG (Return Flow Guard), хотя в настоящее время эта возможность заблокирована
Ошибка целостности при загрузке (0x2D)	Используется одноименной защитной политикой, упоминавшейся ранее; означает, что был загружен непредвиденный образ с измененной сигнатурой или без сигнатуры
Ошибка подавления экспорта CFG (0x2D)	Используется CFG при включении с подавлением экспорта для обозначения того, что подавленный экспорт был целью косвенного перехода
Недействительный управляющий стек (0x2E)	Используется RFG, хотя в настоящее время эта возможность отключена
Отказ в назначении контекста (0x2F)	Используется одноименной защитной политикой, упоминавшейся ранее, хотя в настоящее время эта возможность отключена

Идентификация приложений (AppID)

Исторически решения в сфере безопасности в Windows основывались на идентификации пользователя (в форме пользовательского SID и принадлежности к группе), но растущее количество компонентов системы безопасности (AppLocker, брандмауэр, антивирус, противодействие вредоносным программам, службы управления правами — Rights Management Services и т. д.) вынуждает принимать решения по безопасности на основе выполняемого кода. В прошлом каждый из этих компонентов безопасности использовал свой собственный особый метод для идентификации приложений, что приводило к несовместимости и слишком сложной политике авторства. Цель AppID заключается во внесении согласованности в способах распознавания приложений компонентами системы безопасности путем предоставления единого набора API-интерфейсов и структур данных.

ПРИМЕЧАНИЕ Не путайте эту систему идентификации с системой AppID, используемой приложениями DCOM/COM+, где GUID представляет процесс, совместно используемый несколькими CLSID-идентификаторами, или с системой AppID, используемой приложениями Windows Live.

Так же как пользователь идентифицируется при входе в систему, приложение идентифицируется перед запуском путем генерирования AppID основной программы. Идентификатор AppID может быть сгенерирован из любого из следующих атрибутов приложения:

- **Поля.** Поля внутри сертификата подписи кода, встроенного в файл, позволяют получить различные комбинации имени издателя, имени продукта, имени файла и версии. `APPID://FQBN` является полностью определенным двоичным именем — Fully Qualified Binary Name, представленным в виде строки следующего формата: `{Издатель\Продукт\Имя_файла,Версия}`. Имя издателя является полем `Subject` сертификата x.509, использованного для подписи кода, с использованием следующих полей:
  - **O** — Organization (организация);
  - **L** — Locality (локализация);
  - **S** —State (штат или провинция);
  - **C** — Country (страна).
- **Хеши файлов.** Существует ряд методов, которые можно использовать для создания хешей. Исходным методом является `APPID://SHA256HASH`. Но для обратной совместимости с SRP и большинством сертификатов x.509 по-прежнему поддерживается метод SHA-1 (`APPID://SHA1HASH`). `APPID://SHA256HASH` обозначает хеш файла SHA-256.
- **Частичный или полный путь к файлу.** `APPID://Path` указывает путь с дополнительными метасимволами (`*`).

ПРИМЕЧАНИЕ AppID не служит средством сертификации качества или безопасности приложения. Это всего лишь способ идентификации, чтобы администраторы могли ссылаться на приложение в решениях политики безопасности.

AppID хранится в маркере доступа процесса, позволяя любым компонентам системы безопасности принять решения по авторизации на основе единой согласованной идентификации. Механизм AppLocker использует условные ACE-элементы (рассмотренные ранее) для определения того, разрешено ли конкретной программе быть запущенной тем или иным пользователем.

Когда AppID создается для файла, имеющего цифровую подпись, сертификат этого файла кэшируется и сличается с доверенным корневым сертификатом. Путь сертификата ежедневно перепроверяется, чтобы убедиться, что путь сертификата остался правильным. Кэширование и проверка сертификата записываются в системный журнал событий в разделе Application and Services Logs\Microsoft\Windows\AppID\Operational.

AppLocker

В Windows 8.1 и в Windows 10 (Enterprise-выпуски), а также Windows Server 2012/R2/2016 поддерживается механизм *AppLocker*, позволяющий администратору заблокировать систему для предотвращения запуска неавторизованных программ. В Windows XP появились политика ограниченного использования программ — Software Restriction Policies (SRP), которая была первым шагом по направлению к этой возможности, но SRP была трудна в управлении и не могла применяться к конкретным пользователям или группам (правила SRP распространялись на всех пользователей). AppLocker пришел на замену SRP и пока сосуществует рядом с SRP, а правила AppLocker хранятся отдельно от правил SRP. Если и правила AppLocker, и правила SRP находятся в одном и том же объекте групповой политики — Group Policy object (GPO), то будут применяться только правила AppLocker.

Другим аспектом, определяющим превосходство AppLocker над SRP, является режим аудита AppLocker, позволяющий администратору создавать политику AppLocker и проверять результаты (хранящиеся в системном журнале событий) для определения, будет ли политика выполняться, как ожидалось, фактически не накладывая ограничений. Режим аудита AppLocker может использоваться для отслеживания того, какие приложения используются одним или несколькими пользователями системы.

AppLocker позволяет администратору ограничить запуск файлов следующих типов:

- исполняемых образов (.EXE и .COM);
- динамически подключаемых библиотек (.DLL и .OCX);
- установщика Microsoft Software Installer (.MSI и .MSP) как для установки, так и для удаления установки;
- сценариев Windows PowerShell (.PS1);
- пакетных файлов (.BAT и .CMD);
- сценариев VisualBasic (.VBS);
- сценариев JavaScript (.JS).

AppLocker предоставляет простой GUI-механизм на основе правил, которые очень похожи на правила сетевого брандмауэра, для определения того, каким приложениям или сценариям разрешено запускаться конкретными пользователями и группами, с использованием условных ACE-элементов и AppID-атрибутов. В AppLocker используются правила двух типов:

- разрешить запуск конкретных файлов, запретив все остальное;
- запретить запуск конкретных файлов, разрешив все остальное. «Запрещающие» правила имеют приоритет над «разрешающими».

У каждого правила может также быть список исключений для удаления файлов из правила. Использовав исключение, можно создать правило «Разрешить запускать все в каталогах C:\Windows или C:\Program Files, за исключением Regedit.exe».

Правила AppLocker могут быть связаны с конкретным пользователем или группой. Это позволяет администратору поддерживать требования совместимости путем проверки и принудительной установки круга пользователей, имеющих возможность запуска определенных приложений. Например, можно создать правило «Разрешить пользователям, принадлежащим к группе Finance security, запускать приложения финансового направления». Это правило заблокирует всем, кто не входит в группу Finance security, возможность запуска финансовых приложений (включая администраторов), но по-прежнему предоставит доступ тем, у кого есть обоснованные потребности в запуске приложений. Другим полезным правилом станет предохранение пользователей из группы Receptionists от установки или запуска не получивших одобрения приложений.

Правила AppLocker зависят от условных ACE-элементов и атрибутов, определяемых AppID. Правила могут быть созданы с использованием следующих критериев.

- **Полей внутри сертификата, подписывающего код и встроенного в файл, которые позволяют составлять различные комбинации из имени издателя, имени продукта, имени файла и версии.** Например, правило может быть создано для того, чтобы «Разрешить запускаться всем версиям Contoso Reader, чей номер выше 9.0» или «Разрешить всем из группы graphics запускать установщик или приложение от Contoso под названием GraphicsShop, если версия имеет номер 14*». Например, следующая SDDL-строка запрещает пользователю, вошедшему в систему под учетной записью RestrictedUser (идентифицируемой по SID пользователя), доступ для выполнения к любым, имеющим цифровую подпись программам, изданным Contoso:

```
D:(XD;;FX;;;S-1-5-21-3392373855-1129761602-2459801163-1028;((Exists APPID://FQBN)
&& ((APPID://FQBN) >= ({"O=CONTOSO, INCORPORATED, L=REDMOND,
S=CWASHINGTON, C=US\*\*",0}))))
```

- **Пути к каталогу, разрешающему запуск только тех файлов, которые находятся внутри определенного дерева каталогов.** Этот критерий может также использоваться для идентификации конкретных файлов. Например, следующая SDDL-строка запрещает пользователю, вошедшему в систему под учетной записью RestrictedUser (идентифицируемой по SID пользователя), доступ для выполнения к программам в каталоге C:\Tools:

```
D:(XD;;FX;;;S-1-5-21-3392373855-1129761602-2459801163-1028;(APPID://PATH
Contains "%OSDRIVE%\TOOLS\*"))
```

- **Хеша файла.** Использование хеша также позволит обнаружить внесение в файл изменений и помешать его запуску. Это может рассматриваться как недостаток, если файлы часто подвергаются изменениям, поскольку правило на основе хеша нужно будет часто обновлять. Хеши файлов часто используются для сценариев,

поскольку цифровую подпись имеют лишь некоторые сценарии. Например, следующая SDDL-строка запрещает пользователю, вошедшему в систему под учетной записью RestrictedUser (идентифицируемой по SID пользователя), доступ для выполнения программ с указанными значениями хеша:

```
D:(XD;;FX;;;S-1-5-21-3392373855-1129761602-2459801163-1028;(APPID://SHA256HASH
Any_of {#7a334d2b99d48448eedd308dfca63b8a3b7b44044496ee2f8e236f5997f1b647,
#2a782f76cb94ece307dc52c338f02edbbfdca83906674e35c682724a8a92a76b}))
```

На локальной машине правила AppLocker могут быть определены с помощью MMC-оснастки Локальная политика безопасности (Security Policy, secpol.msc, рис. 7.33) или с помощью сценария Windows PowerShell либо могут быть навязаны машинам, принадлежащим домену с использованием групповой политики. Правила AppLocker хранятся в нескольких местах реестра:

- **HKLM\Software\Policies\Microsoft\Windows\SrpV2.** Этот раздел также зеркально копируется в разделе HKLM\SOFTWARE\Wow6432Node\Policies\Microsoft\Windows\SrpV2. Правила хранятся в формате XML.
- **HKLM\SYSTEM\CurrentControlSet\Control\Srp\Gp\Exe.** Правила хранятся в виде SDDL и двоичного ACE.
- **HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Group PolicyObjects\{GUID}Machine\Software\Policies\Microsoft\Windows\SrpV2.** AppLocker-политика, навязываемая доменом как часть объекта групповой политики — Group Policy Object (GPO), хранится здесь в формате XML.

Сертификаты для файлов, которые были запущены, кэшируются в реестре в разделе HKLM\SYSTEM\CurrentControlSet\Control\AppID\CertStore. AppLocker также создает цепочку сертификатов (хранящуюся в разделе HKLM\SYSTEM\CurrentControlSet\Control\AppID\CertChainStore) от сертификата, найденного в файле, назад к доверенному корневому сертификату.

Есть также ориентированные на AppLocker команды PowerShell (известные также как *командлеты* — cmdlets), позволяющие вести развертывание и тестирование с помощью сценариев. Команда `Import-Module AppLocker` импортирует в PowerShell командлеты AppLocker, в том числе `Get-AppLockerFileInformation`, `Get-AppLockerPolicy`, `New-AppLockerPolicy`, `Set-AppLockerPolicy` и `Test-AppLockerPolicy`.

Службы AppID и SRP сосуществуют в одном двоичном файле (AppIdSvc.dll), который выполняется в процессе SvcHost. Служба запрашивает уведомление о внесении изменений в реестр для отслеживания любого изменения в тех разделах, которые записаны либо GPO, либо AppLocker UI в MMC-оснастке Локальная политика безопасности. При обнаружении изменения служба AppID запускает задание пользовательского режима (AppIdPolicyConverter.exe), которое читает новые правила в формате XML и переводит их в двоичный формат ACE-элементов и SDDL-строк, поддерживаемых компонентами AppID и AppLocker, относящимися как к пользовательскому режиму, так и к режиму ядра. Задание сохраняет пере-

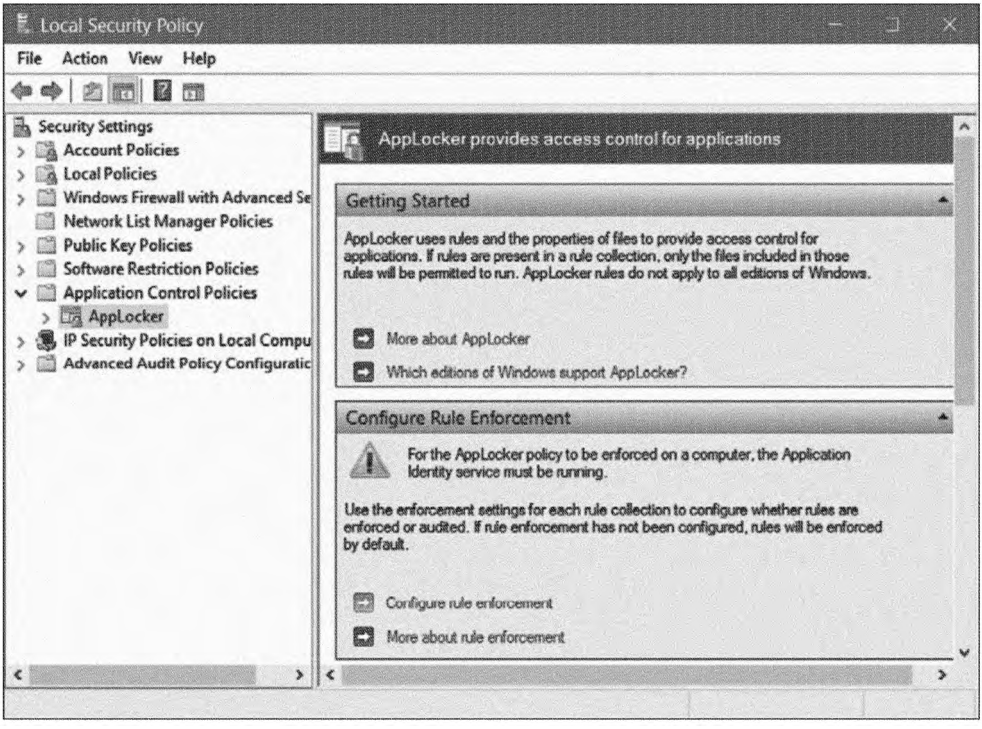


Рис. 7.33. Страница конфигурации AppLocker в оснастке Локальная политика безопасности

веденные правила в разделе HKLM\SYSTEM\CurrentControlSet\Control\Srp\Gp. Этот раздел доступен для записи только группам Система (SYSTEM) и Администраторы (Administrators) и помечен для аутентифицированных пользователей как раздел только для чтения. Компоненты AppID как пользовательского режима, так и режима ядра считывают переведенные правила непосредственно из реестра. Служба также отслеживает хранилище доверенных корневых сертификатов локальной машины и вызывает задание пользовательского режима (AppIdCertStoreCheck.exe) для повторной проверки сертификатов как минимум ежедневно, а также при каждом изменении в хранилище сертификатов. Драйвер AppID режима ядра (%SystemRoot%\System32\drivers\AppId.sys) уведомляется об изменении правил службой AppID посредством `DeviceIoControl`-запроса `APPID_POLICY_CHANGED` (рис. 7.34).

Реализации AppID, AppLocker и SRP не имеют четко обозначенных границ и нарушают принципы строгого разделения по уровням, здесь различные логические компоненты сосуществуют в одних и тех же исполняемых файлах, и система присваивания имен не столь последовательна, как хотелось бы.

Служба AppID запускается как `LocalService`, поэтому она имеет доступ к имеющемуся в системе хранилищу доверенных корневых сертификатов — Trusted

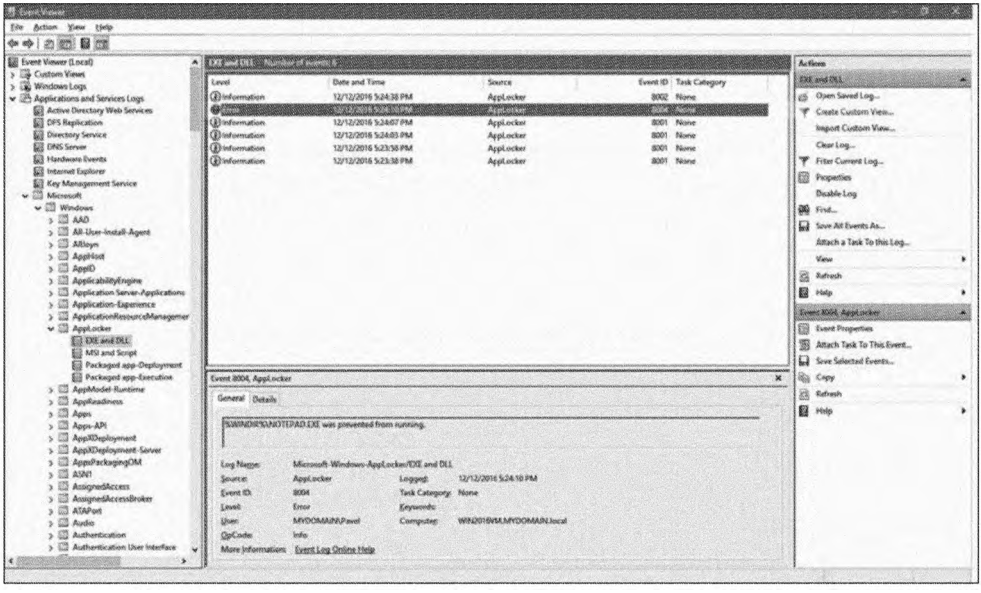


Рис. 7.34. Просмотрщик событий, показывающий AppLocker, разрешающий и запрещающий доступ к различным приложениям. Идентификатор события Event ID со значением 8004 означает «запрещен», а со значением 8002 означает «разрешен»

Root Certificate Store. Это также позволяет ей выполнять проверку сертификатов. Служба AppID отвечает за следующие действия:

- проверку сертификатов издателей;
- добавление в кэш новых сертификатов;
- обнаружение обновления правила AppLocker и уведомление драйвера AppID.

Драйвер AppID выполняет основную часть функциональности AppLocker и зависит от обмена данными (через запросы `DeviceIoControl`) со службой AppID, поэтому его объект устройства защищен с помощью ACL, предоставляя доступ только группам NT SERVICE\AppIDSvc, NT SERVICE\LOCAL SERVICE и BUILTIN\Administrators. Таким образом, драйвер не может быть фальсифицирован вредоносным кодом.

При первой загрузке драйвера AppID он запрашивает создание процесса функции обратного вызова путем вызова процедуры `PsSetCreateProcessNotifyRoutineEx`. Когда вызывается процедура CreateProcessNotifyEx, ей передается структура `PPS_CREATE_NOTIFY_INFO` (описывающая создаваемый процесс). Затем она собирает атрибуты AppID, идентифицирующие исполняемый образ, и записывает их в маркер доступа процесса. Далее она вызывает недокументированную процедуру `SeSrpAccessCheck`, которая проверяет маркер процесса и условные ACE-элементы правил AppLocker, и определяет, нужно ли разрешать запуск процесса. Если запуск процесса не должен быть разрешен, драйвер записывает `STATUS_ACCESS_DISABLED_`

`BY_POLICY_OTHER` в поле статуса (`Status`) структуры `PPS_CREATE_NOTIFY_INFO`, что приводит к прекращению создания процесса (и устанавливает окончательный статус завершения процесса).

Для ограничения выполнения DLL загрузчик образов отправляет запрос `DeviceIoControl` драйверу AppID при загрузке DLL в процессе. Затем драйвер проверяет идентичность DLL по условным ACE-элементам AppLocker, точно так же, как он сделал бы это в отношении исполняемого файла.

ПРИМЕЧАНИЕ Выполнение этих проверок для каждой загрузки DLL требует времени; задержку могут заметить конечные пользователи. Поэтому правила, касающиеся DLL, обычно отключены, и их приходится специально включать через вкладку Дополнительно (Advanced) страницы свойств AppLocker в оснастке Локальная политика безопасности (Local Security Policy).

Ядро сценариев и MSI-установщик были изменены для вызова API-функций SRP пользовательского режима, как только они открывают файл, чтобы проверить, разрешено ли открытие файла. API-функции SRP пользовательского режима вызывают API-функции AuthZ для выполнения проверки прав доступа с помощью условного ACE-элемента.

Политики ограниченного использования программ

Windows также содержит механизм пользовательского режима, который называется *политиками ограниченного использования программ* (Software Restriction Policies) и позволяет администраторам управлять тем, какие образы и сценарии выполняются на их системах. Политики ограниченного использования программ являются узлом редактора локальной политики безопасности, показанным на рис. 7.35, которые служат в качестве интерфейса управления политиками исполняемого кода для всей машины, хотя для каждого пользователя политики также доступны с использованием доменных групповых политик.

В узле политик ограниченного использования программ содержится ряд настроек глобальной политики:

- **Применение** (Enforcement) настраивает применение политик к библиотекам, например к DLL, и определяет применение политик только к пользователям или к администраторам в том числе;
- **Назначенные типы файлов** (Designated File Types) хранит расширения файлов, которые считаются исполняемым кодом;
- **Доверенные издатели** (Trusted Publishers) управляет тем, кто может выбирать, какие из сертификатов издателей являются доверенными.

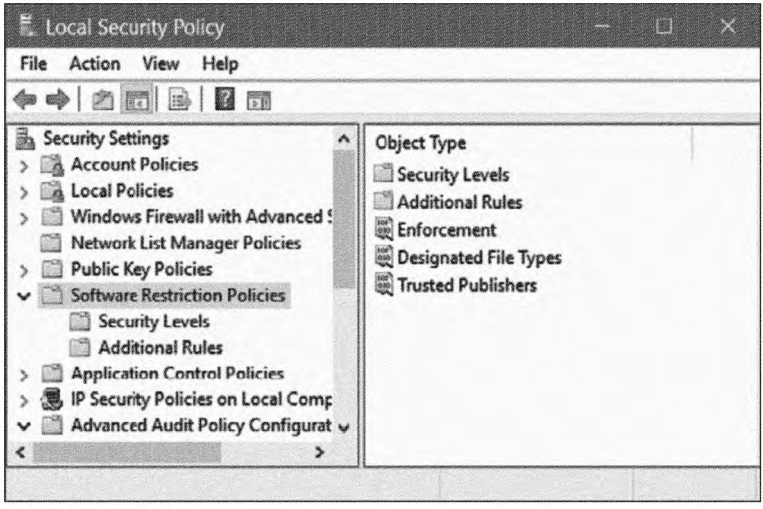


Рис. 7.35. Настройка политик ограниченного использования программ

При настройке политики для конкретного сценария или образа администратор может направить систему на его распознавания с использованием его пути, его хеша, его зоны интернета (в соответствии с ее определением в Internet Explorer) или его криптографического сертификата, и он может определить, связан ли сценарий или образ с политиками безопасности Не разрешено (Disallowed) или Неограниченный (Unrestricted).

Политика SRP реализуется различными компонентами, которые работают с файлами, содержащими исполняемый код. Некоторые из таких компонентов:

- Функция пользовательского режима `CreateProcess` из библиотеки Kernel32.dll применяет ее в отношении исполняемых образов.
- Код загрузки DLL в библиотеке Ntdll применяет ее для DLL-библиотек.
- Окно командной строки Windows (Cmd.exe) применяет ее при выполнении пакетных файлов.
- Компоненты Windows Scripting Host, запускающие сценарии Cscript.exe (для сценариев командной строки), Wscript.exe (для UI-сценариев) и Scrobj.dll (для объектов сценариев), применяют ее при выполнении сценариев.
- Хост PowerShell (PowerShell.exe) применяет ее для выполнения сценариев PowerShell.

Каждый из этих компонентов определяет, включены ли политики ограничений, считывая значение параметра реестра HKEY_LOCAL_MACHINE\Software\Microsoft\Policies\Windows\Safer\CodeIdentifiers\TransparentEnabled, которое при установке в 1 показывает, что политики задействованы. Затем он определяет, соответствует ли код, предназначенный для выполнения, одному из правил, указанных в подразделе

раздела `CodeIdentifiers`, и, если соответствует, определяет, должно быть разрешено выполнение или нет. Если соответствие не обнаружено, определение разрешения на выполнение осуществляется на основе исходной политики, указанной в значении параметра `DefaultLevel` из раздела `CodeIdentifiers`.

Политики ограниченного использования программ — мощный инструмент для предотвращения несанкционированного доступа к коду и к сценариям, но только при условии их правильного применения. Если только политика по умолчанию не настроена на запрет выполнения, пользователь может внести незначительные изменения в образ, помеченный как запрещенный, чтобы обойти правило и выполнить этот образ. Например, пользователь может изменить байт в образе процесса, чтобы хеш-правило не смогло его распознать, или скопировать файл в другое место, чтобы обойти правило, основанное на пути к файлу.

ЭКСПЕРИМЕНТ: ПРОСМОТР ПРИНУДИТЕЛЬНОГО ПРИМЕНЕНИЯ ПОЛИТИК ОГРАНИЧЕННОГО ИСПОЛЬЗОВАНИЯ ПРОГРАММ

Проследить за принудительным применением политик ограниченного использования программ косвенным образом можно, наблюдая за обращениями к реестру при попытке выполнения образа, который был вами запрещен к выполнению.

1. Запустите оснастку secpol.msc, чтобы открыть редактор локальной политики безопасности. Перейдите к узлу Политики ограниченного использования программ (Software Restriction Policies).
2. Если политики не определены, выберите пункт контекстного меню Создать новые политики (Create New Policies).
3. Создайте основанную на пути к файлу запрещающую политику ограничения для %SystemRoot%\System32\Notepad.exe (в узле Дополнительные правила (Additional Rules)).
4. Запустите утилиту Process Monitor и установите включающий фильтр пути для Safer.
5. Откройте окно командной строки и запустите Блокнот (Notepad) из приглашения.

Ваша попытка запустить Notepad приведет к выводу сообщения, в котором говорится, что вы не можете выполнить указанную программу, а Process Monitor должен показать окно командной строки (cmd.exe), запрашивающее политики ограничений на локальной машине.

Защита ядра от модификации

Некоторые драйверы устройств изменяют поведение Windows неподдерживаемыми способами. Например, они исправляют таблицу системных вызовов для перехвата этих вызовов или исправляют образ ядра в памяти для добавления

функциональных возможностей определенным внутренним функциям. Такие модификации принципиально опасны, они могут снизить уровень стабильности и безопасности системы. Кроме того, такие модификации могут вноситься со злым умыслом — либо ненадежными драйверами, либо через эксплойты, обусловленные уязвимостями в драйверах Windows.

Без присутствия сущности более привилегированной, чем само ядро, обнаружение и защита от эксплойтов становится непростым делом. Так как и механизм обнаружения/защиты, и код с нежелательным поведением работают в кольце 0, невозможно определить границу безопасности в истинном смысле слова, так как само нежелательное поведение может использоваться для блокировки, модификации или обмана механизма обнаружения/предотвращения. Тем не менее даже в таких условиях механизм реакции на такие нежелательные операции может быть полезен:

- При фатальном сбое с четко идентифицируемым аварийным дампом режима ядра как пользователи, так и администраторы видят, что нежелательное поведение происходит в ядре, и могут предпринять соответствующие действия. Кроме того, это означает, что законопослушные разработчики программного обеспечения не захотят создавать риск сбоев в системах своих клиентов и будут ориентироваться на поддерживаемые механизмы расширения функциональности ядра (например, использовать диспетчер фильтров для фильтров файловой системы или другие механизмы на базе обратных вызовов).
- Искажение (obfuscation, которое не является границей безопасности) делает более затратным — по времени и/или по сложности — блокировку механизма обнаружения кодом с нежелательным поведением. Эти дополнительные затраты означают, что нежелательное поведение будет более четко идентифицироваться как потенциально вредоносное, а сложность создаст больше проблем для потенциального атакующего.
- Рандомизация и отсутствие документирования конкретных проверок, применяемых механизмом обнаружения/предотвращения для контроля целостности ядра, а также недетерминированность времени применения таких проверок, мешают атакующему убедиться в надежности своих эксплойтов. Атакующему приходится учитывать все возможные недетерминированные переменные и переходы состояний посредством статического анализа, что при применении искажения становится практически невозможным в пределах временного интервала до реализации другого способа искажения или изменения функциональности в механизме.
- Так как аварийные дампы режима ядра автоматически передаются Microsoft, это позволяет компании собирать телеметрические данные реального нежелательного кода. Это позволяет ей идентифицировать разработчиков ПО, которые выпускают код, вызывающий сбои, отслеживать эволюцию вредоносных драйверов (и даже эксплойтов «нулевого дня») в реальных условиях и исправлять ошибки, о которых еще не было объявлено официально, но которые активно эксплуатировались.

PatchGuard

Вскоре после выпуска 64-разрядной Windows для x64 и перед тем, как была разработана богатая экосистема сторонних производителей, компания Microsoft увидела благоприятную возможность для сохранения стабильности 64-разрядной Windows. Для этого в системе был реализован механизм сбора телеметрии и выявления модификаций ядра от исправлений, основанный на технологии *KPP* (Kernel Patch Protection), известной также как *PatchGuard*. С выпуском системы Windows Mobile, работающей на 32-разрядном процессоре ARM, функциональность была адаптирована и для таких систем; она также будет присутствовать в 64-разрядных системах ARM (AArch64). Тем не менее из-за существования слишком большого количества старых 32-разрядных драйверов, использующих неподдерживаемые и опасные методы передачи управления, этот механизм не включается в таких системах — даже в Windows 10. К счастью, эпоха 32-разрядных систем почти завершилась, и серверные версии Windows более не поддерживают эту архитектуру.

Хотя название вроде бы указывает на то, что этот механизм *защищает* систему, важно понимать, что вся защита сводится к аварийному завершению работы, предотвращающему дальнейшее выполнение нежелательного кода. Этот механизм не *предотвращает* атаку, не принимает мер против нее и не отменяет ее. KPP можно рассматривать как систему видеонаблюдения, подключенную к интернету: это сигнализация (сбой) при проникновении в хранилище (ядро), а не непреодолимый замок на двери хранилища.

KPP использует много различных проверок в защищенных системах. Перечислять их все было бы непрактично (из-за сложности статического анализа), вдобавок эта информация могла бы пригодиться потенциальным злоумышленникам (которым придется тратить меньше времени на сбор информации). Тем не менее компания Microsoft документирует некоторые проверки, представленные в табл. 7.23. Вопросы о том, когда, где и как KPP осуществляет эти проверки и на какие конкретные функции и структуры данных они распространяются, выходят за рамки нашего анализа.

Таблица 7.23. Обобщенное описание элементов, защищаемых KPP

Компонент	Допустимое использование	Потенциальное вредоносное использование
Исполняемый код в ядре, его зависимости и основные драйверы, а также таблицы IAT этих компонентов	Стандартные компоненты Windows, играющие ключевую роль для работы режима ядра	Исправление кода этих компонентов может изменить их поведение и создать нежелательные лазейки в системе, скрыть от системы данные или нежелательные коммуникации, а также снизить ее стабильность и даже создать дополнительные уязвимости через сторонний код, содержащий дефекты безопасности

Компонент	Допустимое использование	Потенциальное вредоносное использование
GDT (Global Descriptor Table) (глобальная таблица дескрипторов)	Аппаратная защита центрального процессора для реализации кольцевых уровней привилегий (Ring 0 против Ring 3)	Модификация ожидаемых разрешений и отображений между кодом и уровнями колец, открывающих коду кольца 3 доступ к коду кольца 0
IDT (Interrupt Descriptor Table) (таблица дескрипторов прерываний)	Таблица считывается центральным процессором для предоставления векторов прерываний к правильной процедуре обработки	Перехват нажатий клавиш, сетевых пакетов, операций страничного механизма, системных вызовов, обмена данными с гипервизором и т. д. — всего, что может использоваться для несанкционированного доступа, сокрытия вредоносных данных или коммуникаций или случайного введения уязвимостей из-за дефектов в стороннем коде
SSDT (System Service Descriptor Table) (таблица дескрипторов системных служб)	Таблица, содержащая массив указателей для каждого обработчика системных вызовов	Перехват всех взаимодействий из пользовательского режима с ядром. Те же проблемы, что и в предыдущем пункте
Критические регистры процессора: управляющие регистры, регистр базового адреса вектора и регистры конкретных моделей	Используется для системных вызовов, виртуализации, таких средств безопасности процессора, как SMEP, и для других целей	То же, что выше, плюс отключение критических средств безопасности процессора или защиты гипервизора
Различные указатели на функции ядра	Используются для косвенных вызовов различной внутренней функциональности	Могут использоваться для перехвата внутренних операций ядра с созданием дефектов безопасности и/или нестабильного поведения системы
Различные глобальные переменные ядра	Используются для настройки различных частей ядра, включая некоторые аспекты безопасности	Вредоносный код может заблокировать средства безопасности — например, через эксплойт пользовательского режима, позволяющий выполнять запись в произвольные участки памяти
Список процессов и модулей	Используется для вывода информации об активных процессах и загруженных драйверах в таких программх, как диспетчер задач, Process Explorer и отладчик Windows	Вредоносный код может скрывать существование некоторых процессов или драйверов на машине, делая их невидимыми для пользователя и некоторых приложений (например, средств безопасности)
Стек ядра	Хранение аргументов функций, стека вызовов (адресов, по которым должно возвращаться управление) и переменных	Операции с нестандартным стеком ядра часто являются признаком ROP-эксплойтов (Return-Oriented Programming) с атаками, основанными на смещении стека

Таблица 7.23 (*окончание*)

Компонент	Допустимое использование	Потенциальное вредоносное использование
Диспетчер окон, вызовы графической системы, обратные вызовы и т. д.	Предоставляет функциональность GUI, GDI и DirectX	Те же возможности перехвата, о которых говорилось выше, но ориентированные на графику и стек управления окнами. Проблемы те же, что и с другими типами перехватчиков
Типы объектов	Определения разнообразных объектов (например, процессов и файлов), поддерживаемых системой с помощью диспетчера объектов	Может использоваться как метод перехвата управления, не ориентированный ни на косвенные указатели на функции в секциях данных двоичных файлов, ни на прямую модификацию кода. Те же проблемы
Локальный расширенный контроллер прерываний (APIC)	Используется для получения аппаратных прерываний процессором, получения прерываний таймера и межпроцессорных прерываний (IPI)	Может использоваться для перехвата операций таймера, IPI и прерываний, а также для поддержания существования периодически выполняемого скрытого кода
Фильтры и обратные вызовы сторонних уведомлений	Используются сторонним программным обеспечением безопасности (и Защитником Windows) для получения уведомлений о системных действиях, а в отдельных случаях даже блокировки/защиты от некоторых действий. По сути, предоставляет как поддерживаемый способ для реализации большей части функциональности, от которой защищает KPP	Может использоваться вредоносным кодом для перехвата любых фильтруемых операций, а также для обеспечения существования периодически выполняемого кода
Специализированная конфигурация и флаги	Различные структуры данных, флаги и элементы действительных компонентов, которые обеспечивают безопасность и/или гарантии защиты для этих компонентов	Может использоваться вредоносным кодом для обхода некоторых защитных мер или нарушения некоторых гарантий/ожиданий процессов (например, снятия защиты с защищенного процесса)
Сам механизм KPP	Код, относящийся к проверке системы при нарушениях KPP, выполнении обратных вызовов, связанных с KPP, и т. д.	Изменяя некоторые части системы, используемые KPP, нежелательные компоненты могут попытаться заглушить, обойти или иным образом нарушить работу KPP

Как упоминалось ранее, когда KPP обнаруживает в системе присутствие нежелательного кода, в системе происходит фатальный сбой с четко опознаваемым кодом 0x109 — `CRITICAL_STRUCTURE_CORRUPTION` (критическое изменение структуры). Для анализа аварийного дампа можно воспользоваться отладчиком Windows (за дополнительной информацией обращайтесь к главе 15 части 2). Дамп содержит *некоторую* информацию о поврежденных или некорректно модифицированных частях ядра, но расширенная информация недоступна для пользователей и анализируется группами Microsoft OCA (Online Crash Analysis) и/или WER (Windows Error Reporting).

Сторонние разработчики, использующие средства, подавляемые KPP, могут воспользоваться следующими поддерживаемыми технологиями:

- **(Мини) фильтрами файловой системы** для перехвата всех файловых операций, включая загрузку файлов образов и DLL-библиотек, которые могут быть перехвачены для оперативного уничтожения вредоносного кода или для блокирования чтения известных вредоносных исполняемых файлов. (За дополнительной информацией обращайтесь к главе 13 части 2.)
- **Уведомлениями фильтра реестра** для перехвата всех операций с реестром (за дополнительной информацией об этих уведомлениях обращайтесь к главе 9 части 2). Программы обеспечения безопасности могут заблокировать изменения критических частей реестра, а также эвристически определить вредоносное программное обеспечение за счет шаблонов обращения к реестру или известных «плохих» разделов реестра.
- **Уведомлениями процесса.** Программы обеспечения безопасности могут отслеживать выполнение и завершение всех процессов и потоков системы, а также загружаемые и выгружаемые DLL-библиотеки. Используя расширенные уведомления, добавленные для поставщиков антивирусов и других программ обеспечения безопасности, они также могут блокировать запуск процесса. (За дополнительной информацией об этих уведомлениях обращайтесь к главе 3.)
- **Фильтрацией диспетчера объектов.** Программы обеспечения безопасности могут удалять некоторые права доступа, предоставленные процессам и/или потокам для защиты своих собственных утилит от определенных операций. (Эта тема рассматривается в главе 8 части 2.)
- **Фильтрами NDIS Lightweight Filters (LWF) и Windows Filtering Platform (WFP).** Программное обеспечение безопасности может перехватывать все операции с сокетами (прослушивание, подключение, закрытие и т. д.) и даже сами пакеты. С LWF разработчики средств безопасности получают доступ к низкоуровневым данным Ethernet-фреймов, передаваемым от сетевого адаптера в сеть.
- **ETW (Event Tracing for Windows).** При использовании ETW многие типы операций, обладающие интересными свойствами с точки зрения безопасности, могут потребляться компонентами пользовательского режима, реагирующими на данные практически в реальном времени. В некоторых ситуациях процессы

программного обеспечения безопасности могут получить доступ к специальным защищенным уведомлениям ETW на условиях соглашения о неразглашении информации (NDA) с Microsoft и участия в различных программах безопасности; таким образом они получают доступ к расширенному набору данных трассировки. (ETW рассматривается в главе 8 части 2.)

HyperGuard

В системах со средствами безопасности, основанными на виртуализации (см. раздел «Безопасность на основе виртуализации» этой главы), атака с привилегиями режима ядра уже не выполняется в пределах одной границы безопасности с механизмом обнаружения/предотвращения атак. Атакующий действует на уровне VTL 0, тогда как механизм может быть реализован на уровне VTL 1. В обновлении Windows 10 Anniversary Update (версия 1607) такой механизм действительно существует; он называется *HyperGuard*. У механизма HyperGuard есть несколько интересных свойств, отличающих его от PatchGuard:

- Ему не нужно полагаться на искажение (obfucation). Файлы с символической информацией и имена функций, реализующих HyperGuard, доступны для всех желающих, а код не искажается. Возможен полный статический анализ, поэтому HyperGuard является полноценной границей безопасности.
- Его работа не обязана быть недетерминированной из-за предыдущего свойства. Более того, благодаря детерминированному характеру своей работы HyperGuard может инициировать сбой системы ровно в тот момент, когда будет обнаружено нежелательное поведение. А это означает, что данные сбоя будут содержать четкие и полезные в практическом смысле данные для администратора (и аналитических групп Microsoft) — например, стек ядра, который показывает, какой код вызвал нежелательное поведение.
- Благодаря предыдущему свойству HyperGuard может обнаруживать более широкий спектр атак, потому что вредоносный код не получит возможности восстановить правильное значение за фиксированный промежуток времени — неприятный побочный эффект недетерминированности PatchGuard.

Механизм HyperGuard также используется для расширения функциональности PatchGuard и укрепления возможности выполняться скрытно от атакующих, пытающихся заблокировать его. Обнаружив аномалию, HyperGuard также вызывает сбой системы, хотя и с другим кодом: 0x18C (`HYPERGUARD_VIOLATION`). Как и прежде, будет полезно понимать (хотя бы на общем уровне), что именно обнаруживает HyperGuard; эти компоненты перечислены в табл. 7.24.

В системах с включенной функциональностью VBS существует еще один заслуживающий упоминания механизм, относящийся к безопасности и реализованный в самом гипервизоре: NPIEP (Non-Privileged Instruction Execution Prevention). Этот защитный механизм ориентирован на конкретные инструкции x64, которые

могут использоваться для организации утечки адресов режима ядра GDT, IDT и LDT – SGDT, SIDT и SLDT. С NPIEP выполнение этих инструкций разрешено (по соображениям совместимости), но они возвращают уникальное число для каждого процессора, которое в действительности не является адресом этих структур режима ядра. Это служит защитой от информационных утечек KASLR (Kernel ASLR) от локальных атак.

Таблица 7.24. Элементы, защищаемые HyperGuard

Компонент	Допустимое использование	Потенциальное вредоносное использование
Исполняемый код в ядре, его зависимости и основные драйверы, а также таблицы IAT этих компонентов	См. табл. 7.23	См. табл. 7.23
GDT (Global Descriptor Table) (глобальная таблица дескрипторов)	См. табл. 7.23	См. табл. 7.23
IDT (Interrupt Descriptor Table) (таблица дескрипторов прерываний)	См. табл. 7.23	См. табл. 7.23
Критические регистры процессора: управляющие регистры, GDTR, IDTR, регистр базового адреса вектора и регистры конкретных моделей	См. табл. 7.23	См. табл. 7.23
Исполняемый код, обратные вызовы и области данных в защищенном ядре и его зависимостях, включая сам механизм HyperGuard	Стандартные компоненты Windows, играющие ключевую роль в работе VTL 1 и безопасном использовании режима ядра	Модификация кода в этих компонентах означает, что нападающий получил доступ к некоторой уязвимости VTL 1 — либо через оборудование, либо через гипервизор. Может использоваться для нарушения работы Device Guard, HyperGuard и Credential Guard
Структуры и функциональные возможности, используемые трастлетами	Обмен данными между двумя трастлетами, или трастлетом и ядром, или трастлетами и VTL 0	Подразумевает, что уязвимость может существовать в одном или нескольких трастлетах, что может использоваться для подавления таких механизмов, как Credential Guard или Shielded Fabric/vTPM
Структуры и области гипервизора	Используется гипервизором для взаимодействия с VTL 1	Подразумевает наличие потенциальной уязвимости в компоненте VTL 1 или в самом гипервизоре, который может быть доступен из кольца 0 в VTL 0

Таблица 7.24 (*окончание*)

Компонент	Допустимое использование	Потенциальное вредоносное использование
Битовая карта CFG ядра	Используется для идентификации допустимых функций ядра, которые могут быть целью косвенных вызовов функций или переходов, как описано выше	Подразумевает, что атакующий может выполнить модификацию битовой карты KFCG с защитой VTL1 через оборудование или эксплойт гипервизора
Проверка страниц	Используется для реализации операций, относящихся к HVCI, для Device Guard	Подразумевает, что атакующий каким-то образом атаковал SKCI, что может привести к нарушению защиты Device Guard или неавторизованных трастлетов IUM
NULL-страница	Нет	Подразумевает, что атакующий каким-то образом заставит ядро и/или защищенное ядро выделить виртуальную страницу 0, которая может использоваться для эксплуатации уязвимостей NULL-страниц в VTL 0 или VTL 1

Наконец, следует отметить, что отключить PatchGuard или HyperGuard после их включения уже не удастся. Но поскольку у разработчиков драйверов устройств может возникнуть необходимость во внесении изменений в работающую систему в процессе отладки, PatchGuard не включается при загрузке системы в отладочном режиме с активным удаленным подключением отладки ядра. Аналогичным образом HyperGuard отключается при загрузке гипервизора в отладочном режиме с удаленным подключением отладчика.

Заключение

Windows предоставляет обширный набор функций безопасности, которые отвечают основным требованиям как государственных учреждений, так и коммерческих структур. В данной главе был дан краткий обзор внутренних компонентов, положенных в основу этих функций безопасности.

Марк Руссинович, Дэвид Соломон,
Алекс Ионеску, Павел Йосифович

Внутреннее устройство Windows

7-е издание

Перевел с английского *Е. Матвеев*

Заведующая редакцией	*Ю. Сергиенко*
Ведущий редактор	*К. Тульцева*
Литературный редактор	*И. Карпова*
Художественный редактор	*С. Заматевская*
Корректоры	*С. Беляева, И. Тимофеева*
Верстка	*Л. Егорова*

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 06.2018. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 —
Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01

Подписано в печать 20.06.18. Формат 70×100/16. Бумага офсетная. Усл. п. л. 76,110. Тираж 1200. Заказ 0000.

Отпечатано в ОАО «Первая Образцовая типография». Филиал «Чеховский Печатный Двор».
142300, Московская область, г. Чехов, ул. Полиграфистов, 1.
Сайт: www.chpk.ru. E-mail: marketing@chpk.ru
Факс: 8(496) 726-54-10, телефон: (495) 988-63-87

КНИГА-ПОЧТОЙ

ЗАКАЗАТЬ КНИГИ ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР» МОЖНО ЛЮБЫМ УДОБНЫМ ДЛЯ ВАС СПОСОБОМ:

- на нашем сайте: **www.piter.com**
- по электронной почте: **books@piter.com**
- по телефону: **(812) 703-73-74**

ВЫ МОЖЕТЕ ВЫБРАТЬ ЛЮБОЙ УДОБНЫЙ ДЛЯ ВАС СПОСОБ ОПЛАТЫ:

- Наложенным платежом с оплатой при получении в ближайшем почтовом отделении.
- С помощью банковской карты. Во время заказа вы будете перенаправлены на защищенный сервер нашего оператора, где сможете ввести свои данные для оплаты.
- Электронными деньгами. Мы принимаем к оплате Яндекс.Деньги, Webmoney и Kiwi-кошелек.
- В любом банке, распечатав квитанцию, которая формируется автоматически после совершения вами заказа.

ВЫ МОЖЕТЕ ВЫБРАТЬ ЛЮБОЙ УДОБНЫЙ ДЛЯ ВАС СПОСОБ ДОСТАВКИ:

- Посылки отправляются через «Почту России». Отработанная система позволяет нам организовывать доставку ваших покупок максимально быстро. Дату отправления вашей покупки и дату доставки вам сообщат по e-mail.
- Вы можете оформить курьерскую доставку своего заказа (более подробную информацию можно получить на нашем сайте www.piter.com).
- Можно оформить доставку заказа через почтоматы (адреса почтоматов можно узнать на нашем сайте www.piter.com).

ПРИ ОФОРМЛЕНИИ ЗАКАЗА УКАЖИТЕ:

- фамилию, имя, отчество, телефон, e-mail;
- почтовый индекс, регион, район, населенный пункт, улицу, дом, корпус, квартиру;
- название книги, автора, количество заказываемых экземпляров.

БЕСПЛАТНАЯ ДОСТАВКА:

- курьером по Москве и Санкт-Петербургу при заказе на сумму **от 2000 руб.**
- почтой России при предварительной оплате заказа на сумму **от 2000 руб.**

САЛД

САНКТ-ПЕТЕРБУРГСКАЯ
АНТИВИРУСНАЯ
ЛАБОРАТОРИЯ
ДАНИЛОВА

www.SALD.ru
8 (812) 336-3739

Антивирусные
программные продукты

КЛАССИКА COMPUTER SCIENCE

С момента выхода предыдущего издания этой книги операционная система Windows прошла длинный путь обновлений и концептуальных изменений, результатом которых стала новая стабильная архитектура ядра Windows 10.

Книга «Внутреннее устройство Windows» создана для профессионалов, желающих разобраться во внутренней жизни основных компонентов Windows 10. Опираясь на эту информацию, разработчикам будет проще находить правильные проектные решения, создавая приложения для платформы Windows, и решать сложные проблемы, связанные с их эксплуатацией. Системные администраторы, зная, что находится у операционной системы «под капотом», смогут разобраться с поведением системы и быстрее решать задачи повышения производительности и диагностики сбоев. Специалистам по безопасности пригодится информация о борьбе с уязвимостями операционной системы.

Прочитав эту книгу, вы будете лучше разбираться в работе Windows и в истинных причинах того или иного поведения ОС.

ISBN 978-5-4461-0663-9

youtube.com/ThePiterBooks

С

я!