

Microsoft®



ВНУТРЕННЕЕ УСТРОЙСТВО

Microsoft

Windows

Основные подсистемы ОС

6-е издание М. Руссинович, Д. Соломон, А. Ионеску





Mark Russinovich, David A. Solomon, Alex Ionescu

Windows Internals

6th Edition

Part 2

***Microsoft®
Press***

М. Руссинович, Д. Соломон, А. Ионеску

ВНУТРЕННЕЕ УСТРОЙСТВО

**Microsoft
Windows**

Основные подсистемы ОС

6-е издание



Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Ростов-на-Дону · Екатеринбург · Самара · Новосибирск
Киев · Харьков · Минск

2014

М. Руссинович, Д. Соломон, А. Ионеску

Внутреннее устройство Microsoft Windows. 6-е издание.

Основные подсистемы ОС

Серия «Мастер-класс»

Перевели с английского Н. Вильчинский, И. Рузмайкина

Заведующий редакцией	<i>А. Юрченко</i>
Руководитель проекта	<i>А. Юрченко</i>
Ведущий редактор	<i>Ю. Сергиенко</i>
Литературный редактор	<i>А. Жданов</i>
Художественный редактор	<i>Л. Адуевская</i>
Корректоры	<i>С. Беляева, В. Листова</i>
Верстка	<i>Л. Родионова</i>

ББК 32.973.2-018.2

УДК 004.451

М. Руссинович, Д. Соломон, А. Ионеску

P89 Внутреннее устройство Microsoft Windows. 6-е изд. Основные подсистемы ОС. — СПб.: Питер, 2014. — 672 с.: ил. — (Серия «Мастер-класс»).

ISBN 978-5-496-00791-7

Шестое издание этой легендарной книги посвящено внутреннему устройству и алгоритмам работы основных компонентов операционной системы Microsoft Windows 7, а также Windows Server 2008 R2. Вторая часть книги охватывает основные подсистемы Windows: ввод-вывод, хранение данных, управление памятью, диспетчер кэша и файловые системы. Рассмотрены процессы запуска и завершения работы и дано описание анализа аварийного дампа. Книга предназначена для системных администраторов, разработчиков сложных приложений и всех, кто хочет понять, как устроена операционная система Windows.

12+ (Для детей старше 12 лет. В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ISBN 978-0735665873 англ.

© Authorized Russian translation of the English edition of Windows Internals, 6th Edition. Part 2 © 2012 by David A. Solomon and Mark Russinovich (ISBN 9780735665873). This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

ISBN 978-5-496-00791-7

© Перевод на русский язык ООО Издательство «Питер», 2014
© Издание на русском языке, оформление
ООО Издательство «Питер», 2014

Права на издание получены по соглашению с O'Reilly Media, Inc. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав. Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги.

ООО «Питер Пресс», 192102, Санкт-Петербург, ул. Андреевская (д. Волкова), д. 3, литер А, пом. 7Н.
Налоговая льгота — общероссийский классификатор продукции ОК 005-93, том 2; 95 3005 — литература учебная.
Подписано в печать 24.03.14. Формат 70x100/16. Усл. п. л. 54,180. Тираж 1200. Заказ
Отпечатано в полном соответствии с качеством предоставленных издательством материалов
в Чеховский Печатный Двор. 142300, Чехов, Московская область, г. Чехов, ул. Полиграфистов, д. 1.

Уважаемые читатели! Не удивляйтесь, что эта книга начинается

с восьмой главы. Авторы разделили свой труд на две части.

Первая часть книги вышла в издательстве «Питер» в 2013 году.

В ней рассмотрены следующие темы:

Глава 1. Общие представления и инструментальные средства

Глава 2. Архитектура системы

Глава 3. Системные механизмы

Глава 4. Механизмы управления

Глава 5. Процессы, потоки и задания

Глава 6. Безопасность

Глава 7. Сеть

Оглавление

Введение	15
Глава 8. Подсистема ввода-вывода	21
Компоненты подсистемы ввода-вывода	21
Диспетчер ввода-вывода	24
Стандартная обработка ввода-вывода	24
Драйверы устройств	26
Типы драйверов устройств	26
WDM-драйверы	27
Многоуровневые драйверы	27
Структура драйвера	32
Объекты драйверов и устройств	35
Открытие устройств	40
Обработка ввода-вывода	47
Типы ввода-вывода	47
Синхронный и асинхронный ввод-вывод	47
Быстрый ввод-вывод	48
Ввод-вывод для файлов, отображенных на память, и кэширование файлов	49
Фрагментированный ввод-вывод	50
Пакеты запросов на ввод и вывод	50
Блоки стека IRP-пакетов	52
Управление буфером IRP-пакетов	54

Запрос ввода-вывода к одноуровневому драйверу	55
Обработка прерывания	57
Завершение обработки запроса на ввод-вывод	59
Синхронизация	61
Запросы ввода-вывода к многоуровневым драйверам	62
Независимый от программных потоков ввод-вывод	69
Отмена ввода-вывода	70
Отмена ввода-вывода, инициированная пользователем	71
Отмена ввода-вывода при завершении программного потока	72
Порты завершения ввода-вывода	74
Объект IoCompletion	75
Применение портов завершения	75
Функционирование порта ввода-вывода	77
Определение приоритетов ввода-вывода	80
Приоритеты ввода-вывода	80
Стратегии выбора приоритета	80
Предотвращение инверсии приоритетов ввода-вывода (наследование приоритетов ввода-вывода)	83
Повышение и понижение приоритетов ввода-вывода	84
Резервирование полосы пропускания (планирование файлового ввода-вывода)	86
Уведомления о сеансах	87
Программа Driver Verifier	87
Среда KMDF	90
Структура и функциональность KMDF-драйвера	90
Модель данных в KMDF	92
Модель ввода-вывода в KMDF	97
Среда UMDF	100
PnP-диспетчер	104
Уровень поддержки технологии Plug and Play	105
Поддержка технологии Plug and Play со стороны драйвера	105
Загрузка, инициализация и установка драйвера	107
Параметр Start	108
Перечисление устройств	109
Стеки устройств	113
Загрузка драйверов для стека устройств	114
Установка драйвера	119
Диспетчер электропитания	123
Работа диспетчера электропитания	125
Участие драйверов в управлении электропитанием	126
Управление электропитанием устройств со стороны драйверов и приложений	130
Запросы на изменение режима электропитания	130
Управление электропитанием со стороны центрального процессора	133
Политики парковки ядер	134
Функция полезности	135
Переопределение алгоритма	138
Увеличение/уменьшение числа запаркованных ядер	139

Пороговые значения и варианты настройки политик.	139
Проверка производительности	143
Заключение.	149
Глава 9. Управление внешней памятью	151
Базовая терминология.	151
Дисковые устройства	152
Вращающиеся магнитные диски	152
Формат сектора диска	152
Твердотельные диски	154
Флэш-память типа NAND	155
Удаление файлов и команда Trim	156
Драйверы дисков	158
Программа Winload	158
Драйверы дисковых класса, порта и мини-порта	159
iSCSI-драйверы	160
MPIO-драйверы	161
Объекты устройств для дисков	163
Диспетчер разделов	165
Управление томами.	166
Базовые диски.	166
Схема MBR	166
Схема GPT	167
Диспетчер томов на базовых дисках	168
Динамические диски	169
База данных для LDM	169
Разбиение на разделы в стиле LDM и GPT или в стиле MBR	173
Диспетчер томов для динамических дисков.	174
Управление составными томами	175
Перекрытые тома	175
Чередующиеся тома	176
Зеркальные тома	177
RAID-5.	179
Пространство имен томов	180
Диспетчер монтирования	181
Точки монтирования	182
Монтирование томов	183
Ввод и вывод на томах	187
Служба виртуальных дисков	188
Поддержка виртуального жесткого диска.	190
Присоединение виртуальных жестких дисков	191
Вложенные файловые системы..	192
Шифрование диска BitLocker.	192
Ключи шифрования.	194
Доверенный платформенный модуль.	197
Процесс загрузки BitLocker	200

Восстановление с помощью BitLocker	201
Драйвер шифрования всего тома	202
Управление системой BitLocker	204
Технология BitLocker To Go	205
Служба теневого копирования томов	207
Теневые копии	207
Полные теневые копии	207
Разностные теневые копии	207
Архитектура VSS	208
Функционирование VSS	208
Провайдер теневого копирования	210
Применение в Windows	212
Резервное копирование	212
Предыдущие версии и восстановление системы	213
Заключение	216
Глава 10. Управление внутренней памятью	217
Знакомство с диспетчером памяти	217
Компоненты диспетчера памяти	218
Внутренняя синхронизация	219
Исследование использования памяти	220
Службы диспетчера памяти	224
Большие и малые страницы	224
Резервирование и подтверждение страниц	227
Лимит подтверждения	230
Блокирование памяти	231
Гранулярность выделения памяти	231
Совместно используемая память и отображаемые файлы	232
Защита памяти	235
Защита страниц от выполнения	237
Программное предотвращение выполнения кода	242
Копирование при записи	244
Оконные расширения адресов	245
Кучи режима ядра	248
Размеры пулов	249
Мониторинг использования пулов	251
Ассоциативные списки	255
Диспетчер кучи	257
Типы куч	257
Структура диспетчера кучи	258
Синхронизация кучи	259
Слабо фрагментированная куча	260
Механизмы безопасности куч	261
Средства отладки куч	262
Инструмент pageheap	263
Отказоустойчивая куча	264
Структуры виртуального адресного пространства	265

Структура адресных пространств на платформе x86	267
Структура системного адресного пространства на платформе x86	270
Пространство сеанса на платформе x86	270
Записи системной таблицы страниц	273
Структура адресных пространств 64-разрядных систем	274
Ограничения виртуальной адресации на платформе x64	278
16-терабайтное ограничение для Windows на платформе x64	278
Динамическое управление системным виртуальным адресным пространством	281
Квоты системного виртуального адресного пространства	284
Структура пользовательского адресного пространства	286
Рандомизация образа	288
Рандомизация стека	290
Рандомизация кучи	290
ASLR в адресном пространстве ядра	290
Управление средствами смягчения уровня опасности	290
Преобразование адресов	292
Преобразование виртуальных адресов на платформе x86	292
Каталоги страниц	296
Таблицы страниц и их записи	297
Сравнение аппаратного и программного битов записи	299
Байт внутри страницы	300
Буфер быстрого преобразования адресов	300
Расширение физических адресов	302
Преобразование виртуальных адресов на платформе x64	306
Преобразование виртуальных адресов на платформе IA64	308
Обработка ошибок отсутствия страниц	309
PTE-записи	310
Прототипные PTE-записи	312
Страницочный ввод-вывод	315
Конфликтные ошибки отсутствия страниц	315
Кластерные ошибки отсутствия страниц	316
Страницочные файлы	318
Показатель подтверждения и системный лимит подтверждения	319
Показатель подтверждения и размер страницочного файла	323
Стеки	325
Пользовательские стеки	326
Стеки ядра	327
DPC-стек	328
Дескрипторы виртуальных адресов	328
Дескрипторы виртуальных адресов процесса	329
Чередующиеся дескрипторы виртуальных адресов	331
NUMA	331
Объекты разделов	333
Программа проверки драйверов	340
База данных номеров страницочных блоков	345
Динамика списков страниц	349

Приоритеты страниц	359
Подсистема записи измененных страниц	362
Структура данных PFN-записи	364
Лимиты физической памяти	370
Лимиты памяти клиентских версий Windows	371
Фактические лимиты памяти на 32-разрядных клиентских системах	372
Рабочие наборы	375
Подкачка по требованию	375
Компонент логической предвыборки	376
Политика размещения	380
Управление рабочими наборами	381
Диспетчер настройки баланса и поток подкачки	385
Системные рабочие наборы	386
События уведомлений в памяти	387
Упреждающее управление памятью (супервыборка)	390
Компоненты	390
Трассировка и протоколирование	393
Сценарии	394
Приоритеты страниц и перебалансировка	395
Устойчивое функционирование	397
Служба ReadyBoost	400
Технология ReadyDrive	402
Унифицированное кэширование	402
Отражение процессов	405
Заключение	409
Глава 11. Диспетчер кэша	410
Основные возможности диспетчера кэша	410
Единый централизованный системный кэш	411
Диспетчер памяти	411
Согласованность кэша	412
Кэширование виртуальных блоков	413
Кэширование на основе потоков данных	414
Поддержка самовосстанавливающихся файловых систем	414
Управления виртуальной памятью кэша	415
Размер кэша	417
Виртуальный размер кэша	417
Размер рабочего набора кэша	418
Физический размер кэша	419
Структуры данных кэша	421
Общесистемные структуры данных кэша	422
Структуры данных кэша, относящиеся к каждому файлу	425
Интерфейсы файловых систем	431
Копирование в кэш и из кэша	432
Кэширование через интерфейсы отображения и фиксации	432
Кэширование через интерфейсы прямого доступа к памяти	433

Быстрый ввод-вывод	433
Упреждающее чтение и отложенная запись	435
Интеллектуальное упреждающее чтение	436
Кэширование с обратной записью и отложенная запись	437
Отключение режима отложенной записи для файла	445
Принудительное включение в кэше режима сквозной записи на диск	445
Сброс отображаемых файлов	445
Ограничение записи	446
Системные программные потоки	448
Заключение	449
Глава 12. Файловые системы	450
Форматы файловых систем в Windows	451
CDFS	451
UDF	452
FAT12, FAT16 и FAT32	452
exFAT	456
NTFS	456
Архитектура драйверов файловой системы	457
Локальные FSD-драйверы	458
Удаленные FSD-драйверы	459
Блокировка	461
Работа файловой системы	467
Явный ввод-вывод	468
Подсистема записи модифицированных и отображенных страниц	472
Подсистема отложенной записи	473
Программный поток опережающего чтения	473
Обработчик ошибок страниц	474
Фильтрующие драйверы файловой системы	474
Программа Process Monitor	474
Решение проблем файловой системы	476
Базовый и расширенный режимы программы Process Monitor	476
Устранение неисправностей с помощью Process Monitor	477
Файловая система с типовым протоколированием	478
Маршалирование	478
Типы журналов	479
Структура журнала	481
Регистрационные номера транзакций в журнале	482
Блоки журнала	483
Страницы владельца	484
Преобразование виртуальных LSN-номеров в физические	485
Политики управления	486
Цели разработки и особенности NTFS	487
Требования к профессиональной файловой системе	487
Восстанавливаемость	487
Безопасность	487

Избыточность данных и отказоустойчивость	488
Нетривиальные возможности NTFS	488
Множественные потоки данных	489
Имена на базе Unicode	491
Универсальный механизм индексации	491
Динамическое переназначение поврежденных кластеров	492
Жесткие ссылки	492
Символические (мягкие) ссылки и соединения	493
Сжатие и разреженные файлы	495
Протоколирование изменений	496
Квоты томов для пользователей	496
Отслеживание связей	498
Шифрование	498
Поддержка POSIX	499
Дефрагментация	499
Динамическое разбиение на разделы	501
Драйвер файловой системы NTFS	502
NTFS-структура на диске	505
Тома	505
Кластеры	506
Главная таблица файлов	507
Индексы файловых записей	511
Файловые записи	511
Имена файлов	514
Резидентные и нерезидентные атрибуты	518
Сжатие данных и разреженные файлы	521
Сжатие разреженных данных	522
Сжатие неразреженных данных	524
Разреженные файлы	526
Файл журнала изменений	526
Индексация	529
Идентификаторы объектов	531
Отслеживание квот	531
Консолидированная система безопасности	533
Точки повторной обработки	535
Поддержка транзакций	535
Изоляция	536
Транзакционные API-интерфейсы	538
Диспетчеры ресурсов	539
Реализация на диске	541
Реализация протоколирования	543
Реализация восстановления	543
Поддержка восстановления в NTFS	544
Техническое решение	545
Протоколирование метаданных	546
Служба файла журнала	546

Типы записей журнала	548
Восстановление	551
Анализ	551
Повторение	552
Отмена	553
Восстановление поврежденных кластеров в NTFS	555
Самовосстановление	559
Безопасность в шифрующей файловой системе	560
Первое шифрование файла	563
Шифрование файловых данных	564
Процесс дешифрирования	565
Резервное копирование шифрованных файлов	566
Копирование зашифрованных файлов	567
Заключение	567
Глава 13. Запуск и завершение работы системы	568
Процесс загрузки	568
Начальные этапы загрузки систем на базе BIOS	568
Загрузочный сектор систем на базе BIOS и Bootmgr	572
Загрузка в UEFI-системах	587
Загрузка с iSCSI-устройств	588
Инициализация ядра и исполнительных подсистем	589
Smss, Csrss и Wininit	597
ReadyBoot	603
Автоматически запускаемые образы	603
Анализ проблем при загрузке и запуске системы	605
Последняя удачная конфигурация	605
Безопасный режим	605
Загрузка драйверов в безопасном режиме	606
Программы с поддержкой безопасного режима	608
Протоколирование загрузки в безопасном режиме	608
Среда восстановления Windows	609
Решение распространенных проблем загрузки	613
Повреждение MBR	613
Повреждение загрузочного сектора	614
Неправильная конфигурация BCD	614
Повреждение системных файлов	615
Повреждение куста System	617
Сбой или зависание после вывода экранной заставки	617
Завершение работы	619
Заключение	622
Глава 14. Анализ аварийного дампа	623
Почему в Windows случаются сбои?	623
Синий экран	624
Причины сбоев в Windows	625

Устранение проблем при сбоях	627
Файлы аварийного дампа	629
Генерация аварийного дампа	635
Передача в Microsoft отчетов об ошибках.	638
Анализ сбоев через Интернет	639
Базовый анализ аварийного дампа	640
Программа Notmyfault	641
Базовый анализ	642
Детальный анализ	643
Инструменты устранения сбоев	645
Переполнение буфера, повреждение памяти и особый пул	646
Перезапись кода и защита системного кода от записи	649
Углубленный анализ аварийных дампов	651
Засорение стека	652
Зависание, или отсутствие отклика	654
Если аварийный дамп отсутствует	659
Анализ распространенных стоп-кодов	662
Код 0xD1 – DRIVER_IRQL_NOT_LESS_OR_EQUAL	662
Код 0x8E – KERNEL_MODE_EXCEPTION_NOT_HANDLED	664
Код 0x7F – UNEXPECTED_KERNEL_MODE_TRAP	665
Код 0xC5 – DRIVER_CORRUPTED_EXPOOL	667
Отказы аппаратуры	670
Заключение	671
Об авторах	672

Введение

Шестое издание этой книги предназначено для специалистов в области ИТ (разработчиков и системных администраторов), желающих разобраться в принципах функционирования ключевых компонентов операционных систем Microsoft Windows 7 и Windows Server 2008 R2. Эта информация поможет лучше понять обоснование проектных решений при разработке приложений для платформы Windows. Также она будет полезна системным администраторам и разработчикам, занимающимся отладкой сложных проблем, — ведь понимание принципов работы системы «изнутри» дает возможность оценить ее поведение с точки зрения производительности и упрощает устранение неполадок. После прочтения этой книги вы начнете намного лучше понимать, как работает операционная система Windows и почему она ведет себя так, а не иначе.

Структура книги

Впервые книга оказалась разделенной на две части. Это сделано для того, чтобы вы быстрее могли получить информацию, ведь обновление всей книги при выходе очередной версии Windows требует значительного времени.

Первая часть¹ начинается с двух глав, в которых вводятся ключевые концепции, рассказывается об используемых в книге инструментах, описываются общая архитектура и компоненты системы. Следующие две главы посвящены основополагающим системным механизмам и механизмам управления. Завершается первая часть рассмотрением трех ключевых компонентов операционной системы: во-первых, это процессы, программные потоки и задания, во-вторых, безопасность, и в-третьих, работа в сети.

Остальные ключевые подсистемы, к которым относятся механизмы ввода-вывода, долговременного хранения и управления памятью, а также диспетчер кэша и файловые системы, рассматриваются во второй части. Завершает вторую часть описание процессов запуска и остановки операционной системы, а также средств анализа аварийного дампа.

Предыстория

Это шестое издание книги, которая изначально называлась «Inside Windows NT» (Microsoft Press, 1992) и была написана Хелен Кастер (Helen Custer) еще до выхода Microsoft Windows NT 3.1. Она стала первой книгой по Windows NT и представляла

¹ Руцкинович М., Соломон Д. Внутреннее устройство Microsoft Windows. 6-е изд. СПб.: Питер, 2013. Далее мы будем называть эту книгу просто часть I. — Примеч. ред.

собой подробный обзор архитектуры этой системы. Второе издание, «Inside Windows NT» (Microsoft Press, 1998), было написано Дэвидом Соломоном. В него вошла новая информация о Windows NT 4.0, а сам материал книги стал гораздо глубже.

Третье издание, «Inside Windows 2000» (Microsoft Press, 2000), было подготовлено Дэвидом Соломоном и Марком Руссиновичем. В нем появилось много новых тематических разделов, в числе прочего посвященных этапам загрузки и завершения работы системы, внутреннему устройству служб и реестра, драйверам файловой системы и сетевой поддержке. Также были рассмотрены новые функциональные возможности Windows 2000, в том числе модель драйверов Windows (WDM), технология Plug and Play, управление электропитанием, инструментарий управления Windows (WMI), шифрование, объекты заданий и терминальные службы. Четвертое издание, вышедшее уже под названием «Windows Internals» (в русском переводе — «Внутреннее устройство Windows», издательство «Питер», 2008), было переработано и дополнено, причем основной упор был сделан на информации, помогающей IT-специалистам использовать свои знания внутреннего устройства Windows. К примеру, подробно рассматривались программные инструменты, созданные в Windows Sysinternals (www.microsoft.com/technet/sysinternals), и приемы анализа аварийных дампов. Пятое издание, «Windows Internals», было связано с обновлением Windows Vista и Windows Server 2008. В книгу вошли новые материалы, посвященные загрузчику изображений, механизму отладки пользовательского режима и средству виртуализации Hyper-V.

Изменения в шестом издании

В шестом издании рассказывается об изменениях в ядре операционных систем Windows 7 и Windows Server 2008 R2. В соответствии с изменившимся инструментарием обновлены практические эксперименты.

Эксперименты

Даже без доступа к исходному коду существующие инструменты, такие как отладчик ядра, а также служебные программы, разработанные в Sysinternals и Winsider Seminars & Solutions, позволяют многое прояснить во внутреннем устройстве Windows. В том месте, где для демонстрации какого-либо аспекта поведения Windows используется та или иная служебная программа, во врезке «Эксперимент» даются инструкции по ее применению. Такие врезки часто встречаются в книге, и мы рекомендуем вам в процессе чтения выполнять все эксперименты: наглядно увидев, как ведет себя Windows в конкретной ситуации, вы гораздо лучше усвоите прочитанный материал.

Что не вошло в книгу

Windows — большая и сложная операционная система. Нельзя объять необъятное, и поэтому основное внимание в книге уделяется только базовым системным компонентам.

Например, мы не рассматриваем ни модель COM+, ни инфраструктуру объектно-ориентированного программирования распределенных приложений для Windows, ни платформу .NET Framework, ни платформу для приложений с управляемым кодом.

Поскольку наша книга рассказывает о внутреннем устройстве Windows, а не о том, как пользоваться этой операционной системой, программировать для нее или администрировать системы, созданные на ее основе, вы не найдете здесь никаких сведений об использовании, программировании и конфигурировании Windows.

Предупреждение

В книге описываются недокументированные внутренние структуры и функции ядра, архитектура и различные аспекты внутренней работы Windows, однако указанные структуры и функции могут отчасти измениться в следующем выпуске этой операционной системы. (Впрочем, внешние интерфейсы вроде Windows API всегда сохраняют совместимость с аналогичными интерфейсами новых версий.)

Говоря «могут измениться», мы не имеем в виду, что детали устройства системы обязательно изменятся в следующем выпуске, а лишь обращаем внимание на то, что достоверность информации гарантируется исключительно для описываемых версий. Любое программное обеспечение с недокументированными интерфейсами может перестать работать в будущих версиях Windows. Более того, такое программное обеспечение, если оно функционирует в режиме ядра (как, например, драйверы устройств), может привести к неработоспособности новых версий Windows.

Благодарности

В первую очередь хотелось бы поблагодарить Джейми Ханрахан (Jamie Hanrahan) и Брайана Катлина (Brian Catlin) из Azius, LLC за участие в нашем проекте — без их помощи эта книга вряд ли была бы закончена. Именно они внесли основной вклад в обновление глав, посвященных безопасности и работе в сети, а также помогли в работе над главами, описывающими механизмы управления, процессы и программные потоки. Компания Azius подготовила учебный курс по внутреннему устройству Windows и драйверам устройств. Подробно об этом можно узнать на сайте www.azius.com.

Хотим также выразить признательность Алексу Ионеску (Alex Ionescu), ставшему полноправным соавтором этого издания. Он интенсивно работал с нами и над предыдущей версией книги.

Спасибо Дэниэлю Пирсону (Daniel Pearson), обновившему главу об анализе аварийного дампа. Его многолетний опыт работы в этой области позволил предоставить массу полезной с практической точки зрения информации.

Благодарим Эрика Траута (Eric Traut) и Джона ДеВана (Jon DeVaan) за представление Дэвиду Соломуна доступа к внутреннему коду Windows и за разработку курсов по внутреннему устройству Windows, которые он ведет.

Мы не поблагодарили трех основных рецензентов, Аруна Кишана (Arun Kishan), Лэнди Ванг (Landy Wang) и Аарона Маргосиса (Aaron Margosis), за их рецензии и вклад

в предыдущее издание. Еще раз большое вам всем спасибо! Особая благодарность Аруну и Лэнди за их подробные рецензии и помощь при подготовке данного издания.

Мы не смогли бы достичь такой глубины и точности изложения технических сведений без поддержки, замечаний и предложений ключевых членов команды разработчиков Microsoft Windows. Поэтому мы хотели бы поблагодарить за технические обзоры и вклад в книгу многих людей, среди которых:

- ❑ Грег Коттингэм (Greg Cottingham);
- ❑ Джо Хамбург (Joe Hamburg);
- ❑ Джэфф Ламберт (Jeff Lambert);
- ❑ Павел Лебедински (Pavel Lebedinsky);
- ❑ Джозеф Ист (Joseph East);
- ❑ Эди Олтин (Adi Oltean);
- ❑ Алексей Пахунов (Alexey Pakhunov);
- ❑ Валери Си (Valerie See);
- ❑ Брэд Уотерс (Brad Waters);
- ❑ Брюс Вортигтон (Bruce Worthington);
- ❑ Робин Александер (Robin Alexander);
- ❑ Бернард Урганлян (Bernard Ourghanlian).

Спасибо Скотту Ли (Scott Lee), Тому Шульцу (Tim Shoultz) и Эрику Кратцеру (Eric Kratzer) за помощь в подготовке главы об анализе аварийного дампа.

А за главу о работе в сети хотелось бы отдельно поблагодарить Джанлуиджи Нуска (Gianluigi Nusca) и Тома Джолли (Tom Jolly), помощь которых выходила за рамки их служебных обязанностей. Джанлуиджи, спасибо тебе за колоссальный вклад в материал о технологии BranchCache и за ряд предложений (а также за многочисленные написанные лично тобой абзацы). Том, мы благодарим тебя не только за обзоры и предложения (отличного качества), но и за получение рецензий от множества других разработчиков. Вот те, кто внес свой вклад в главу о работе в сети:

- ❑ Рупеш Баттерати (Roopesh Battepati);
- ❑ Молли Браун (Molly Brown);
- ❑ Грег Коттингем (Greg Cottingham);
- ❑ Дотан Елхаррап (Dotan Elharrar);
- ❑ Эрик Хансон (Eric Hanson);
- ❑ Том Джолли (Tom Jolly);
- ❑ Маной Кадам (Manoj Kadam);
- ❑ Грег Крамер (Greg Kramer);
- ❑ Давид Круз (David Kruse);

- Джефф Ламберт (Jeff Lambert);
- Дарен Льюис (Darene Lewis);
- Дан Ловингер (Dan Lovering);
- Джанлуиджи Нуска (Gianluigi Nusca);
- Амос Ортал (Amos Ortal);
- Иван Пашов (Ivan Pashov);
- Ганеш Прасад (Ganesh Prasad);
- Пауль Сван (Paul Swan);
- Шива Кумар Тхангапанди (Shiva Kumar Thangapandi).

Амос Ортал и Дотан Элхаррар оказали значительную помощь в освещении технологии NAP, а Шива Кумар Тхангапанди сильно помог с описанием инфраструктуры EAP.

Спасибо Жерару Мерфи (Gerard Murphy) за обзор механизмов остановки работы Windows 7 и пояснения по поводу групповой политики.

Благодарим Тристана Брауна (Tristan Brown) из группы Power Management Microsoft за часы, проведенные в офисе с Алексом за разбором алгоритмов и вариантов поведения при парковке ядер, а также за предоставленную им огромную диаграмму.

Спасибо Апурве Доши (Apurva Doshi) за присланную им Алексу подробную документацию по изменениям, которые произошли в Windows 7 с диспетчером кэша. Именно они позволили зафиксировать ряд новых вариантов поведения, описание которых вошло в данную книгу.

Благодарим Мэттью Сюиша (Matthieu Suiche) за его базу данных символов ядра, позволившую Алексу обнаружить в ключевых структурах данных ядра новые и удаленные поля и, как следствие, выявить изменения в основной функциональности.

Спасибо Дженку Эргану (Cenk Ergan), Майклу Фортину (Michel Fortin) и Мехмету Айигану (Mehmet Iyigun) за их обзоры и вклад в освещение механизма супервыборки.

Скрупулезная проверка, проведенная Кристофом Назаром (Christophe Nasarre), нашим техническим редактором, оказала большое влияние на точность предоставленных в книге сведений и на целостность материала.

И еще хотелось бы поблагодарить Ильфака Гульфанова (Ilfak Guifanov) из Hex-Rays (www.hex-rays.com) за предоставленные им Алексу лицензии по IDA Pro Advanced и Hex-Rays, ускорившие выполненную им декомпиляцию ядра Windows.

Наконец, авторы хотели бы сказать спасибо сотрудникам издательства Microsoft Press, благодаря которым эта книга увидела свет. Девон Масгрейв (Devon Musgrave) выполнял обязанности рецензента издательства и научного редактора, в то время как Кэрол Диллингем (Carol Dillingham) курировала нас в качестве руководителя проекта. Большой вклад в обеспечение качества данной книги внесли главный редактор Кертис Филипс (Curtis Phillips), литературный редактор Джон Пирс (John Pierce), корректор Андреа Фокс (Andrea Fox).

Последнюю по месту, но не по важности благодарность хотелось бы выразить Бену Райану (Ben Ryan), издателю Microsoft Press, который до сих пор уверен в важности предоставления читателям столь подробных сведений об операционной системе Windows!

Техническая поддержка

Мы приложили максимум усилий, чтобы не допустить неточностей и ошибок в книге. Все ошибки, о которых было сообщено с момента выхода книги, опубликованы на сайте Microsoft Press:

<http://go.microsoft.com/fwlink/?LinkId=258649>

От издательства

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

Мы будем рады узнать ваше мнение!

На веб-сайте издательства <http://www.piter.com> вы найдете подробную информацию о наших книгах.

Глава 8¹. Подсистема ввода-вывода

Подсистема ввода-вывода в Windows состоит из набора компонентов исполнительной системы, которые совместно управляют устройствами и предоставляют этим устройствам интерфейсы для приложений и системы. 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. Завершает главу сведения о способах 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:

- Стандартные средства безопасности и именования устройств предназначены для защиты общих ресурсов (политики безопасности Windows описываются в главе 6 части I).
- 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.

¹ Уважаемые читатели! Не удивляйтесь, что эта книга начинается с восьмой главы. Авторы разделили свой труд на две части. Первая часть книги вышла в издательстве «Питер» в 2013 году.

- ❑ Подсистема управления электропитанием позволяет системе или отдельным устройствам переходить в состояния с низким энергопотреблением.
- ❑ Поддерживается установка различных файловых систем, в том числе FAT, CDFS, UDF и NTFS (типы и архитектура файловых систем подробно рассматриваются в главе 12).
- ❑ Благодаря поддержке технологии WMI (Windows Management Instrumentation – инструментарий управления Windows) и средств диагностики управление драйверами и текущий контроль осуществляются при помощи WMI-приложений и WMI-сценариев (подробно технология WMI рассматривается в главе 4 части I). Для реализации этой функциональности в подсистеме ввода-вывода Windows предусмотрен ряд компонентов и драйверов устройств (рис. 8.1).

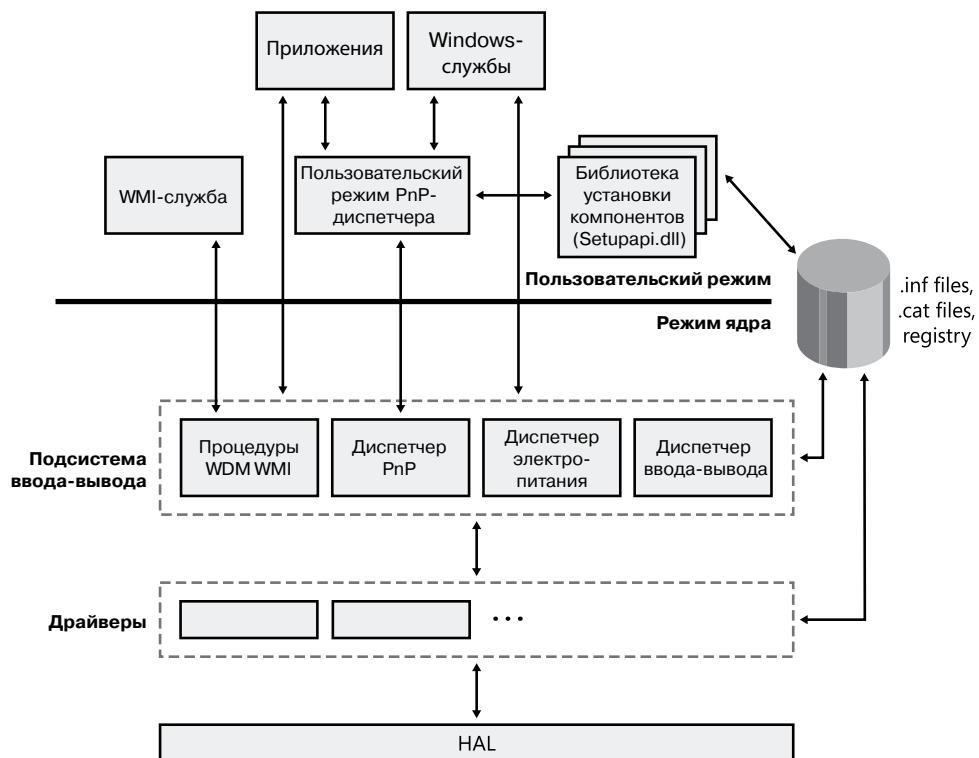


Рис. 8.1. Компоненты подсистемы ввода-вывода

- ❑ Сердцем подсистемы ввода-вывода является одноименный диспетчер; он соединяет приложения и системные компоненты с виртуальными, логическими и физическими устройствами, создает поддерживающую драйверы устройств инфраструктуру.

- ❑ Драйвер устройства, как правило, обеспечивает конкретные устройства интерфейсом ввода-вывода. Он представляет собой программный модуль, интерпретирующий высокоуровневые команды, такие как `read` или `write`, 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 службы 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 провайдер WDM WMI (подробно технология WMI рассматривается в соответствующем разделе главы 4 части I).
- ❑ Реестр представляет собой базу данных с описанием основных подключенных к подсистеме устройств, а также параметров инициализации драйверов и конфигурации (подробно они рассматриваются в главе 4 части I).
- ❑ Файлы установки драйверов (с расширением .inf) связывают аппаратные устройства с драйверами, управляющими этими устройствами. Содержимое такого файла состоит из напоминающих сценарий инструкций, описывающих 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 контролируются другими драйверами.

Диспетчер ввода-вывода

Центральным элементом подсистемы ввода-вывода является *диспетчер ввода-вывода* (I/O manager), задающий инфраструктуру 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 позволяетциальному 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 на ввод и вывод» далее в этой главе).

Диспетчер ввода-вывода представляет операции ввода и вывода в памяти в виде IRP-пакетов. При этом он передает указатель на IRP нужному драйверу и после завершения операции удаляет пакет. А 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 время ожидания для драйверов и регистрирует загруженные в операционную систему устанавливаемые файловые системы. Диспетчер ввода-вывода содержит почти сотню процедур, которые могут вызываться драйверами устройств.

Также диспетчер ввода-вывода предоставляет гибкие службы ввода-вывода, позволяющие подсистемам окружения (например, Windows и POSIX) реализовывать соответствующие функции. В частности, сюда относятся тщательно разработанные службы асинхронного ввода-вывода, предоставляющие разработчикам возможность создавать высокопроизводительные масштабируемые серверные приложения.

Единый 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 усовершенствованных механизмов, 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 операции ввода и вывода программные потоки выполняют с виртуальными файлами. Термин «виртуальный файл» относится к любому источнику или приемнику запроса на ввод-вывод, который рассматривается как файл (это может быть файл, папка, канал или почтовая ячейка). Операционная система рассматривает все запросы на ввод и вывод как операции над виртуальным файлом, так как диспетчер ввода-вывода ни с чем другим работать не умеет. При этом за преобразование файловых команд (`open`, `close`, `read`, `write`) в команды для конкретного устройства отвечает драйвер. Подобная абстракция обеспечивает единый программный интерфейс для всех устройств. Приложения в режиме пользователя (как в Windows, так и в POSIX) 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 драйверам соответствующих устройств. Рисунок 8.2 демонстрирует базовую схему обработки запроса на ввод-вывод.

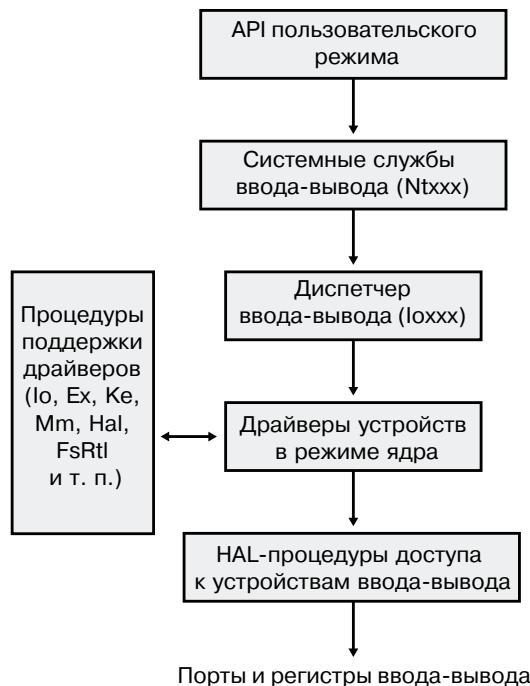


Рис. 8.2. Схема обработки типичного запроса на ввод-вывод

Далее мы детально рассмотрим эти компоненты, 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.

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.

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 несколько типов драйверов пользовательского режима:

- ❑ Драйверы принтеров в 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 аппаратным обеспечением на уровне 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-устройств* (Plug and Play drivers) работают с аппаратным обеспечением и объединены с диспетчером электропитания и PnP-диспетчером. В эту категорию входят драйверы запоминающих устройств, видеоадаптеров, устройств ввода и сетевых адаптеров.
- ❑ *Драйверы устройств, не поддерживающих технологию 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 при-

кладного программного интерфейса, а также описанный в главе 4 части I драйвер Process Monitor.

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.
- *Функциональные драйверы* (function drivers) управляют устройствами конкретного типа. Драйверы шины представляют устройства функциональным драйверам через PnP-диспетчера. 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 при помощи фильтрующего драйвера клавиатуры, работающего поверх функционального драйвера клавиатуры.

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 стандартизованными 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.



Рис. 8.3. Многоуровневое представление драйвера файловой системы и драйвера диска

- *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», а Dxgport/Videoprt – драйвером «порта» DirectX/video.
- *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.

Проиллюстрируем работу драйверов устройств на высшем уровне на примере, упрощенном для демонстрационных целей. Драйвер файловой системы принимает запрос на запись данных в определенное место конкретного файла. Его преобразуют в запрос на запись определенного числа 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. Эта схема показана на рис. 8.3.

Схема иллюстрирует разделение обязанностей между драйверами. Диспетчер ввода-вывода получает запрос на запись, связанный с началом 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 может быть добавлен еще один драйвер без изменения существующих драйверов или подсистемы ввода-вывода. Этот драйвер, к примеру, может превратить несколько дисков в один большой диск. Такой логический драйвер диспетчера томов располагается между драйвером файловой системы и драйвером диска, как показано на диаграмме, представленной на рис. 8.4. (Реальная диаграмма стека драйверов устройств показана на рис. 9.3 в главе 9.) Драйверы диспетчера томов подробно описываются в главе 9.

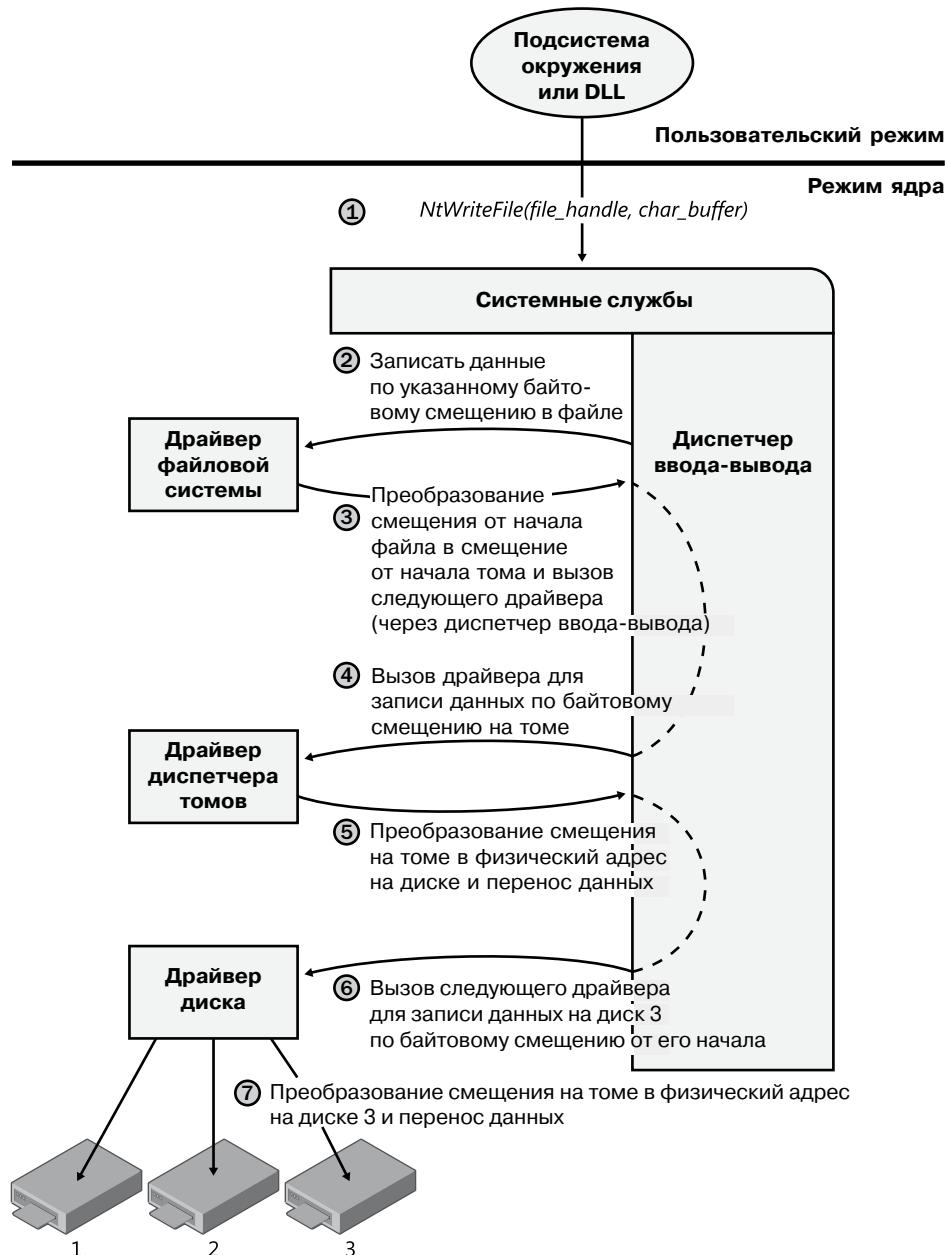
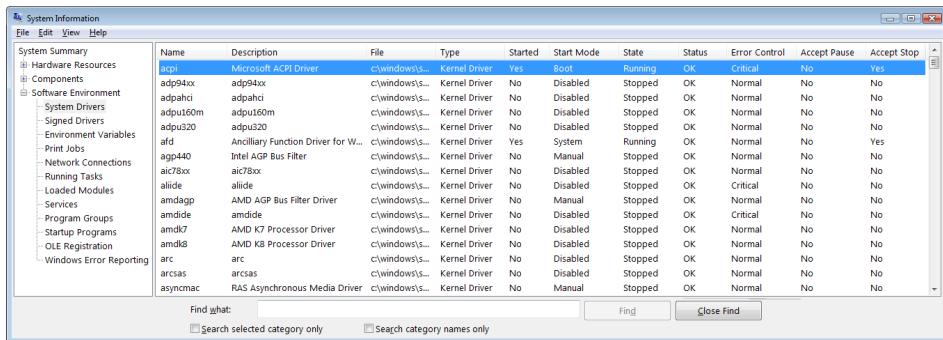


Рис. 8.4. Добавление промежуточного драйвера

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 (Программная среда) строку System Drivers (Системные драйверы) — справа появится список всех драйверов системы. Загруженные драйверы отмечены словом Yes (Да) в столбце Started (Работает).

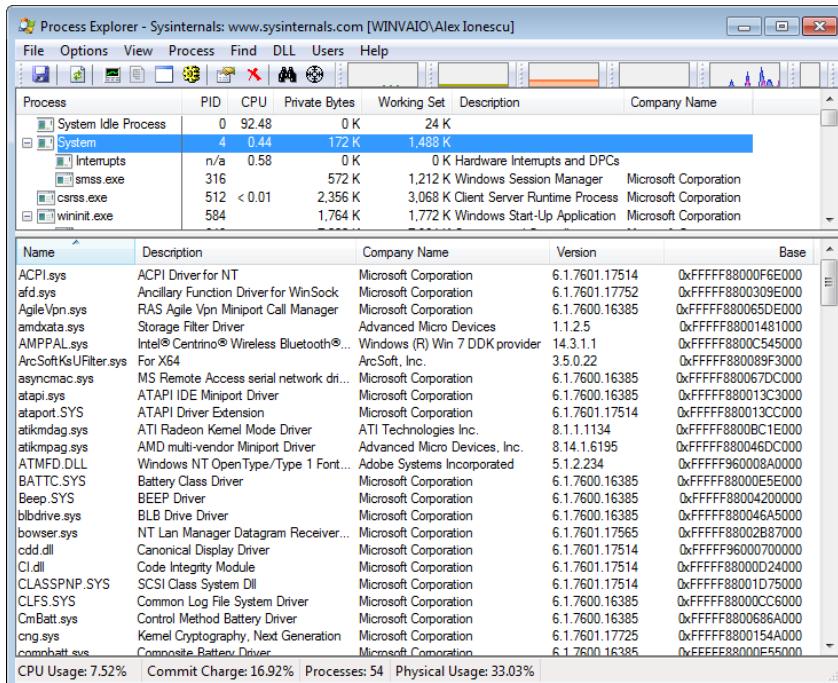


The screenshot shows the Windows System Information window. On the left, there's a tree view with categories like System Summary, Hardware Resources, Components, and Software Environment. Under Software Environment, the 'System Drivers' node is selected. The main pane displays a table of drivers:

Name	Description	File	Type	Started	Start Mode	Status	Error Control	Accept Pause	Accept Stop
acpi	Microsoft ACPI Driver	c:\windows\sys...	Kernel Driver	Yes	Boot	Running	OK	Critical	No
adp94xx	adp94xx	c:\windows\sys...	Kernel Driver	No	Disabled	Stopped	OK	Normal	No
adphaci	adphaci	c:\windows\sys...	Kernel Driver	No	Disabled	Stopped	OK	Normal	No
adput60m	adput60m	c:\windows\sys...	Kernel Driver	No	Disabled	Stopped	OK	Normal	No
adpu320	adpu320	c:\windows\sys...	Kernel Driver	No	Disabled	Stopped	OK	Normal	No
afd	Ancillary Function Driver for W...	c:\windows\sys...	Kernel Driver	Yes	System	Running	OK	Normal	No
agp440	Intel AGP Bus Filter	c:\windows\sys...	Kernel Driver	No	Manual	Stopped	OK	Normal	No
aic78xx	aic78xx	c:\windows\sys...	Kernel Driver	No	Disabled	Stopped	OK	Normal	No
alide	alide	c:\windows\sys...	Kernel Driver	No	Disabled	Stopped	OK	Critical	No
amdags	AMD AGP Bus Filter Driver	c:\windows\sys...	Kernel Driver	No	Manual	Stopped	OK	Normal	No
amidde	amidde	c:\windows\sys...	Kernel Driver	No	Disabled	Stopped	OK	Critical	No
amdI7	AMD K7 Processor Driver	c:\windows\sys...	Kernel Driver	No	Disabled	Stopped	OK	Normal	No
amdI8	AMD K8 Processor Driver	c:\windows\sys...	Kernel Driver	No	Disabled	Stopped	OK	Normal	No
arc	arc	c:\windows\sys...	Kernel Driver	No	Disabled	Stopped	OK	Normal	No
arcas	arcas	c:\windows\sys...	Kernel Driver	No	Disabled	Stopped	OK	Normal	No
asyncmac	RAS Asynchronous Media Driver	c:\windows\sys...	Kernel Driver	No	Manual	Stopped	OK	Normal	No

At the bottom, there are search fields: 'Find what:' and 'Find', and checkboxes for 'Search selected category only' and 'Search category names only'.

Список загруженных драйверов для режима ядра можно увидеть с помощью приложения Process Explorer, созданного компанией Windows Sysinternals (<http://www.microsoft.com/technet/sysinternals>). После его запуска следует выделить строку System в столбце Process и выбрать в меню View команду Lower Pane View ▶ DLLs.



The screenshot shows the Process Explorer window. The top menu bar includes File, Options, View, Process, Find, DLL, Users, and Help. The main pane displays a table of processes:

Process	PID	CPU	Private Bytes	Working Set	Description	Company Name
System Idle Process	0	92.48	0 K	24 K		
System	4	0.44	172 K	1,483 K		
Interrupts	n/a	0.58	0 K	0 K	Hardware Interrupts and DPCs	
smss.exe	316	572 K	1,212 K	Windows Session Manager	Microsoft Corporation	
csrss.exe	512	< 0.01	2,356 K	3,068 K	Client Server Runtime Process	Microsoft Corporation
wininit.exe	584		1,764 K	1,772 K	Windows Start-Up Application	Microsoft Corporation

Below this, another table lists individual drivers:

Name	Description	Company Name	Version	Base
ACPI.sys	ACPI Driver for NT	Microsoft Corporation	6.1.7601.17514	0xFFFFF88000F6E000
afdi.sys	Ancillary Function Driver for WinSock	Microsoft Corporation	6.1.7601.17752	0xFFFFF8800309E000
AgileVpn.sys	RAS Agile Vpn Miniport Call Manager	Microsoft Corporation	6.1.7600.16385	0xFFFFF880065DE000
amdiata.sys	Storage Filter Driver	Advanced Micro Devices	1.1.2.5	0xFFFFF88001481000
AMPAL.sys	Intel® Centino® Wireless Bluetooth®...	Windows (R) Win 7 DDK provider	14.3.1.1	0xFFFFF8800C545000
ArcSoftKsFilter.sys	For X64	ArcSoft, Inc.	3.5.0.22	0xFFFFF880089F3000
asyncmac.sys	MS Remote Access serial network dri...	Microsoft Corporation	6.1.7600.16385	0xFFFFF880067DC000
atapi.sys	ATAPI IDE Miniport Driver	Microsoft Corporation	6.1.7600.16385	0xFFFFF880013CC000
ataport.SYS	ATAPI Driver Extension	Microsoft Corporation	6.1.7601.17514	0xFFFFF880013CC000
atkmdag.sys	ATI Radeon Kernel Mode Driver	ATI Technologies, Inc.	8.1.1.1134	0xFFFFF8800BC1E000
atkmpag.sys	ATI Multi-vendor Miniport Driver	Advanced Micro Devices, Inc.	8.14.1.6195	0xFFFFF880046DC000
ATMFD.DLL	Windows NT OpenType/Type 1 Font...	Adobe Systems Incorporated	5.1.2.234	0xFFFFF960008A0000
BATT.CSY	Battery Class Driver	Microsoft Corporation	6.1.7600.16385	0xFFFFF88000E5E000
Beep.SYS	BEEP Driver	Microsoft Corporation	6.1.7600.16385	0xFFFFF88004200000
blbdrive.sys	BLB Drive Driver	Microsoft Corporation	6.1.7600.16385	0xFFFFF880046A5000
bowserv.sys	NT Lan Manager Datagram Receiver...	Microsoft Corporation	6.1.7601.17565	0xFFFFF88002B87000
cdq.dll	Canonical Display Driver	Microsoft Corporation	6.1.7601.17514	0xFFFFF96000700000
Ci.dll	Code Integrity Module	Microsoft Corporation	6.1.7601.17514	0xFFFFF88000D24000
CLASSPNP.SYS	SCSI Class System Dll	Microsoft Corporation	6.1.7601.17514	0xFFFFF88001D75000
CLFS.SYS	Common Log File System Driver	Microsoft Corporation	6.1.7600.16385	0xFFFFF88000C6C000
CmBatt.sys	Control Method Battery Driver	Microsoft Corporation	6.1.7600.16385	0xFFFFF8800686A000
cng.sys	Kernel Cryptography, Next Generation	Microsoft Corporation	6.1.7601.17725	0xFFFFF88001544000
lcompmatt.sus	Composite Battery Driver	Microsoft Corporation	6.1.7601.16385	0xFFFFF88000FE5E000

At the bottom, status information is displayed: CPU Usage: 7.52%, Commit Charge: 16.92%, Processes: 54, Physical Usage: 33.03%.

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:

```
lkd> lm kv
start end module name
82007000 823c0000 nt (pdb symbols)
c:\programming\symbols\ntkrpamp.pdb\37D328E3BAE5460F8E662756ED80951D2\ntkrpamp.
pdb
Loaded symbol image file: ntkrpamp.exe
Image path: ntkrpamp.exe
Image name: ntkrpamp.exe
Timestamp: Fri Jan 18 21:30:58 2008 (47918B12)
CheckSum: 00372038
ImageSize: 003B9000
File version: 6.0.6001.18000
Product version: 6.0.6001.18000
File flags: 0 (Mask 3F)
File OS: 40004 NT Win32
File type: 1.0 App
File date: 00000000.00000000
Translations: 0409.04b0
CompanyName: Microsoft Corporation
ProductName: Microsoft® Windows® Operating System
InternalName: ntkrpamp.exe
OriginalFilename: ntkrpamp.exe
ProductVersion: 6.0.6001.18000
FileVersion: 6.0.6001.18000 (longhorn_rtm.080118-1840)
FileDescription: NT Kernel & System
LegalCopyright: © Microsoft Corporation. All rights reserved.
823c0000 823f3000 hal (deferred)
Image path: halmacpi.dll
Image name: halmacpi.dll
Timestamp: Fri Jan 18 21:27:20 2008 (47918A38)
CheckSum: 0003859F
ImageSize: 00033000
Translations: 0000.04b0 0000.04e0 0409.04b0 0409.04e0
82600000 82671000 ksecdd (deferred)
Image path: \SystemRoot\System32\Drivers\ksecdd.sys
Image name: ksecdd.sys
Timestamp: Fri Jan 18 21:41:20 2008 (47918D80)
CheckSum: 0006E742
ImageSize: 00071000
Translations: 0000.04b0 0000.04e0 0409.04b0 0409.04e0
```

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. 8.5.

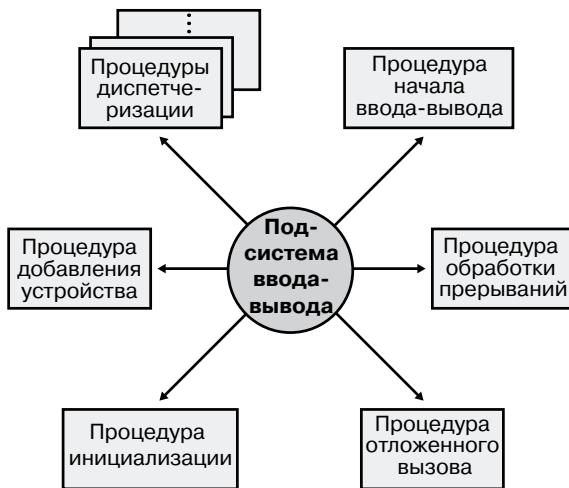


Рис. 8.5. Основные процедуры драйвера устройства

- **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 драйвер. Именно она регистрирует остальные процедуры драйвера в диспетчере ввода-вывода и при необходимости выполняет глобальную инициализацию этого драйвера.
- **Процедура добавления устройства.** Драйверы PnP-устройств реализуют процедуру добавления устройства. PnP-диспетчер при обнаружении устройства, за которое отвечает конкретный драйвер, посыпает этому драйверу уведомление. В рамках данной процедуры драйвер, как правило, создает соответствующий устройству объект, но об этом мы поговорим чуть позже.
- **Процедуры диспетчеризации.** Это основные точки входа для драйвера устройства. В качестве примера таких процедур можно привести открытие, закрытие, чтение, запись и прочие возможности устройства, файловой системы или сети. 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. Подробно IRQL-уровень рассматривается в главе 3 части I. 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 отложенного вызова.** Основную часть обработки прерывания, оставшейся после ISR-процедуры, выполняет процедура отложенного вызова. DPC-процедура работает на более низком IRQL-уровне (на уровне DPC/dispatch), чем запускаемая на уровне устройства ISR-процедура, чтобы не блокировать без необходимости другие прерывания. 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. 8.5:
- ❑ **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, вызвав функцию `CancelIo`), диспетчер ввода-вывода выполняет связанную с IRP процедурой 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 11), например драйверы файловой системы, обычно имеют процедуры быстрой диспетчеризации (fast dispatch routines), позволяющие ядру при доступе к драйверу обходить стандартную

обработку ввода-вывода. К примеру, такие операции, как чтение или запись, можно быстро выполнить путем непосредственного 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 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 очистку 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.

ПРИМЕЧАНИЕ

Большинство драйверов устройств, работающих в режиме ядра, написаны на языке С. Начиная с набора средств разработки Windows Driver Kit 8.0 драйверы можно писать на языке C++ благодаря поддержке новыми компиляторами C++ в режиме ядра. Использовать язык ассемблера не рекомендуется, так как он привносит дополнительную сложность и затрудняет перенос драйвера между такими аппаратными архитектурами, как x86, x64 и IA64.

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, какой драйвер следует вызвать 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* в системе соответствует отдельному драйверу. Именно он дает диспетчеру ввода-вывода адрес процедур диспетчеризации (точек входа) всех драйверов.
- ❑ *Объект устройства* представляет собой физическое или логическое устройство в системе и описывает его характеристики, 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 в систему 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. При удалении последнего объекта устройства и отсутствии ссылок на драйвер диспетчер ввода-вывода производит выгрузку этого драйвера.

В момент создания объекта устройства драйвер может присвоить ему имя. В этом случае объект помещается в пространство имен диспетчера объектов (оно подробно рассматривается в главе 3 части I). Имя объекта задается явным образом 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 рассматривается в главе 9.

Чтобы предоставить приложениям доступ к объекту устройства, драйвер должен создать в папке `\Global??` символьную ссылку на имя этого объекта в папке `\Device`. (Префиксы `\??` рассматриваются в главе 3 части I.) Драйверы, не поддерживающие технологию Plug and Play, и драйверы файловой системы обычно выбирают для символьной ссылки общеизвестное имя (например, `\Device\Hardware2`). 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 служебной программой `Uuidgen`, входящей в наборы WDK и Windows SDK. Учитывая предоставляемый 128 битами диапазон чисел, можно говорить о том, что каждый создаваемый GUID-идентификатор практически гарантированно является уникальным.

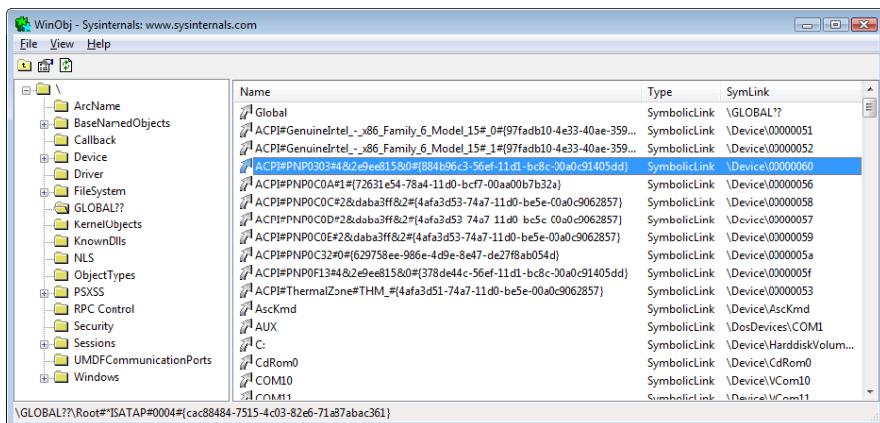
Функция `IoRegisterDeviceInterface` генерирует символьную ссылку, связанную с экземпляром объекта устройства. Но прежде чем диспетчер ввода-вывода действительно создаст ссылку, драйвер должен вызывать эту функцию, чтобы разрешить ис-

пользование интерфейсом устройства. Драйверы обычно делают это при получении от PnP-диспетчера пакета, инициирующего работу устройства, в данном случае — пакета **IRP_MJ_PNP**, **IRP_MN_START_DEVICE**.

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**, чтобы открыть устройство и получить его дескриптор.

ЭКСПЕРИМЕНТ: ПРОСМОТР ОБЪЕКТОВ УСТРОЙСТВ

Увидеть содержимое папки **\Device** в пространстве имен диспетчера объектов можно при помощи служебной программы **WinObj** от Sysinternals или команды **!object** отладчика ядра. На рисунке 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:

```
lkd> !object \Device
Object: 8b611b88 Type: (84d10d40) Directory
ObjectHeader: 8b611b70 (old version)
HandleCount: 0 PointerCount: 365
Directory Object: 8b602470 Name: Device
Hash Address Type Name
----- -----
00 85557a00 Device KsecDD
855589d8 Device Ndis
8b6151b0 SymbolicLink {941D252A-0BDA-4772-B3CB-30697579BD4A}
86859030 Device 0000009b
```

продолжение ➔

```

88c92da8 Device SrvNet
886723f0 Device Beep
8b71fb90 SymbolicLink ScsiPort2
84d17a98 Device 00000032
84d15f00 Device 00000025
84d13030 Device 00000019
01 86d44030 Device NDMP10
8d291eb0 SymbolicLink {E85EEE75-32E3-4A94-8905-52709C2C9BCC}
886da3c8 Device Netbios
86862030 Device 0000009c
84d177c8 Device 00000033
84d15c70 Device 00000026
02 86de9030 Device NDMP11
84d19320 Device 00000040
88633ca0 Device NetBT_Tcpip_{033C65A4-C1D6-4824-B420-DDAEADFF873E}
8b7dcdd0 SymbolicLink Ip
84d17500 Device 00000034
84d159a8 Device 00000027
03 86df3380 Device NDMP12
8515ede0 Device WMIAdminDevice
84d1a030 Device 00000041
8862e040 Device Video0
86eaec28 Device KeyboardClass0
84d03b00 Device KMDF0
84d17230 Device 00000035
84d156e0 Device 00000028
04 86e0d030 Device NDMP13
86e65030 Device NDMP20
85541030 Device VolMgrControl
86e6c358 Device Tun0
84d1ad68 Device 00000042
8862ec48 Device Video1
88e15158 Device 0000009f
9badd848 SymbolicLink MailslotRedirector
86e1d488 Device KeyboardClass1
...

```

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 хэш-таблице, основой для которой служит хэш имен объектов. Именно поэтому вы получаете перечень объектов, хранящихся в каждой корзине хэш-таблицы папки.

Как показано на рис. 8.6, 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 позволяет найти объект драйвера, соответствующий обслуживающему устройство драйверу. После этого происходит обращение к объекту драйвера через указанный в исходном запросе номер функции. Каждый номер функции соответствует точке входа драйвера. (Показанные на рис. 8.6 номера функции подробно рассматриваются в разделе «Блоки стека IRP-пакетов».)

С объектом драйвера часто связывается несколько объектов устройств. Список последних представляет собой перечень физических и логических устройств, управляемых

емых данным драйвером. Например, для каждого раздела жесткого диска существует отдельный объект устройства с информацией, касающейся данного раздела. Но при этом для доступа ко всем разделам используется один драйвер жесткого диска. При выгрузке драйвера из системы диспетчер ввода-вывода прибегает к очереди объектов устройств, чтобы определить, на какие устройства повлияет удаление данного драйвера.



Рис. 8.6. Объект 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:

```
1kd> !drvobj kbdclass
Driver object (86e379a0) is for:
\Driver\kbdclass
Driver Extension List: (id , addr)

Device Object list:
86e1d488 86eaec28

1kd> !devobj 86eaec28
Device object (86eaec28) is for:
KeyboardClass0 \Driver\kbdclass DriverObject 86e379a0
Current Irp 00000000 RefCount 0 Type 0000000b Flags 00002044
DevExt 86eaec0 DevObjExt 86eaedc0
```

продолжение ↗

```

ExtensionFlags (0x000000800)
Unknown flags 0x000000800
AttachedDevice (Upper) 86e15a40 \Driver\ctrl2cap
AttachedTo (Lower) 86e15020 \Driver\i8042prt
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).

Благодаря записи информации о драйверах в объекты диспетчеру ввода-вывода уже не требуются сведения о деталях реализации драйверов. Он находит драйвер по указателю, может легко загружать новые драйверы и обеспечивать их переносимость.

Открытие устройств

Такая структура данных режима ядра, как файловый объект, для устройства представляет собой дескриптор и точно соответствует определению объектов в Windows — это системный ресурс, доступный двум и более процессам в пользовательском режиме, у него может быть имя, его безопасность обеспечивается моделью защиты объектов, кроме того, он поддерживает синхронизацию. Ресурсы совместного использования в подсистеме ввода-вывода, как и в других компонентах исполнительной подсистемы Windows, обрабатываются в виде объектов. (Описание диспетчера объектов можно найти в главе 3 части I, а в главе 6 части I вы найдете сведения о безопасности объектов.)

Файловые объекты обеспечивают нас таким представлением ресурсов в памяти, которое напоминает интерфейс, ориентированный на ввод-вывод и реализующий чтение или запись. В табл. 8.1 перечислены некоторые атрибуты файлового объекта. Объявления полей и их размеры можно посмотреть в определении структуры FILE_OBJECT в файле WDM.h.

Таблица 8.1. Атрибуты файлового объекта

Атрибут	Назначение
Имя файла	Идентифицирует физический файл, на который ссылается файловый объект, переданный в CreateFile 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	Указывают тип ввода-вывода — синхронный или асинхронный, кэшируемый или нет, с последовательным или прямым доступом и т. п.
Указатель на объект устройства	Указывает тип устройства, в котором находится файл

Атрибут	Назначение
Указатель на блок параметров тома (Volume Parameter Block, VPB)	Указывает том или раздел, в котором находится файл
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 шаблоны чтения или приоритет страницы для процесса. Информацию о приоритете страницы вы найдете в главе 10 «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 связанную с контекстом 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, что дало нам шесть дополнительных атрибутов. Они перечислены в табл. 8.2.

Таблица 8.2. 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 (подробно рассматривается в разделе «Порты завершения ввода-вывода»). Задается процедурой SetFileIoOverlappedRange
Общий	Содержит информацию, связанную с фильтрующим драйвером и добавляемыми к вызывающей программе расширенными параметрами создания. Задается процедурой IoCreateFileEx

продолжение ↗

Таблица 8.2 (продолжение)

Расширение	Назначение
Запланированный ввод-вывод файла	Сохраняет сведения о резервировании полосы пропускания (см. далее), которые используются системой хранения для оптимизации, и гарантирует пропускную способность мультимедийных приложений. 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 12

При открытии вызывающей программой файла или простого устройства диспетчер ввода-вывода возвращает дескриптор файлового объекта. Происходящие при открытии файла процессы показаны на рис. 8.7.

В данном примере программа на языке С вызывает из стандартной библиотеки функцию `fopen` (1), которая, в свою очередь, вызывает Windows-функцию `CreateFile` (2). Далее DLL (в данном случае — Kernel32.dll) подсистемы Windows вызывает из модуля Ntdll.dll собственную функцию `NtCreateFile` (3). Она содержит команды, заставляющие перейти в диспетчер системных сервисов в режиме ядра, который вызывает настоящую функцию `NtCreateFile` из файла Ntoskrnl.exe (4). (О диспетчеризации системных служб рассказывается в главе 3 части I.) Напоследок эта функция обрабатывает параметры и флаги таким образом, что функция `IoCreateFile` диспетчера ввода-вывода получает возможность выполнить предписанную операцию.

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 задают собственные представления.

Файлы, как и прочие объекты исполнительной системы, защищены дескрипторами безопасности, которые включают в себя список контроля доступа (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 потоку дескриптором файла (5, 6). 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. Это предполагает новую проверку безопасности. (Вопросы защиты объектов рассматриваются в главе 6 части I.)

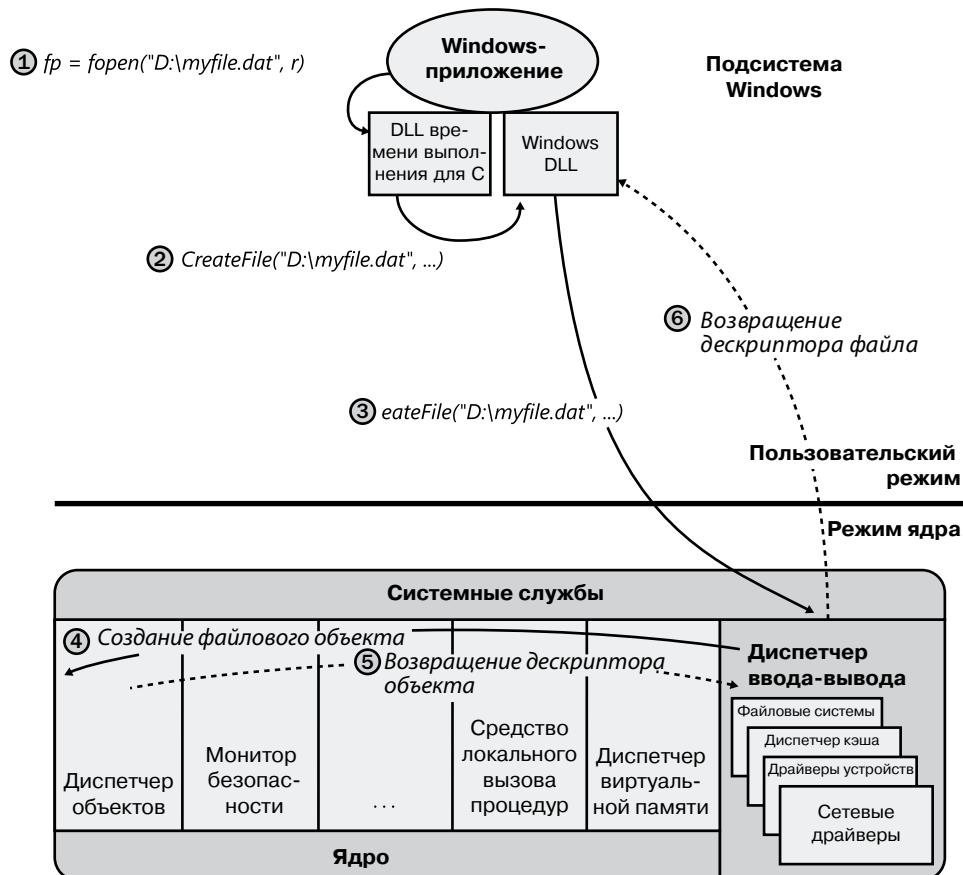
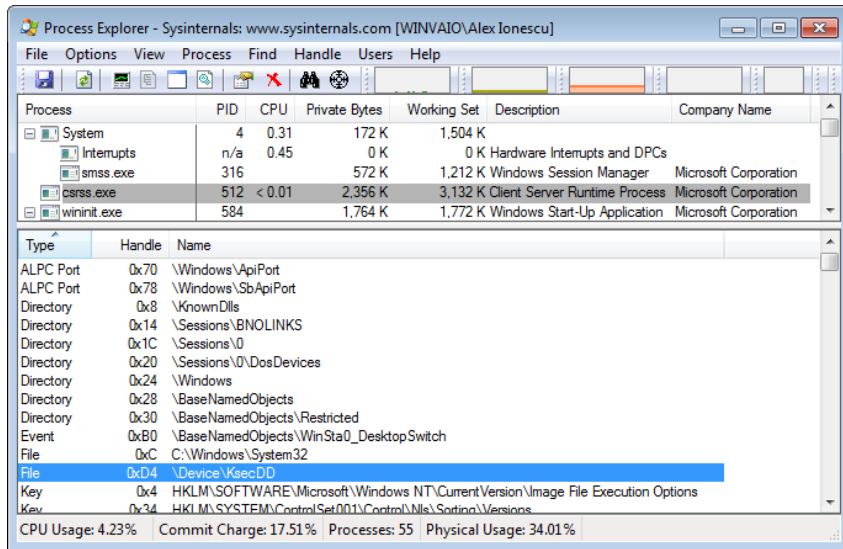


Рис. 8.7. Открытие файлового объекта

ЭКСПЕРИМЕНТ: ПРОСМОТР ДЕСКРИПТОРОВ УСТРОЙСТВ

Для любого процесса, открывшего дескриптор устройства, в таблице дескрипторов появится файловый объект. Его можно увидеть в приложении Process Explorer. Достаточно выделить процесс и выбрать 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 помечены как File.



В данном примере процесс Csrss 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 0xD4), который принадлежит процессу Csrss.exe с идентификатором 512 (0x200):

```
1kd> !handle d4 f 200
```

```
Searching for Process with Cid == 200
PROCESS ffffffa800bf35b30
SessionId: 0 Cid: 0200 Peb: 7fffffd8000 ParentCid: 0188
DirBase: 1dba5000 ObjectTable: fffff8a000f28d80 HandleCount: 630.
Image: csrcss.exe
```

```
Handle table at fffff8a000f28d80 with 630 entries in use
```

```
00d4: Object: ffffffa800c9cc9f0 GrantedAccess: 00100001 Entry: fffff8a001409350
Object: ffffffa800c9cc9f0 Type: (fffffa800737a080) File
ObjectHeader: ffffffa800c9cc9c0 (new version)
HandleCount: 1 PointerCount: 1
```

Так как мы имеем дело с файловым объектом, данные о нем можно получить командой !fileobj:

```
1kd> !fileobj ffffffa800c9cc9f0
```

```
Device Object: 0xffffffa8007da1550 \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 разделяемого 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 будет задавать в файле место начала следующей операции чтения или записи с использованием указанного дескриптора. Каждый дескриптор имеет собственное значение этого атрибута, несмотря на открытый доступ к файлу. Кроме того, файловый объект уникален для каждого процесса. Исключением является случай, когда дескриптор дублируется процессом для передачи другому процессу (при помощи Windows-функции `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.

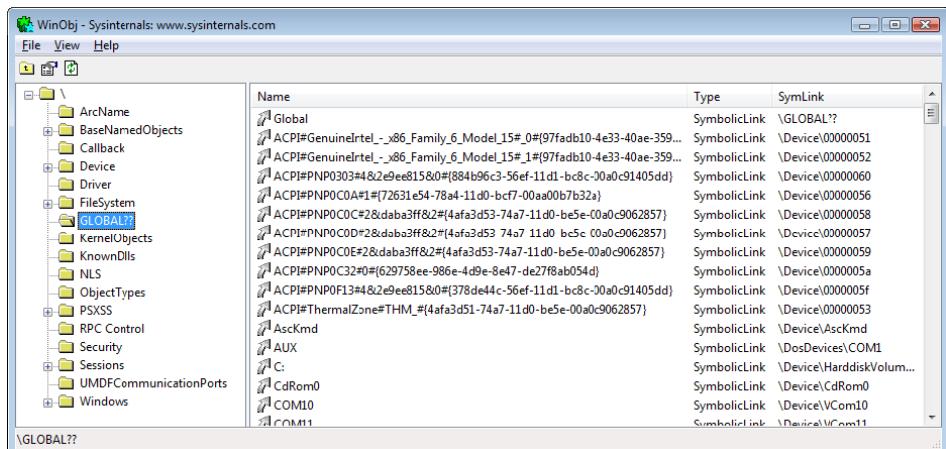
Дескриптор файла уникален для процесса, а вот определяемый им физический ресурс — нет. Следовательно, как при доступе к любым общим ресурсам, программным потокам требуется синхронизировать свое обращение к совместно используемым файлам, папкам и устройствам. Например, если поток осуществляет запись в файл, при открытии файла он должен получить эксклюзивный доступ на запись, лишив все остальные программные потоки возможности выполнить эту операцию одновременно с ним. Заблокировать на время записи часть файла позволяет Windows-функция `LockFile`.

Имя открытого файла включает в себя имя объекта устройства, на котором находится файл. Например, имя `\Device\HarddiskVolume1\Myfile.dat` относится к файлу `Myfile.dat` на диске С. 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 символические ссылки на реальные внутренние имена устройств в Windows. Как уже упоминалось, за создание ссылок в этой папке отвечают драйверы устройств, обеспечивая Windows-приложениям доступ к подотчетным им устройствам. Ссылки можно посмотреть и даже отредактировать через такие Windows-функции, как `QueryDosDevice` и `DefineDosDevice`.

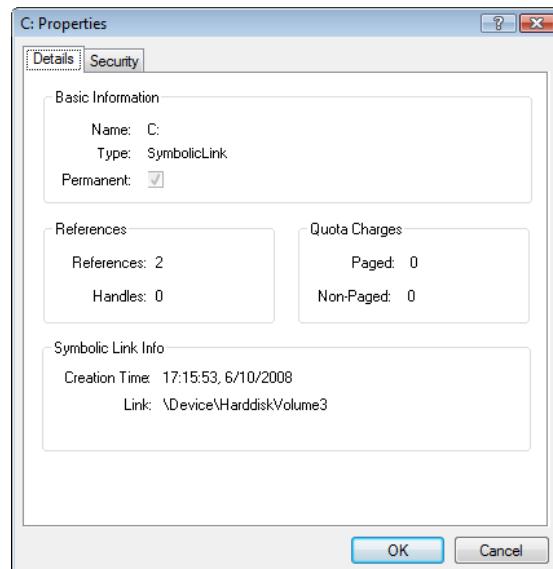
ЭКСПЕРИМЕНТ: ОТОБРАЖЕНИЕ WINDOWS-ИМЕН НА ВНУТРЕННИЕ ИМENA УСТРОЙСТВ

Доступ к символическим ссылкам, определяющим пространство имен устройств в Windows, дает служебная программа `WinObj`, разработанная в `Sysinternals`. Запустите ее и перейдите в папку `\Global??`.

продолжение ↗



Обратите внимание на расположенные справа символические ссылки. Щелкните правой кнопкой мыши на устройстве С и выберите команду Properties. Должно открыться вот такое диалоговое окно.



Как видите, С представляет собой символическую ссылку на внутреннее устройство \Device\HarddiskVolume3, то есть на первый том первого жесткого диска в системе. Показанный в приложении WinObj элемент COM1 является символической ссылкой на устройство \Device\Serial0 и т. п. Попробуйте создать собственные ссылки в командной строке с помощью команды subst.

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 делится на несколько очевидных этапов. Наборы выполняемых на каждом этапе операций 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.

Типы ввода-вывода

Существуют различные типы запросов на ввод и вывод. Более того, диспетчер ввода-вывода позволяет драйверам реализовывать сокращенный интерфейс ввода-вывода, что зачастую сокращает необходимые для обработки данных запросы IRP-пакетов. Рассмотрим различные типы более подробно.

Синхронный и асинхронный ввод-вывод

Большинство операций ввода-вывода, запрашиваемые приложениями, являются *синхронными* (этот тип предлагается по умолчанию); то есть 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 передать запрос на ввод или вывод и продолжить свою работу, не дожидаясь передачи данных устройством. Это повышает эффективность приложения, позволяя его программному потоку решать другие задачи параллельно с операцией ввода-вывода. Для асинхронного ввода-вывода при вызове Windows-функции `CreateFile` 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 запросом на завершение ввода-вывода, наблюдая за дескриптором объекта синхронизации (роль этого объекта могут играть событие, 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; то есть после выдачи запроса на ввод или вывод драйвер устройства возвращается в подсистему ввода-вывода. Будет ли управление немедленно возвращено вызывающей программе, зависит от того, какой именно ввод-вывод — синхронный или асинхронный — инициирован дескриптором. Схема управления для операции чтения показана на рис. 8.8. Обратите внимание, что наступление состояния ожидания

зависит от флага FILE_FLAG_OVERLAPPED в файловом объекте и реализуется функцией `NtReadFile` в режиме ядра.

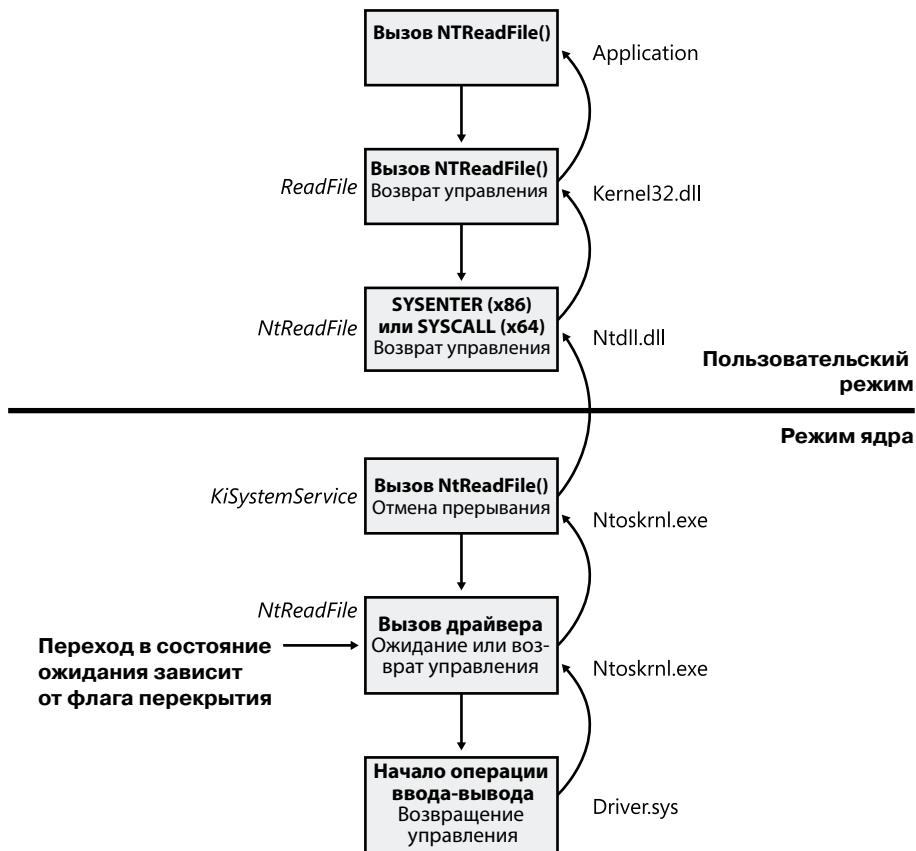


Рис. 8.8. Схема управления при операции ввода-вывода

Проверка состояния асинхронной операции ввода-вывода в режиме ожидания выполняется макросом `HasOverlappedIoCompleted`. При использовании портов завершения ввода-вывода для этой цели применяются функции `GetQueuedCompletionStatus` и `GetQueuedCompletionStatusEx`.

Быстрый ввод-вывод

При быстром вводе-выводе подсистема ввода-вывода может обойтись без генерации IRP-пакетов и напрямую обратиться к стеку драйверов для завершения запроса на ввод или вывод. Подробно быстрый ввод-вывод рассматривается в главах 11 и 12. 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. Вот пример таблицы быстрого ввода-вывода для драйверного объекта файловой системы NTFS:

```
!drvobj \FileSystem\Ntfs
Driver object (fffffa8007d9fbe0) is for:
\FileSystem\Ntfs
DriverEntry: ffffff880017d406c Ntfs!GsDriverEntry
DriverStartIo: 00000000
DriverUnload: 00000000
AddDevice: 00000000
Dispatch routines:
...
Fast I/O routines:
FastIoCheckIfPossible ffffff88001782230 Ntfs!NtfsFastIoCheckIfPossible
FastIoRead ffffff880016efd60 Ntfs!NtfsCopyReadA
FastIoWrite ffffff880016f2a10 Ntfs!NtfsCopyWriteA
FastIoQueryBasicInfo ffffff880016e42e8 Ntfs!NtfsFastQueryBasicInfo
...
ReleaseForModWrite ffffff8800166fee4 Ntfs!NtfsReleaseFileForModWrite
AcquireForCcFlush ffffff8800167133c Ntfs!NtfsAcquireFileForCcFlush
ReleaseForCcFlush ffffff880016713a0 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, происходит стандартное формирование IRP-пакета.

Ввод-вывод для файлов, отображенных на память, и кэширование файлов

Важной задачей подсистемы ввода-вывода является ввод в файлы и вывод из файлов, отображенных на память. Такой ввод-вывод поддерживается как самой подсистемой, так и диспетчером памяти. (Детали, касающиеся отображения файлов на память, рассматриваются в главе 10.) Само понятие *ввода-вывода для файлов, отображенных на память* (mapped file I/O), относится к возможности рассматривать файл на диске как часть 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 делает запись 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` и `MapViewOfFile`. В самой операционной системе такой ввод-вывод применяется при выполнении важных операций, например кэшировании файлов и активации образов (загрузке и запуске исполняемых программ). Еще ввод-вывод данного типа востребован диспетчером кэша. Файловые системы обращаются к этому диспетчеру для отображения данных на виртуальную память, что ускоряет функционирование 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 вариабельность стала возможной благодаря тому, что диспетчер кэша при автоматическом расширении (или уменьшении) размера кэша опирается на диспетчера памяти. Подробно данный механизм рассматривается в главе 10. Система подкачки страниц диспетчера памяти выполняет часть обязанностей диспетчера кэша, что уменьшает количество продлеваемых последним операций. (Внутреннее устройство диспетчера кэша подробно рассматривается в главе 11.)

Фрагментированный ввод-вывод

В Windows поддерживается также особый вид высокопроизводительного ввода-вывода, который называется *фрагментированным* (*scatter/gather I/O*). Он реализуется через Windows-функции `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 и выровнять пользовательские буферы по границам страниц. Кроме того, ввод-вывод должен быть асинхронным (перекрывающимся). 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 на ввод и вывод

Пакет запроса ввода-вывода (I/O Request Packet, IRP) представляет собой место, где 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 ассоциативных списков, отдельных для каждого процессора и хранящихся в пуле неподкачиваемой памяти. Ассоциативный список малых IRP-пакетов хранит пакеты в одном блоке стека (о том, что такое блок стека, мы поговорим чуть позже). Для средних IRP-пакетов имеется список пакетов с четырьмя блоками стека (при этом их можно применять для IRP-пакетов, требующих всего два или три блока стека), а для больших IRP-пакетов ассоциативный список содержит IRP-пакеты с более чем четырьмя блоками стека. По умолчанию в последнем случае система сохраняет IRP-пакеты с 10 блоками стека, но раз в минуту это число приводится в соответствие с количеством запрошенных блоков и может быть увеличено до 20. Кроме того, существует резервная копия этих списков

в глобальных ассоциативных списках, что обеспечивает передачу IRP-пакетов между процессорами. Если IRP-пакет требует больше блоков стека, чем в ассоциативном списке больших IRP-пакетов, диспетчер ввода-вывода выделяет под пакеты память из пула неподкачиваемой памяти. После создания и инициализации IRP-пакета диспетчер ввода-вывода сохраняет там указатель на файловый объект вызывающей программы.

ПРИМЕЧАНИЕ

Если определен DWORD-параметр реестра HKLM\System\CurrentControlSet\Session Manager\I/O System\LargerIrpStackLocations, он указывает, сколько блоков стека содержится в IRP-пакетах, хранящихся в ассоциативном списке больших IRP-пакетов.

Пример запроса ввода-вывода, демонстрирующий взаимосвязь между IRP и объектами файла, устройства и драйвера, показан на рис. 8.9. В данном случае рассматривается запрос к одноуровневому драйверу, в то время как большинство операций ввода-вывода не настолько примитивны и управляются одним или даже несколькими многоуровневыми драйверами. (Этот случай рассматривается чуть позже.)

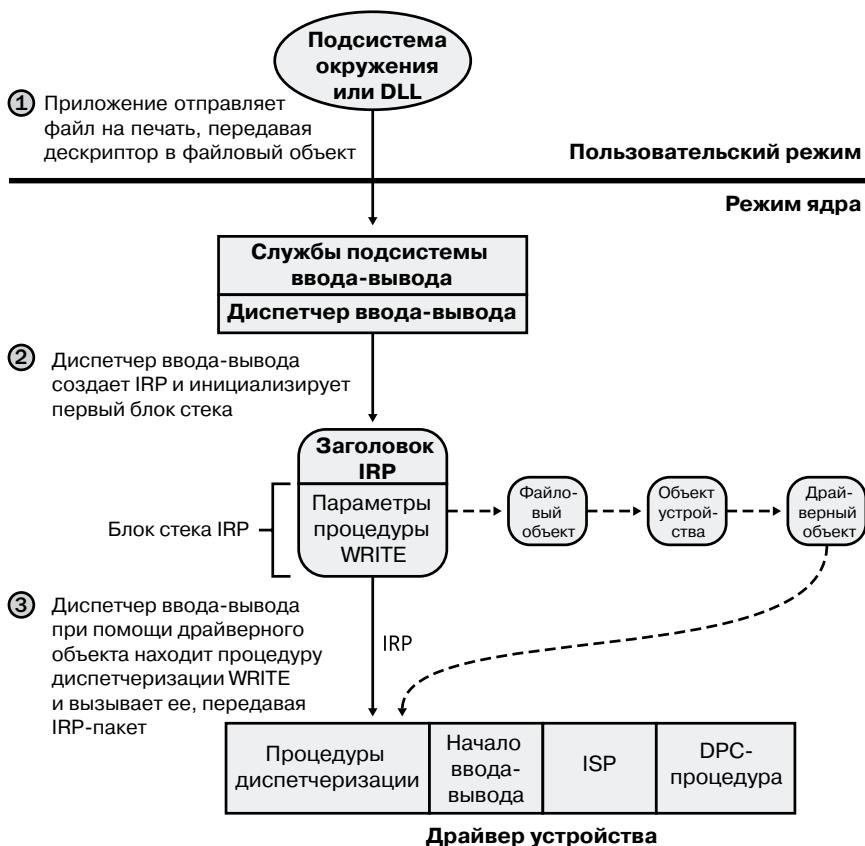


Рис. 8.9. Структуры данных, участвующие в запросе на ввод-вывод, адресованном одноуровневому драйверу

Блоки стека IRP-пакетов

IRP-пакет состоит из двух частей: фиксированного заголовка (его еще называют телом IRP-пакета) и одного или нескольких блоков стека. Фиксированная часть содержит данные о типе и размере запроса, сведения о том, синхронным или асинхронным он является, указатель на буфер для буферизованного ввода-вывода и информацию о состоянии, меняющуюся по ходу обработки запроса. Блок стека IRP-пакета содержит номер функции (составленный из основного и дополнительного номеров), связанные с функцией параметры и указатель на файловый объект вызывающей программы. *Основной номер функции* (major function code) определяет, какую из процедур диспетчеризации драйвера диспетчер ввода-вывода вызывает при передаче IRP-пакета драйверу. Необязательный *дополнительный номер функции* (minor function code) иногда применяется как модификатор основного номера. Он всегда включается в команды управления электропитанием и PnP-устройствами.

Большинство драйверов применяют процедуры диспетчеризации только для подмножества возможных основных номеров функций, включая функции создания (открытия), чтения, записи, управления вводом-выводом на устройстве, управления электропитанием и PnP-операциями, управления системой (для WMI-команд), очистки и закрытия. (Полный список основных номеров функций вы найдете в описании следующего эксперимента.) Драйверы файловой системы можно привести как пример типа драйверов, которые определяют функции для всех или почти всех точек входа. А вот драйвер простого USB-устройства, скорее всего, предоставит только процедуры открытия, закрытия, чтения, записи и отправки управляющих кодов ввода-вывода. Диспетчер ввода-вывода записывает в не заполненные драйверами точки входа указатели на свою функцию `IopInvalidDeviceRequest`. Она завершает IRP-пакет, возвращая код ошибки, уведомляющий о попытке обращения к функции, не поддерживаемой данным устройством.

ЭКСПЕРИМЕНТ: ПРОСМОТР ПРОЦЕДУР ДИСПЕТЧЕРИЗАЦИИ ДРАЙВЕРА

Для получения списка всех функций, определенных драйвером для его процедур диспетчеризации, следует ввести 7 после имени (или адреса) драйверного объекта в команде отладчика ядра `!drvobj`. Показанный далее результат действия этой команды демонстрирует, что драйверы поддерживают 28 типов процедур.

```
1kd> !drvobj \Driver\kbdclass 7
Driver object (fffffa800adc2e70) is for:
\Driver\kbdclass
Driver Extension List: (id , addr)

Device Object list:
fffffa800b04fce0 ffffffa800abde560

DriverEntry: fffff880071c8ecc kbdclass!GsDriverEntry
DriverStartIo: 00000000
DriverUnload: 00000000
AddDevice: fffff880071c53b4 kbdclass!KeyboardAddDevice

Dispatch routines:
[00] IRP_MJ_CREATE fffff880071bedd4 kbdclass!KeyboardClassCreate
```

```
[01] IRP_MJ_CREATE_NAMED_PIPE ffffff800036abc0c nt!IoPInvalidDeviceRequest
[02] IRP_MJ_CLOSE ffffff880071bf17c kbdclass!KeyboardClassClose
[03] IRP_MJ_READ ffffff880071bf804 kbdclass!KeyboardClassRead
...
[19] IRP_MJ_QUERY_QUOTA ffffff800036abc0c nt!IoPInvalidDeviceRequest
[1a] IRP_MJ_SET_QUOTA ffffff800036abc0c nt!IoPInvalidDeviceRequest
[1b] IRP_MJ_PNP ffffff880071c0368 kbdclass!KeyboardPnP
```

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 или вывода.) 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 означает встроенное прерывание вместо APC-запроса.

ЭКСПЕРИМЕНТ: ПРОСМОТР НЕЗАВЕРШЕННЫХ IRP-ПАКЕТОВ ПРОГРАММНОГО ПОТОКА

Команда !thread выводит на экран все связанные с программным потоком IRP-пакеты. Запустите отладчик ядра в работающей системе и найдите в строках, сгенерированных командой !process, процесс диспетчера управления службами (Services.exe):

```
1kd> !process 0 0
**** NT ACTIVE PROCESS DUMP ****
...
PROCESS 8623b840 SessionId: 0 Cid: 0270 Peb: 7ffd6000 ParentCid: 0210
DirBase: ce21e080 ObjectTable: 964c06a0 HandleCount: 198.
Image: services.exe
...
```

Теперь создайте для процесса дамп потоков, выполнив применительно к объекту процесса команду !process. Вы увидите множество программных потоков, большая часть которых имеет IRP-пакеты. Эти пакеты перечислены в разделе IRP List сведений о потоке (обратите внимание, что отладчик выводит только первые 17 IRP-пакетов для потока, количество незавершенных запросов на ввод и вывод у которого превышает 17):

```
1kd> !process 8623b840
PROCESS 8623b840 SessionId: 0 Cid: 0270 Peb: 7ffd6000 ParentCid: 0210
DirBase: ce21e080 ObjectTable: 964c06a0 HandleCount: 198.
Image: services.exe
VadRoot 862b1358 Vads 71 Clone 0 Private 466. Modified 14. Locked 2.
DeviceMap 8b0087d8
...
THREAD 86a1d248 Cid 0270.053c Peb: 7ffdc000 Win32Thread: 00000000
WAIT: (UserRequest) UserMode Alertable
86a40ca0 NotificationEvent
86a40490 NotificationEvent
```

продолжение ↴

```
IRP List:
86a81190: (0006,0094) Flags: 00060900 Mdl: 00000000
...
```

Выберите IRP-пакет и проанализируйте его при помощи команды !irp:

```
1kd> !irp 86a81190
Irp is active with 1 stacks 1 is current (= 0x86a81200)
No Mdl: No System Buffer: Thread 86a1d248: Irp stack trace.
cmd flg cl Device File Completion-Context
>[ 3, 0] 0 1 86156328 86a4e7a0 00000000-00000000 pending
\FileSystem\Npfs
Args: 00000800 00000000 00000000 00000000
```

Основной номер функции в этом IRP-пакете равен 3, что соответствует функции `IRP_MJ_READ`, которая находится, например, в файле `WDM.h`. Пакет содержит один блок стека и адресован устройству, принадлежащему драйверу `Npfs` (Named Pipe File System). Подробно этот драйвер рассматривается в главе 7 части I.

Управление буфером IRP-пакетов

В случае неявного создания IRP-пакета приложением или устройством при помощи системной службы `NtReadFile`, `NtWriteFile` или `NtDeviceIoControlFile` (или при помощи соответствующих этим службам API-функций Windows, то есть `ReadFile`, `WriteFile` и `DeviceIoControl`) диспетчер ввода-вывода определяет, должен ли он участвовать в управлении буферами ввода и вывода вызывающей программы. В этом диспетчере поддерживаются три механизма управления буферами:

- ❑ **Буферизованный ввод-вывод.** Диспетчер ввода-вывода выделяет буфер в пуле неподкачиваемой памяти, который по размеру совпадает с буфером вызывающей программы. Создавая IRP-пакет при операциях записи, диспетчер ввода-вывода копирует в выделенный буфер данные из буфера вызывающей программы. Завершая обработку IRP-пакета при операциях чтения, диспетчер копирует в пользовательский буфер данные из выделенного буфера, освобождая последний. Указателем на буфер в пуле неподкачиваемой памяти является поле `AssociatedIrp.SystemBuffer` IRP-пакета.
- ❑ **Прямой ввод-вывод.** Создавая IRP-пакет, диспетчер ввода-вывода блокирует пользовательский буфер в памяти (делает его неподкачиваемым). После завершения работы с IRP-пакетом буфер деблокируется. Диспетчер ввода-вывода хранит данные об этой памяти в виде *списка дескрипторов памяти* (Memory Descriptor List, MDL). В MDL указывается объем физической памяти, занятой буфером (подробно о MDL можно почитать в документации из WDK). Устройствам, использующим прямой доступ к памяти (Direct Memory Access, DMA), требуется только физическое описание буфера, поэтому для выполнения на них различного рода операций вполне достаточно списка дескрипторов памяти. (Устройства, поддерживающие DMA, передают данные в память компьютера через DMA-контроллер, минуя процессор.) Но для доступа драйвера к содержимому буфера достаточно отобразить буфер на адресное пространство системы.

- **Ввод-вывод без управления.** В этом случае диспетчер ввода-вывода не участвует в управлении буферами. Ответственность за управление возлагается на драйвер устройства, который позволяет вручную выполнить операции, реализуемые диспетчером в других механизмах управления буфером.

При любом варианте управления буфером диспетчер ввода-вывода помещает в IRP-пакет ссылки на буфера ввода и вывода. Выбор варианта управления диспетчером зависит от того, что было запрошено драйвером для операции определенного типа. Драйвер регистрирует нужный ему вариант управления буферами для операций чтения и записи в объекте устройства. Операции управления вводом и выводом для устройства (запрос которых выполняется вызовом функции `NtDeviceIoControlFile`) определяются управляющими кодами ввода-вывода драйвера. Такой код содержит данные о варианте управления буфером, применяемом диспетчером ввода-вывода.

Для запросов размером менее одной страницы (4 Кбайт на процессорах x86) и для устройств, не поддерживающих DMA, драйверы, как правило, используют буферизованный ввод-вывод. Для запросов большего размера и для устройств с поддержкой DMA применяется прямой ввод-вывод. Драйверы файловой системы обычно пользуются третьим вариантом управления, так как это позволяет сократить расходы на управление драйверами при копировании данных из кэша файловой системы в буфер вызывающей программы. Но большинство драйверов предпочитают другие варианты, так как указатель на буфер вызывающей программы действителен лишь на время выполнения программным потоком этой программы.

Расположенные в пользовательском пространстве драйверы, использующие ввод-вывод третьего типа, должны проверять, что адреса буферов корректны и не ссылаются на память в режиме ядра. Впрочем, скалярные значения можно передавать без проблем, хотя они имеются у считанного количества драйверов. Отсутствие проверки может привести к краху системы или проблемам с безопасностью, поскольку приложения, имея доступ к памяти в режиме ядра, получают возможность внедрять в ядро свой код. Предоставляемые ядром драйверам функции `ProbeForRead` и `ProbeForWrite` проверяют, может ли буфер полностью поместиться в пользовательской части адресного пространства. Чтобы избежать краха системы из-за ссылки на недопустимый адрес, драйверы могут обращаться к буферам в режиме пользователя из кода обработки исключений (в языке С из блоков `try/except`), перехватывающего любые попытки доступа по неправильным адресам и преобразующего их в возвращаемые приложением коды ошибок. Кроме того, драйверы должны фиксировать все входящие данные в буфере ядра, не полагаясь на адреса пользовательского режима. Ведь вызывающая программа всегда может внести изменения в данные, минуя драйвер, даже при корректных адресах памяти.

Запрос ввода-вывода к одноуровневому драйверу

Рассмотрим процесс обработки запроса синхронного ввода-вывода к одноуровневому драйверу устройства в режиме ядра. В простейшем виде он состоит из семи этапов:

1. Запрос на ввод или вывод передается через DLL-подсистемы.
2. DLL-подсистемы вызывают службу `NtWriteFile` диспетчера ввода-вывода.

3. Диспетчер ввода-вывода создает описывающий запрос IRP-пакет и посыпает его драйверу (в данном случае – драйверу устройства), вызывая собственную функцию `IoCallDriver`.
4. Драйвер передает на устройство данные из IRP-пакета и начинает операцию ввода-вывода.
5. Устройство уведомляет о завершении ввода или вывода, генерируя прерывание.
6. Драйвер устройства обслуживает прерывание.
7. Драйвер вызывает функцию `IoCompleteRequest` диспетчера ввода-вывода, чтобы уведомить о завершении обработки запроса из IRP-пакета, после чего диспетчер ввода-вывода завершает запрос.

Эти этапы иллюстрирует рис. 8.10.

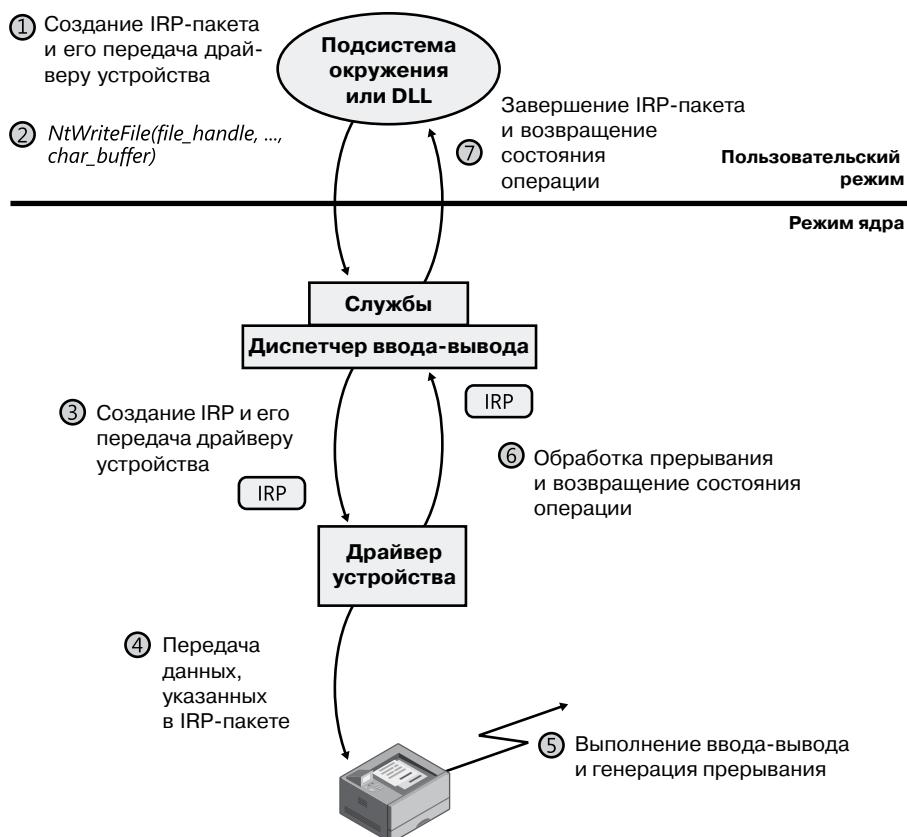


Рис. 8.10. Обработка синхронного запроса на ввод или вывод

Теперь, когда вы знаете, как инициируется ввод-вывод, рассмотрим процесс обработки прерывания и завершение операций ввода-вывода.

Обработка прерывания

Завершив передачу данных, устройство ввода-вывода генерирует прерывание, после чего в дело вступают ядро Windows, диспетчер ввода-вывода и драйвер устройства. Первая фаза процесса показана на рис. 8.11. Механизм диспетчеризации прерываний, в том числе DPC, описывается в главе 3 части I, а здесь мы коротко повторим данный материал, так как отложенный вызов процедур является ключевым в обработке ввода-вывода на устройствах, управляемых прерываниями.

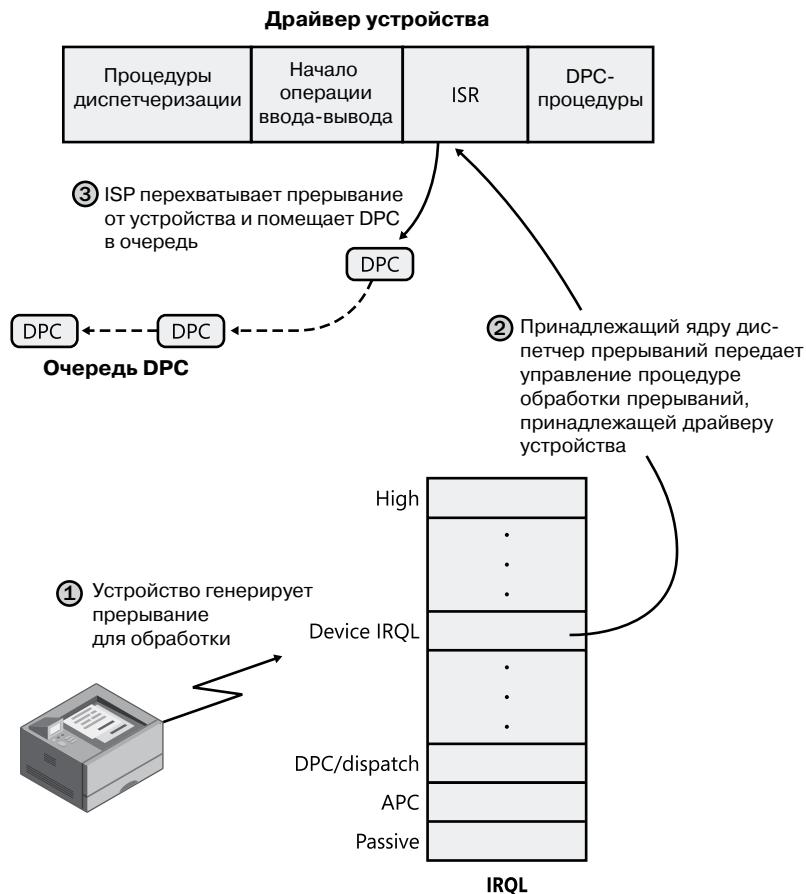


Рис. 8.11. Обработка прерываний устройства (фаза 1)

При возникновении прерывания устройства процессор передает управление обработчику исключений и прерываний ядра, который по таблице диспетчеризации прерываний находит ISR для этого устройства. В Windows обычно ISR обрабатывают прерывания устройств в два этапа. При первом вызове ISR, как правило, остается на уровне IRQL устройства ровно столько времени, сколько требуется для сохранения состояния устройства и запрета его дальнейших прерываний. Затем DPC помещается в очередь

и завершается, закрыв прерывание. Впоследствии, когда вызывается DPC-процедура на уровне 2, устройство завершает обработку прерывания и вызывает диспетчер ввода-вывода, чтобы закончить ввод или вывод и удалить IRP-пакет. При этом устройство может начать выполнение следующего в очереди устройства запроса на ввод или вывод.

Применение DPC для выполнения большинства вариантов обслуживания устройств делает возможными любые блокируемые прерывания уровня от «device IRQL» до «DPC/dispatch» прежде, чем начнется обработка DPC с более низким приоритетом. В результате быстрее обслуживаются прерывания промежуточных уровней, что сокращает время ожидания в системе. Рисунок 8.12 иллюстрирует вторую фазу ввода-вывода (обработку DPC).

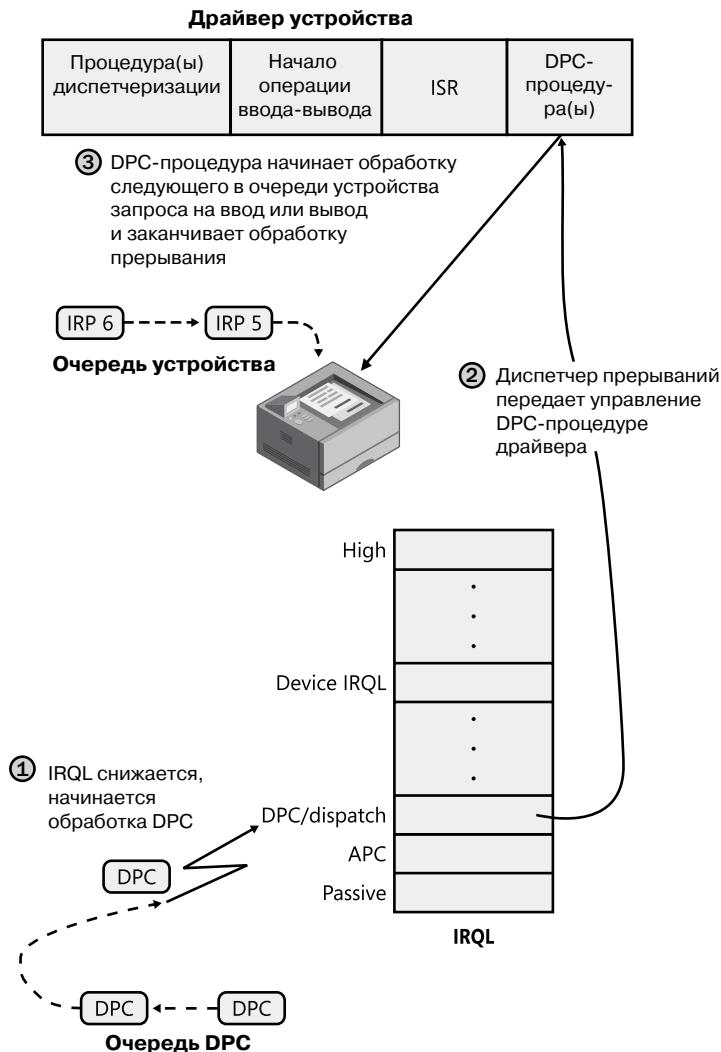


Рис. 8.12. Обработка прерываний устройства (фаза 2)

Завершение обработки запроса на ввод-вывод

После выполнения DPC-процедуры драйвером устройства остается ряд действий, необходимых для завершения запроса на ввод или вывод. Это третья фаза обработки данного запроса, называемая завершением, начинается с вызова драйвером функции `IoCompleteRequest` для уведомления диспетчера ввода-вывода о конце обработки указанного в IRP-пакете запроса (и о принадлежащих ему блоках стека). Выполняемые на этом этапе действия зависят от конкретной операции ввода-вывода. Например, все драйверы ввода-вывода записывают результат операции в блок состояния ввода-вывода — структуру данных, хранящуюся в IRP и в процессе завершения ввода-вывода копирующую в буфер вызывающей программы. Аналогичным образом некоторые выполняющие буферизованный ввод-вывод драйверы требуют возвращения данных вызывающему потоку через подсистему ввода-вывода.

В обоих случаях подсистема ввода-вывода должна скопировать данные из системной памяти в виртуальное адресное пространство вызывающей программы. При синхронном завершении IRP-пакета это пространство является текущим и доступно напрямую, а вот в случае асинхронного завершения диспетчеру ввода-вывода приходится откладывать завершение IRP-пакета до получения доступа к адресному пространству вызывающей программы. Для этого диспетчер ввода-вывода должен передавать данные «в контексте вызывающего потока», то есть в ходе выполнения программного потока вызывающей программы (предполагается, что процесс вызывающей программы является текущим, а его адресное пространство активно на процессоре). Эту задачу диспетчер ввода-вывода решает, ставя в очередь к потоку специальный асинхронный вызов процедуры (APC) в режиме ядра. Данный процесс иллюстрирует рис. 8.13.

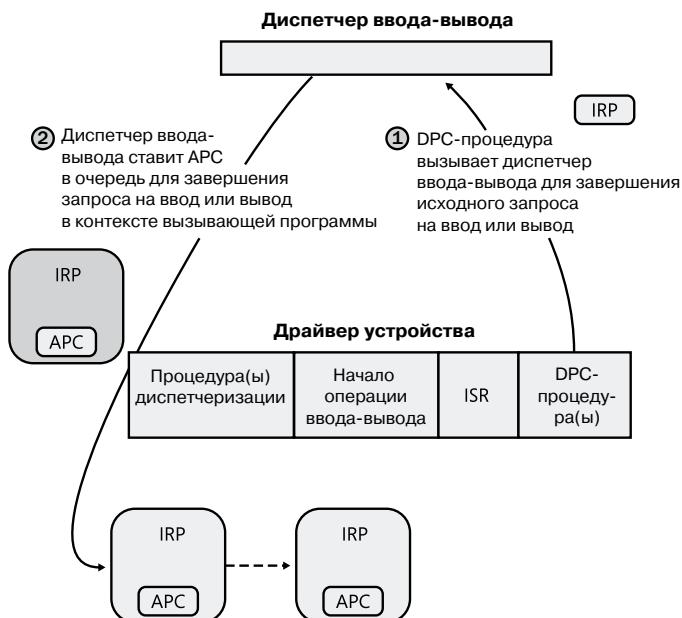


Рис. 8.13. Завершение обработки запроса на ввод-вывод (фаза 1)

Как объясняется в главе 3 части I, APC выполняется в контексте определенного программного потока, в то время как DPC – в контексте любого потока. Это означает, что DPC-процедура не касается адресного пространства процесса в режиме пользователя. Также вспомните, что приоритет программного прерывания (IRQL) у DPC выше, чем у APC.

В следующий раз, когда поток начнет выполнять при низком IRQL-уровне (ниже уровня DISPATCH_LEVEL), ему доставляется отложенный APC-вызов. Ядро передает управление APC-процедуре диспетчера ввода-вывода, которая копирует данные (для запроса на чтение) и код возвращения в адресное пространство исходной вызывающей программы, освобождает представляющий данную операцию IRP-пакет и либо переводит дескриптор файла вызывающей программы (и любое предлагаемое вызывающей программой событие) в свободное состояние для синхронного ввода-вывода, либо ставит элемент в очередь порта завершения ввода-вывода вызывающей программы. После этого операция ввода-вывода считается завершенной. Исходная вызывающая программа или любые другие программные потоки, ожидающие освобождения дескриптора файла (или другого объекта), переходят из состояния ожидания в состояние готовности к выполнению. Рисунок 8.14 иллюстрирует вторую фазу завершения ввода-вывода.

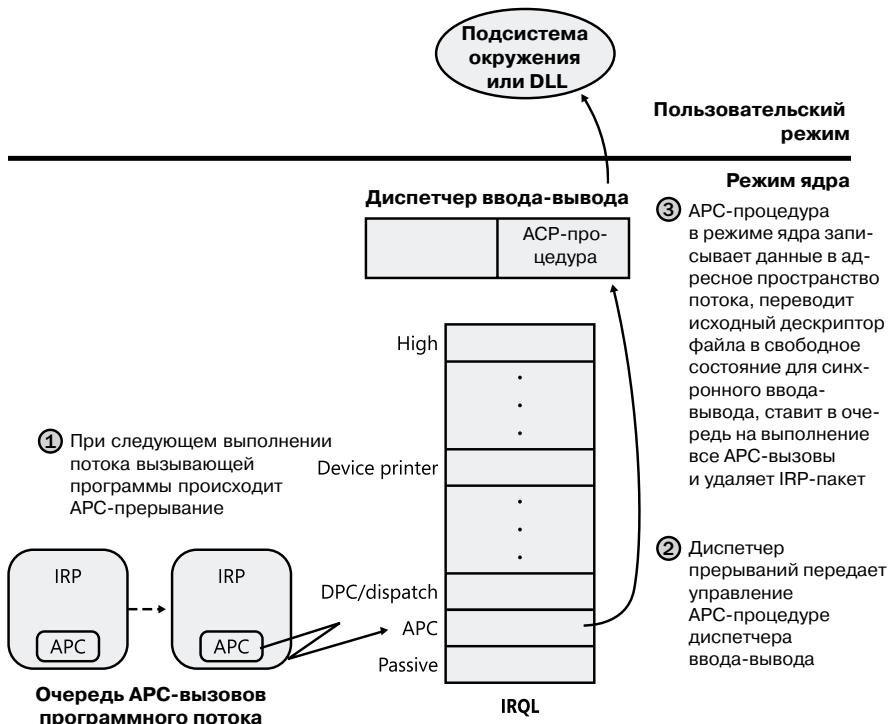


Рис. 8.14. Завершение запроса на ввод-вывод (фаза 2)

Это обычный способ завершения ввода-вывода, но если операция завершается в том же программном потоке, который выдал запрос на ввод или вывод, Windows действует по сокращенной схеме. Пока не отключен режим APC (для совместимости с более старыми версиями Windows, в которых даже в подобных ситуациях всегда применялся асинхронный вызов процедур), происходит встроенный вызов механизма завершения ввода-вывода на второй фазе.

Напоследок следует заметить, что функции асинхронного ввода-вывода `ReadFileEx` и `WriteFileEx` позволяют вызывающей программе задействовать в качестве параметра APC в режиме пользователя. При передаче данного параметра диспетчер ввода-вывода на последнем шаге завершения операции ставит данный APC-вызов в соответствующую очередь вызывающего потока. Это позволяет вызывающей программе определять подпрограмму, вызываемую после завершения или отмены запроса на ввод или вывод. Процедуры завершения APC в режиме пользователя выполняются в контексте реализующего запрос программного потока и доставляются, только если поток находится в режиме дежурного ожидания (например, при вызове функций Windows `SleepEx`, `WaitForSingleObjectEx` или `WaitForMultipleObjectsEx`).

Синхронизация

Драйверам нужно 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. Этую проблему иллюстрирует рис. 8.15.

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 обновлением 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 ядре Windows существует специальная процедура синхронизации `KeSynchronizeExecution`, вызываемая драйверами устройства при доступе к общим с ISR данным. Эта процедура не дает выполнять ISR-процедуру, пока идет работа с общими данными. Также драйвер может использовать процедуру `KeAcquireInterruptSpinLock` для непосредственной циклической блокировки объекта прерывания, хотя в общем

случае лучше, когда драйверы синхронизируются с ISR при помощи процедуры `KeSynchronizeExecution`. Дело в том, что вызов данной функции на уровне `PASSIVE_LEVEL` приведет к синхронизации с функцией `KEVENT` в структуре объекта прерывания, вместо того чтобы повысить IRQL-уровень.

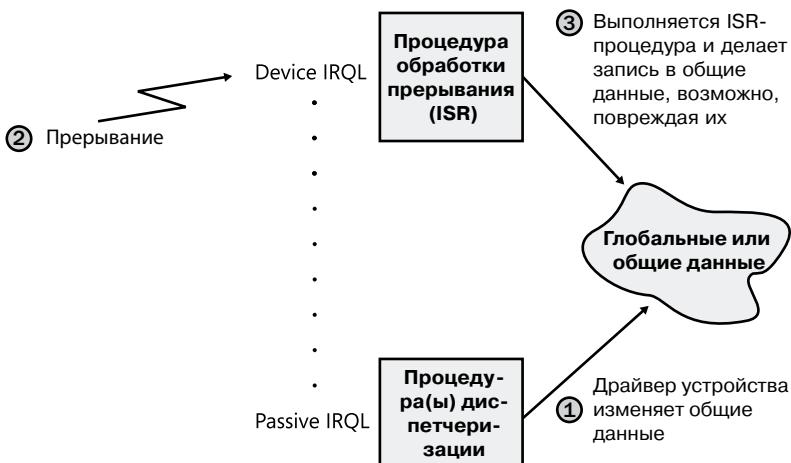


Рис. 8.15. Параллельный доступ к общим данным со стороны процедуры диспетчеризации драйвера устройства и ISR-процедуры

Надеемся, теперь вы понимаете, что особого внимания требуют не только ISR-процедуры. Любые используемые драйвером устройства данные могут стать объектом доступа со стороны фрагмента того же драйвера, выполняемого на другом процессоре. Поэтому для кода драйвера устройства крайне важно синхронизировать свое обращение к любым глобальным или общим данным (и любые обращения к самому физическому устройству). Если эти данные также использует ISR-процедура, драйвер устройства должен вызвать функцию `KeSynchronizeExecution` или `KeAcquireInterruptSpinLock`. В качестве альтернативы он может воспользоваться стандартными циклическими блокировками ядра (получаемыми на втором IRQL-уровне `DISPATCH_LEVEL`).

Запросы ввода-вывода к многоуровневым драйверам

В предыдущем разделе мы рассмотрели процесс обработки запроса на ввод-вывод, адресованного простому устройству, управляемому единственным драйвером. Обработка ввода-вывода в случае устройств, работающих с файлами, или запросов к другим многоуровневым драйверам происходит примерно таким же образом. Просто появляются несколько дополнительных операций.

На рис. 8.16 показан упрощенный пример прохождения через многоуровневые драйверы асинхронного запроса на ввод-вывод. В данном случае рассматривается диск, управляемый файловой системой.

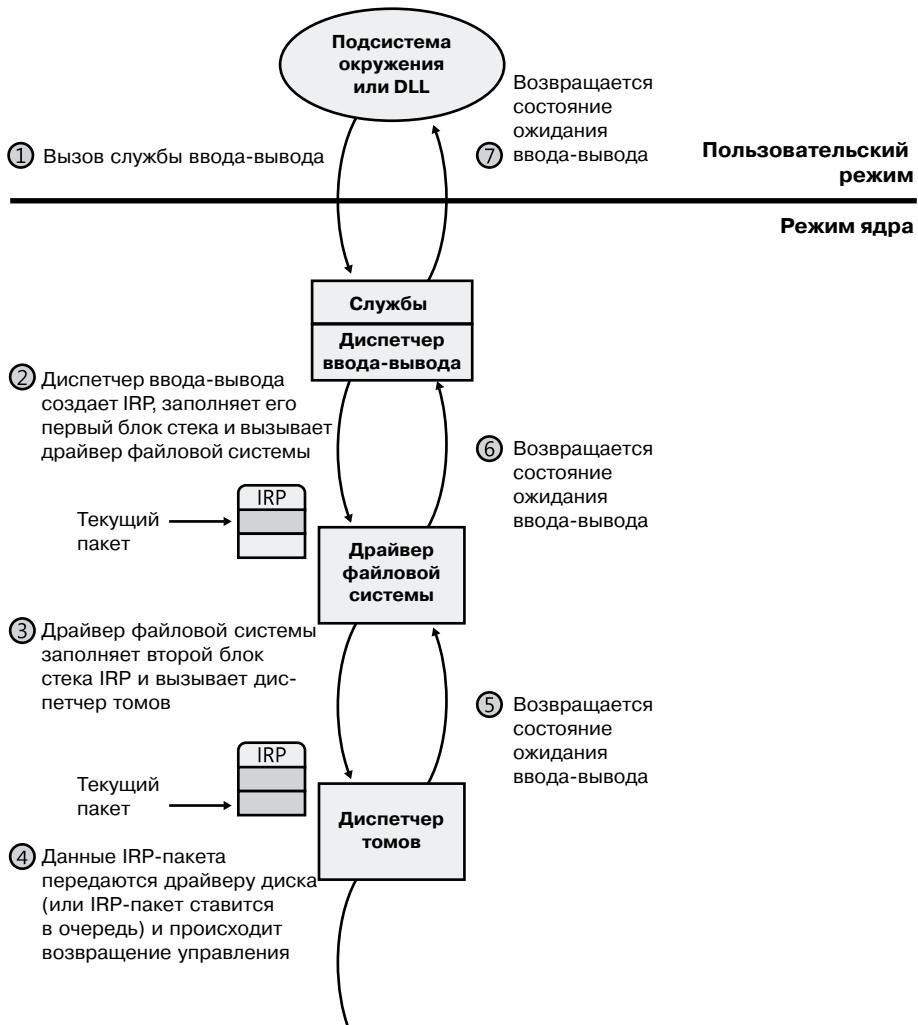


Рис. 8.16. Постановка в очередь асинхронного запроса к многоуровневым драйверам

В очередной раз диспетчер ввода-вывода получает запрос и создает для его представления пакет запросов на ввод и вывод. 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 посыпает драйверу диска уже имеющийся IRP-пакет или генерирует дополнительные IRP-пакеты и поочередно их передает.

Если полученный новый запрос допускает преобразование в единообразный понятный 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 попросит его прочитать один сектор из того места в томе, где начинается рассматриваемый файл.

Чтобы запрос к многоуровневым драйверам повторно использовался различными драйверами, IRP содержит набор *блоков стека* (stack locations) – не путайте этот стек со стеком процессора, который используется программными потоками для хранения параметров функций и адресов возврата. Эти области данных – по одной на каждый вызываемый драйвер – содержат информацию, которая требуется каждому из драйверов для выполнения своей части запроса, например номер функции, параметры и контекстная информация драйвера. Как показано на рис. 8.16, дополнительные блоки стека заполняются по мере передачи IRP от одного драйвера к другому. По способу добавления и удаления из него данных IRP-пакет напоминает стек. Но он не связан с каким бы то ни было конкретным процессом и имеет фиксированный размер. В начале операции ввода-вывода диспетчер ввода-вывода выделяет для IRP память в одном из ассоциативных списков или в пуле неподкачиваемой системной памяти.

ПРИМЕЧАНИЕ

Так как количество устройств в конкретном стеке известно заранее, диспетчер ввода-вывода выделяет один блок стека для каждого драйвера устройства. Но иногда IRP направляется в стек нового драйвера, как в сценарии с диспетчером фильтров, позволяющим фильтру передать IRP соседу (например, при переходе от локальной файловой системы к сетевой). Данная функциональность поддерживается предоставляемым диспетчером ввода-вывода API-интерфейсом `IoAdjustStackSizeForRedirection`, который добавляет блоки стека при обнаружении устройств в перенаправленном стеке.

ЭКСПЕРИМЕНТ: ПРОСМОТР СТЕКА УСТРОЙСТВА

Команда `!devstack` отладчика ядра выводит на экран стек устройства для выбранного многоуровневого объекта устройства. В рассматриваемом примере стек связан с объектом устройства `\device\KeyboardClass0`, принадлежащим драйверу класса клавиатуры:

```
1kd> !devstack KeyboardClass0
!DevObj !DrvObj !DevExt ObjectName
fffffa800a5e2040 \Driver\Ctrl2cap ffffffa800a5e2190
> ffffffa800a612ce0 \Driver\kbdclass ffffffa800a612e30 KeyboardClass0
fffffa800a612040 \Driver\i8042prt ffffffa800a612190
fffffa80076e0a00 \Driver\ACPI ffffffa80076f3a90 00000005c
!DevNode ffffffa800770f750 :
DeviceInst is "ACPI\PNP0303\4&b0a2531&0"
ServiceName is "i8042prt"
```

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. Как правило, IRP-пакеты передаются по стеку сверху вниз.

ЭКСПЕРИМЕНТ: АНАЛИЗ IRP-ПАКОТОВ

В этом эксперименте в системе находятся незавершенные IRP-пакеты и 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.

В системе в любой момент времени присутствует несколько незавершенных IRP-пакетов, поскольку существует множество устройств, которым приложения могут посыпал IRP-

пакеты, а драйвер обрабатывает запрос только после определенного события, например после получения доступа к данным. Например, это чтение в блокирующем режиме из конечной точки сети. Для просмотра незавершенных IRP-пакетов в системе используйте команду !irpfind отладчика ядра:

```
1kd> !irpfind
```

```
Scanning large pool allocation table for Tag: Irp? (86c16000 : 86d16000)
Searching NonPaged pool (80000000 : ffc00000) for Tag: Irp?
```

```
Irp [ Thread ] irpStack: (Mj,Mn) DevObj [Driver] MDL Process
862d2380 [8666dc68] irpStack: ( c, 2) 84a6f020 [ \FileSystem\Ntfs]
862d2bb0 [864e3d78] irpStack: ( e,20) 86171348 [ \Driver\AFD] 0x864dbd90
862d4518 [865f7600] irpStack: ( d, 0) 86156328 [ \FileSystem\Npfs]
862d4688 [867133f0] irpStack: ( 3, 0) 86156328 [ \FileSystem\Npfs]
862dd008 [00000000] Irp is complete (CurrentLocation 4 > StackCount 3)
0x00420000
862dee28 [864fc030] irpStack: ( 3, 0) 84baf030 [ \Driver\kbdclass]
```

Выделенная строка описывает IRP-пакет, адресованный драйверу Kbdclass, то есть данный пакет, скорее всего, инициирован программным потоком подсистемы Windows, читающим данные непосредственно с клавиатуры. Исследуют этот IRP-пакет командой !irp:

```
1kd> !irp 862dee28
Irp is active with 3 stacks 3 is current (= 0x862deee0)
No Mdl: System buffer=864f5108: Thread 864fc030: Irp stack trace.
cmd flg cl Device File Completion-Context
[ 0, 0] 0 0 00000000 00000000 00000000-00000000

Args: 00000000 00000000 00000000 00000000
[ 0, 0] 0 0 00000000 00000000 00000000-00000000
Args: 00000000 00000000 00000000 00000000
>[ 3, 0] 0 1 84baf030 864f52f8 00000000-00000000 pending
\Driver\kbdclass
Args: 00000078 00000000 00000000 00000000
```

Активный блок стека находится в самом низу. (Отладчик помечает его символом «» в первом столбце.) Основной номер функции равен 3, что соответствует IRP_MJ_READ.

Теперь нам нужно выяснить, какому объекту устройства адресован IRP-пакет. В этом нам поможет команда !devobj, в качестве параметра которой мы укажем взятый из активного блока стека адрес объекта устройства:

```
1kd> !devobj 84baf030
Device object (84baf030) is for:
KeyboardClass1 \Driver\kbdclass DriverObject 84b706b8
Current Irp 00000000 RefCount 0 Type 0000000b Flags 00002044
Dacl 8b0538b8 DevExt 84baf0e8 DevObjExt 84baf1c8
ExtensionFlags (0x00000800)
Unknown flags 0x00000800

AttachedTo (Lower) 84badaa0 \Driver\TermDD
Device queue is not busy.
```

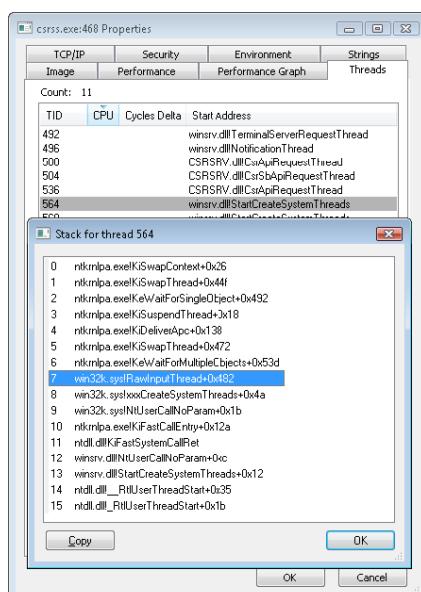
Итак, данный IRP-пакет адресован устройству KeyboardClass1. Присоединенный ниже объект устройства, принадлежащий драйверу Termdd, указывает, что мы имеем дело с вводом через клиент Terminal Server, а не с физической клавиатурой.

Получить подробную информацию о программном потоке и выдавшем IRP-пакет процессе нам помогут команды !thread и !process:

```
1kd> !thread 864fc030
THREAD 864fc030 Cid 01d4.0234 Peb: 7ffd9000 Win32Thread: ffac4008
WAIT: (WrUserRequest) KernelMode Alertable
8623c620 SynchronizationEvent
864fc3a8 NotificationTimer
864fc378 SynchronizationTimer
864fc360 SynchronizationEvent
IRP List:
86af0e28: (0006,01d8) Flags: 00060970 Mdl: 00000000
86503958: (0006,0268) Flags: 00060970 Mdl: 00000000
862dee28: (0006,01d8) Flags: 00060970 Mdl: 00000000
Not impersonating
DeviceMap 8b0087d8
Owning Process 0 Image: <Unknown>
Attached Process 864d2d90 Image: csrss.exe
Wait Start TickCount 171909 Ticks: 29 (0:00:00.000.452)
Context Switch Count 121222
UserTime 00:00:00.000
KernelTime 00:00:00.717
Win32 Start Address 0x764d9a30
Stack Init 96f46000 Current 96f45c28 Base 96f46000 Limit 96f43000 Call 0
Priority 15 BasePriority 13 PriorityDecrement 0 IoPriority 2 PagePriority 5

1kd> !process 864d2d90
PROCESS 864d2d90 SessionId: 1 Cid: 0208 Peb: 7ffd000 ParentCid: 0200
DirBase: ce21e0a0 ObjectTable: 964a6e68 HandleCount: 284.
Image: csrss.exe
```

Найдите этот поток в приложении Process Explorer, открыв для файла Csrss.exe диалоговое окно Properties и перейдя на вкладку Threads. Судя по именам функций в его стеке, мы действительно имеем дело с потоком непосредственного ввода подсистемы Windows.



После того как DMA-адаптер дискового контроллера завершит передачу данных, этот контроллер генерирует прерывание, вызывая ISR-процедуру, которая выдает запрос на обратный DPC-вызов, завершающий IRP-процедуру, как показано на рис. 8.17.

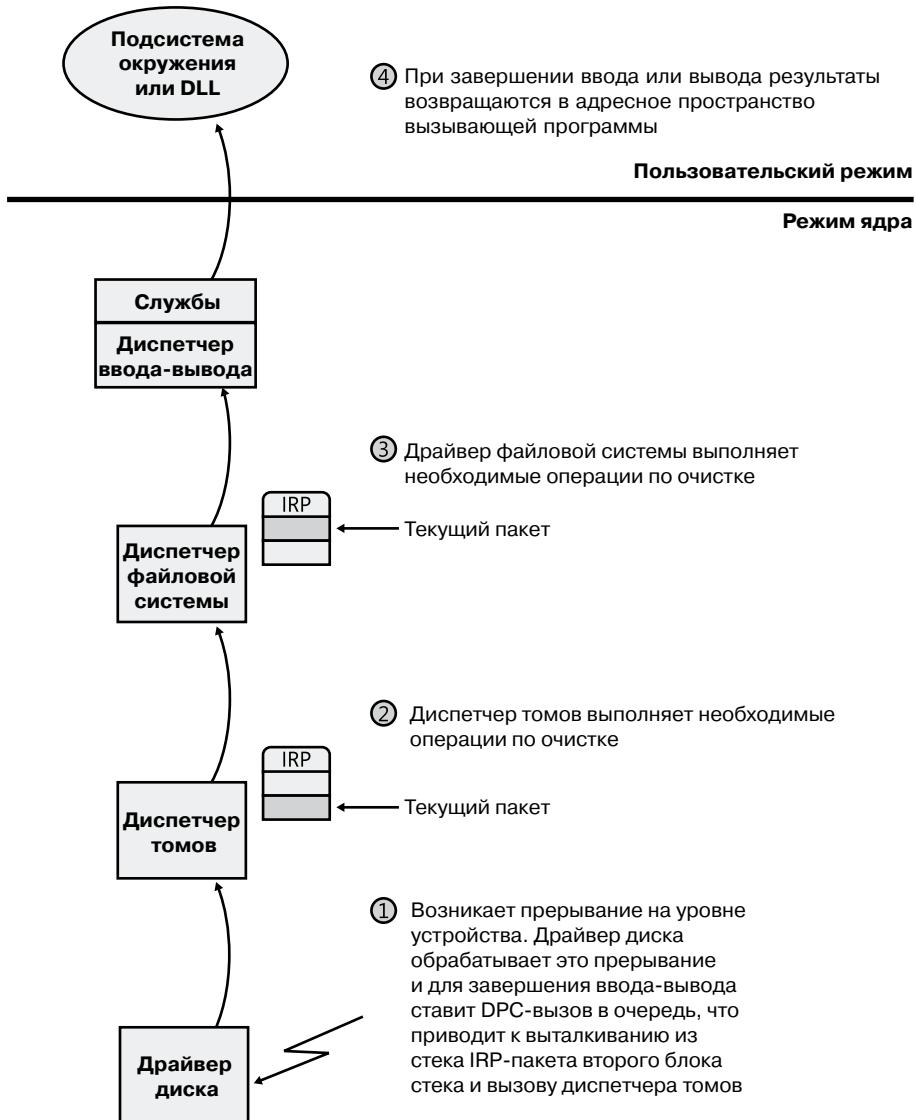


Рис. 8.17. Завершение многоуровневого запроса на ввод-вывод

В качестве альтернативы повторному использованию одного IRP-пакета файловая система может создать группу связанных IRP-пакетов, которые будут параллельно обрабатываться в рамках одного запроса на ввод или вывод. 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. 8.18.

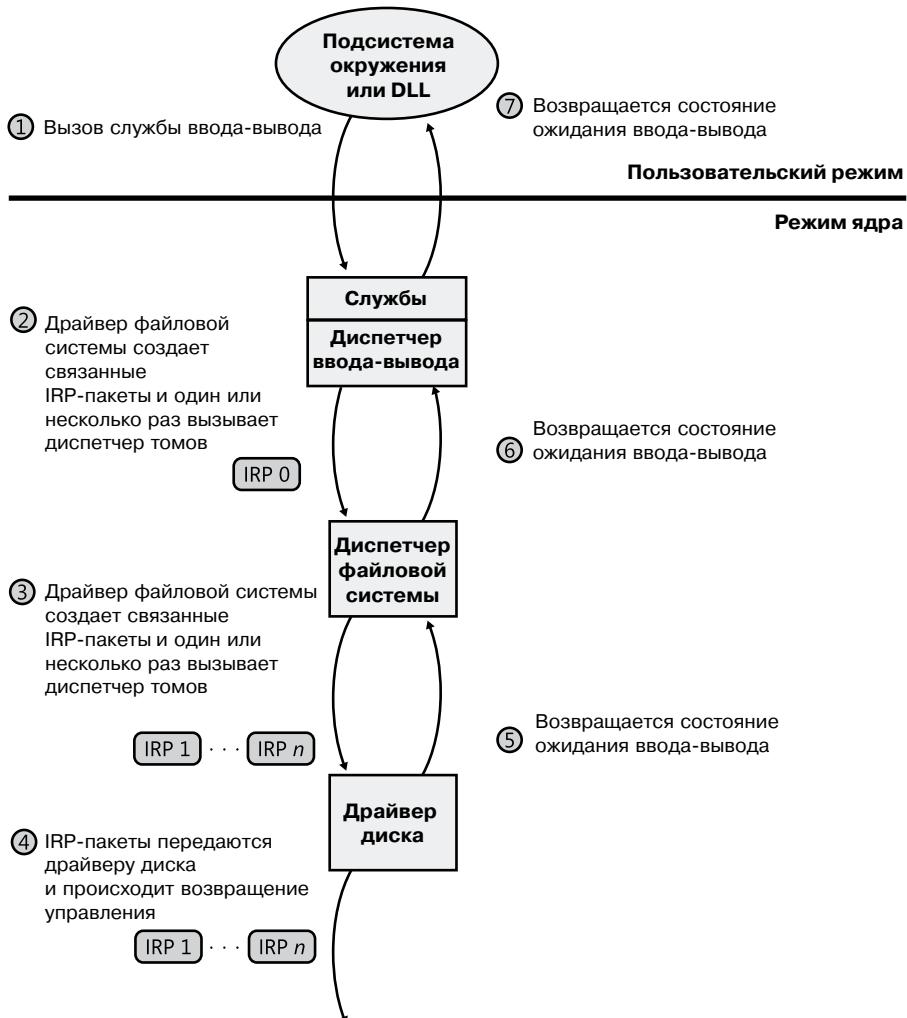


Рис. 8.18. Постановка в очередь связанных IRP-пакетов

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. 8.19.

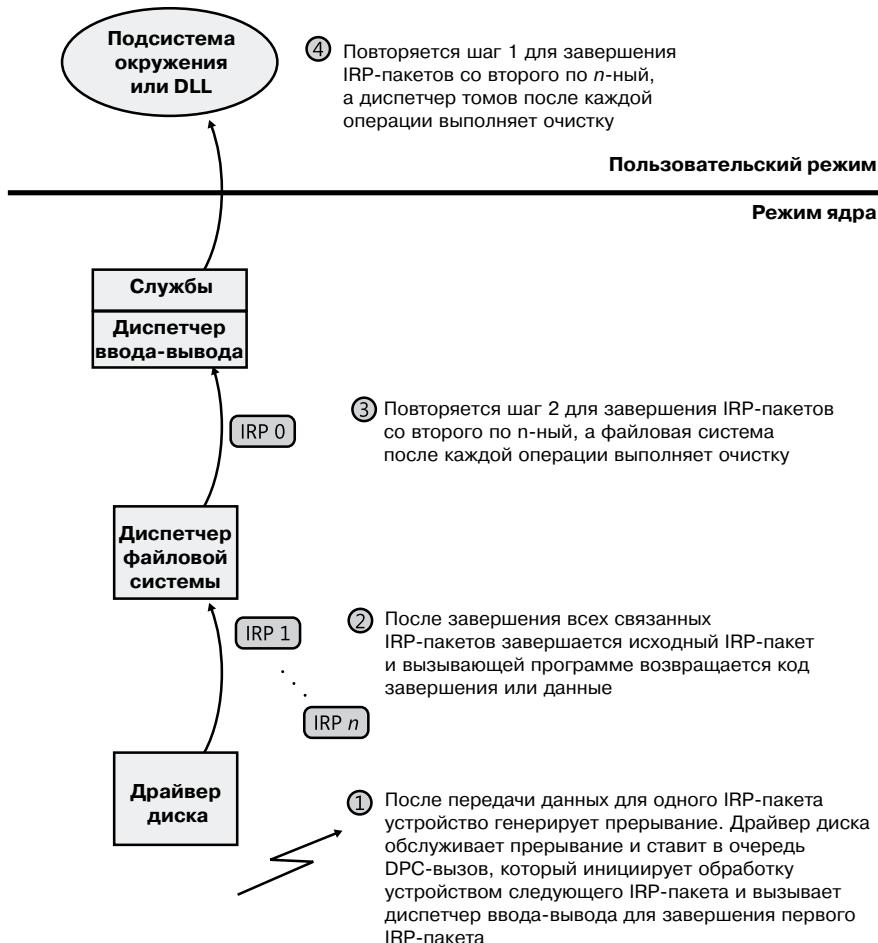


Рис. 8.19. Завершение связанных IRP-пакетов

ПРИМЕЧАНИЕ

Все драйверы, управляющие в Windows дисковыми файловыми системами, являются частью, как минимум, трехуровневого стека драйверов. На верхнем уровне находится драйвер файловой системы, затем следует диспетчер томов, а завершает все драйвер диска. Между этими драйверами может располагаться любое число 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 9.

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 дис-

петчер ввода-вывода выполнял асинхронный вызов процедуры. То есть завершение обработки осуществлялось в контексте процесса и программного потока. Связанная с потоком обработка ввода-вывода, как правило, в достаточной степени обеспечивает производительность и расширяемость большинства приложений, но в Windows также поддерживается *независимый от программных потоков ввод-вывод* (thread agnostic I/O), причем для этого применяются два подхода:

- ❑ 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 на ядро 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 `SE_LOCK_MEMORY` 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 множество способов, позволяющих обработать IRP-пакеты и завершить запрос на ввод или вывод, но несмотря на это часто операции ввода-вывода не завершаются, а отменяются. Например, при наличии активных IRP-пакетов может потребоваться удалить устройство; 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 ожидающие обработки операции ввода или вывода.

Работающий с драйверами диспетчер ввода-вывода должен эффективно и надежно обрабатывать такие запросы, обеспечивая целостность пользовательского интерфейса. Драйверы при этом регистрируют для допускающих отмену операций ввода-вывода (обычно это операции, стоящие в очереди, но еще не запущенные на выполнение) процедуру отмены, которая при необходимости и вызывается диспетчером ввода-вывода. Неверно функционирующий в этом сценарии драйвер приводит к появлению процессов, которые невозможно убить. Визуально они никак не проявляются, но работают в системе. Увидеть их можно в диспетчере задач или в приложении `Process Explorer`. (Подробно процессы и программные потоки рассматриваются в главе 5 части I.)

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 изрядное время, например для передачи больших объемов данных или сетевых манипуляций, в Windows предусмотрена отмена как синхронных, так и асинхронных операций. Программный поток может отменить собственные ожидающие выполнения операции ввода и вывода, вызвав функцию `CancelIo`. Она аннулирует все асинхронные операции ввода-вывода для определенного дескриптора файла, какому бы потоку они не 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.

Для отмены 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. Схема отмены синхронного и асинхронного ввода-вывода демонстрируется на рис. 8.20 и 8.21. (Для драйвера все варианты отмены выглядят одинаково.)

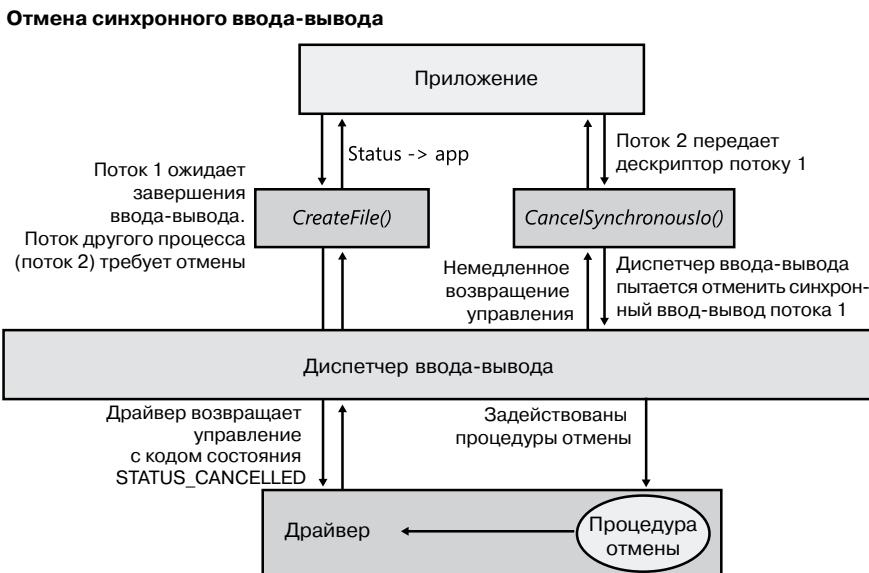


Рис. 8.20. Отмена синхронного ввода-вывода

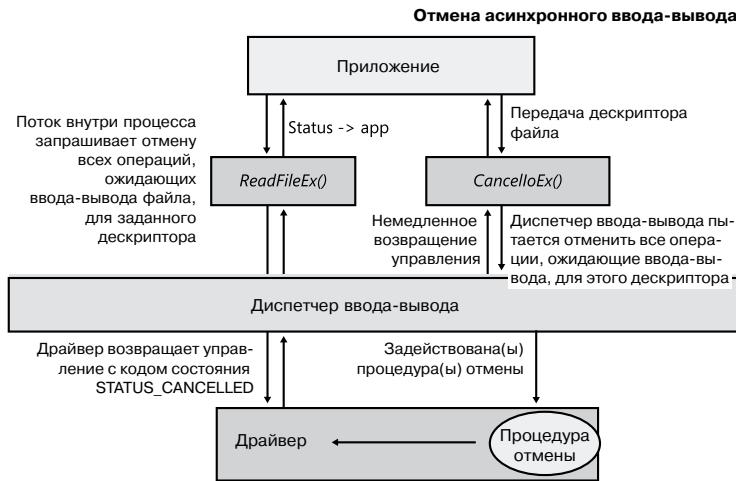


Рис. 8.21. Отмена асинхронного ввода-вывода

Отмена ввода-вывода при завершении программного потока

Вторым сценарием, при котором требуется отмена операций ввода-вывода, является прекращение работы программного потока, причем как непосредственное, так и вследствие завершения процесса (сопровождаемое завершением всех принадлежащих процессу потоков). Так как с каждым потоком связан список IRP-пакетов, диспетчер ввода-вывода может выбрать в этом списке допускающие отмену пакеты и аннулировать их. В отличие от функции `CancelIoEx`, возвращающей управление, 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. Сценарий завершения процесса иллюстрирует рис. 8.22.

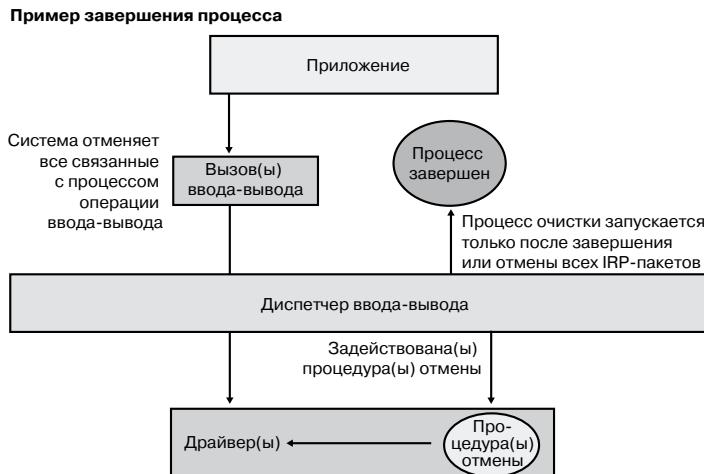


Рис. 8.22. Отмена при завершении процесса

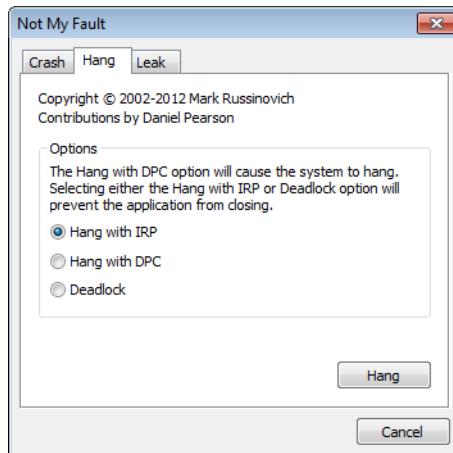
ПРИМЕЧАНИЕ

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 выявления неубиваемого процесса (подробно эта программа рассматривается в главе 14). Мы заставим драйвер Myfault.sys (которым пользуется программа Notmyfault.exe) неопределенно долго удерживать IRP-пакет из-за отсутствия для него зарегистрированной процедуры отмены.

Запустите файл Notmyfault.exe, на вкладке Hang открывшегося диалогового окна установите переключатель Hang with IRP, как показано на рисунке, и щелкните на кнопке Hang.



Визуально ничего не произойдет, и щелчком на кнопке Cancel вы можете выйти из приложения. Но в диспетчере задач или в приложении Process Explorer вы увидите процесс Notmyfault. Попытки его удаления ни к чему не приведут, так как Windows будет бесконечно ждать завершения IRP-пакета, поскольку драйвер Myfault не зарегистрировал процедуру отмены.

Воспользуемся отладчиком WinDbg, чтобы определить, чем занят поток. Откройте локальный сеанс в режиме ядра и воспользуйтесь командой !process для получения данных о процессе Notmyfault.exe:

```
lkd> !process 0 7 notmyfault.exe
PROCESS 86843ab0 SessionId: 1 Cid: 0594 Peb: 7ffd8000 ParentCid: 05c8
DirBase: ce21f380 ObjectTable: 9cfb5070 HandleCount: 33.
Image: NotMyfault.exe
VadRoot 86658138 Vads 44 Clone 0 Private 210. Modified 5. Locked 0.
DeviceMap 987545a8
...
```

продолжение ↗

```

THREAD 868139b8 Cid 0594.0230 Teb: 7ffdde000 Win32Thread: 00000000
WAIT: (Executive) KernelMode Non-Alertable
86797c64 NotificationEvent
IRP List:
86a51228: (0006,0094) Flags: 00060000 Mdl: 00000000
...
ChildEBP RetAddr Args to Child
88ae4b78 81cf23bf 868139b8 86813a40 00000000 nt!KiSwapContext+0x26
88ae4bbc 81c8fcf8 868139b8 86797c08 86797c64 nt!KiSwapThread+0x44f
88ae4c14 81e8a356 86797c64 00000000 00000000 nt!KeWaitForSingleObject+0x492
88ae4c40 81e875a3 86a51228 86797c08 86a51228 nt!IoCancelAlertedRequest+0x6d
88ae4c64 81e87cba 00000103 86797c08 00000000 nt!IoSyncrhonousServiceTail+0x267
88ae4d00 81e7198e 86727920 86a51228 00000000 nt!IoPxxxControlFile+0x6b7
88ae4d34 81c92a7a 0000007c 00000000 00000000 nt!NtDeviceIoControlFile+0x2a
88ae4d34 77139a94 0000007c 00000000 00000000 nt!KiFastCallEntry+0x12a
01d5fecc 00000000 00000000 00000000 00000000 ntdll!KiFastSystemCallRet
...

```

Данная трассировка стека показывает, что инициировавший ввод-вывод поток обнаружил завершение IRP-пакета (функция IoSyncrhonousServiceTail вызвала функцию IoCancelAlertedRequest) и теперь ожидает собственной 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:

```

1kd> !irp 86a51228
Irp is active with 1 stacks 1 is current (= 0x86a51298)
No Mdl: No System Buffer: Thread 868139b8: Irp stack trace.
cmd flg cl Device File Completion-Context
>[ e, 0] 5 0 86727920 86797c08 00000000-00000000
\Driver\MYFAULT
Args: 00000000 00000000 83360020 00000000

```

Отсюда видно, что причиной проблемы стал драйвер \Driver\MYFAULT, то есть Myfault.sys. Его имя (дословно — «моя вина») подчеркивает, что ситуация возникла из-за проблем с драйвером, а не из-за ошибок приложения. К сожалению, эта информация не дает нам возможности устраниТЬ проблему — требуется перезагрузка системы, так как Windows не в состоянии допустить, что еще не случившаяся отмена можно просто проигнорировать. IRP-пакет может в любой момент вернуть управление и повредить системную память. Если вы столкнетесь с подобной ситуацией в реальной жизни, попробуйте поискать более новую 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 числа потоков на сервере. К примеру, если сервер создаст для обработки запросов всего один программный поток, работа клиентов сильно замедлится, так как в единицу времени будет рассматриваться 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 себе создание сервером огромного пула потоков, когда для обработки каждого запроса выделяется отдельный программный поток. Такой сценарий обычно ведет к пробуксовке: множество потоков порождается, выполняет обработку данных и блокируется в ожидании ввода-вывода, а после обработки запроса блокируется в ожидании следующего. Наличие огромного числа потоков уже само по себе приводит к избыточному переключению контекста, возникающего из-за того, что планировщику приходится делить время процессора между всеми активными потоками.

Сервер должен, с одной стороны, уменьшить количество переключений контекста, заставив потоки избегать ненужной блокировки, а с другой — обеспечить максимальный параллелизм за счет большого числа 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 подобной схемы нужно, чтобы при блокировке потока, обрабатывающего клиентский запрос, в ожидании ввода или 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 дают такое преимущество, как параллелизм, то есть 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 число 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, то есть процессоры работают практически на полной мощности.

Применение портов завершения

На рис. 8.23 представлена высокоуровневая схема работы порта завершения. Порт завершения в Windows создается путем вызова 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), то есть следующий пакет получает поток, заблокированный 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 блокирующими потоками.

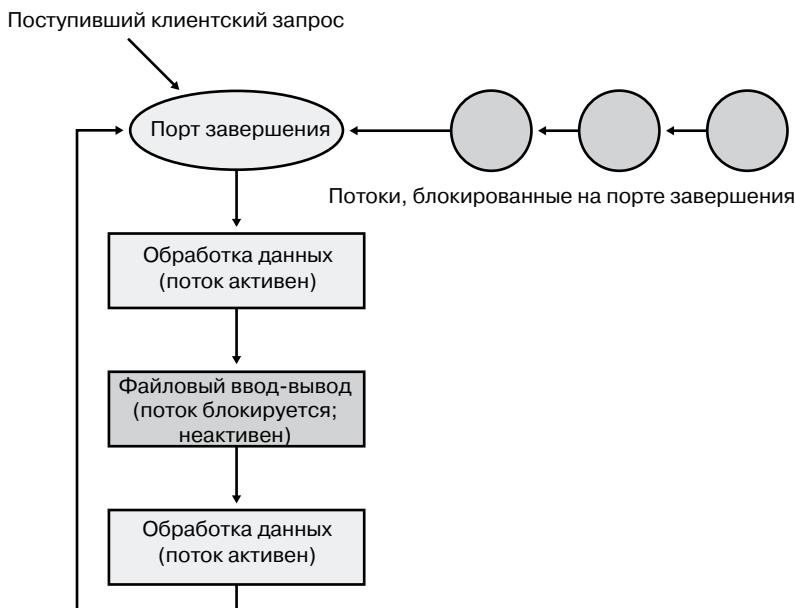


Рис. 8.23. Схема работы порта завершения ввода-вывода

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`. 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. Обнаружив подобные действия, Windows 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.

Согласно рекомендациям Microsoft, максимальное число активных потоков следует задавать примерно равным количеству процессоров в системе. Имейте в виду, что это число может быть превышено. Представьте, что вы задали максимальное значение чис-

ла активных потоков равным 1. При поступлении клиентского запроса активируется выделенный для его обработки поток. Если в данный момент поступит второй запрос, активировать второй поток, ожидающий на порте, будет невозможно, так как лимит уже достигнут. В какой-то момент произойдет блокировка первого потока из-за ожидания операции ввода или вывода, то есть он станет неактивным. Это приведет к освобождению второго потока. Но при этом завершится файловый ввод-вывод первого потока, то есть он снова станет активным. С этого момента — и до блокировки одного из потоков — число активных потоков превысит установленный предел на 1. В большинстве случаев количество активных потоков равно предельному или немного превышает его.

Предусмотренный для порта завершения прикладной программный интерфейс 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 сортами 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, вызывая API-функцию `CreateIoCompletionPort` и задавая в качестве дескриптора порта завершения значение `NULL`. В результате выполняется системная служба `NtCreateIoCompletion`. Объект исполнительной системы `IoCompletion` включает в себя синхронизирующий объект ядра, называемый *очередью ядра* (*kernel queue*). Соответственно, системная служба 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 начальной области памяти 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 служба `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.

Напоследок 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 путем вызова 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 служба `NtRemoveIoCompletion`. После проверки параметров и преобразования дескриптора порта завершения в указатель на этот порт данная служба вызывает функцию `IoRemoveIoCompletion`, которая, в конечном счете, вызывает функцию `KeRemoveQueueEx`. В высокопроизводительных сценариях возможно завершение набора 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. Изнутри это реализуется при помощи функции `NtRemoveIoCompletionEx`, которая вызывает функцию `IoRemoveIoCompletion`, оснащенную счетчиком количества позиций в очереди и передающую это значение в функцию `KeRemoveQueueEx`.

Как видите, работу порта завершения обеспечивают функции `KeRemoveQueueEx` и `KeInsertQueue`. Они определяют, не нужно ли активировать поток, ожидающий пакета завершения ввода-вывода. Объект очереди поддерживает внутренний 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. 8.24. 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.

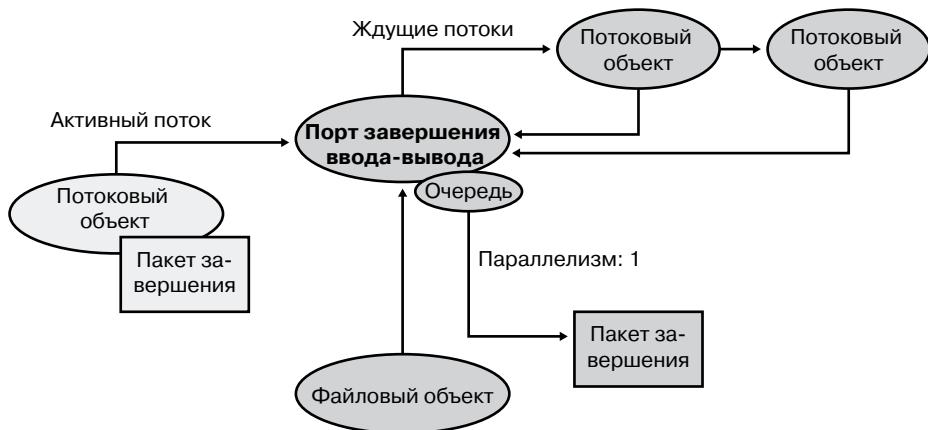


Рис. 8.24. Схема работы порта завершения ввода-вывода

Ну и, наконец, 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. Существуют три варианта оптимизации режима уведомлений (табл. 8.3). Следует заметить, что все они задаются на уровне дескриптора файла и не модифицируются.

Таблица 8.3. Режимы уведомлений порта завершения ввода-вывода

Режим уведомления	Значение
Пропуск порта завершения при успешном выполнении	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 <code>ERROR_PENDNG</code>
Пропуск события для дескриптора	Диспетчер ввода-вывода не задает событие для файлового объекта, если запрос возвращается с кодом успешного завершения или с ошибкой <code>ERROR_PENDNG</code> , а вызванная функция не относится к синхронным. Если для запроса предусмотрено явное событие, о нем сообщается
Пропуск пользовательского события при быстром вводе-выводе	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 <code>ERROR_PENDNG</code> , а вызываемая функция не является синхронной

Определение приоритетов ввода-вывода

Если не задать приоритеты ввода-вывода, фоновые операции, такие как индексация поиска, 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. 8.4, 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 8.4. Приоритеты ввода-вывода

Приоритет	Использование
Critical	Диспетчер памяти
High	Не используется
Normal	Нормальный ввод-вывод приложений
Low	Не используется
Very Low	Запланированные задания, служба Superfetch, дефрагментация, индексация 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 индексация 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 пол-секунды). 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, представленный на рис. 8.25. 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. 8.25. Пример глобальной очереди ввода-вывода

Пользовательские приложения задают приоритет ввода-вывода через три объекта. Объекты `SetPriorityClass` и `SetThreadPriority` устанавливают приоритет для

всех операций ввода-вывода, генерируемых процессом в целом или определенными программными потоками (этот приоритет хранится в IRP-пакете каждого запроса). Объект `SetFileInformationByHandle` задает приоритет 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 является не более чем рекомендацией. 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 (%SystemRoot%\System32\Ataport.sys) и USB (%SystemRoot%\System32\Usbstor.sys) поддерживают данную стратегию, а драйверы порта SCSI и накопителей (%SystemRoot%\System32\Scsiport.sys и %SystemRoot%\System32\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 поддерживать 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 приоритетов ввода-вывода, в то время как устройства на основе SCSI, Fibre Channel и iSCSI – нет.

В то же время стратегию на основе простого реализует драйвер класса хранилищ (%SystemRoot%\System32\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 простого 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 8.26 демонстрирует упрощенное представление стека драйверов внешней памяти и областей реализации обеих стратегий. Подробно стек драйверов внешней памяти рассматривается в главе 9.

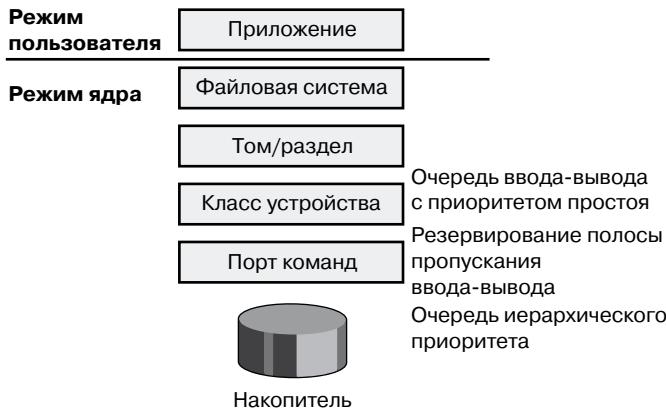


Рис. 8.26. Реализация приоритетов ввода-вывода в стеке драйверов внешней памяти

Предотвращение инверсии приоритетов ввода-вывода (наследование приоритетов ввода-вывода)

Чтобы избежать инверсии приоритетов ввода-вывода (из-за которого 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 изменением приоритетов.

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`.

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 выполнить 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.
- ❑ 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 `Superfetch` (которая всегда использует ввод-вывод с приоритетом простоя). В этом случае IRP-пакет будет иметь нормальный приоритет, даже если приоритет текущего потока ниже.

В следующем эксперименте вы увидите пример приоритета `Very Low` и просмотрите приоритеты различных запросов с помощью приложения Process Monitor.

ЭКСПЕРИМЕНТ: ПРОИЗВОДИТЕЛЬНОСТЬ ВВОДА-ВЫВОДА С ПРИОРИТЕТАМИ VERY LOW И NORMAL

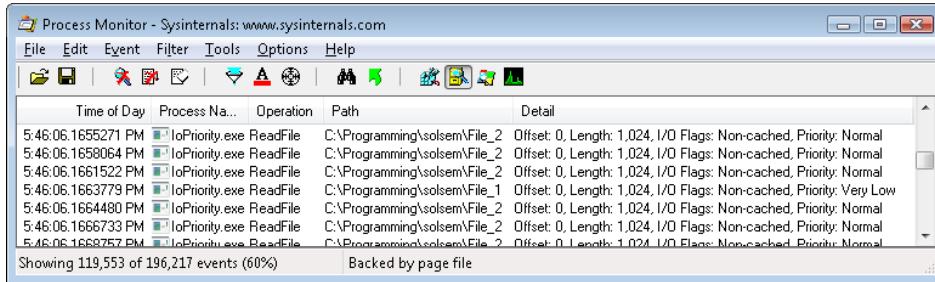
Для определения производительности программных потоков с различными приоритетами ввода-вывода можно воспользоваться приложением `IO Priority` (включенным в служебные программы для данной книги). Запустите файл `IoPriority.exe`, убедитесь, что для потока `Thread 1` установлен флагок `Low priority`, и щелкните на кнопке `Start IO`. Вы обнаружите заметное различие в быстродействии двух потоков, как показано на следующем рисунке.



Обратите внимание, что производительность потока `Thread 1` остается практически постоянной, в районе 2 Кбайт/с. Это можно легко объяснить тем, что приложение `IO Priority` выполняет его операции ввода-вывода на скорости 2 Кбайт/с в соответствии со стратегией выбора приоритета на основе простоя, гарантирующей, по меньшей мере, одну операцию ввода или вывода в полсекунды. В противном случае поток `Thread 2` задержал бы любую предпринятую потоком `Thread 1` попытку ввода или вывода.

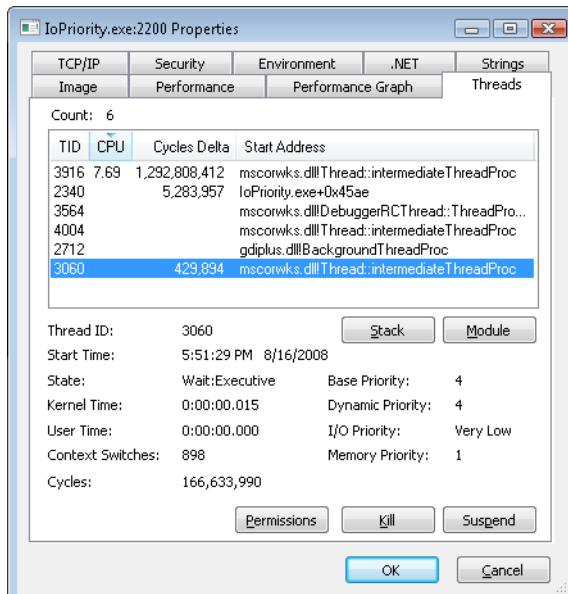
Обратите внимание, что при низком приоритете обоих потоков и относительном простое системы производительность потоков в этом примере будет примерно равна производительности одного потока с нормальным приоритетом ввода-вывода. Дело в том, что при отсутствии конкуренции с более высокоприоритетными потоками потоки с низким приоритетом ввода-вывода перестают искусственно ограничиваться или тормозиться.

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 поток Thread 1 осуществляет запись в файл File_1, а поток Thread 2 — в файл File_2. Прокрутите экран вниз, пока не обнаружите запись в File_1, и вы увидите примерно такой результат.



Видно, что связанный с файлом File_1 ввод-вывод имеет приоритет Very Low. Посмотрев в столбец Time Of Day, вы обнаружите, что все операции ввода-вывода разделены промежутком в 0,5 секунды — еще один признак стратегии на основе простоты.

Наконец, приложение Process Explorer позволяет идентифицировать поток Thread 1 в процессе IoPriority путем просмотра приоритетов ввода-вывода для всех потоков на вкладке Threads диалогового окна Properties данного процесса. Также можно заметить, что приоритет интересующего нас потока ниже заданного по умолчанию значения 8 (нормальный), что указывает на возможную работу потока в фоновом режиме. Вот примерно то, что вы должны увидеть.



Примечательно, что если приложение IO Priority задаст приоритет файла File_1, а не вызывающего его потока, оба потока будут выглядеть одинаково. И только приложение Process Monitor позволит увидеть разницу в приоритетах ввода-вывода.

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 IoPagingWriteHighPriorityBumpedCount;
- IoBoostedThreadedIrpCount и IoBoostedPagingIrpCount;
- IoBlanketBoostCount.

Для просмотра их значений используйте команду dd, выводящую содержимое памяти.

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 может быть размер пакетов.

Подсистема ввода-вывода не обслуживает другие операции ввода-вывода, пока не удовлетворены требования приложений, выполнивших резервирование на целевом накопителе. Рисунок 8.27 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 операциями придется подождать следующего цикла.



Рис. 8.27. 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, то есть 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 нашине 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. В Microsoft программа Driver Verifier применяется для проверки как собственных драйверов устройств производства Microsoft, так и присыпаемых производителями для тестирования в лаборатории Windows Hardware Quality Labs (WHQL). Это гарантирует совместимость драйверов с Windows и отсутствие в них распространенных ошибок. (Имеется также программа Application Verifier, позволившая повысить качество кода в пользовательском режиме Windows, но ее описание выходит за рамки темы данного издания.)

Главным образом, программа Driver Verifier используется как инструмент, помогающий разработчикам драйверов устройств обнаруживать ошибки в коде, но полезна она и системным администраторам при крахе системы. О ее роли в анализе системных сбоев рассказывается в главе 14.

Программа Driver Verifier поддерживается несколькими системными компонентами: параметры проверки драйверов, которые можно активировать, присутствуют в диспетчере памяти, диспетчере ввода-вывода и в HAL. Их конфигурирование осуществляется через диспетчер проверки драйверов (`%SystemRoot%\System32\Verifier.exe`).

При запуске программы Driver Verifier без аргументов командной строки появляется интерфейс, показанный на рис. 8.28.

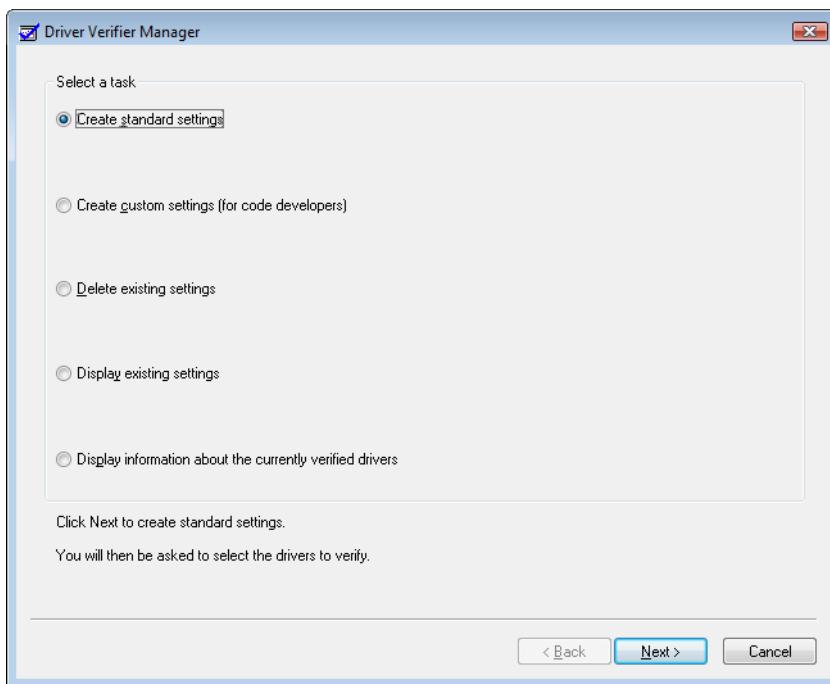


Рис. 8.28. Диалоговое окно Driver Verifier Manager

Включать и отключать программу Driver Verifier и просматривать ее текущие параметры можно также через интерфейс командной строки. Для просмотра доступных ключей используйте команду `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 свободной памяти, выделение места под объекты синхронизации в памяти типа `NonPagedPoolSession`, ссылки на освобожденные объекты, задержка выключения компьютера более чем на 20 минут и запросы блоков памяти нулевого размера.

Давайте посмотрим на описание параметров проверки операций, связанных с вводом-выводом (рис. 8.29). Параметры, относящиеся к управлению памятью, вместе с механизмом перенаправления диспетчером памяти системных вызовов драйвера к определенным версиям программы Driver Verifier описаны в главе 10.

Эти параметры оказывают следующее действие:

- I/O Verification (Проверка ввода-вывода). После установки этого флагка диспетчер ввода-вывода выделяет память под IRP-пакеты для проверенных драйверов из

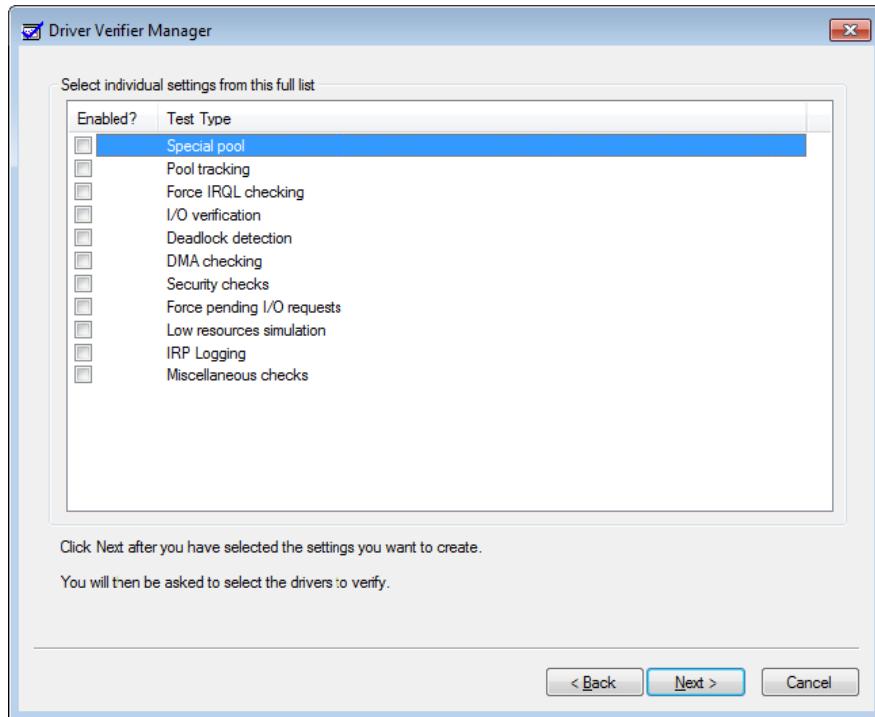


Рис. 8.29. Параметры программы Driver Verifier,
связанные с операциями ввода-вывода

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 флагка 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 онидерживаются 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 флагка 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 флагка 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 флагок 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`.

Среда KMDF

Набор инструментальных средств Windows Driver Foundation (WDF) уже обсуждался в главе 2 части I. Теперь мы более подробно рассмотрим компоненты и функциональность той части инфраструктуры, которая работает в режиме ядра, а именно — среды KMDF (Kernel-Mode Driver Framework). Впрочем, в этом разделе вы найдете только краткое описание базовой архитектуры KMDF. Детально данная тема рассматривается на странице <http://msdn.microsoft.com/en-us/library/windows/hardware/gg463370.aspx>.

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 обладать следующими возможностями:

- ❑ **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 обратный вызов функции `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 3 части I), являясь для инфраструктуры внутренними.

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 поддерживать эту операцию и предоставить обратный вызов, а может предпочесть KMDF-код, предлагаемый по умолчанию, который сообщает 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 KMDF-ДРАЙВЕРОВ

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 типичном компьютере с операционной системой Windows:

```
1kd> !wdfkd.wdfldr
LoadedModuleList 0xfffff880010682d8
-----
```

продолжение ↗

```

LIBRARY_MODULE ffffffa8002776120
Version v1.9 build(7600)
Service \Registry\Machine\System\CurrentControlSet\Services\Wdf01000
ImageName Wdf01000.sys
ImageAddress 0xfffffff88000c00000
ImageSize 0xa4000
Associated Clients: 16
ImageName Version WdfGlobals FxGlobals
ImageAddress ImageSize
peauth.sys v1.7(6001) 0xfffffff8004754210 0xfffffff80047540c0
0xfffffff880074cc000 0x0000a6000
scfilter.sys v1.5(6000) 0xfffffff8002ef34e0 0xfffffff8002ef3390
0xfffffff880040b3000 0x0000e0000
WinUSB.sys v1.9(7600) 0xfffffff8002eefd20 0xfffffff8002eefbd0
0xfffffff88004000000 0x00011000
monitor.sys v1.9(7600) 0xfffffff8004854a10 0xfffffff80048548c0
0xfffffff8800412a000 0x0000e0000
vmswitch.sys v1.5(6000) 0xfffffff8002de5d60 0xfffffff8002de5c10
0xfffffff88003e9b000 0x00068000
vmbus.sys v1.5(6000) 0xfffffff8002d7fcf0 0xfffffff8002d7fba0
0xfffffff88003e5f000 0x0003c000
Vid.sys v1.5(6000) 0xfffffff8002ddacf0 0xfffffff8002ddaba0
0xfffffff88002a00000 0x00033000
umbus.sys v1.9(7600) 0xfffffff8002e57e70 0xfffffff8002e57d20
0xfffffff880035db000 0x00012000
storvsp.sys v1.5(6000) 0xfffffff8002e48b10 0xfffffff8002e489c0
0xfffffff88003575000 0x00023000
CompositeBus.sys v1.9(7600) 0xfffffff8002d79160 0xfffffff8002d79010
0xfffffff88002936000 0x00010000
HDAudBus.sys v1.7(6001) 0xfffffff8002e357f0 0xfffffff8002e356a0
0xfffffff880037a9000 0x00024000
intelppm.sys v1.9(7600) 0xfffffff8002c518f0 0xfffffff8002c517a0
0xfffffff880027e7000 0x00016000
cdrom.sys v1.9(7600) 0xfffffff80028bf8f0 0xfffffff80028bf7a0
0xfffffff880011c4000 0x0002a000
vmstorfl.sys v1.5(6000) 0xfffffff8002b2cdd0 0xfffffff8002b2cc80
0xfffffff8800144a000 0x00010000
vdrvroot.sys v1.9(7600) 0xfffffff80027887c0 0xfffffff8002788670
0xfffffff8800139c000 0x0000d000
msisadrv.sys v1.9(7600) 0xfffffff80029c5430 0xfffffff80029c52e0
0xfffffff8800135f000 0x0000a000
-----
Total: 1 library loaded

```

Модель данных в KMDF

Основой модели данных в KMDF, как и модели ядра, являются объекты, но в данном случае не используется диспетчер объектов. Своими объектами KMDF управляет самостоятельно, предоставляем драйверам дескрипторы, но не показывая реальной структуры данных. Для каждого типа объектов в инфраструктуре существуют процедуры, выполняющие операции с объектами, например процедура `WdfDeviceCreate` создает устройство. Кроме того, у объектов могут присутствовать специальные поля данных или члены, доступ к которым осуществляется через API Get/Set (для модификаций,

которые должны осуществляться без сбоев) или Assign/Retrieve (для модификаций, в которых возможны сбои). Например, функция `WdfInterruptGetInfo` возвращает информацию о данном объекте прерывания (`WDF_INTERRUPT`).

В отличие от изолированных друг от друга объектов ядра, все KMDF-объекты принадлежат иерархии — большинство типов объектов связано с родителем. Корневым объектом является структура `WDF_DRIVER`, описывающая реальный драйвер. Построению и предназначению она аналогична предоставляемой диспетчером ввода-вывода структуре `DRIVER_OBJECT`, а все прочие KMDF-структуры являются ее потомками. Следующим по важности является объект `WDF_DEVICE`, 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. 8.5.

Таблица 8.5. Типы KMDF-объектов

Объект	Тип	Описание
Список потомков	<code>WDF_CHILD_LIST</code>	Список связанных с устройством потомков объекта <code>WDF_DEVICE</code> . Используется только драйверами шины
Коллекция	<code>WDF_COLLECTION</code>	Список объектов одного типа, например отфильтрованная группа объектов <code>WDF_DEVICE</code>
Отложенный вызов процедуры	<code>WDF_DPC</code>	Экземпляр DPC-объекта (подробно DPC-вызовы рассматриваются в главе 3 части I)
Устройство	<code>WDF_DEVICE</code>	Экземпляр устройства
Общий DMA-буфер	<code>WDF_COMMON_BUFFER</code>	Область памяти, к которой могут обращаться устройство и драйвер при прямом доступе к памяти
Средство включения DMA	<code>WDF_DMA_ENABLER</code>	Включает прямой доступ к памяти для драйвера на указанном канале
DMA-транзакция	<code>WDF_DMA_TRANSACTION</code>	Экземпляр DMA-транзакции
Драйвер	<code>WDF_DRIVER</code>	Корневой объект для драйвера; представляет драйвер с его параметрами, обратными вызовами и другими элементами
Файл	<code>WDF_FILE_OBJECT</code>	Экземпляр файлового объекта, который может использоваться в качестве канала между приложением и драйвером
Обобщенный объект	<code>WDF_OBJECT</code>	Позволяет поместить определенные драйвером нестандартные данные в объектную модель данных инфраструктуры в виде объекта

продолжение ↴

Таблица 8.5 (продолжение)

Объект	Тип	Описание
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
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
Сpin-блокировка	WDFSPINLOCK	Описывает spin-блокировку (подробно она рассматривается в главе 3 части I)
Строка	WDFSTRING	Описывает структуру Unicode-строки
Таймер	WDFTIMER	Описывает таймер исполнительной системы (см. главу 3 части I)
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	WDFWMINSTANCE	Представляет блок данных для объекта WDFWMIPROVIDER

Объект	Тип	Описание
WMI-провайдер	WDFWMIPROVIDER	Описывает схему WMI для всех поддерживаемых драйвером объектов WDFWMINSTANCE
Рабочий элемент	WDFWORKITEM	Описывает исполнительный рабочий элемент

К каждому из этих объектов можно присоединить другой KMDF-объект 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. 8.30 представлена иерархия KMDF-объектов.

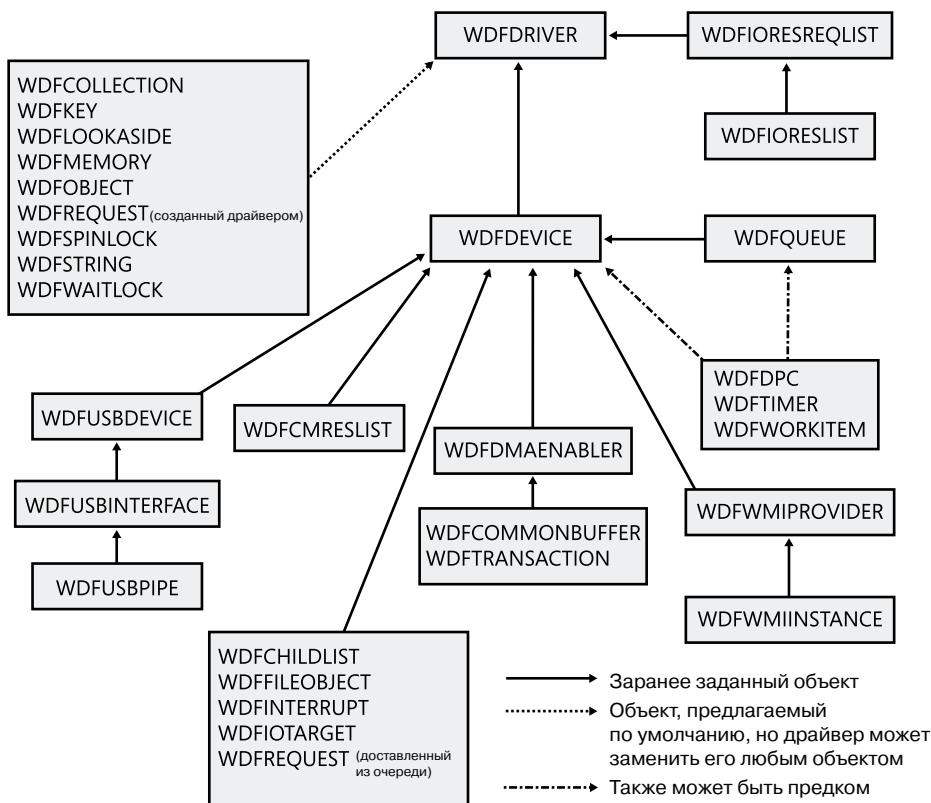


Рис. 8.30. Иерархия KMDF-объектов

Следует заметить, что упомянутые ранее и показанные на рисунке 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 инфраструктурой.

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-блокировок или строки. Подобная расширяемость позволяет отвечающим за обработку ввода-вывода библиотекам или слоям кода независимо взаимодействовать с другим кодом, основываясь на контекстной области, с которой они работают, поддерживая механизм, подобный наследованию.

Наконец, с KMDF-объектами связан набор атрибутов (табл. 8.6). 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 8.6. Атрибуты KMDF-объектов

Атрибут	Описание
<code>ContextSizeOverride</code>	Размер контекстной области объекта
<code>ContextTypeInfo</code>	Тип контекстной области объекта
<code>EvtCleanupCallback</code>	Обратный вызов, уведомляющий драйвер об очистке объекта перед его удалением (могут существовать ссылки на объект)
<code>EvtDestroyCallback</code>	Обратный вызов, уведомляющий драйвер о неминуемом удалении объекта (счетчик ссылок равен 0)
<code>ExecutionLevel</code>	Описывает максимальный IRQL-уровень, на котором KMDF может активировать обратные вызовы
<code>ParentObject</code>	Определяет предка объекта
<code>Size</code>	Размер объекта
<code>SynchronizationScope</code>	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. 8.31.

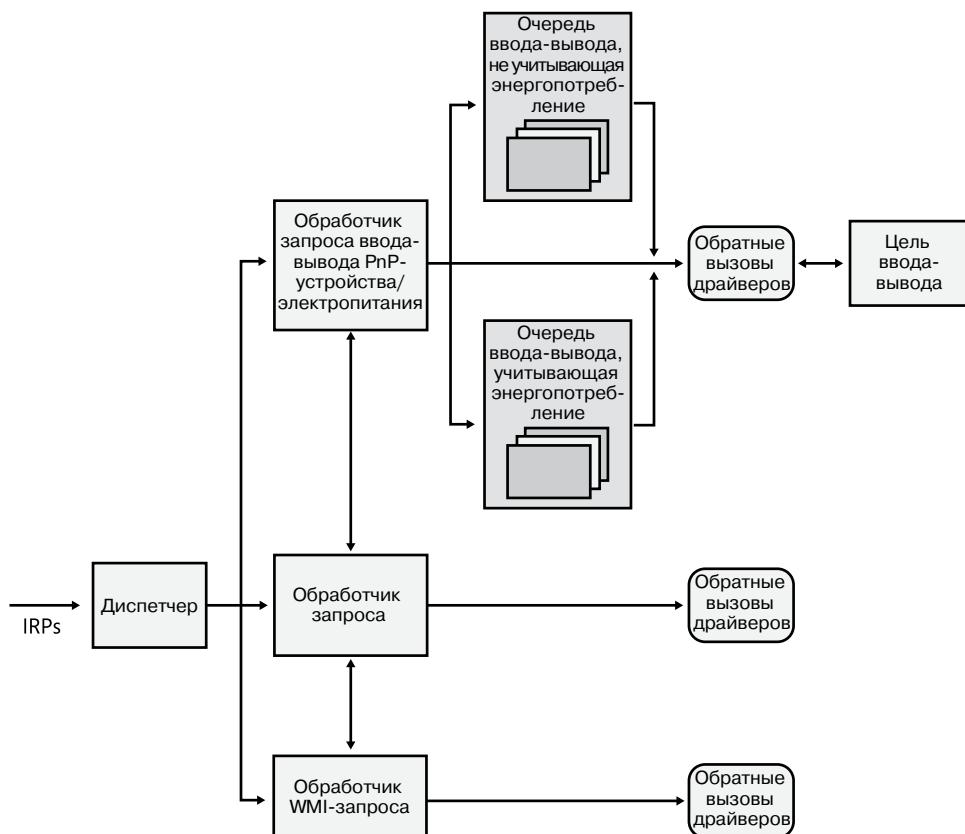


Рис. 8.31. Процесс ввода-вывода и обработка IRP-пакетов в KMDF

На основе механизма обработки IRP-пакетов для WDM-драйверов, о котором рассказывалось ранее, KMDF выполняет одно из следующих действий:

- ❑ Посыпает IRP-пакет обработчику ввода-вывода, выполняющему стандартные операции с устройствами.
- ❑ Посыпает IRP-пакет обработчику PnP-устройств и электропитания, который обслуживает эти события и уведомляет другие драйверы в случае изменения состояния.
- ❑ Посыпает 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 разделе, посвященном завершению ввода-вывода, а также в главе 3 части I). 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 одновременно) и синхронизацию ввода-вывода (как отмечено в списке атрибутов объектов в табл. 8.6). 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 в 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.

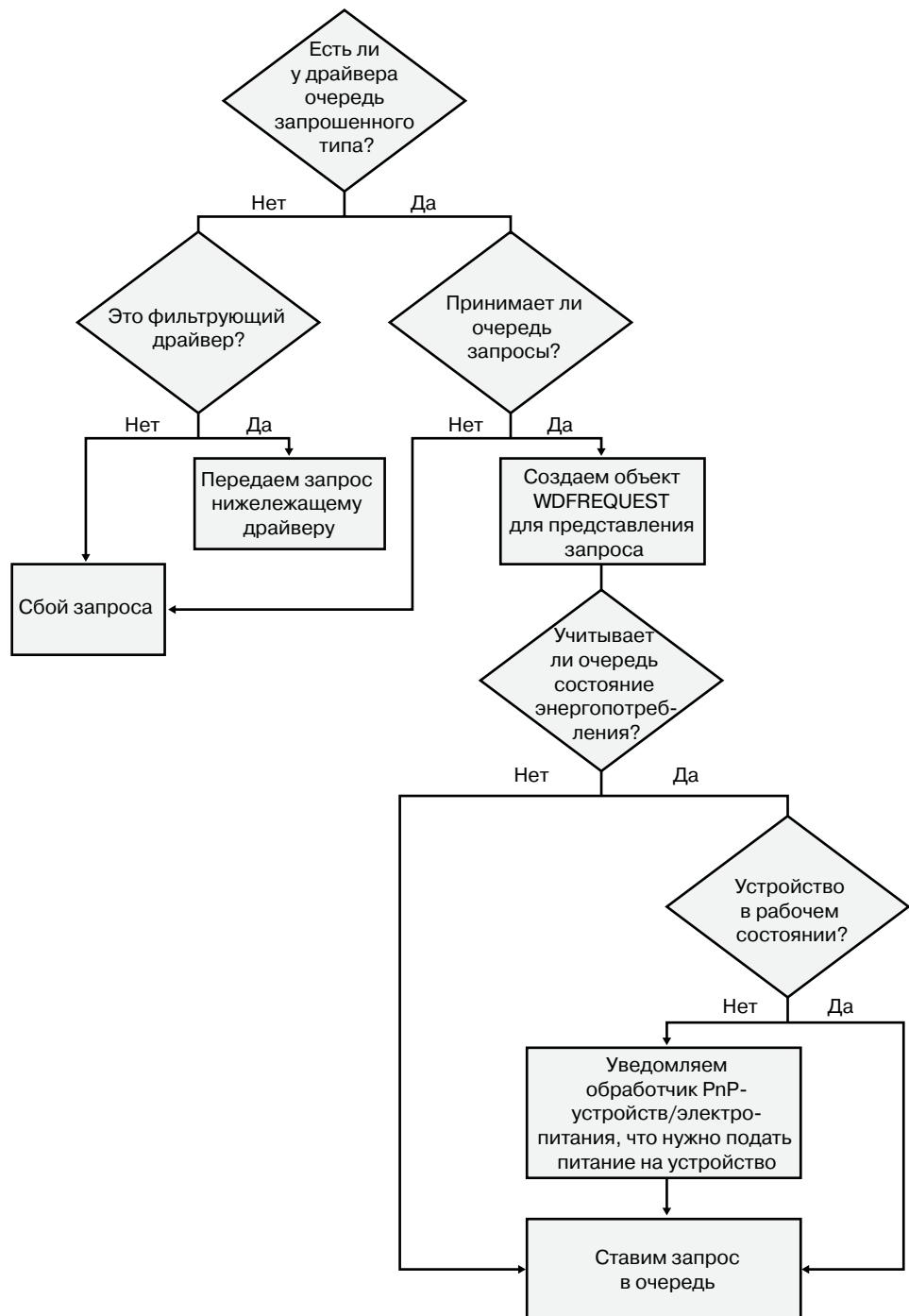


Рис. 8.32. Обработка в KMDF запросов ввода-вывода на чтение, на запись и на IOCTL

- ❑ 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 просит немедленного уведомления 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. 8.32 иллюстрирует ход выполнения запроса на ввод-вывод для KMDF-драйвера в случае наиболее распространенных операций (коды чтения, записи и управления вводом-выводом).

Среда UMDF

В этой главе основной упор делается на драйверы режима ядра, но в Windows появляется все больше драйверов, работающих в режиме пользователя через упоминавшуюся ранее среду UMDF (User-Mode Driver Framework), входящую в состав WDF. Перед завершением обсуждения драйверов сделаем небольшой обзор архитектуры и функциональности UMDF. Более подробную информацию по данной теме вы найдете на странице <http://msdn.microsoft.com/en-us/library/windows/hardware/gg463370.aspx>.

Среда UMDF специально разработана для поддержки так называемых *классов устройств протокола* (protocol device classes). Этот термин относится к устройствам, использующим один и тот же стандартизованный, 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 музыкаль-

ные 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.

С целью упростить передачу кода из режима ядра в режим пользователя и для поддержания целостности архитектуры в UMDF используется та же концептуальная модель программирования драйверов, что и в KMDF, но с другими компонентами, интерфейсами и структурами данных. К примеру, в KMDF входят объекты, существующие только в режиме ядра, в то время как UMDF включает в себя объекты, присутствующие только в режиме пользователя. В UMDF доступны прямая обработка прерываний, DMA, невыгружаемый пул, поддерживаются строгие требования к синхронизации. Более того, UMDF-драйвер не может находиться в стеке драйверов ядра или быть клиентом другого драйвера или самого ядра.

В отличие от KMDF-драйверов, работающих как драйверные объекты, представляющие файл образа (с расширением .sys), UMDF-драйверы работают в хост-процессе драйвера, аналогичном хост-процессу службы. Хост-процесс включает в себя сам драйвер (реализованный в виде внутреннего COM-компоненты), инфраструктуру драйвера в режиме пользователя (реализованную как динамически подключаемая библиотека, содержащая для каждого UMDF-объекта COM-подобные компоненты) 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 общается со средой исполнения в хост-процессах драйвера режима пользователя (ALPC описывается в главе 3 части I). Архитектура модели UMDF-драйвера показана на рис. 8.33.

На рисунке 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 KMDF- или WDM-устройства. Приложения не подозревают, что общение с устройствами происходит на базе UMDF, и вызовы отправляются в диспетчер ввода-вывода ядра.
- **Ядро Windows (диспетчер ввода-вывода)** формирует IRP-пакеты для операций, как и любое стандартное устройство.
- **Отражателем** называется стандартный фильтрующий WDM-драйвер, расположенный поверх стека каждого устройства, управляемого UMDF-драйвером. 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 адекватное время ответа на запросы и предотвращая зависание драйверов и приложений.

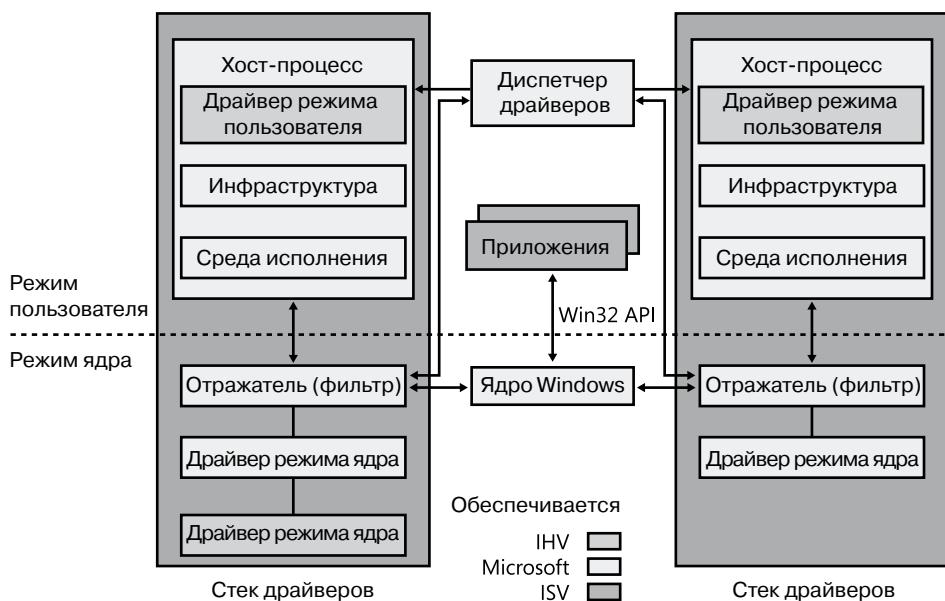


Рис. 8.33. UMDF-архитектура

- ❑ **Диспетчер драйверов** отвечает за запуск и остановку хост-процессов драйвера в зависимости от имеющихся UMDF-устройств, а также за управление содержащейся 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 драйвера функционирует как стандартная Windows-служба 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 реальным драйверам пространство адресов и среду выполнения. 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 обеспечение оборудования стеком драйверов внешней памяти в режиме пользователя, видимым всем приложениям в системе. В текущем выпуске UMDF каждый экземпляр устройства имеет собственный стек драйверов внешней памяти, запущенный в отдельном хост-процессе. В будущем набор экземпляров сможет 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 процесс **WUDFHost.exe**, соответствующий хост-процессу драйвера. Переключитесь на вывод DLL-библиотек и воспользуйтесь прокруткой, чтобы найти библиотеки, аналогичные показанным на рис. 8.34.

The screenshot shows two windows from the Windows Task Manager. The top window is titled 'Process' and lists several processes along with their PID, CPU usage, description, and company name. The bottom window is titled 'Name' and 'Description' and lists various DLL files and their descriptions, also showing their company names. Both windows show that the DLLs are loaded into the host process WUDFHost.exe.

Process	PID	CPU	Description	Company Name
svchost.exe	504		Host Process for Windo...	Microsoft Corporation
wlanext.exe	1408		Windows Wireless LAN ...	Microsoft Corporation
dwm.exe	2940		Desktop Window Manag...	Microsoft Corporation
WUDFHost.exe	16140		Windows Driver Founda...	Microsoft Corporation
svchost.exe	578		Host Process for Windo...	Microsoft Corporation
taskeng.exe	1632		Task Scheduler Engine	Microsoft Corporation
taskeng.exe	2404		Task Scheduler Engine	Microsoft Corporation
taskeng.exe	18008		Task Scheduler Engine	Microsoft Corporation

Name	Description	Company Name
WMASF.DLL	Windows Media ASF DLL	Microsoft Corpration
wmvcore.dll	Windows Media Playback/Authoring DLL	Microsoft Corpration
WpdRapi2.dll	Windows Mobile WPD Rapi Driver	Microsoft Corpration
WS2_32.dll	Windows Socket 2.0 32-Bit DLL	Microsoft Corpration
wshtcpip.dll	Winsock2 Helper DLL (TL/IPv4)	Microsoft Corpration
WSOCK32.dll	Windows Socket 32-Bit DLL	Microsoft Corpration
WTSAPI32.dll	Windows Terminal Server SNK APIs	Microsoft Corpration
WUDFHost.exe	Windows Driver Foundation - User-mode Driver...	Microsoft Corpration
WUDFHost.exe.mui	Windows Driver Foundation - User-mode Driver...	Microsoft Corpration
WUDFPlatform.dll	Windows Driver Foundation - User-mode Platfor...	Microsoft Corpration
WUDFx.dll	WDF:UMDF Framework Library	Microsoft Corpration

Рис. 8.34. DLL в хост-процессе среды UMDF

Можно выделить три основных компонента, соответствующих описанной архитектуре:

- **WUDFx.dll** — сама инфраструктура;
- **WUDFPlatform.dll** — среда выполнения;
- **WpdRapi2.dll** — COM-компонент, представляющий WPD-драйвер, который демонстрирует содержимое USB-накопителя в Windows-оболочке и медиа-приложениях.

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 стыковочную станцию портативному компьютеру с системой Windows автоматически обнаружить дополнительные устройства стыковочной 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 кшине 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.
- ❑ Еще одной функцией PnP-диспетчера является загрузка соответствующих драйверов. 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 накопителей 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 лишить систему возможности использовать другие устройства. В табл. 8.7 систематизированы результаты различных комбинаций устройств и драйверов, как с поддержкой технологии Plug and Play, так и без таковой.

Таблица 8.7. 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 стыковочного узла, засыпание 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 перераспределить 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.)

Поддержка технологии Plug and Play со стороны драйвера

Для поддержки технологии Plug and Play в драйвере должны быть реализованы процедуры PnP-диспетчеризации, диспетчеризации управления электропитанием (описанной в разделе «Диспетчер электропитания») и добавления устройства. Кроме того, драйверы шин должны поддерживать типы PnP-запросов, отличные от поддерживаемых функциональными и фильтрующими драйверами. Например, при перечислении устройств в процессе загрузки (этот процесс подробно описывается далее) PnP-диспетчер запрашивает у драйверов шин описание обнаруженных им на шинах

устройств. В это описание входят данные, уникальным образом идентифицирующие каждое устройство, а также требования устройств к аппаратным ресурсам. На основе этой информации PnP-диспетчер загружает функциональные или фильтрующие драйверы, установленные для обнаруженных устройств. Затем для каждого из них он вызывает процедуру добавления.

Во время процедуры добавления устройства функциональные и фильтрующие драйверы готовятся к управлению своими устройствами, но пока еще не вступают с ними в контакт. Они ждут от PnP-диспетчера команды `start-device`, которая должна быть передана их процедуре диспетчеризации. До отправки этой команды PnP-диспетчер выполняет арбитраж ресурсов, распределяя ресурсы между устройствами. Команда `start-device` содержит данные о результатах этого распределения. Получив команду `start-device`, драйвер настраивает свое устройство на использование указанных ресурсов. Если приложение пытается открыть устройство, запуск которого еще не завершен, ему возвращается сообщение об ошибке, указывающее, что устройства не существует.

После запуска устройства PnP-диспетчер может посыпать драйверу дополнительные PnP-команды, в том числе относящиеся к удалению устройства из системы или перераспределению ресурсов. Например, после запуска пользователем программы удаления/извлечения устройства (для этого следует щелкнуть правой кнопкой мыши на значке USB connector панели задач и выбрать команду `Eject USB Mass Storage Device`), как показано на рис. 8.35, для извлечения флэш-диска с USB-интерфейсом PnP-диспетчер посыпает уведомление `query-remove` всем приложениям, зарегистрированным для получения PnP-уведомлений на этом устройстве. Как правило, приложения регистрируются для получения уведомлений через свои дескрипторы, которые закрываются при получении уведомления `query-remove`. Если ни одно из приложений не налагает на запрос `query-remove` запрета, PnP-диспетчер посыпает команду `query-remove` драйверу, управляющему извлекаемым устройством. На этом этапе драйвер может отказаться от удаления устройства, чтобы завершить на нем все операции ввода-вывода и прекратить дальнейший прием направляемых устройству запросов на ввод и вывод. Если драйвер отвечает согласием на запрос об удалении, а открытые дескрипторы на устройство отсутствуют, PnP-диспетчер отправляет драйверу команду `remove`, требующую прекратить обращение к устройству и освободить все выделенные для него ресурсы.

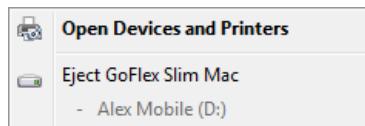


Рис. 8.35. Служба удаления/извлечения

Когда возникает необходимость перераспределить ресурсы для устройства, PnP-диспетчер сначала запрашивает драйвер, может ли он временно приостановить операции на устройстве, посыпая ему команду `query-stop`. Если нет риска потерять или повредить данные, драйвер отвечает на запрос согласием. В противном случае запрос

отклоняется. Как и в случае с командой `query-remove`, согласившись на выполнение запроса, драйвер заканчивает незавершенные операции ввода-вывода и перестает передавать устройству запросы на ввод и вывод. Новые запросы драйвер обыч но ставит в очередь, чтобы перегруппировка ресурсов была прозрачной для обращающихся в данный момент к устройству приложений. Затем PnP-диспетчер посыпает драйверу команду `stop`. На этом этапе он может заставить драйвер выделить устройству другие ресурсы, а потом снова послать ему команду `start-device`.

По сути, различные PnP-команды вызывают переход устройства в определенные состояния, формируя четкую таблицу переходных состояний, которая в упрощенной форме представлена на рис. 8.36. (Некоторые возможные переходы и PnP-команды для ясности опущены. Кроме того, эта диаграмма состояний реализуется функциональными драйверами. В случае с драйверами шин диаграмма выглядит гораздо сложнее.) На рисунке фигурирует еще не рассмотренное нами состояние, возникающее после выполнения команды `surprise-remove`. Данная команда возникает, когда пользователь без предупреждения удаляет устройство из системы, например при извлечении платы PCMCIA без запуска службы удаления/извлечения или в случае сбоя. Команда `surprise-remove` заставляет драйвер немедленно прекратить все взаимодействия с устройством, так как оно больше не подключено к системе. Все незавершенные запросы на ввод и вывод при этом отменяются.

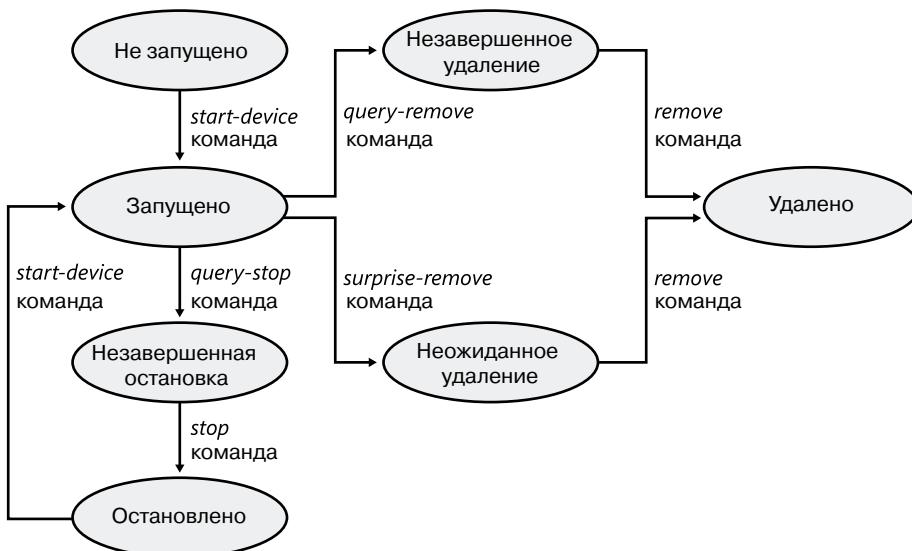


Рис. 8.36. Переходы между состояниями PnP-устройства

Загрузка, инициализация и установка драйвера

Загружать и инициализировать драйверы в Windows можно двумя способами: явным образом и на основе перечисления. Явную загрузку определяет ветвь реестра `HKEY_LOCAL_MACHINE\`

SYSTEM\CurrentControlSet\Services, как описано в разделе «Приложения службы» главы 4 части I. Второй вариант возникает, когда PnP-диспетчер динамически загружает драйверы для устройств, о наличии которых сообщает драйвер шины.

Параметр Start

Как отмечено в главе 4 части I, у каждого драйвера и у каждой службы в Windows есть свой раздел в ветви реестра Services текущего набора управляющих параметров. В этом разделе содержатся параметры, задающие тип образа (например, Windows-служба, драйвер или файловая система) и путь к файлу образа драйвера или службы, а также контролирующие порядок загрузки драйвера или службы. Между явной загрузкой драйвера устройства и загрузкой Windows-службы есть два основных различия:

- ❑ Только для драйверов устройств можно указать значения параметра **Start**: 0 – запуск при загрузке системой; 1 – запуск системы.
- ❑ Драйверы устройств могут использовать параметры **Group** и **Tag** для управления порядком загрузки при запуске системы, но в отличие от служб они не могут указывать параметры **DependOnGroup** и **DependOnService**.

В главе 13 описываются этапы загрузки и объясняется, что равный нулю параметр **Start** драйвера указывает на загрузку драйвера загрузчиком операционной системы. Единичное значение указывает, что драйвер загружается диспетчером ввода-вывода после инициализации компонентов исполнительной подсистемы. Диспетчер ввода-вывода вызывает процедуры инициализации драйвера в порядке загрузки драйверов при запуске системы. Подобно Windows-службам, драйверы пользуются параметром **Group** в своем разделе реестра для указания группы, к которой они принадлежат; порядок загрузки групп на этапе запуска системы определяется параметром HKLM\SYSTEM\CurrentControlSet\Control\ServiceGroupOrder\List.

Еще больше детализировать порядок своей загрузки драйвер может при помощи параметра **Tag**, указывающего положение драйвера в группе. Диспетчер ввода-вывода сортирует драйверы в группе именно по значениям этого параметра, заданным в разделах реестра драйверов. Не имеющие этого параметра драйверы перемещаются в конец группы. Можно предположить, что сначала диспетчер ввода-вывода инициализирует драйверы с меньшими значениями параметра **Tag**, но это может быть и не так. Приоритет параметров **Tag** в группе определен в разделе HKLM\SYSTEM\CurrentControlSet\Control\GroupOrderList. Данный раздел предоставляет специалистам из Microsoft и разработчикам драйверов свободу в определении собственной системы целых чисел.

Вот правила, согласно которым драйверы устанавливают параметр **Start**:

- ❑ Не поддерживающие технологию Plug and Play драйверы присваивают этому параметру такое значение, чтобы система загружала их на определенном этапе своего запуска.
- ❑ Драйверы, загружаемые системным загрузчиком при запуске операционной системы, присваивают параметру **Start** значение 0. В качестве примера можно привести драйверы системных шин и драйвер загрузки файловой системы.

- Драйвер, который не требуется для загрузки системы и который распознает устройство, не перечисляемое драйвером системной шины, присваивает параметру **Start** значение 1. В качестве примера можно привести драйвер последовательного порта, информирующий PnP-диспетчера о присутствии стандартных последовательных портов, обнаруженных программой установки и записанных в реестр.
- Драйвер, не поддерживающий технологию Plug and Play, или драйвер файловой системы, не обязательный для ее загрузки, присваивает параметру **Start** значение 2. В качестве примера можно привести драйвер многосетевого UNC-провайдера (Multiple UNC Provider), поддерживающий UNC-имена удаленных ресурсов (такие, как \\REMOTECOMPUTERNAME\SHARE).
- PnP-драйверы, не требующиеся для загрузки системы, присваивают параметру **Start** значение 3 (запуск по требованию). В качестве примера можно привести драйверы сетевых адаптеров.

Единственным предназначением параметра **Start** для PnP-драйверов и драйверов перечисляемых устройств является загрузка драйвера через загрузчик операционной системы, если такой драйвер нужен для ее успешного запуска. Кроме этого процесс перечисления устройств PnP-диспетчера, о котором речь пойдет далее, определяет порядок загрузки PnP-драйверов.

Перечисление 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) и отражающее 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. 8.37. Эта система ACPI-совместима, поэтому перечислителем основной шины является уровень аппаратных абстракций, соответствующий спецификации ACPI (Advanced Configuration and Power Interface — усовершенствован-

ванный интерфейс управления конфигурированием и энергопотреблением). Основной является шина PCI, к которой подсоединены шины USB, ISA и SCSI.

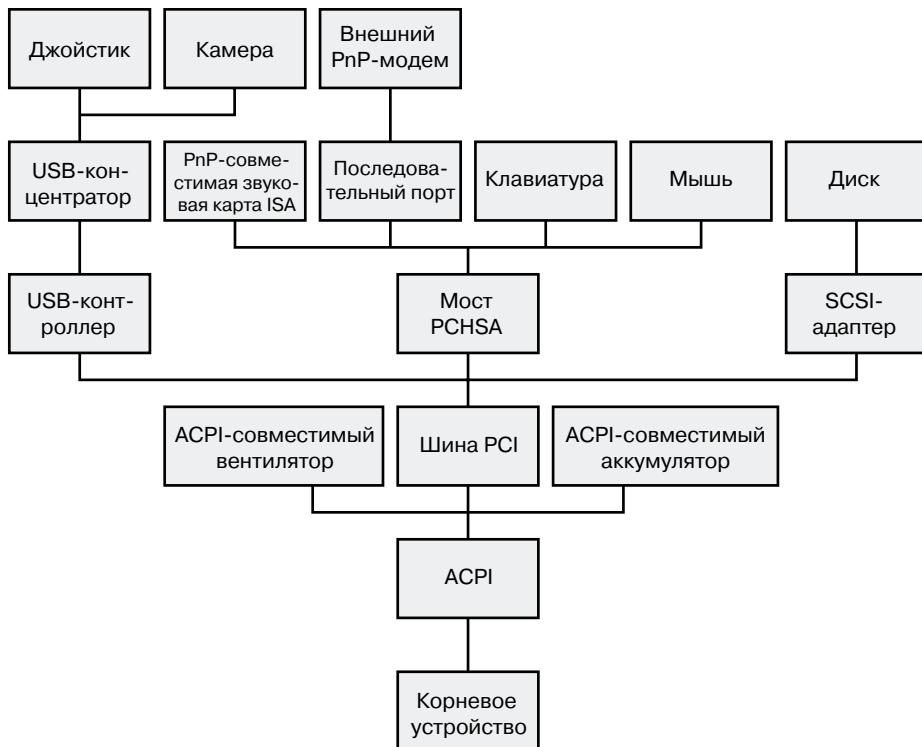


Рис. 8.37. Пример дерева устройств

Служебная программа Device Manager (Диспетчер устройств), доступная из оснастки Computer Management (Управление компьютером), которая вызывается командой Programs ▶ Administrative Tools (Программы ▶ Администрирование) меню Start (Пуск), а также по ссылке Device Manager (Диспетчер устройств) панели управления, демонстрирует список устройств в конфигурации, предлагаемой по умолчанию. Кроме того, можно выбрать в меню View (Вид) диспетчера устройств вариант Devices By Connection (Устройства по подключению) и представить список в виде иерархического дерева, как показано на рис. 8.38.

С учетом перечисления устройств загрузка и инициализация драйверов происходит в следующем порядке:

1. Диспетчер ввода-вывода вызывает входную процедуру каждого драйвера, запускаемого при загрузке системы. При наличии у такого драйвера дочерних устройств диспетчер ввода-вывода перечисляет их, сообщая о них PnP-диспетчеру. Дочерние устройства конфигурируются и запускаются, если их драйверы запускаются при загрузке системы. Если же драйвер такого устройства не запускается при загрузке

системы, PnP-диспетчер создает для этого устройства узел, но не запускает его и не загружает его драйвер.

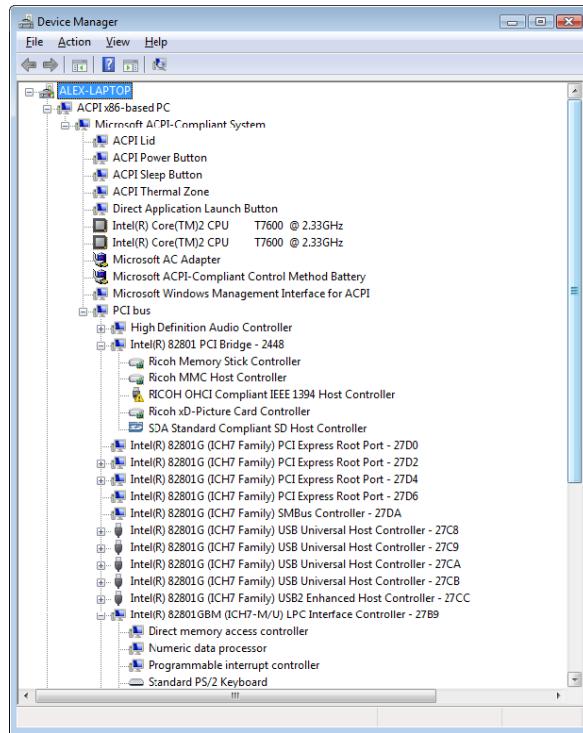


Рис. 8.38. Вид дерева устройств в диспетчере устройств

2. После инициализации запускаемых при загрузке системы драйверов PnP-диспетчер проходит по дереву устройств, загружая драйверы для узлов, не загруженных на первом этапе, и запускает соответствующие устройства. При запуске каждого из них PnP-диспетчер перечисляет связанные с ними дочерние устройства, если таковые имеются. Для этого он запускает соответствующие драйверы и при необходимости перечисляет их дочерние устройства. На данном этапе PnP-диспетчер загружает драйверы для обнаруженных устройств вне зависимости от значения их параметра **Start**. (Единственным исключением является значение **disabled**.) В конце этого этапа драйверы всех PnP-устройств загружены и работают. Исключением являются драйверы не перечисляемых устройств и их дочерних устройств.
3. PnP-диспетчер загружает все еще не загруженные драйверы со значением параметра **Start**, равным 1. Эти драйверы определяют не перечисляемые обычным образом устройства и сообщают о них. После этого PnP-диспетчер загружает драйверы для этих устройств, пока все они не пройдут этап конфигурирования и запуска.
4. Диспетчер управления службами загружает автоматически запускаемые драйверы.

Дерево устройств используется как PnP-диспетчером, так и диспетчером электропитания при выдаче устройствам IRP-пакетов, связанных с самонастройкой и управлением электропитанием. Как правило, поток IRP-пакетов распространяется от верхней части узла устройства вниз, и в некоторых случаях драйвер в одном из узлов создает новые IRP-пакеты для передачи другим узлам (всегда в направлении корня). О потоках этих IRP-пакетов мы поговорим чуть позже.

Все устройства, обнаруженные после установки системы, регистрируются в подразделах раздела HKLM\SYSTEM\CurrentControlSet\Enum. Им присваиваются имена вида <Перечислитель>\<ID устройства>\<ID экземпляра>. Здесь перечислитель — драйвер шины, ID устройства — уникальный идентификатор устройств данного типа, а ID экземпляра — уникальный идентификатор различных экземпляров одного устройства.

ЭКСПЕРИМЕНТ: ПРОСМОТР ДАМПА ДЕРЕВА УСТРОЙСТВ

Более подробно, чем в диспетчере устройств, рассмотреть дерево устройств можно при помощи команды !devnode отладчика ядра. Указав параметр 0 1, вы получите дамп внутренних структур узлов дерева. При этом элементы структур выводятся с отступами, отражающими их положение в общей иерархии:

```
1kd> !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». Каждый стек устройств имеет верх и низ, и, как показано на рис. 8.39, он состоит по меньшей мере из двух, а иногда и большего числа объектов устройств.

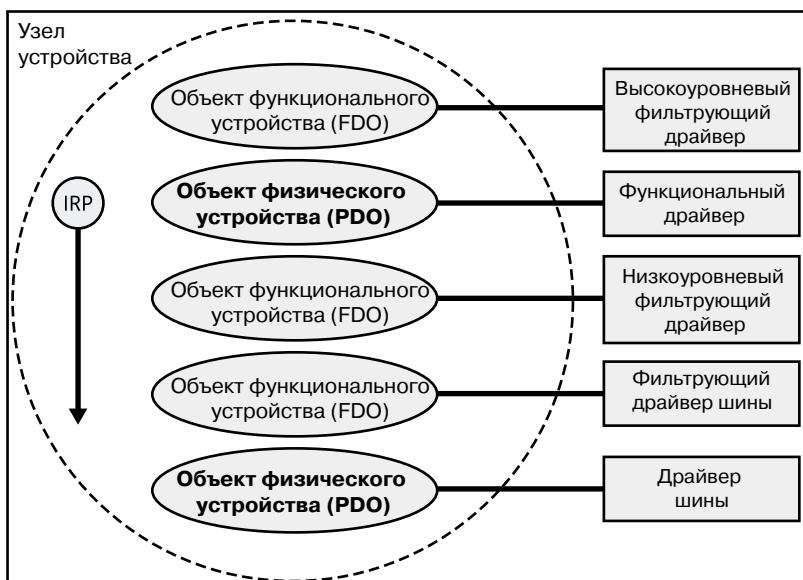


Рис. 8.39. Внутреннее строение стека устройств

- *Объект физического устройства* (Physical Device Object, PDO) создается драйвером шины по заданию PnP-диспетчера, когда этот драйвер, перечисляя устройства на своейшине, 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.
- Один или несколько необязательных объектов фильтрующих устройств (FiDO), которые находятся между PDO и FDO, но выше объектов фильтрующих устройств, созданных фильтрующими драйверами шин, создаются низкоуровневыми фильтрующими драйверами.
- 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. 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 устройство 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.

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. Например, функциональный драйвер может обработать запрос на чтение, не пересылая IRP-пакет драйверу шины. IRP-пакет проходит весь путь сверху вниз и далее к узлу, содержащему драйвер шины, только если функциональному драйверу требуется помочь драйвера шины.

Загрузка драйверов для стека устройств

До сих пор мы не ответили на два важных вопроса: как PnP-диспетчер определяет, какой функциональный драйвер следует загрузить для конкретного устройства и как фильтрующие драйверы регистрируют свое присутствие, чтобы их можно было загружать при создании стека устройств?

Ответ на оба вопроса нужно искать в реестре. Перечисляя устройства, драйвер шины сообщает PnP-диспетчеру их идентификаторы, которые зависят от конкретной шины. Например, для шины USB идентификатор составляется из *идентификатора продукта* (Product ID, PID), который производитель присваивает устройству, и собственно *идентификатора производителя* (Vendor ID, VID). (Подробно форматы

идентификаторов устройств рассматриваются в 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 ветви перечисления (`HKEY\SYSTEM\CurrentControlSet\Enum`). Пример такого раздела для клавиатуры показан на рис. 8.40. Раздел содержит данные, характеризующие устройство, и получаемые из INF-файла параметры *Service* и *ClassGUID*, помогающие PnP-диспетчеру локализовать драйверы данного устройства.

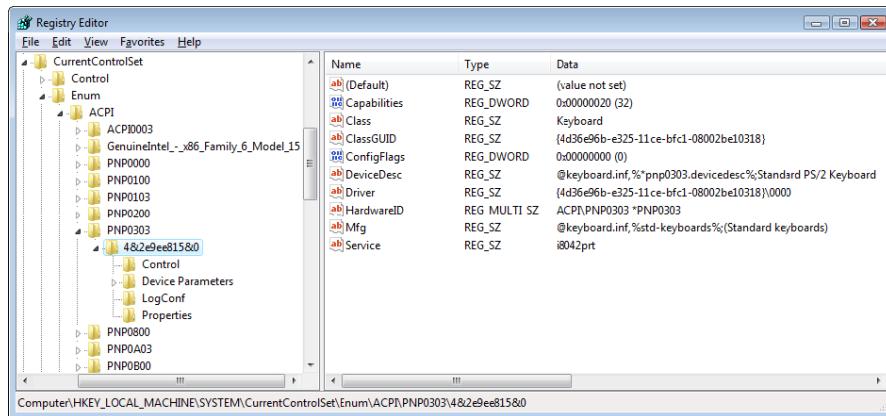


Рис. 8.40. Созданный в процессе перечисления раздел реестра для клавиатуры

Для работы с многофункциональными устройствами (такими, как некоторые принтеры или сотовые телефоны с интегрированными фотокамерами и плеерами) Windows поддерживает *идентификаторы контейнеров* (container ID), привязываемые к узлу устройства. Это уникальный для конкретного экземпляра физического устройства глобальный уникальный идентификатор (GUID), используемый всеми принадлежащими этому устройству функциональными узлами, как показано на рис. 8.41.

Идентификатор контейнера, как и идентификатор экземпляра, сообщается драйвером шины соответствующего аппаратного обеспечения. После перечисления 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.

**Многофункциональное устройство**

- Принтер
- Сканер
- Факс

Рис. 8.41. Многофункциональный принтер с уникальным с точки зрения PnP-диспетчера идентификатором

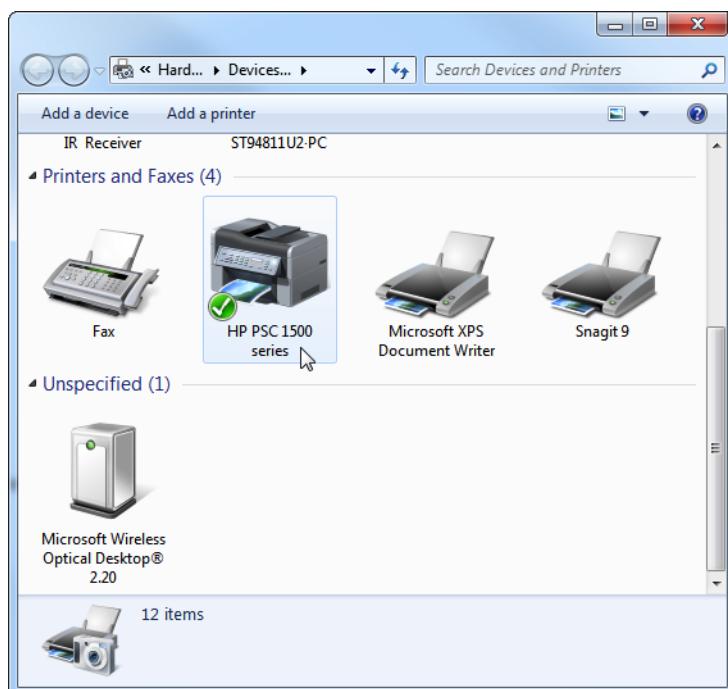


Рис. 8.42. Устройства и принтеры

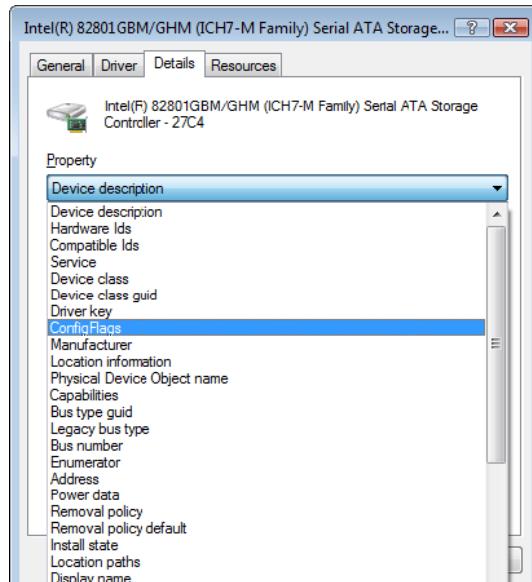
Наконец, в ситуации, когда драйверы устройств не предоставляют идентификатор контейнера, Windows может сделать обоснованное предположение, по возможности запросив топологию шины через такой механизм, как ACPI. Поняв, является ли определенное устройство потомком другого, относится ли оно к сменным, допускает 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 Devices And Printers (Устройства и принтеры) 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. 8.42 принтер HP PSC 1500 series идентифицируется как одно устройство.

ЭКСПЕРИМЕНТ: ПРОСМОТР ДЕТАЛЬНЫХ СВЕДЕНИЙ ОБ УЗЛАХ УСТРОЙСТВ В ДИСПЕТЧЕРЕ УСТРОЙСТВ

По умолчанию в окне Device Manager (Диспетчер устройств), доступном по ссылке Hardware (Оборудование) окна свойств системы, выводятся подробные сведения об узле устройства на вкладке Details (Сведения). 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 Details (Сведения), демонстрирующий доступную вам информацию.



Параметр **ClassGUID** позволяет PnP-диспетчеру локализовать раздел класса устройства в ветви **HKLM\SYSTEM\CurrentControlSet\Control\Class**. Раздел класса клавиатур показан на рис. 8.43.

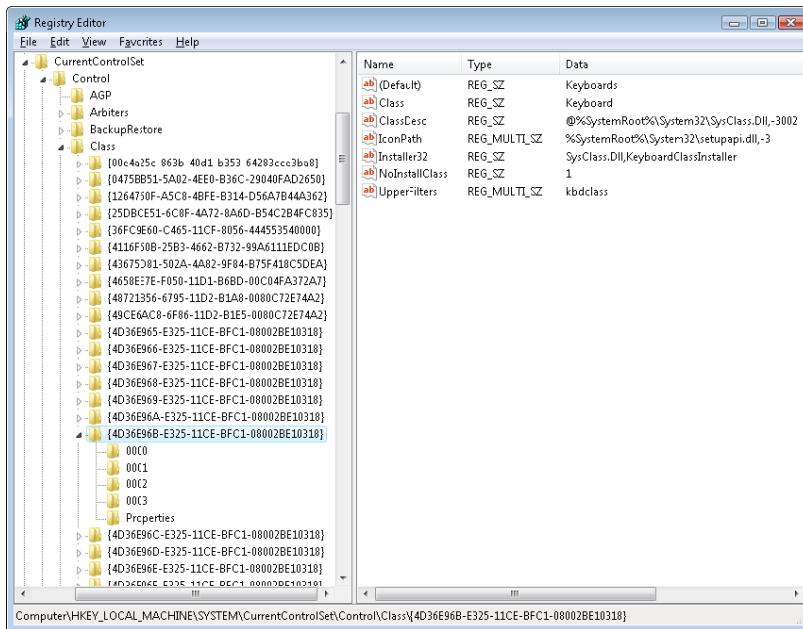


Рис. 8.43. Раздел класса клавиатур

Благодаря разделу, созданному для устройства в процессе перечисления, а также разделу классов PnP-диспетчер получает сведения, на основе которых он загружает драйверы, нужные узлу данного устройства. Загрузка происходит в следующем порядке:

1. Все низкоуровневые фильтрующие драйверы, указанные в параметре **LowerFilters** раздела, созданного для устройства в процессе перечисления.
2. Все низкоуровневые фильтрующие драйверы, указанные в параметре **LowerFilters** раздела класса данного устройства.
3. Функциональный драйвер, указанный в параметре **Service** раздела, созданного для устройства в процессе перечисления. Этот параметр интерпретируется как раздел драйвера в ветви **HKLM\SYSTEM\CurrentControlSet\Services**.
4. Все высокоуровневые фильтрующие драйверы, указанные в параметре **UpperFilters** раздела, созданного для устройства в процессе перечисления.
5. Все высокоуровневые фильтрующие драйверы, указанные в параметре **UpperFilters** раздела класса данного устройства.

Во всех случаях ссылка на драйверы происходит по имени их раздела в ветви **HKLM\SYSTEM\CurrentControlSet\Services**.

ПРИМЕЧАНИЕ

В WDK раздел, созданный для устройства в процессе перечисления, называется аппаратным ключом (hardware key), а раздел класса — программным ключом (software key).

У устройства клавиатуры, показанного на рис. 8.40 и 8.43, низкоуровневые фильтрующие драйверы отсутствуют. Но у него есть функциональный драйвер *i8042prt*, а в разделе класса фигурируют два высокоуровневых фильтрующих драйвера: *kbdclass* и *vmkbd2*.

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 *%SystemRoot%\System32\Umpnprmgr.dll* и выполняется как служба в хост-процессе (*Svhost.exe*).

Участвующие в установке драйвера компоненты показаны на рис. 8.44. 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 пользователяского 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 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 *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*.

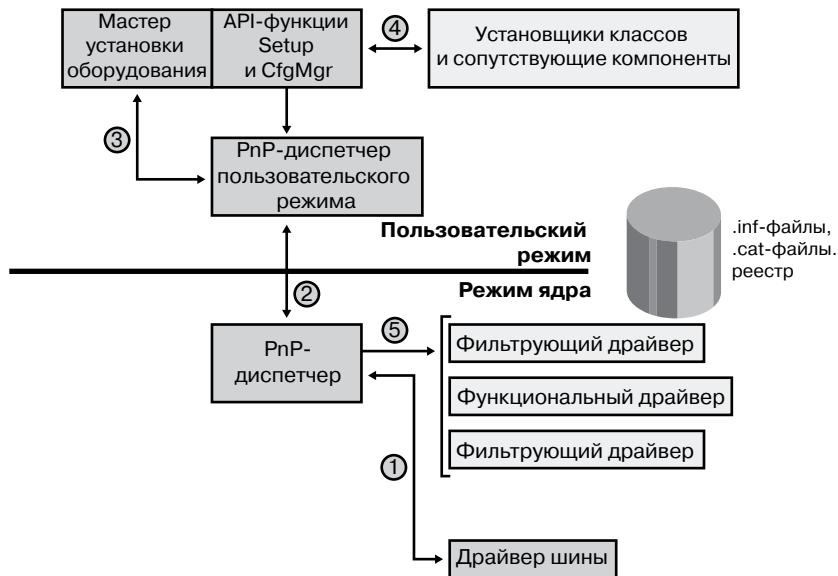


Рис. 8.44. Компоненты, участвующие в установке драйвера

Для поиска драйверов нового устройства процесс установки получает от драйвера шины список идентификаторов оборудования и идентификаторов совместимых устройств. Они 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, предпочтение отдается более полным вариантам, INF-файлам с цифровой подписью, причем среди последних преимущество имеют подписаные позже остальных. 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. 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 с перечнем его параметров.

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 Notepad, и вы увидите примерно следующее:

```
; Copyright (c) Microsoft Corporation. All rights reserved.

[Version]
Signature="$Windows NT$"
Class=Keyboard
ClassGUID={4D36E96B-E325-11CE-BFC1-08002BE10318}
Provider=%MS%
DriverVer=06/21/2006,6.1.7601.17514

[SourceDiskNames]
3426=windows cd
...
```

Если поискать в этом файле сочетание «.sys», найдется запись, указывающая PnP-диспетчеру пользовательского режима, что он должен установить драйверы i8042prt.sys и kbdclass.sys:

```
...
[STANDARD_CopyFiles]
i8042prt.sys,,,0x100
kbdclass.sys,,,0x100
...
...
```

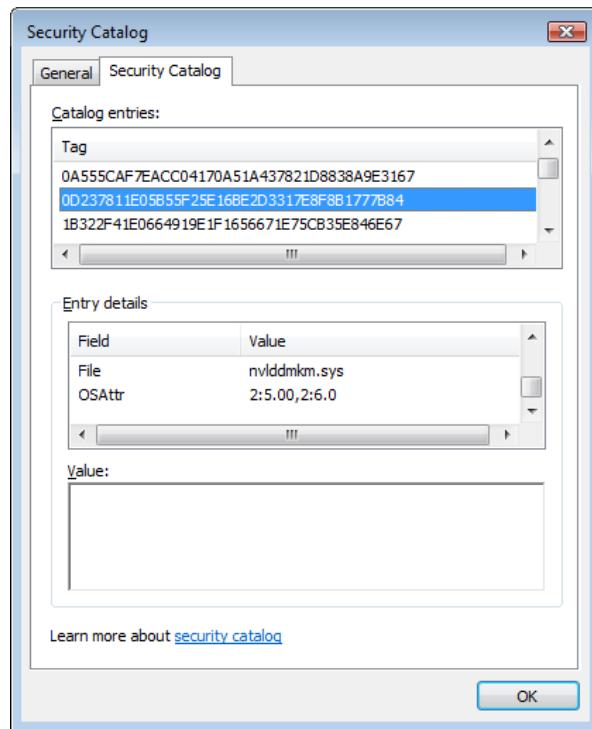
Перед установкой драйвера PnP-диспетчер пользовательского режима проверяет системную политику подписи драйверов. Если параметры, указанные для системы, блокируют установку неподписанных драйверов или предупреждают о попытках их установить, PnP-диспетчер пользовательского режима проверяет в INF-файле 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 получает хэш или уникальное значение для представления файлов драйвера, в том числе его образа. Затем этот хэш 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, файлы Nt5.cat и Nt5ph.cat хранят подписи и хэши страниц для системных файлов Windows.

Открыв один из CAT-файлов, вы увидите диалоговое окно с двумя вкладками. На вкладке General (Общие) представлены сведения о подписи в данном файле. На вкладке Security Catalog (Каталог безопасности) перечислены хэши компонентов, подписанных с помощью этого CAT-файла. На следующем снимке экрана показан CAT-файл для видеодрайверов NVIDIA и содержимое хэша для драйвера мини-порта в режиме ядра. Остальные хэши 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 открытого ключа Microsoft и сравнивает полученный хэш с хэшем файла устанавливаемого драйвера. При совпадении хешей драйвер считается проверенным на соответствие требованиям WHQL. Если же совпадения не происходит, 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 3 части I. Политика же проверки подписей драйверов распространяется только на драйверы, устанавливаемые с помощью INF-файлов.

После установки драйвера PnP-диспетчер режима ядра (этап 5 на рис. 8.44) запускает драйвер и вызывает процедуру добавления устройства, чтобы информировать драйвер о наличии устройства, для управления которым он был загружен. После этого конструируется стек устройств, о котором речь шла ранее.

ПРИМЕЧАНИЕ

PnP-диспетчер пользовательского режима также проверяет, не входит ли устанавливаемый драйвер в защищенный список драйверов (protected driver list), которые поддерживаются Windows Update. При положительном результате проверки установка блокируется, а для пользователя выводится предупреждение. В список вносятся драйверы, гарантированно несовместимые или имеющие ошибки.

Диспетчер электропитания

Как и технологии Plug and Play, управлению электропитанием требуется аппаратная поддержка, отвечающая спецификации ACPI (с ней можно ознакомиться на странице <http://www.acpi.info>). Этот стандарт определяет различные уровни энергопотребления для системы и устройств. В табл. 8.8 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 активную фазу».
- Аппаратная задержка** (hardware latency). Время, необходимое для возвращения компьютера в полностью активное состояние.

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.

Таблица 8.8. 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
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

продолжение ➔

Таблица 8.8 (продолжение)

Состояние	Энергопотребление	Возобновление работы ПО	Аппаратная задержка
S4 (сон)	Ток подается только на кнопку включения электропитания и в контур пробуждения	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

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 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 *Winresume*, 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.

Компьютер никогда не совершает прямых переходов из состояния S1 в S4. Сначала ему нужно перейти в состояние S0. Как показано на рис. 8.45, переход системы из состояний S1–S5 в S0 называется *пробуждением* (*waking*), а переход из состояния S0 в любое из состояний S1–S5 — *засыпанием* (*sleeping*).

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, сколько в состоянии D0, а в состоянии D2 устройство должно потреблять столько же или меньше энергии, сколько в состоянии D1. Совместно с крупными производителями аппаратного обеспечения специалисты Microsoft определили набор спецификаций управления электропитанием, в которых описываются состояния энергопотребления всех устройств конкретного класса (для основных классов устройств, включая монитор, сеть, SCSI и т. п.). Некоторые устройства могут быть либо полностью включены, либо полностью выключены, соответственно, промежуточные состояния для них не предусмотрены.

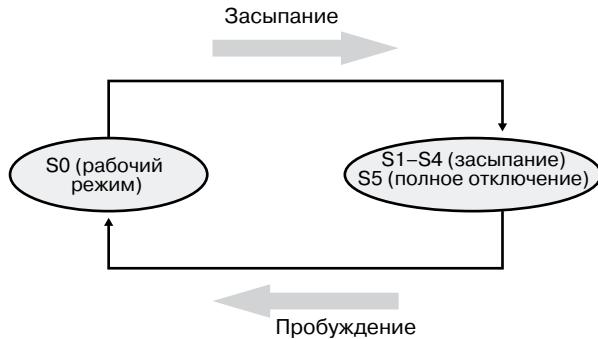


Рис. 8.45. Переходы системы между различными состояниями энергопотребления

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;
- параметры электропитания, заданные 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. 8.9 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 адAPTERы, поддерживают способность вывода системы из состояния сна. О наличии подобной способности также сообщается при перечислении устройств.

Таблица 8.9. 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)	D2
S4 (сон)	D3 (полное отключение)
S5 (полное отключение)	D3 (полное отключение)

Участие драйверов в управлении электропитанием

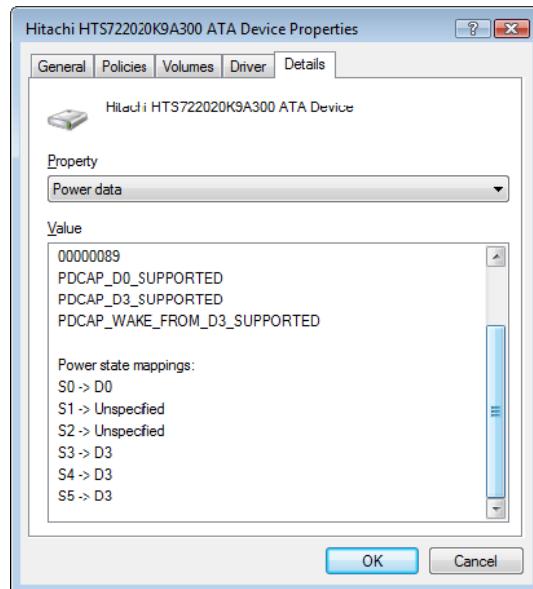
Принимая решение о переходе в другое состояние, диспетчер электропитания посылает команды процедуре драйвера, отвечающей за диспетчеризацию электропитания. Управлять устройством могут несколько драйверов, но только один из них назначается проводником политики управления электропитанием устройства. Этот драйвер определяет состояние устройства в зависимости от состояния энергопотребления системы. Например, при переходе из состояния S0 в S1 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. 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 Properties (Свойства) и выберите вариант Power Data (Сведения о питании) в раскрывающемся списке на вкладке Details (Сведения).

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 будущий 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 ACPI-совместимого портативного компьютера:

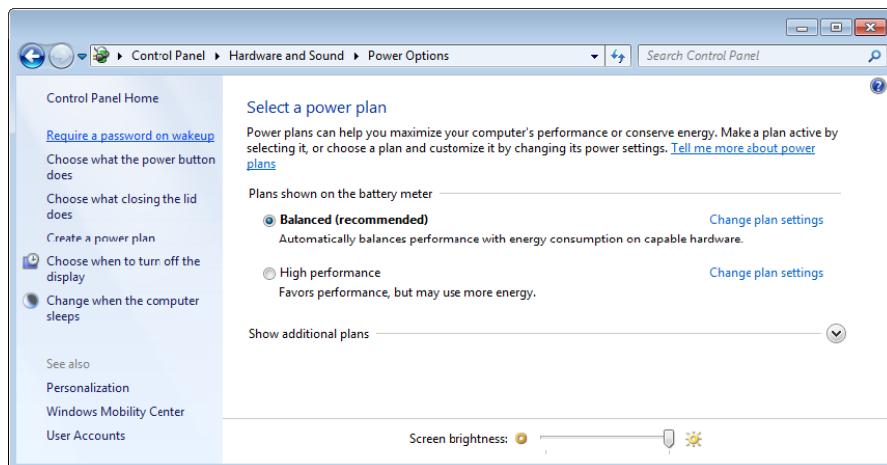
```
1kd> !pocaps
PopCapabilities @ 0x82114d80
Misc Supported Features: PwrButton SlpButton Lid S3 S4 S5 HiberFile
FullWake VideoDim
Processor Features: Thermal
```

продолжение ↗

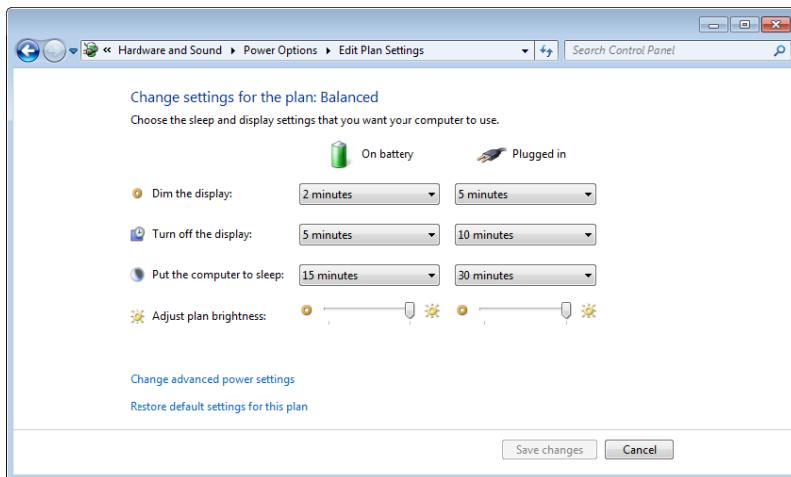
```
Disk Features: SpinDown
Battery Features: BatteriesPresent
Battery 0 - Capacity: 0 Granularity: 0
Battery 1 - Capacity: 0 Granularity: 0
Battery 2 - Capacity: 0 Granularity: 0
Wake Caps
Ac OnLine Wake: Sx
Soft Lid Wake: Sx
RTC Wake: S4
Min Device Wake: Sx
Default Wake: Sx
```

Строка Misc Supported Features говорит о том, что кроме S0 система поддерживает состояния S1, S3, S4 и S5 (состояние S2 не реализовано) и имеет действительный файл гибернации, в котором можно сохранить содержимое системной памяти при переходе в спящий режим (состояние S4).

Показанная ниже страница Power Options (Электропитание), 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.



Меняя заранее заданный план управления электропитанием, можно устанавливать интервалы простоя, по истечении которых отключается монитор, останавливаются жесткие диски, происходит переход в ждущий (состояние S1) и в спящий (состояние S4) режимы. Кроме того, параметр Change Plan Settings (Настройка плана электропитания) позволяет задать поведение системы при нажатии кнопок включения электропитания или перехода в спящий режим, а также при закрытии крышки ноутбука.



Возможности настройки, к которым вы получаете доступ, перейдя по ссылке Change Advanced Power Settings (Изменить дополнительные параметры питания), непосредственно влияют 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) @ 0x82107994
PowerButton: Sleep Flags: 00000000 Event: 00000000
SleepButton: Sleep Flags: 00000000 Event: 00000000
LidClose: Sleep Flags: 00000000 Event: 00000000
Idle: Sleep Flags: 00000000 Event: 00000000
OverThrottled: None Flags: 00000000 Event: 00000000
IdleTimeout: 384 IdleSensitivity: 90%
MinSleep: S3 MaxSleep: S3
LidOpenWake: S0 FastSleep: S0
WinLogonFlags: 1 S4Timeout: fd20
VideoTimeout: 300 VideoDim: 0
SpinTimeout: 258 OptForPower: 0
FanTolerance: 0% ForcedThrottle: 0%
SpinTimeout: 258 OptForPower: 0
MinThrottle: 0% DyanmicThrottle: None
```

Первые строки отписывают реакцию системы на нажатие кнопок включения электропитания и перехода в спящий режим. В данной системе обе эти кнопки, как и закрытие крышки ноутбука, переводят систему в спящий режим.

Показанные в конце значения тайм-аутов выражаются в секундах и выводятся в шестнадцатеричной системе. Они соответствуют указанным на странице Power Options (Электропитание) параметрам (портативный компьютер при этом работает от батареек). К примеру, тайм-аут для монитора равен 300, что означает его отключение после 300 секунд, или 5 минут, простоя. Жесткий диск перестает работать после тайм-аута 0x258, что соответствует 600 секундам, или 10 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, что упрощает логику программирования, необходимую для достижения 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 сделанный 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 посыпает драйверу уведомление о переходе, не спрашивая его согласия на это действие.

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.

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 детальность запросов на управление электропитанием. Кроме того, драйверы не могли посыпать 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 3 части I, можно посмотреть тип объекта `PowerRequest` в папке `\ObjectTypes`, а для подтверждения его действительности следует 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`).

Через эти API-функции реализуются три типа запросов: запрос к системе, запрос к монитору и запрос на переход в «режим отсутствия». В первом случае выясняется, что система не переходит в ждущий режим по таймеру простоя (хотя пользователю никто не мешает, например, закрыть крышку ноутбука и перевести его в состояние засыпания). Второй запрос требует аналогичной функциональности для монитора. «Режим отсутствия» представляет собой модификацию поведения в обычном ждущем режиме (состояние S3). При этом компьютер полностью снабжается электроэнергией, но монитор и звуковая карта отключаются, создавая у пользователя впечатление, что машина находится в ждущем режиме. Такое поведение, как правило, используется только в компьютерных приставках или мультимедийных центрах, в которых доставка содержимого должна продолжаться даже после нажатия пользователем кнопки перехода в ждущий режим. Со временем в Windows будут поддерживаться и другие виды запросов.

ЭКСПЕРИМЕНТ: ПРОСМОТР ЗАПРОСОВ НА УПРАВЛЕНИЕ ЭЛЕКТРОПИТАНИЕМ В ОТЛАДЧИКЕ

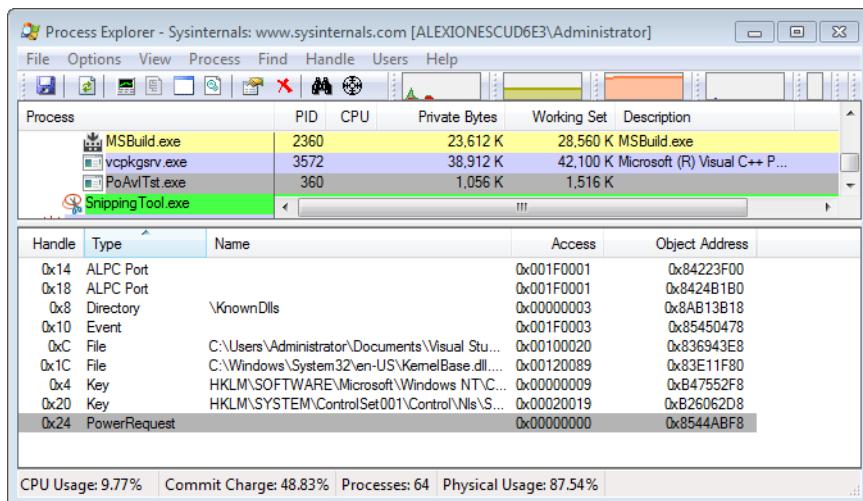
Так как запросы на управление электропитанием отправляются через объекты при помощи диспетчера объектов, приложения открывают для них дескрипторы, вызывая API-функцию `PowerCreateRequest`. Эти дескрипторы можно найти в приложении Process Explorer, используя функциональность поиска по DLL/дескриптору, о которой мы говорили в предыдущих главах.

Можно провести поиск по выражению «`PowerRequest`», обнаружив службы и приложения, от которых исходят запросы на управление электропитанием. (Драйверы в этом списке фигурировать не будут, так как API-функции ядра не пользуются дескрипторами.)

продолжение ➔

К примеру, службами, имеющими объекты запросов на управление электропитанием, являются Print Spooler (Spoolsvc.exe) и Windows Media Player Network Sharing Service (Wmpntwk.exe).

Запустив программу тестирования Poavltst.exe из пакета Book Tools и проведя поиск через Process Explorer, вы обнаружите, что и у нее есть открытый дескриптор. В нижней панели можно видеть адрес этого объекта в ядре — в данном случае 0x8544ABF8.



Затем, как показано далее, можно воспользоваться локальной отладкой ядра для получения дампа объекта запроса на управление электропитанием. К сожалению, основная структура данных ядра в символьных файлах отсутствует, поэтому возможен только шестнадцатеричный дамп. Тем не менее понять компоновку объекта несложно: двунаправленный список (первые два указателя), несколько флагов и указатель на сам запрос, предоставленный программой для тестирования. Он выделен полужирным шрифтом.

```
kd> dc 8544ABF8
855d01a8 819586c0 85448ea0 00000001 00000007 .....D.....
855d01b8 00000000 00000000 00000000 00000000 .....D.....
855d01c8 b13e9b50
```

Воспользовавшись этой же командой для указателя, вы увидите причину запроса на управление электропитанием: «Computation in progress» (Выполняются вычисления).

```
kd> dc b13e9b50
b13e9b50 00000001 8556b030 00000000 00000044 ....0.V....D...
b13e9b60 00000001 00000014 00000000 80080001 .....D.....
b13e9b70 00000000 006f0043 0070006d 00740075 ....C.o.m.p.u.t.
b13e9b80 00740061 006f0069 0020006e 006e0069 a.t.i.o.n. .i.n.
b13e9b90 00700020 006f0072 00720067 00730065 .p.r.o.g.r.e.s
```

Также можно воспользоваться командой dl для первого указателя в дампе объекта, чтобы получить список дампов всех запросов на управление электропитанием в системе, которые связаны символом PopPowerRequestObjectList в ядре. Это позволит увидеть запросы, не представленные в приложении Process Explorer, например созданные драйверами.

ЭКСПЕРИМЕНТ: ПРОСМОТР ЗАПРОСОВ НА УПРАВЛЕНИЕ ЭЛЕКТРОПИТАНИЕМ ЧЕРЕЗ POWERCFG

Как видите, создание дампа запросов на управление электропитанием требует погружения в ядро. К счастью, аналогичную функциональность предоставляет приложение Powercfg через чуть более простую в использовании командную строку. Вот результат, полученный в этом приложении, если одновременно просматривать общие ресурсы с другой машины, воспроизводить MP-файл и запустить приложение Poavltst.exe:

```
C:\Users\Administrator>powercfg -requests
DISPLAY:
[PROCESS] \Device\HarddiskVolume1\Users\Administrator\PoAvltst.exe
Computation in progress
[PROCESS] \Device\HarddiskVolume1\Program Files\Windows Media Player\wmplayer.exe

SYSTEM:
[DRIER] Parallel Audio Controller (x32) (PCI\VEN_8086&DEV_2445&SUBSYS_04001AB8
&REV_02\3&11583659&0&FC)
An audio stream is currently in use.
[DRIER] \FileSystem\srvenet
An active remote client has recently sent requests to this machine.
[PROCESS] \Device\HarddiskVolume1\Program Files\Windows Media Player\wmplayer.exe

AWAYMODE:
None.
```

Обратите внимание на уже знакомую строку «Computation in progress», а также на то, что SMB-драйвер и аудиодрайвер посылают запросы на управление электропитанием и объясняют, почему они это делают. В то же время приложение Windows Media Player использует устаревший API-интерфейс, поэтому в данном случае информация о причинах запроса не предоставляется.

Управление электропитанием со стороны центрального процессора

До этого момента мы рассматривали только управление электропитанием, производимое устройствами и системой, но в современных операционных системах существует еще один важный вариант управления электропитанием — со стороны процессора. В Windows реализован диспетчер электропитания центрального процессора (Processor Power Manager, PPM), отвечающий за контроль С- и Р-состояний (состояние простоя и активное состояние процессора), а также за взаимодействие с ACPI-программами и при необходимости с предоставленным производителем драйвером управления электропитанием (например, для процессоров от Intel это драйвер Intelppm.sys). Выбираемое состояние обычно определяется комбинацией внутренних алгоритмов и параметров реестра Windows, большая часть которых задается OEM-производителями и администраторами. Об этих настраиваемых значениях политик рассказывается в этом разделе чуть позже.

Хотя подробное описание PPM выходит за рамки темы данной книги, к тому же зачастую PPM зависит от конкретного оборудования, имеет смысл детально рассмотреть одну уникальную для Windows технологию: *парковку ядер* (core parking). По сути, это

механизм, запускаемый внутри PPM в зависимости от нагрузки и предлагающий два варианта решений:

- ❑ Какое P-состояние следует выбрать для конкретного процессора и каким образом будет осуществляться управление электропитанием в домене питания. Домен представляет собой набор функциональных единиц, связанных с конкретным ядром процессора (включая само ядро), работающих на одном генераторе тактовой частоты с одним делителем, и, соответственно, с одной частотой. Это может быть целый набор процессоров, половина набора или даже одно SMT-ядро с несколькими логическими процессорами.
- ❑ Когда планировщику (планирование рассматривается в главе 5 части I) следует запретить доступ к определенным ядрам, чтобы уменьшить количество попыток снова загрузить эти ядра. Такие ядра называются *запаркованными* (parked cores). Следует заметить, что строгие варианты настройки привязки заставляют планировщик выбирать из этих «недоступных» ядер. Но об этом мы поговорим позже.

ПРИМЕЧАНИЕ

В текущей реализации парковка ядер не перераспределяет баланс прерываний и не сдвигает программные таймеры, но это, возможно, реализуется в будущем.

Фактически, при парковке ядер процессоры принудительно переводятся в состояние глубокого простоя и по возможности тамдерживаются.

Политики парковки ядер

Так как требования к электропитанию и модели использования настольных компьютеров и серверов различаются, при парковке ядер реализуются две внутренние политики управления ядрами процессора. Первая называется *переопределением парковки ядер* (core parking override) и по умолчанию используется в клиентских системах. В ней ниже пороги простоя, при которых начинается парковка (то есть она происходит более агрессивно), и, что самое важное, один программный поток в SMT-наборе всегда остается незапаркованным. Другими словами, эта политика отвечает за отключение функции Hyper-Threading на процессорах Intel, пока она не потребуется при загрузке. Данный эффект показан на рис. 8.46: процессоры CPU 1 и CPU 3 запаркованы, так как они соответствуют второму программному потоку SMT-наборов CPU 0 и CPU 2.

Вторая политика парковки ядер соответствует поведению, предлагаемому по умолчанию, при котором, скажем так, не делается никаких специальных предположений по поводу SMT-ядер. Также с ней связаны менее агрессивные пороговые значения, большие подходящие для серверных рабочих нагрузок, когда большую часть времени загрузка процессоров невелика, но при этом они постоянно должны быть готовы к ее увеличению до пиковых значений.

Кроме того, механизм настроен так, чтобы избежать чрезмерной объединенной обработки одним или несколькими узлами. Хотя объединение работы дает энергетические преимущества, так как энергия в меньшей степени распределяется по системе и меньше тратится впустую, такое объединение сопровождается значительной конкуренцией с контроллерами памяти, которые в распределенной системе NUMA были бы заняты

меньше благодаря алгоритмам выбора наиболее подходящего узла и порождающего процесса. (Эта тема подробно рассматривается в главе 5 части I.) Соответственно, при парковке ядер нужно найти баланс между снижением энергопотребления, повышением эффективности кэша и доступа к памяти, а также ослаблением конкуренции на ресурсах локального узла. Именно ради этого механизм парковки всегда оставляет хотя бы одно ядро доступным для NUMA-узла, сохраняя результат усилий планировщика по распределению процессов и поддерживая приложения, специально разделяющие рабочую нагрузку между узлами, через учитывающую архитектуру NUMA привязку программных потоков и выделение памяти.

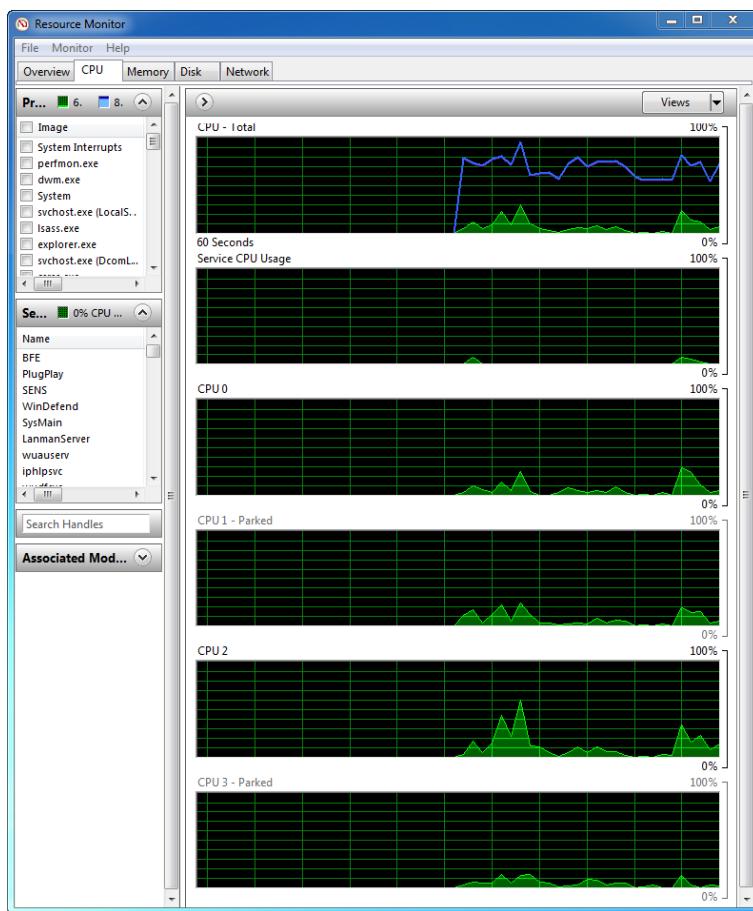


Рис. 8.46. Монитор ресурсов показывает эффект от парковки ядер в SMT-системах

Функция полезности

Принимаемые механизмом PPM решения об изменении состояния энергопотребления ядра и о ядрах, которые следует запарковать или перевести в активное состояние,

основываются на единственном принципе — принципе полезности. В представлении механизма PPM полезность процессора отражает загрузку ядра и вычисляется как произведение средней частоты (в процентах от максимального времени активности) на время загрузки (в процентах от всего времени активности). Так как перемножаются два значения в процентах, максимальная полезность составляет 10 000. И практически все вычисления PPM осуществляют путем сравнения полезности (позднее мы покажем производное от нее значение) с неким порогом или средним показателем.

ПРИМЕЧАНИЕ

На современных процессорах для получения средней частоты вызывается обработчик обратной связи, принадлежащий текущему домену питания, который управляет предоставленным производителем драйвером (например, Intelppm.sys). При отсутствии доступа к механизму обратной связи используется частота текущего домена.

Так как полезность процессора, очевидно, может быстро меняться со временем, PPM создает историю полезности каждого из ядер, фиксируя также их среднюю частоту. Хранит PPM и растущую сумму полезностей, соответственно, итоговая средняя полезность вычисляется как эта сумма, поделенная на количество записей в журнале.

ЭКСПЕРИМЕНТ: ПРОСМОТР СВЕДЕНИЙ О ПОЛЕЗНОСТИ И ЧАСТОТЕ

Как и большинство связанный с PPM информации, данные о полезности, а также ее история хранятся в структуре данных, называемой KPRCB. Помимо всего прочего, она легко визуализируется некоторыми расширениями отладчика.

Команда !ppm отладчика ядра выдает результат, аналогичный показанному для LP 0:

```
1kd> !ppm  
  
Processor 0  
  
Idle States (3)  
0: C1 - intelppm  
1: C2 - intelppm  
2: C3 - intelppm  
Last Used Idle State: 2  
  
Current Frequency: 100%  
HardwareFeedback: 55%  
Maximum Policy: 100%  
Platform Cap: 100%  
Minimum Policy: 5%  
Minimum Performace: 44%  
Minimum Throttle: 5%  
  
Utility: 5400
```

Полужирным шрифтом выделены упомянутые три значения. Полезность этого процессора равна 5400, и в настоящее время его рабочая частота составляет 100 % от максимальной. В строке HardwareFeedback указана средняя частота описанного ранее обработчика обратной связи. Эту частоту предоставленный производителем PPM-драйвер вычислил как 55 %.

Также можно посмотреть на PPM-информацию для других процессоров. В удаленном отладочном сеансе для переключения между процессорами используется команда ~ (тильда). При работе с локальным отладчиком ядра нужно вручную получать дамп структуры KPRCB и выводить список подструктуры .PowerState, как показано ниже. В рассматриваемом примере фигурирует дамп PPM-состояния для LP 1:

```
lkd> !running -i
System Processors: (0000000f)
Idle Processors: (0000000a)

Prcre Current (pri) Next (pri) Idle
0 8376cd20 87f0b030 (12) 83776380 .....
1 8b404120 8b409800 ( 0) 8b409800 .....
2 8b43a120 86e6ed48 (11) 8b43f800 .....
3 8b470120 8b475800 ( 0) 8b475800 .....

lkd> dt nt!_KPRCB 8b404120 PowerState.
+0x33a0 PowerState :
+0x000 IdleStates : 0x877ff890 _PPM_IDLE_STATES
+0x008 IdleTimeLast : 0xed
+0x010 IdleTimeTotal : 0xadae7baa
...

```

ЭКСПЕРИМЕНТ: ПРОСМОТР ИСТОРИИ ПОЛЕЗНОСТИ И ЧАСТОТЫ

Если текущая политика парковки ядер позволяет отслеживать историю (по умолчанию на клиентских системах этот режим отключен), можно посмотреть, как полезность и частота менялись со временем. Для этого используется еще одно расширение ядра — команда !ppmstate.

Вот результат применения этой команды для серверной системы с включенным режимом парковки ядер:

```
lkd> !ppmstate
Prcb.PowerState - 0x837700c0

IdleStates: 0x877fe1b0
IdleTimeLast: 0.000.006us (0x860 )
IdleTimeTotal: 11:35.968.474us (0x6bc4ae5f )
IdleAccounting: 0x874d8008

Hypervisor State: 0x0
LastPerfCheck: 13:20.311.497us (0x7becdf55)
PerfDomain: 0x874d9c50
PerfConstraint: 0x874d9cc8
Utility: 0xf6c

PerfHistory: 0x88604300
PerfHistory contents (3 slots, oldest to newest)

Slot Utility Frequency
0 3435 82%
1 10800 108%
2 10900 109%
```

продолжение ↗

```

ThermalConstraint: 100%
PerfActionDPC: 0x83770120
PerfActionMask: 0x0
WmiDispatchPtr: nt!PpmWmiDispatch
WmiInterfaceEnabled: 0x1
CurrentKernelUserTime: 0xc59e
CurrentIdleThreadKTime: 0xb556

```

В отличие от команды !ppm команда !ppmstate можно легко использовать при локальной отладке ядра, так как данное расширение принимает в качестве параметра адреса поля PowerState любой структуры KPRCB.

При парковке ядер и возвращении их в активное состояние применяется вторая характеристика, называемая *обобщенной полезностью* (generic utility). Это сумма всех функций полезности по всем процессорам, которые оказались затронутыми алгоритмом парковки ядер. Она используется для оценки общего уровня активности системы и позднее преобразуется в проценты (об этом мы поговорим при рассмотрении алгоритма). Соответственно, так как администраторы и пользователи задают политики электропитания всей системы, а не отдельных процессоров (в то время как парковка выполняется на уровне процессоров), обобщенная полезность нужна для преобразования функции полезности уровня процессоров в общесистемный вид.

Переопределение алгоритма

Так как парковка ядра не связана с планировщиком (этот аспект контролируется разработчиками), существуют отдельные сценарии, в которых приоритет управления отдается планировщику. Во-первых, это вынужденная привязка. При обсуждении алгоритмов планировщика в главе 5 части I упоминалось, что если запаркованное ядро является идеальным процессором для программного потока, а незапаркованные ядра отсутствуют, планировщик принудительно выбирает запаркованное ядро. Об этом узнает механизм парковки ядер, так как счетчик привязок в состоянии электропитания структуры KPRCB увеличивается на единицу. С течением времени механизм строит взвешенную историю ядер (в соответствии с конфигурацией политики), которые постоянно становились объектами принудительной привязки. А переход определенного порога, также заданного политикой, заставляет механизм соответствующим образом реагировать (эта реакция описывается далее в разделе, посвященном алгоритмам).

Второе переопределение возникает в случае, когда ядро запарковано (что означает низкие или нулевые значения полезности), но при этом вычисленная полезность превышает заданный в конфигурации порог. Данная ситуация не контролируется планировщиком. По факту она означает, что истечение программных таймеров, отложенные вызовы процедуры, прерывания и прочие подобные сценарии заставили запаркованное ядро запустить код, не входящий в сферу контроля планировщика. На подобную ситуацию механизм реагирует по-разному в соответствии с заданным алгоритмом. Кроме того, сохраняется история такой «сверхполезности», которая затем сравнивается с текущей политикой, и если оказывается, что превышен определенный порог, в алгоритм вносятся изменения.

Если вернуться к рис. 8.46 со снимком экрана в приложении Resource Monitor, можно заметить, как процессоры 1 и 3, даже будучи запаркованными, все равно по-

требляют какое-то процессорное время. В зависимости от текущей политики один или несколько таких процессоров могут быть рассмотрены как чрезмерно загруженные.

Увеличение/уменьшение числа запаркованных ядер

Каждый раз, когда механизму PPM требуется увеличить или уменьшить число запаркованных ядер или повлиять на текущее состояние производительности ядра, выполняется одно из следующих вариантов действий:

- В **идеальной модели** механизм пытается при выборе состояния производительности (`PERFSTATE_POLICY_CHANGE_IDEAL`) достичь средней производительности (частоты) между порогами уменьшения и увеличения. Механизм меняет состояние такого количества ядер, чтобы распределение обобщенной полезности активных ядер оказалось немного ниже или выше порога увеличения или уменьшения соответственно (`CORE_PARKING_POLICY_CHANGE_IDEAL`).
- В **пошаговой модели** механизм увеличивает или уменьшает производительность (частоту) по одному шагу (если величина шага задана через ACPI) или по 5 % (`PERFSTATE_POLICY_CHANGE_STEP`). Парковка ядер и перевод их в активное состояние также осуществляются по одному (`CORE_PARKING_POLICY_CHANGE_STEP`).
- В **реактивной модели** механизм переводит ядро в состояние максимальной или минимальной производительности (частоты) (`PERFSTATE_POLICY_CHANGE_ROCKET`). В этом случае осуществляется парковка всех ядер сразу (исключая одно на узел или в соответствии с настройкой текущей политики), как и перевод в активное состояние (`CORE_PARKING_POLICY_CHANGE_ROCKET`).

Далее при рассмотрении реального алгоритма парковки ядра вы увидите, когда предпринимаются эти действия по увеличению и уменьшению количества ядер.

Пороговые значения и варианты настройки политик

В конечном счете, изменение состояния производительности и парковка ядер зависят от пороговых значений и выбора политик в реестре, которые настраиваются как отдельно для каждого производителя и типа процессора, так и в целом для клиентских и серверных систем и различных схем управления электропитанием (например, High Performance, Balanced или Low Power). Варианты настройки политик и пороговые значения для парковки ядер перечислены в табл. 8.10–8.14.

Таблица 8.10. Политики производительности процессора (GUID_PROCESSOR_PERF)

GUID	Значение
INCREASE/DECREASE_THRESHOLD	Задает порог занятости, после которого происходит изменение состояния процессора
INCREASE/DECREASE_POLICY	Задает алгоритм выбора нового состояния производительности в ситуации, когда идеальное состояние производительности начинает отличаться от текущего

продолжение ↗

Таблица 8.10 (продолжение)

GUID	Значение
INCREASE/DECREASE_TIME	Задает минимальное количество интервалов проверки производительности с момента последнего изменения данного параметра, после которого состояние производительности снова можно менять
TIME_CHECK	Задает время до следующей оценки состояний производительности процессора и запаркованных ядер (в миллисекундах)
BOOST_POLICY	Задает количество процессоров, которые, когда это позволяют условия функционирования, могут рационально поднять частоту выше максимума
ALLOW_THROTTLING	Позволяет процессорам использовать состояния снижения питания (T-состояния) в добавок к состояниям производительности
HISTORY	Задает количество интервалов проверки производительности процессора, которые будут использоваться при вычислении средней полезности

Таблица 8.11. Политики управления состоянием простоя (GUID_PROCESSOR_IDLE)

GUID	Значение
ALLOW_SCALING	Указывает, следует ли масштабировать значения увеличения и уменьшения приоритета простоя, базируясь на текущем состоянии производительности
DISABLE	Указывает, следует ли отключить состояния простоя
TIME_CHECK	Указывает время с момента последнего увеличения или уменьшения приоритета состояния простоя, после которого приоритет этого состояния снова можно менять (в микросекундах)
DEMOTE/PROMOTE_THRESHOLD	Задает порог загрузки, после которого следует изменение состояния простоя процессора

Таблица 8.12. Политики парковки ядра (GUID_PROCESSOR_CORE_PARKING)

GUID	Значение
INCREASE/DECREASE_THRESHOLD	Задает порог загрузки, после которого меняется количество активных ядер

GUID	Значение
INCREASE/DECREASE_POLICY	Задает алгоритм выбора ядер, которые при необходимости паркуются или переводятся в активное состояние
MAX/MIN_CORES	Задает допустимое количество активных ядер (в процентах)
INCREASE/DECREASE_TIME	Задает минимальное количество интервалов проверки производительности, после которого паркуются или переводятся в активное состояние дополнительные ядра
CORE_OVERRIDE	Гарантирует хотя бы один активный процессор на ядро
PERF_STATE	Указывает, в каком состоянии производительности окажется процессор после парковки

Таблица 8.13. Политики истории привязки
(GUID_PROCESSOR_CORE_PARKING_AFFINITY_HISTORY)

GUID	Значение
DECREASE_FACTOR	Определяет, во сколько раз будет сокращена история привязки на каждом ядре после проверки производительности
THRESHOLD	Задает порог, выше которого считается, что запаркованное ядро имеет большую склонность к выполнению запланированных для него заданий
WEIGHTING	Задает вес каждого случая принудительного запуска запаркованного ядра

Таблица 8.14. Политики чрезмерной загрузки
(GUID_PROCESSOR_CORE_PARKING_OVER_UTILIZATION)

GUID	Значение
HISTORY_DECREASE_FACTOR	Определяет, во сколько раз будет сокращена история чрезмерной загрузки на каждом ядре после проверки производительности
HISTORY_THRESHOLD	Задает порог, выше которого считается, что запаркованное ядро недавно было чрезмерно загруженным
WEIGHTING	Задает вес каждого случая чрезмерной загрузки запаркованного ядра
THRESHOLD	Задает порог загрузки, после которого запаркованное ядро считается чрезмерно загруженным

ЭКСПЕРИМЕНТ: ПРОСМОТР ТЕКУЩЕЙ ПОЛИТИКИ ПАРКОВКИ ЯДРА

Рассмотренная ранее команда !popolicy позволила нам посмотреть только системные политики электропитания. Полный список политик, в который входит и PPM, остался за кадром. Команда dt с корректно указанным типом структуры позволяет увидеть PPM-политики, в которых встречаются GUID-идентификаторы из приведенных таблиц. Так как системная политика электропитания начинается со смещения 4, достаточно просто вычесть 4 из возвращенного командой !popolicy указателя.

```
lkd> !popolicy
SYSTEM_POWER_POLICY (R.1) @ 0x8377a6c4

lkd> dt nt!_POP_POWER_SETTING_VALUES 8377a6c0
...
+0x10c AllowThrottling : 0 ''
+0x10d PerfHistoryCount : 0x20 ''
+0x110 PerfTimeCheck : 0xf
+0x114 PerfIncreaseTime : 1
+0x118 PerfDecreaseTime : 1
+0x11c PerfIncreaseThreshold : 0x1e ''
+0x11d PerfDecreaseThreshold : 0xa ''
+0x11e PerfIncreasePolicy : 0x2 ''
+0x11f PerfDecreasePolicy : 0x1 ''
+0x120 PerfMinPolicy : 0x5 ''
+0x121 PerfMaxPolicy : 0x64 'd'
+0x124 PerfBoostPolicy : 0x64
+0x128 CoreParkingIncreaseThreshold : 0x55 'U'
+0x129 CoreParkingDecreaseThreshold : 0x32 '2'
+0x12a CoreParkingMaxCores : 0x64 'd'
+0x12b CoreParkingMinCores : 0xa ''
+0x12c CoreParkingIncreasePolicy : 0 ''
+0x12d CoreParkingDecreasePolicy : 0 ''
+0x130 CoreParkingIncreaseTime : 7
+0x134 CoreParkingDecreaseTime : 0x14
+0x138 CoreParkingAffinityHistoryDecreaseFactor : 0x2 ''
+0x13a CoreParkingAffinityHistoryThreshold : 0x96
+0x13c CoreParkingAffinityWeighting : 0x64
+0x13e CoreParkingOverUtilizationHistoryDecreaseFactor : 0x2 ''
+0x140 CoreParkingOverUtilizationHistoryThreshold : 0x28
+0x142 CoreParkingOverUtilizationWeighting : 0x64
+0x144 CoreParkingOverUtilizationThreshold : 0x3c '<'
+0x145 ParkingCoreOverride : 0x1 ''
+0x146 ParkingPerfState : 0 ''
```

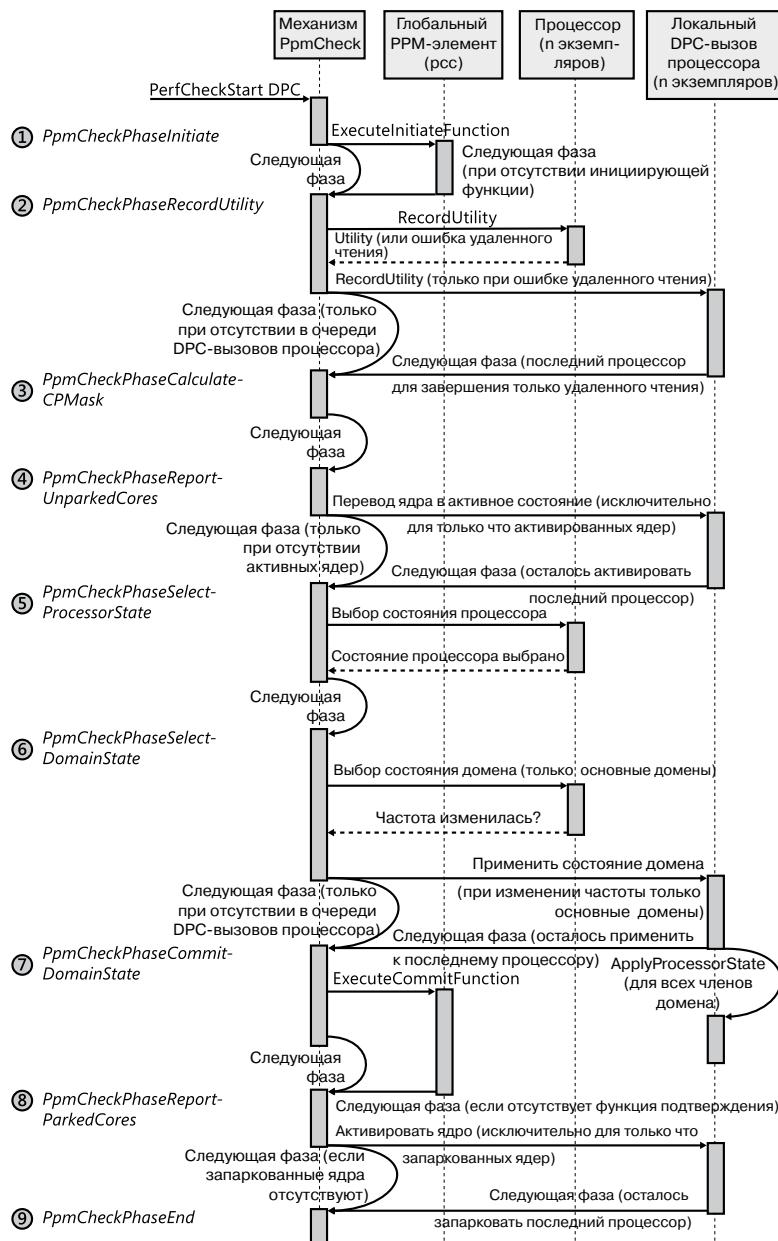
Также можно получить более ограниченный набор текущих политик, воспользовавшись расширением !ppmperfpolicy, которое выводит еще несколько вариантов настройки политик ядра:

```
lkd> !ppmperfpolicy

MaxPerf: 100%
MinPerf: 5%
TimeCheck: 15 ms
IncreaseTime: 1 time check period(s)
DecreaseTime: 1 time check period(s)
IncreaseThreshold: 30%
DecreaseThreshold: 10%
IncreasePolicy: 2
DecreasePolicy: 1
HistoryCount: 1
BoostPolicy: 100
```

Проверка производительности

Алгоритм, лежащий в основе механизма PPM, называется *проверкой производительности* (performance check). Он выполняется обратным вызовом таймера PpmCheckStart, который запускается с периодичностью, заданной в текущей политике. Этот обратный вызов получает блокировку политики и задает начальную фазу функции PpmCheckPhaseInitiate, что приводит к вызову функции PpmCheckRun, которая запускает показанный на следующей диаграмме алгоритм.



Показанные на диаграмме шаги с перечислением PPM_CHECK_PHASE представлены в табл. 8.15.

Таблица 8.15. Фазы проверки PPM

Название фазы	Смысл
PpmCheckPhaseInitiate	Уведомляет предоставленный производителем драйвер, управляющий электропитанием, что механизм парковки ядра готов начать проверку производительности
PpmCheckPhaseRecordUtility	Запускается на каждом процессоре, чтобы вычислить полезность каждого ядра
PpmCheckPhaseCalculateCoreParkingMask	На основе данных о полезности, текущем состоянии парковки ядер, привязках и истории чрезмерной загрузки распределяет ядра по наборам, что позволяет определить, какие ядра лучше запарковать, а какие перевести в активное состояние. Затем выполняет перевод ядер в активное состояние
PpmCheckPhaseReportUnparkedCores	Запускается на каждом активном процессоре, чтобы уведомить планировщик о его активном состоянии
PpmCheckPhaseSelectProcessorState	Вычисляет новое состояние производительности (целевую частоту) для каждого процессора, базируясь на его парковочном состоянии и полезности
PpmCheckPhaseSelectDomainState	Выбирает для всех процессоров в данном домене наилучшее состояние производительности, основываясь на ограничениях и переключениях в новое состояние производительности процессора
PpmCheckPhaseCommitDomainState	Вызывает предоставленный производителем драйвер управления электропитанием для подтверждения новых состояний производительности процессоров
PpmCheckPhaseReportParkedCores	Запускается на каждом запаркованном процессоре для уведомления планировщика о переходе ядра в активное состояние. С ядра снимается любая текущая или поставленная в очередь потоковая активность
PpmCheckPhaseEnd	Снимает блокировку политики и осуществляет переключение в нерабочую fazу
PpmCheckPhaseNotRunning	Указывает, что проверка производительности не проводится

Некоторые из упомянутых в таблице шагов требуют более детального рассмотрения.

Шаг 2. Запись полезности. Функция `PpmCheckRecordAllUtility` перечисляет все процессоры, относящиеся к зарегистрированному в данный момент набору механизма парковки ядер, и определяет, полезность каких из них будет запрашиваться удаленно (то есть из текущего ядра, исполняющего алгоритм проверки); или же целевой DPC-вызов запросит полезность локально. Такое определение выполняется путем вызова функции `PpmPerfRecordUtility` и базируется на значениях простоя и текущей полезности ядра. Так как эти значения в конце перемножаются, чем выше становится загрузка ядра (увеличивается полезность), тем выше будет погрешность от отсутствия точных изменений частоты. Последнее является побочным эффектом от запуска проверки не на локальном, а на удаленном ядре.

Кроме того, при локальном запуске функция проверяет наличие пропусков тактов процессора, находящихся вне компетенции PPM и обычно указывающих на ошибки в программном обеспечении или драйверах (или на существование стратегии управления электропитанием, не контролируемой операционной системой).

Остальные связанные с записью полезности проверки имеют отношение к вычислению значения, описанного в разделе «Функция полезности», и к отслеживанию истории этого параметра, если эта функция включена в политиках.

Шаг 4. Выбор активируемых ядер. На этом этапе работают две функции. Первая, `PpmPerfCalculateCoreParkingMask`, вычисляет, сколько ядер следует перевести в активное состояние, и строит различные наборы, которые могут использоваться при определении приоритетов этого перехода:

- ❑ **Чрезмерно загруженные ядра.** Их полезность превышает заданный политикой порог, как описано в разделе «Переопределение алгоритма».
- ❑ **Предыдущие чрезмерно загруженные ядра.** Ядра, которые были чрезмерно загружены при предыдущей проверке производительности, как описано в разделе «Переопределение алгоритма».
- ❑ **Ядра с привязкой.** Ядра, принудительно выбираемые планировщиком благодаря переопределению привязки, что также описывается в разделе «Переопределение алгоритма».
- ❑ **Активные ядра.** Ядра, которые уже переведены в активное состояние.
- ❑ **Активные ядра с высокой полезностью.** Активные ядра с высокими значениями функции полезности.

Затем функция вычисляет обобщенную полезность (см. раздел «Функция полезности») и определяет, выше или ниже заданных политикой пороговых значений оказывается этот параметр, выраженный в процентах (определяется как обобщенная полезность, поделенная на сумму частот загрузки всех ядер). В зависимости от того, какой из порогов был превышен, если таковое имело место, выполняются заданные политикой действия по увеличению/уменьшению количества запаркованных ядер.

Полученное значение обобщенной полезности и описанные ранее наборы передаются функции `PpmPerfChooseCoresToUnpark`, которая определяет, какие из процессоров следует перевести в активное состояние в зависимости от распределения обобщенной полезности. Прежде всего алгоритм определяет, превосходит ли количество уже

активных ядер значение целевого счетчика. При положительном результате проверки функция возвращает управление. В противном случае она сохраняет активируемые ядра, пока группы чрезмерно использующихся ядер не станет достаточно для обработки остальных запросов на активацию. Другими словами, чрезмерно загруженные ядра всегда переходят в активное состояние, а алгоритм должен выбрать, какие ядра, еще не используемые чрезмерно, следует перевести в активное состояние.

Для этого запускается следующая отборочная процедура. Каждый шаг предпринимается только в случае ненулевого пересечения (то есть при наличии других кандидатов):

- игнорировать любые процессоры, которые еще не являются чрезмерно загруженными;
- игнорировать любые процессоры, которые еще не имеют высоких показателей полезности;
- игнорировать любые процессоры, которые еще не активированы;
- игнорировать любые процессоры, которые ранее не были чрезмерно загруженными;
- игнорировать любые процессоры, у которых отсутствует принудительная привязка к программным потокам.

В наиболее удачном случае останется набор процессоров, которые чрезмерно загружены, имеют высокую полезность, ранее чрезмерно загружались или обладают принудительной привязкой. Другими словами, этот набор содержит процессоры, выгода от парковки которых сомнительна. Из этого набора механизм парковки выбирает наименьший номер процессора и снова начинает процедуру отбора, пока не будет соблюдено указанное условие.

В конце работы этого алгоритма, после того как все чрезмерно загруженные и неисключенные ядра будут переведены в активное состояние, обобщенная полезность сбалансируется (равномерно распределится) по всем новым активным процессорам.

Шаг 5. Выбор состояния процессора. Функция `PpmPerfSelectProcessorStates` перечисляет все процессоры, которые являются частью данного прогона, и для каждого из них вызывает функцию `PpmPerfSelectProcessorState`. В этом случае алгоритм может работать удаленно (без локального обратного DPC-вызова для ядра), так как все данные доступны в структуре KPRCB. Эта функция должна решить, какое из состояний является оптимальным для данного процессора, исходя из его ожидаемой функции полезности.

При первой проверке выясняется, был ли процессор на шаге 3 выбран для парковки. В случае положительного результата в соответствии с политикой выбирается целевое состояние энергопотребления для запаркованных ядер. Возможны три варианта:

- Самый легкий.** Запаркованный процессор предназначается для работы со стопроцентной частотой.
- Самый глубокий.** Запаркованный процессор предназначается для работы с однопроцентной частотой.
- Без предпочтений.** Запаркованный процессор воспринимается как и любой другой, продолжая обычный алгоритм.

Если предположить, что алгоритм продолжает свою работу, следующим шагом будет вычисление загруженности процессора. Так как функция полезности равна загрузке в процентах, умноженной на среднюю частоту, нужное нам значение можно получить, поделив полезность процессора на его среднюю частоту. Затем мы его сравним с порогами увеличения и/или уменьшения, заданными политикой, и будем действовать в соответствии с одной из трех моделей, идеальной, пошаговой или реактивной (см. раздел «Увеличение/уменьшение числа запаркованных ядер»).

Затем применяется обратный вызов обработчика производительности домена (им обладает предоставленный производителем драйвером процессора) с указанием новых частот и того, допускает ли политика пропуск тактов.

Шаг 6. Выбор состояния домена. Как уже было показано, этот этап также состоит из набора вспомогательных шагов. Во-первых, при удаленном выполнении он осуществляется функцией `PpmPerfSelectDomainStates`, которая выбирает основные домены и вызывает для них функцию `PpmPerfSelectDomainState`. Последняя по очереди просматривает все процессоры в домене и выбирает один с максимальным состоянием производительности (наивысшей желаемой частотой). Затем она указывает это значение как желаемую частоту для всего домена.

После этого управление возвращается функции `PpmPerfSelectDomainStates`, которая формирует очередь из локальных DPC-вызовов для всех основных доменов, реализованных функцией `PpmPerfApplyDomainState`. Это второй шаг. Данная функция принимает во внимание действительные P-состояния (и T-состояния, если в политике включен режим пропуска тактов), обрезая любые состояния, выходящие за рамки ограничителей для данного процессора, к которым относятся максимальное значение в процентах и максимальная температура. После выбора наилучшей целевой частоты (и консультации с обратным вызовом обработчика производительности домена) функция создает очередь DPC-вызовов для всех процессоров во всех доменах, чтобы применить выбранную производительность к каждому ядру.

На этом третьем этапе, реализованном DPC-процедурой `PpmPerfApplyProcessorState`, для переключения состояний применяется обратный вызов обработчика производительности домена. Наконец, вызывается функция `PpmScaleIdleStateValues`. Если политика позволяет масштабирование простоя, она масштабирует C-состояния процессора (состояния простоя) в соответствии с указанными в политике процентными показателями приоритета увеличения/уменьшения.

ЭКСПЕРИМЕНТ: ПРОСМОТР ДАННЫХ ТЕКУЩЕЙ РРМ-ПРОВЕРКИ

Отладчик ядра содержит расширение `!ppmcheck`, позволяющее проверять, включена ли функция парковки ядра, а также узнавать, какие ядра запаркованы в данный момент. Кроме того, она показывает состояние алгоритма проверки внутренней производительности. Вот пример результата работы данной функции:

```
1kd> !ppmcheck
PpmCheckArmed: TRUE
PpmCheckStartDpc: 0x8377aa58
PpmCheckDpc: 0x8377aa78
PpmCheckTimer: 0x8377aa30
PpmCheckMakeupCount: -
```

продолжение ↗

```
PpmCheckLastExecutionTime: -
PpmCheckTime: 08:40.738.783us (0x50a26d3d)
PpmCheckPhase: 9
PpmCheckRegistered: 0x8376b408
{[0000000F]}
PpmPerfStatesRegistered: 0x8376b390
{[0000000F]}
CoreParkingEnabled: TRUE
CoreParkingMask: 0x8376b35c
{[0000000A]}
```

Также можно увидеть полную PPM-информацию для данного процессора, обратив внимание на поле PowerState структуры PRCB и исследовав члены Domain и PerfConstraint. Это даст вам сведения о состоянии производительности выбранного домена, ограничителях (по температуре и частоте) и другие имеющие значение данные. Чтобы увидеть эту информацию для текущей структуры PRCB, воспользуйтесь командой dt nt!_KPRCB @\$prcb PowerState:

```
+0x33a0 PowerState :
+0x000 IdleStates : 0x877fe1b0 _PPM_IDLE_STATES
+0x008 IdleTimeLast : 0xa6
+0x010 IdleTimeTotal : 0x97789fc9
+0x018 IdleTimeEntry : 0
+0x020 IdleAccounting : 0x874d8008 _PROC_IDLE_ACCOUNTING
+0x024 Hypervisor : 0 ( ProcHypervisorNone )
+0x028 PerfHistoryTotal : 0
+0x02c ThermalConstraint : 0x64 'd'
+0x02d PerfHistoryCount : 0x1 ''
+0x02e PerfHistorySlot : 0 ''
+0x02f Reserved : 0 ''
+0x030 LastSystime : 0xfa86
+0x034 WmiDispatchPtr : 0x837c5464
+0x038 WmiInterfaceEnabled : 0n1
+0x040 FFHThrottleStateInfo : _PPM_FFH_THROTTLE_STATE_INFO
+0x060 PerfActionDpc : _KDPC
+0x080 PerfActionMask : 0n0
+0x088 IdleCheck : _PROC_IDLE_SNAP
+0x098 PerfCheck : _PROC_IDLE_SNAP
+0x0a8 Domain : 0x874d9c50 _PROC_PERF_DOMAIN
+0x0ac PerfConstraint : 0x874d9cc8 _PROC_PERF_CONSTRAINT
+0x0b0 Load : (null)
+0x0b4 PerfHistory : (null)
+0x0b8 Utility : 0xbab
+0x0bc OverUtilizedHistory : 0
+0x0c0 AffinityCount : 0
+0x0c4 AffinityHistory : 0

1kd> dt 0x874d9c50 _PROC_PERF_DOMAIN
nt!_PROC_PERF_DOMAIN
+0x000 Link : _LIST_ENTRY [ 0x8376b39c - 0x8376b39c ]
+0x008 Master : 0x8b470120 _KPRCB
+0x00c Members : _KAFFINITY_EX
+0x018 FeedbackHandler : 0x93d19d08 unsigned char +0
+0x01c GetFFHThrottleState : 0x93d1804e void +0
+0x020 BoostPolicyHandler : 0x93d18104 void +0
+0x024 PerfSelectionHandler : 0x93d19bee unsigned long +0
```

```
+0x028 PerfHandler : 0x93d19d40 void +0
+0x02c Processors : 0x874d9cc8 _PROC_PERF_CONSTRAINT
+0x030 PerfChangeTime : 0xaa90c1ed
+0x038 ProcessorCount : 4
+0x03c PreviousFrequencyMhz : 0x532
+0x040 CurrentFrequencyMhz : 0xa65
+0x044 PreviousFrequency : 0x31
+0x048 CurrentFrequency : 0x64
+0x04c CurrentPerfContext : 0
+0x050 DesiredFrequency : 0x64
+0x054 MaxFrequency : 0xa65
+0x058 MinPerfPercent : 0x2c
+0x05c MinThrottlePercent : 5
+0x060 MaxPercent : 0x64
+0x064 MinPercent : 5
+0x068 ConstrainedMaxPercent : 0x64
+0x06c ConstrainedMinPercent : 0x2c
+0x070 Coordination : 0x1 ''
+0x074 PerfChangeIntervalCount : 0n0

1kd> dt 0x874d9cc8 _PROC_PERF_CONSTRAINT
ntdll!_PROC_PERF_CONSTRAINT
+0x000 Prcb : 0x8376cd20 _KPRCB
+0x004 PerfContext : 0x877febe0
+0x008 PercentageCap : 0x64
+0x00c ThermalCap : 0x64
+0x010 TargetFrequency : 0x36
+0x014 AcumulatedFullFrequency : 0x46c3df
+0x018 AcumulatedZeroFrequency : 0xd51828
+0x01c FrequencyHistoryTotal : 0
+0x020 AverageFrequency : 0x36
```

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 увеличивает переносимость.

Благодаря наличию общей структуры драйверы могут располагаться друг над другом, обеспечивая модульность и уменьшая дублирование функций между драйверами. Кроме того, все драйверы устройств в Windows должны разрабатываться с учетом необходимости корректной работы в многопроцессорных системах.

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.

Три следующие главы мы посвятим смежной тематике, связанной с подсистемой ввода-вывода: управлению устройствами внешней памяти, файловым системам (в частности, подробно рассмотрим NTFS) и диспетчеру кэша.

Глава 9. Управление внешней памятью

Управление внешней памятью определяет способ взаимодействия операционной системы с энергонезависимыми устройствами хранения данных и носителями. Сам термин *внешняя память* (storage) охватывает множество различных устройств, включая оптические носители, флэш-накопители с USB-интерфейсом, гибкие и жесткие диски, твердотельные диски (Solid State Disks, SSD), сетевые хранилища, например iSCSI, сети хранения данных (Storage Area Networks, SAN) и виртуальные хранилища, такие как виртуальные жесткие диски (Virtual Hard Disks, VHD). Операционная система Windows отдельно поддерживает каждый из этих классов носителей внешней памяти. Так как книга посвящена в основном компонентам ядра Windows, в этой главе рассматриваются только фундаментальные принципы работы подсистемы управления внешней памятью, включая поддержку внешних дисков и флэш-накопителей. Поддержку съемных носителей и внешних хранилищ (автономное архивирование) Windows реализует в основном в пользовательском режиме.

В этой главе исследуется взаимодействие драйверов устройств режима ядра с драйверами файловой системы и дисками, обсуждается деление дисков на разделы, описываются принципы абстрагирования и управления томами, применяемые диспетчером томов. Кроме того, демонстрируется реализация средств управления дисками с несколькими разделами, к которым относятся также средства репликации и распределения данных файловой системы между физическими дисками, применяемые для повышения надежности и производительности. В заключение рассказывается о том, как драйверы файловой системы монтируют тома, за управление которыми они отвечают, а также обсуждаются технология шифрования дисков в Windows и поддержка автоматических резервного копирования и восстановления.

Базовая терминология

Для понимания материала этой главы нужно знать базовую терминологию:

- **Диском** (disk) называется физическое устройство внешней памяти, такое как жесткий диск, компакт-диск, DVD-диск, диск формата Blu-ray, твердотельный диск (SSD) или флэш-память.
- **Секторы** (sectors) делят диск на адресуемые блоки фиксированного размера. Размер сектора определяется видом диска. Например, размер секторов жесткого диска, как правило, составляет 512 байт (но сейчас происходит переход к 4096 байт), а для CD-ROM он равен 2048 байт. Более подробно о переходе к жестким дискам с 4096-байтными секторами можно узнать на странице <http://support.microsoft.com/kb/2510009/ru>.

- *Разделами* (partitions) называются наборы непрерывных секторов на диске. Адрес начального сектора, размер и другие характеристики раздела хранятся в таблице разделов или в другой базе данных управления диском. Эта база находится на одном диске с разделом.
- *Простые тома* (simple volumes) являются объектами, представляющими секторы одного раздела, которым драйверы файловой системы управляют как единым целым.
- *Составные тома* (multipartition volumes) – это объекты, представляющие секторы нескольких разделов, которыми драйверы файловой системы управляют как единым целым. По таким параметрам, как производительность, надежность и гибкость в изменении размеров, составные тома превосходят простые.

Дисковые устройства

С точки зрения Windows диск представляет собой устройство, которое обеспечивает адресуемое долговременное хранение блоков данных и доступ к которому осуществляется при помощи драйверов файловой системы. Другими словами, отдельные байты диска не имеют своего адреса, а вот у блоков адреса есть. Эти блоки называются секторами и являются базовыми единицами хранения и передачи информации с устройства и на устройство (то есть все передаваемые фрагменты данных должны делиться в соответствии с размером секторов). При этом не имеет значения, каким способом реализован диск – на вращающихся магнитных носителях (жесткий диск, дискета) или на твердотельной памяти (флэш-диск, флэш-накопитель).

В Windows поддерживается множество связанных друг с другом механизмов присоединения диска к системе, в том числе SCSI, SAS (Serial Attached SCSI), SATA (Serial Advanced Technology Attachment), USB, SD/MMC и iSCSI.

Вращающиеся магнитные диски

Типичный дисковый накопитель (часто называемый *жестким диском*) состоит из одной или нескольких вращающихся пластин, покрытых ферромагнитным материалом. Считывающая головка перемещается вперед и назад по поверхности пластины, читая и записывая биты, которые сохраняются магнитным способом.

Формат сектора диска

В то время как интерфейсы жестких дисков с момента появления в 1956 году первого диска производства IBM эволюционировали, став и быстрее, и умнее, основной формат диска практически не изменился, если не считать постоянное увеличение плотности записи (количества битов на квадратный дюйм). После появления дисковых накопителей порция данных сектора диска составляла 512 байт.

Плотность записи дискового накопителя, составлявшая в 1956 году 2000 бит на квадратный дюйм, к 2011 году возросла до 650 миллиардов битов на квадратный дюйм, причем основной прогресс произошел за последние 15 лет. Производители дисков

достигли физических пределов в рамках существующей технологии, поэтому изменения коснулись только формата: размер сектора увеличился с 512 до 4096 байт, а размер поля, выделяемого для кода коррекции ошибок (Error Correcting Code, ECC), — с 50 до 100 байт. Это так называемый расширенный формат дисков. Выбранный для него размер сектора совпадает с размером страницы в архитектуре x86 и с размером кластера в файловой системе NTFS. Расширенный формат обеспечивает 10 % приращения мощности за счет уменьшения количества служебной информации в каждом секторе (к служебной информации относится все, кроме данных) и благодаря улучшенному алгоритму исправления ошибок. (Одно поле ECC размером 100 байт лучше, чем восемь полей ECC размером 50 байт каждое.) К недостаткам данного формата можно отнести потенциальную потерю места при сохранении маленьких файлов, но в главе 12 вы увидите, что файловая система NTFS снабжена механизмом, обеспечивающим эффективность данной операции.

Диски расширенного формата поддерживают режим эмуляции (известный как 512e) для сохранения совместимости с устаревшими операционными системами, рассчитанными на работу только с секторами размером 512 байт. Благодаря этому режиму компьютер не подозревает, что диск использует секторы размером 4096 байт; он выполняет процедуры чтения и записи секторов размером 512 байт (которые называются логическими блоками). Контроллер диска преобразует номер логического блока в корректный физический сектор. К примеру, при поступлении запроса на чтение логического блока номер 6 контроллер диска запишет данные из нулевого сектора во внутренний буфер и вернет только порцию размером 512 байт, соответствующую логическому блоку номер 6, как показано на рис. 9.1.

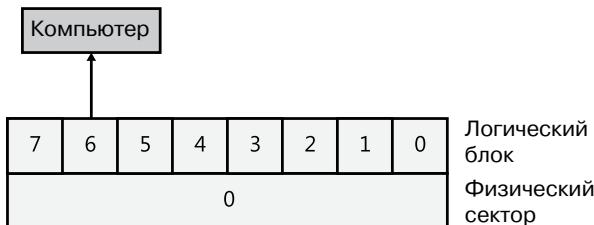


Рис. 9.1. Расширенный формат сектора в режиме эмуляции 512e

Процесс записи выглядит несколько сложнее, так как при этом контроллеру диска приходится выполнять процесс «чтение-изменение-запись», как показано на рис. 9.2.

1. Компьютер записывает в контроллер логический блок 6.
2. Контроллер совмещает логический блок 6 с физическим сектором 0 и считывает в память весь сектор.
3. Контроллер копирует логический блок 6 на предназначено ему место в копии хранящегося в памяти контроллера физического сектора.
4. Контроллер записывает физический сектор размером 4096 байт из памяти на диск.



Рис. 9.2. Операция «чтение-изменение-запись» в режиме эмуляции 512e

Очевидно, что в данном случае за эмуляцию приходится платить снижением производительности, зато диски расширенного формата могут работать со старыми версиями операционных систем.

Windows поддерживает собственный расширенный формат секторов размером 4096 байт, поэтому затраты на операцию «чтение-изменение-запись» отсутствуют. В главе 12 вы узнаете, что файловая система NTFS создавалась для поддержки секторов, размер которых превышает 512 байт, и операции дискового ввода-вывода там по умолчанию выполняются для кластера размером 4096 байт. Диспетчер кэша Windows (о котором пойдет речь в главе 11) пытается сократить издержки для приложений, работающих с 512-байтовыми секторами; но приложения следует обновить, чтобы они узнавали размер секторов диска (делая запрос на ввод-вывод с параметром `IOCTL_STORAGE_QUERY_PROPERTY` и проверяя возвращаемое значение `BytesPerPhysicalSector`) и при выполнении ввода-вывода секторов не рассматривали размер 512 байт. Инструменты разбиения должны учитывать размер физических секторов и выравнивать разделы по границам физического сектора, так как в каждом секторе должно помещаться целое число разделов.

Твердотельные диски

В последнее время стоимость производства флэш-памяти упала настолько, что производители начали выпуск подсистем хранения с дисковым интерфейсом, то есть **твердотельных дисков** (SSD), или **флэш-дисков**. С точки зрения Windows, они сильно отличаются от вращающихся дисков. Но перед тем как углубиться в детали поддержки твердотельных дисков, посмотрим на способ их реализации.

Флэш-память в некотором смысле напоминает оперативную память (Random Access Memory, RAM), но отключение электроэнергии не сопровождается потерей данных, что означает ее энергонезависимость. Шире всего распространены варианты флэш-памяти NOR и NAND. С точки зрения эксплуатации первая ближе к RAM, так как допускает доступ к отдельным ячейкам памяти, в то время как в NAND память разбита на блоки, как в случае с диском. Поэтому обычно флэш-память типа NOR применяется для хранения BIOS на материнской плате компьютера, а типа NAND — в твердотельных накопителях.

В отличие от оперативной памяти, допускающей практически бесконечное количество операций чтения и записи, флэш-память допускает перезапись менее 100 000 раз. (Для некоторых типов флэш-памяти эта цифра равна всего 1000.) На практике флэш-память изнашивается, поэтому ее следует рассматривать как носитель с ограниченным временем жизни (такой, как дискета), в отличие от RAM или магнитного диска. Еще флэш-память невозможно обновить методом замещения; перед записью новой информации в блок оттуда требуется удалить старую (даже для флэш-памяти типа NOR). Флэш-память намного быстрее магнитных дисков (обычно в 100 000 раз; к примеру, время доступа к ней составляет 50 наносекунд, в отличие от 5 миллисекунд, необходимых для доступа к магнитному диску), но медленнее, чем RAM (примерно в 50 раз). С практической точки зрения время доступа к памяти является не единственным аспектом, на который следует обращать внимание, так как флэш-память не подключается к системной шине памяти. Для нее нашине ввода-вывода существует интерфейс контроллера, напоминающий диск, поэтому в реальности скорость работы флэш-памяти может всего в 1000 раз превосходить скорость работы магнитных дисков, а в некоторых вариантах загрузки производительность таких дисков превосходит производительность дешевых моделей SSD.

Флэш-память типа NAND

В твердотельных накопителях чаще всего используется флэш-память типа NAND, поэтому подробно мы рассмотрим именно ее. Она бывает двух видов:

- В каждой одноуровневой ячейке (Single-Level Cell, SLC) хранится 1 бит информации, допускается большое количество циклов перезаписи (порядка 100 000), а скорость работы значительно выше, чем в случае MLC, но такие микросхемы стоят довольно дорого.
- В каждой многоуровневой ячейке (Multilevel Cell, MLC) хранится несколько битов информации, а стоимость этой микросхемы ниже, чем SLC. MLC-микросхемы требуют большего количества ECC-битов, допускают меньше, чем SLC, циклов перезаписи (порядка 5000) и потребляют больше энергии.

Флэш-память типа NAND разбита на страницы размером 4096 байт (которые могут быть представлены в виде восьми секторов по 512 байт или один сектор на 4096 байт), являющиеся минимальными фрагментами чтения или записи. Страницы группируются в блоки (в одном блоке может присутствовать от 64 до 1024 страниц), а на одной микросхеме располагаются тысячи блоков. Как и в случае с магнитными дисками, на каждой странице содержится служебная информация с полем ECC и резервной областью. Информация стирается блоками, поэтому для изменения сектора страницы нужно стереть и перезаписать целый блок. (Запись в флэш-ячейки осуществляется только после их очистки.) Это означает, что сектор очень быстро записывается в пустой блок, но если такие блоки отсутствуют, контроллер должен произвести следующие действия:

1. Прочитать целый блок в свою внутреннюю оперативную память.
2. Стереть блок из флэш-памяти.
3. Обновить блок в RAM, добавив туда содержимое нового сектора.
4. Записать весь блок во флэш-память.

Обратите внимание, что от записи сектора (512 байт) мы перешли к записи целого блока. И если предположить, что блок состоит из 128 страниц и заполнен до конца, запись займет в 1023 раза больше времени (блок состоит из 1024 секторов), чем запись одного сектора в пустой блок. Разумеется, данный пример иллюстрирует самый худший вариант развития событий и нормой не является, но он демонстрирует нам важное свойство SSD: чем больше SSD-памяти расходуется, тем значительнее количество перезаписываемых данных превосходит один сектор. То есть твердотельные диски по мере заполнения начинают работать медленнее. Это приводит к важным последствиям, о которых мы поговорим чуть позже, в разделе «Удаление файлов и команда Trim».

По мере износа блока все труднее становится стирать с него информацию. Кроме того, чем чаще блок подвергается перезаписи, тем медленнее он начинает работать (это обусловлено способом реализации флэш-памяти). Соответственно, твердотельный диск по мере его использования функционирует все медленнее и медленнее — даже с пустыми блоками. К примеру, в случае флэш-диска типа MLC емкостью 1 Гбайт со 128 страницами в блоке (что дает нам 2048 блоков) перезапись одного блока в секунду приведет к износу всех блоков за 23,7 дня (предполагается, что для одного блока допустимо 1000 циклов перезаписи, как в дешевых флэш-дисках). Перезапись одного и того же блока раз в секунду изнашивает блок всего за 16,6 минуты! Твердотельные диски обычно снабжаются резервными блоками (часто они составляют 20 % от всей емкости), на которые переносятся данные с истертых блоков. То есть очевидно, что флэш-памятью невозможно пользоваться таким же способом, как оперативной памятью или магнитным диском.

Для контроллера флэш-памяти была реализована технология *выравнивания износа* (wear-leveling), распределяющая перезаписываемую информацию по поверхности накопителя. Дело в том, что на диск, как правило, записываются статичные данные; то есть данные, которые меняются не часто (их в основном считывают, что не ведет к износу). Разумеется, присутствуют там и динамичные данные (например, журналы регистрации). Существуют различные типы алгоритмов, по которым данные записываются в разные ячейки памяти и включаются в общую ротацию, но их рассмотрение выходит за рамки темы данной книги. Вам достаточно просто понять, что благодаря технологии выравнивания износа контроллер распределяет данные по флэш-памяти, увеличивая общий срок службы твердотельного диска. Однако в результате из-за того, что в циклах перезаписи принимает участие больше блоков, больше блоков изнашивается, но это происходит более равномерно. Имейте в виду, что индустрия твердотельных накопителей постепенно развивается в сторону более надежных моделей, которые остаются доступными для чтения даже после того, как запись становится невозможной.

Удаление файлов и команда Trim

Файловая система следит за тем, какие участки диска заняты конкретным файлом, и далеко не все эти участки освобождаются при удалении файла — в противном случае удаление больших файлов занимало бы больше времени, чем удаление маленьких, кроме того, не смогли бы функционировать программы восстановления. Поэтому драйвер файловой системы помечает как доступные такие участки в своих структурах данных (обычно их называют *метаданными*; мы поговорим о них в главе 12). Для магнитных

дисков со встроенными механизмами чтения и записи секторов это не проблема, а вот твердотельные диски таких механизмов лишены (вспомните, что размер доступного для записи фрагмента, то есть страницы, намного меньше, чем размер стираемой области – блока).

В процессе обновления секторов твердотельным дискам приходится управлять содержимым страниц и блоков. И это становится серьезной проблемой, так как накопитель не знает, свободна ли страница, если с нее не было стерто содержимое. Поэтому он продолжает хранить «удаленные» данные в процессе обновления сектора или выравнивания износа, уменьшая объем доступного его контроллеру свободного места. В итоге скорость работы падает до момента, пока не произойдет (хотя бы по одному разу) доступ ко всем секторам. Бороться с этим можно только очисткой всего диска. Именно такое поведение отличало ранние версии твердотельных накопителей.

Решением проблемы стала команда `trim`. Файловая система узнает о том, что накопитель поддерживает эту команду, отправляя запрос на ввод-вывод `IOCTL_STORAGE_QUERY_PROPERTY` с идентификатором `StorageDeviceTrimProperty` в стек драйверов внешней памяти (о нем мы поговорим чуть позже). При удалении файла на диске, поддерживающем команду `trim`, или уменьшении его размера файловая система отправляет драйверу диска список секторов, которые занимал данный файл. Для этого используется запрос на ввод-вывод `IOCTL_STORAGE_MANAGE_DATA_SET_ATTRIBUTES` с параметром действия `DeviceDsmAction_Trim`. При получении такого запроса драйвер диска отправляет накопителю команду `trim`, уведомляя о том, что перечисленные сектора теперь свободны и могут быть очищены и повторно задействованы под запись данных. В результате накопитель пользуется ими при обновлении или во время процедуры выравнивания износа, повышая свою производительность. Имейте в виду, что команду `trim` нельзя поставить во внутреннюю очередь контроллера, то есть она выполняется синхронно, заметно тормозя работу накопителя при удалении больших файлов.

Хотя Windows поддерживает твердотельные диски, Microsoft рекомендует как можно чаще прибегать к резервному копированию, особенно если речь идет о хранении важных данных. С твердотельными дисками нельзя использовать стандартную программу дефрагментации диска, так как она приводит к быстрому износу флэш-памяти. Дефрагментатор Windows не будет пытаться обработать SSD. (В целом дефрагментация твердотельных дисков не имеет особого смысла, так как для них при фрагментации доступ к файлу не замедляется, как в случае магнитных дисков.) Как вы увидите в главе 12, поскольку при проектировании NTFS существование недолговечных дисков (флэш-памяти) не учитывалось, файловая система выполняет множество мелких записей в журнал транзакций, что повышает ее надежность, но приводит к дополнительному износу флэш-памяти. Превратив SSD в диск C, вы можете радикально повысить скорость работы операционной системы, но имейте в виду, что твердотельный диск придет в негодность намного быстрее магнитного.

ПРИМЕЧАНИЕ

Магнитные диски последних моделей могут в ряде случаев показывать более высокую производительность, чем дешевые твердотельные диски, так как последние плохо справляются с хаотичной записью небольших фрагментов данных, которая сопровождает типичный рабочий процесс в Windows.

Драйверы дисков

Драйверы устройств, участвующие в управлении конкретным накопителем, вместе образуют *стек драйверов внешней памяти* (storage stack). На рис. 9.3 перечислены все типы драйверов, которые могут присутствовать в стеке, с кратким описанием их назначения. В этой главе рассматривается поведение драйверов устройств, расположенных в стеке ниже уровня файловой системы. (Драйверам файловой системы посвящена глава 12.)



Рис. 9.3. Стек драйверов устройств внешней памяти в Windows

Программа Winload

Как вы уже знаете (см. главу 4 части I), за начальный этап загрузки Windows отвечает программа Winload. Не относясь с технической точки зрения к стеку внешней памяти, эта программа участвует в управлении ею, поскольку предоставляет поддержку доступа к дисковым устройствам до момента, пока не начнет работать подсистема ввода-вывода Windows. Winload находится на загрузочном томе; расположенный на системном томе код загрузочного сектора запускает диспетчер загрузки Bootmgr, который считывает информацию хранилища конфигурационных данных загрузки (Boot

Configuration Database, BCD) с системного тома или EFI и предлагает пользователю варианты загрузки. Имя выбранного пользователем варианта Bootmgr преобразует в соответствующий загрузочный раздел и запускает Winload для загрузки в память системных файлов Windows (начиная с реестра, Ntoskrnl.exe и зависящих от него файлов и драйверов загрузки). Во всех случаях Winload использует микропрограммы компьютера для чтения содержащего системный том диска.

Драйверы дисковых класса, порта и мини-порта

В процессе инициализации диспетчер ввода-вывода запускает драйверы жестких дисков. Драйверы устройств внешней памяти в Windows имеют архитектуру класс/порт/мини-порт. В этом случае Microsoft предоставляет драйвер класса внешней памяти, реализующий общую для всех устройств внешней памяти функциональность, и драйвер порта, поддерживающий связанную с классом функциональность, общую для определенной шины, например SATA (Serial Advanced Technology Attachment), SAS (Serial Attached SCSI) или Fibre Channel. А изготовители оборудования предоставляют драйверы мини-порта, подключаемые к драйверам порта и создающие в Windows интерфейс для конкретных реализаций контроллера.

В архитектуре драйверов дисковой памяти стандартными интерфейсами драйверов устройств в Windows обладают только драйверы класса. Драйверы мини-портов вместо интерфейса драйверов устройств пользуются интерфейсом драйверов порта, последние же просто реализуют набор процедур, служащих интерфейсом между Windows и драйверами мини-порта. Такой подход упрощает разработку драйверов мини-порта, так как Microsoft предоставляет драйверы порта для каждой операционной системы, позволяя разработчикам сконцентрироваться на логических схемах драйверов, связанных с аппаратным обеспечением. В Windows входит драйвер класса, реализующий стандартную функциональность дисков (%SystemRoot%\System32\Drivers\Disk.sys). Также Windows предоставляет разнообразные драйверы порта для дисков. Например, %SystemRoot%\System32\Drivers\Scsiport.sys — это устаревший драйвер порта для дисков на шинах SCSI (Scsiport в настоящее время не рекомендуется к применению), а %SystemRoot%\System32\Drivers\Ataport.sys — драйвер порта для систем на базе IDE. Большинство более новых драйверов используют драйвер порта %SystemRoot%\System32\Drivers\Storport.sys вместо Scsiport.sys. Драйвер Storport.sys был спроектирован для реализации функциональности высокопроизводительных аппаратных RAID-контроллеров и адаптеров Fibre Channel. Модель Storport аналогична Scsiport, что упрощает производителям перенос существующих драйверов мини-порта Scsiport на Storport. Драйверы мини-порта, создаваемые разработчиками для Storport, пользуются преимуществами набора механизмов Storport, повышающих производительность, в частности поддержкой параллельной инициации и завершения запросов на ввод-вывод в многопроцессорных системах, более управляемой архитектурой очереди запросов на ввод-вывод и выполнением большей части кода при более низком значении IRQL для минимизации длительности маскирования аппаратных прерываний. Драйвер Storport также поддерживает динамическое перенаправление прерываний и отложенных вызовов процедуры на самый лучший (наиболее локальный) узел NUMA (часто называемый NUMA I/O).

Оба драйвера, Scsiport.sys и Ataport.sys, реализуют алгоритм планирования запросов к жесткому диску C-LOOK. Драйверы помещают запросы на дисковый ввод и вывод в списки, отсортированные по первому сектору, также известному как *адрес логического блока* (Logical Block Address, LBA), которому адресован запрос. Они используют функции KeInsertByKeyDeviceQueue и KeRemoveByKeyDeviceQueue (документированные в Windows Driver Kit), чтобы представить запросы на ввод и вывод как элементы, в то время как начальный сектор запроса играет роль требуемого этими функциями ключа. В процессе обслуживания запросов драйвер проходит по всему списку, от нижнего сектора к самому верхнему. Достигнув конца, он возвращается к началу, так как в списке могли появиться новые запросы. В случае распределенных по всему диску адресов запросов головке диска приходится постоянно перемещаться от внешних цилиндров к внутренним. Драйвер Storport.sys не касается дискового планирования, так как в основном служит для управления вводом и выводом, адресованным массивам накопителей, у которых нет четкого определения начала и конца диска.

Вместе с Windows поставляется несколько драйверов мини-порта. В системах с хотя бы одним устройством IDE на базе ATAPI функциональность мини-порта представлена драйверами %SystemRoot%\System32\Drivers\Atapi.sys, %SystemRoot%\System32\Drivers\Pciidex.sys и %SystemRoot%\System32\Drivers\Pciide.sys. В большинстве версий Windows присутствует один или несколько этих драйверов.

iSCSI-драйверы

Разработка iSCSI как транспортного протокола для дисков позволила интегрировать протокол SCSI в стек протоколов TCP/IP. В результате компьютеры получили возможность взаимодействовать с блочными накопителями, в том числе дисками, через IP-сети. Сеть хранения данных (SAN) обычно строится на базе сети Fibre Channel, но администраторы через протокол iSCSI могут создавать относительно недорогие сети хранения данных, взявшись за основу такие сетевые технологии, как Gigabit Ethernet. Это обеспечивает масштабируемость, аварийную защиту, эффективное резервное копирование и защиту данных. Поддержка протокола iSCSI в Windows осуществляется в форме программного iSCSI-инициатора производства Microsoft, включенного во все редакции Windows.

Этот программный инициатор состоит из следующих компонентов:

- **Инициатор (initiator).** Необязательный компонент, состоящий из драйвера порта Storport и драйвера мини-порта iSCSI (%SystemRoot%\System32\Drivers\Msiscsi.sys). Для реализации программного протокола iSCSI поверх стандартных адаптеров Ethernet и адаптеров TCP/IP с аппаратным ускорением сетевых операций применяется драйвер TCP/IP.
- **Служба инициатора (initiator service).** Служба, реализованная в программе %SystemRoot%\System32\lscsicli.exe, управляет обнаружением и защитой всех iSCSI-инициаторов, а также началом и завершением сеансов. Функциональность обнаружения устройств реализована в файле %SystemRoot%\System32\lscsium.dll. Важной задачей службы iSCSI-инициатора является обеспечение общей инфраструктуры открытия-управления, не зависящей от используемого драйвера протокола. В качестве последнего может выступать программный эмулятор контроллера или НВА-драйвер (адаптер главной шины; протокол iSCSI, перекладывающий

обработку на оборудование, в роли которого обычно выступают мини-порты Storport). В этом контексте iSCSI также предоставляет интерфейсы Win32 и WMI для управления и конфигурирования. Служба iSCSI-инициатора поддерживает четыре механизма обнаружения:

- **iSNS** (Internet Storage Name Service — служба имен хранилищ Интернета). Адреса iSNS-серверов, которыми будет пользоваться служба iSCSI-инициатора, статически конфигурируются командой `iscsicli AddiSNSServer`.
- **SendTargets**. Порталы SendTarget статически настраиваются с помощью команды `iscsicli AddTargetPortal`.
- **Обнаружение адаптеров шины**. Адаптеры iSCSI-шины, соответствующие интерфейсам в iSCSI-инициаторе, могут участвовать в обнаружении целевых объектов при помощи интерфейса между адаптером шины и службой iSCSI-инициатора.
- **Настроенные вручную целевые объекты**. Целевые iSCSI-объекты можно настраивать вручную при помощи команды `iscsicli AddTarget` или графического пользовательского интерфейса iSCSI-инициатора.

□ **Управляющие приложения**. В эту категорию попадают утилита `Iscsicli.exe` (утилита командной строки для управления соединениями iSCSI-устройств и их защитой) и соответствующее приложение панели управления.

Некоторые производители выпускают iSCSI-адAPTERЫ с аппаратным ускорением операций по протоколу iSCSI. С этими адаптерами работает служба инициатора, поэтому они должны поддерживать протокол iSNS (RFC 4171), чтобы все iSCSI-устройства, в том числе обнаруженные службой iSCSI-инициатора и iSCSI-оборудованием, можно было распознавать и контролировать через стандартные интерфейсы Windows.

MPIO-драйверы

Для большинства дисковых устройств путь к компьютеру всего один — он пролегает через набор адаптеров, кабелей и коммутаторов. Серверы, требующие высоких показателей надежности, пользуются решениями с несколькими путями (Multipath I/O, MPIO). То есть между компьютером и диском находится несколько наборов соединительного оборудования, чтобы в случае отказа одного из путей система все равно могла обращаться к диску. Однако без поддержки со стороны операционной системы или дисковых драйверов диск с двумя путями будет восприниматься как два разных диска. Операционная система Windows поддерживает ввод-вывод по разным путям, что позволяет управлять набором дисков как единой сущностью. Поддержка реализуется через драйверы сторонних производителей, так называемые *модули для конкретных устройств* (Device-Specific Modules, DSM). Именно эти модули берут на себя всю специфику управления путями, в том числе политику балансировки нагрузки, на основе которой выбираются как путь передачи запросов, так и механизмы обнаружения ошибок, уведомляющие Windows об отказе того или иного пути. В Windows встроены DSM-модули (%SystemRoot%\System32\Drivers\Msdsm.sys), работающие со всеми массивами хранения, которые соответствуют промышленному стандарту (спецификации T10 SPC4), описывающему ассиметричные логические массивы (Asymmetric Logical Unit Arrays, ALUA). Производители массивов хранения данных

должны писать собственные DSM-модули для несовместимых с ALUA модулей. Средства написания DSM в настоящее время являются частью набора Windows Driver Kit. Поддержка MPIO не является обязательной и доступна в Windows Server 2008/R2 при установке через диспетчер серверов. В клиентских версиях Windows поддержка MPIO отсутствует.

В показанном на рис. 9.4 стеке драйверов внешней памяти Windows MPIO-драйвер диска включает в себя функциональность MPIO-устройств, которая в предыдущих версиях Windows реализовывалась отдельным драйвером (`Mpdev.sys`).

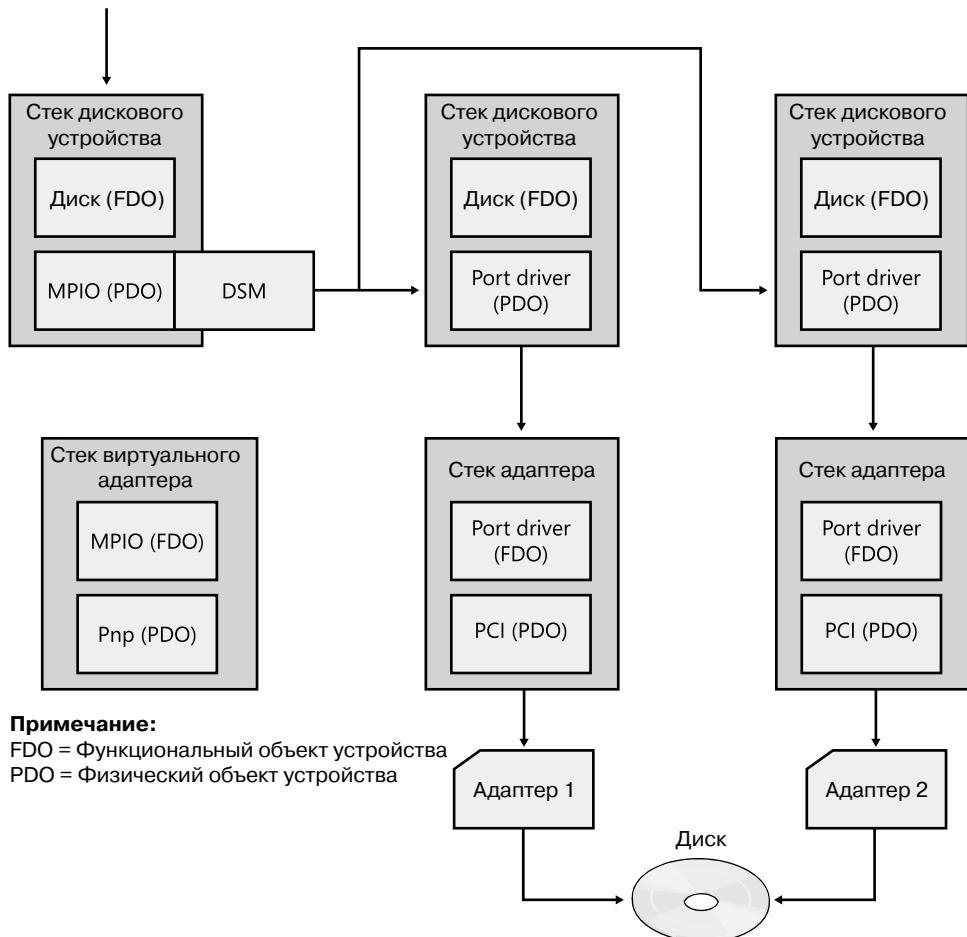


Рис. 9.4. Стек MPIO-драйверов в Windows

Драйвер `Disk.sys` отвечает за монопольное использование объектов устройств (которые представляют диски с набором путей, гарантирующие, что для представления всех дисков достаточно будет одного объекта устройства) и за размещение соответствующего DSM-модуля, предназначенного для управления путями к устройству. Драйвер

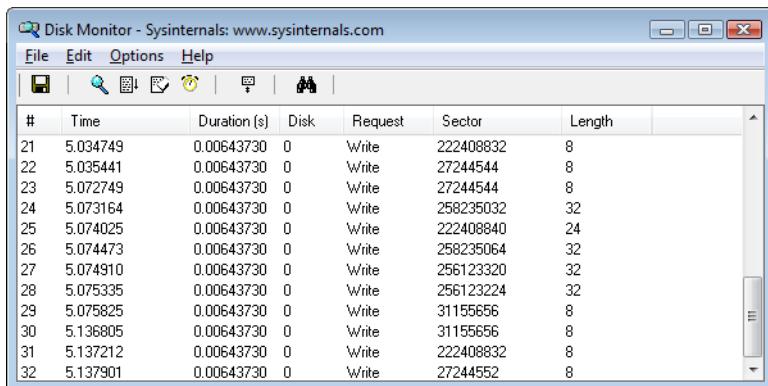
шины с несколькими путями (%SystemRoot%\System32\Drivers\Mpio.sys) управляет соединениями между компьютером и устройством, а также электропитанием устройства. Драйвер Disk.sys уведомляет драйвер Mpio.sys о наличии устройства, которым последний должен управлять. Драйвер порта (и нижележащий драйвер мини-порта) для диска с несколькими путями не распознает MPIO и не принимает участия в обработке набора путей. Всего существуют три варианта стека дискового устройства, два из которых представляют физические пути (дочерние для стеков адаптера), а один — диск (дочерний для стека MPIO-адаптера). При получении запроса последний использует DSM для определения пути, по которому следует перенаправить этот запрос. DSM осуществляет выбор на базе политики устройств, направляя запрос в соответствующий стек дискового устройства. Этот стек, в свою очередь, передает его устройству через нужный адаптер.

Аварийный дамп памяти системы и механизмы перехода в спящий режим функционируют в крайне ограниченной среде (их поддержка операционной системой и драйверами устройств невелика). Работающие в этой среде драйверы осведомлены о MPIO, но поддержка реализована только до определенного предела. К примеру, при отказе одного из путей Windows может переключиться только на диск, контролируемый тем же самым драйвером мини-порта.

Управление MPIO-конфигурациями осуществляется при помощи программы MP-Claim (%SystemRoot%\System32\Mpclaim.exe) и на вкладке свойств диска в Проводнике.

ЭКСПЕРИМЕНТ: ВВОД И ВЫВОД НА ФИЗИЧЕСКОМ ДИСКЕ

Приложение Diskmon, разработанное компанией Windows Sysinternals (www.microsoft.com/technet/sysinternals), при помощи системы трассировки событий для Windows (Event Tracing for Windows, ETW), о котором рассказывается в главе 3 части I, отслеживает ввод и вывод на физических дисках и выводит данные о них в своем окне. Содержимое этого окна обновляется один раз в секунду. Для каждой операции показываются время, продолжительность, номер целевого диска, тип, смещение и длина, как показано на снимке экрана.



The screenshot shows the 'Disk Monitor - Sysinternals' window. The menu bar includes File, Edit, Options, and Help. Below the menu is a toolbar with icons for Stop, Refresh, and others. The main area is a table with the following columns: #, Time, Duration (s), Disk, Request, Sector, and Length. The data in the table is as follows:

#	Time	Duration (s)	Disk	Request	Sector	Length
21	5.034749	0.00643730	0	Write	222408832	8
22	5.035441	0.00643730	0	Write	27244544	8
23	5.072749	0.00643730	0	Write	27244544	8
24	5.073164	0.00643730	0	Write	258235032	32
25	5.074025	0.00643730	0	Write	222408840	24
26	5.074473	0.00643730	0	Write	258235064	32
27	5.074910	0.00643730	0	Write	256123320	32
28	5.075335	0.00643730	0	Write	256123224	32
29	5.075825	0.00643730	0	Write	31155656	8
30	5.136805	0.00643730	0	Write	31155656	8
31	5.137212	0.00643730	0	Write	222408832	8
32	5.137901	0.00643730	0	Write	27244552	8

Объекты устройств для дисков

Драйвер дискового класса в Windows создает для представления дисков объекты устройств. Имена таких объектов имеют вид \Device\HarddiskX\DRX, где X — номер

диска. Для совместимости с приложениями, использующими старый способ именования, драйвер дискового класса создает символические ссылки с именами в формате Windows NT 4, ведущие к созданным драйвером объектам устройств. К примеру, драйвер диспетчера тома создает ссылку `\Device\Harddisk0\Partition0` для указания на устройство `\Device\Harddisk0\DR0` и ссылку `\Device\Harddisk0\Partition1` для указания на объект устройства первого раздела первого диска. Для обратной совместимости с приложениями, ожидающими старых вариантов имен, драйвер дискового класса создает также символические ссылки, представляющие физические накопители, которые могли бы создаваться в системах Windows NT 4. Соответственно, к примеру, ссылка `\GLOBAL??\PhysicalDrive0` указывает на устройство `\Device\Harddisk0\DR0`. Рисунок 9.5 демонстрирует служебную программу WinObj производства компании Sysinternals, которая показывает содержимое папки Harddisk базового диска. На правой панели находятся объекты устройств для физического диска и раздела.

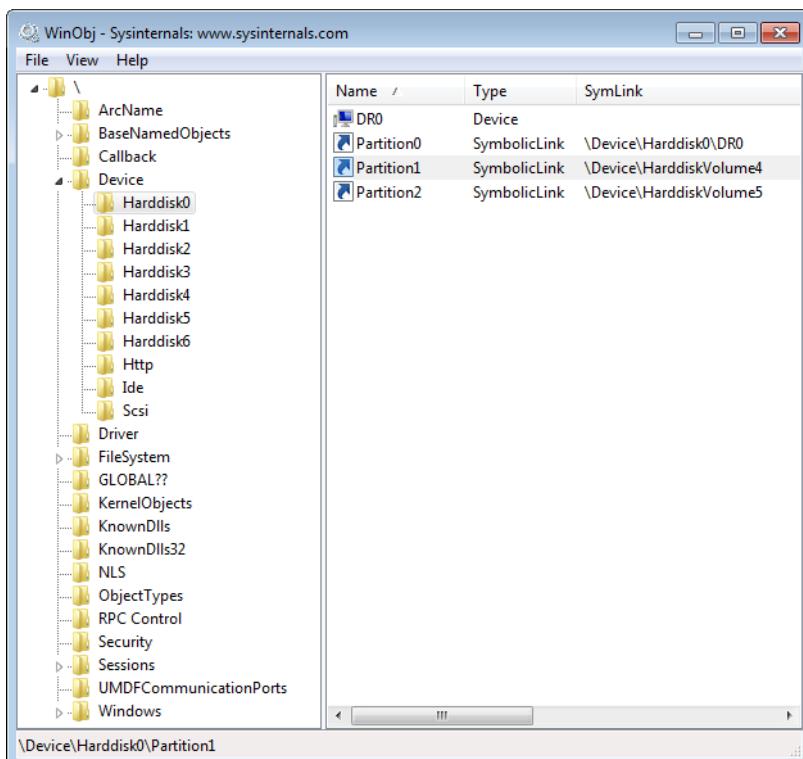


Рис. 9.5. Приложение WinObj демонстрирует содержимое папки Harddisk базового диска

Как упоминалось в главе 3 части I, Windows API не имеет представления о пространстве имен диспетчера объектов. Операционная система резервирует два подкаталога пространства имен, один из которых называется `\Global??`. (В другую группу входит набор подкаталогов для конкретных сеансов `\BaseNamedObjects`, который

мы рассматривали в главе 3.) В этом подкаталоге Windows предоставляет приложениям доступ к объектам устройств, а также к дискам, COM-портам и параллельным портам. Так как на самом деле объекты, представляющие диски, находятся в других подкаталогах, Windows пользуется символическими ссылками для связывания имен в папке `\Global??` с объектами, расположенными в других папках пространства имен. Для каждого физического диска в системе диспетчер ввода-вывода создает ссылку вида `\Global??\PhysicalDriveX`, указывающую на каталог `\Device\HarddiskX\DRX`, где X – числа начиная с 0. Windows-приложения, напрямую обращающиеся к секторам диска, открывают диск вызовом функции `CreateFile`, в качестве параметра которой указывается имя `\.\PhysicalDriveX`, где X – номер диска. (Следует заметить, что для прямого доступа к секторам присоединенного диска требуются права администратора.) Прикладной уровень Windows преобразует имя к виду `\Global??\PhysicalDriveX` и только потом передает его диспетчеру объектов.

Диспетчер разделов

Диспетчер разделов (`%SystemRoot%\System32\Drivers\Partmgr.sys`) отвечает за обнаружение, создание и удаление разделов, а также за управление разделами. Для обнаружения раздела он действует как функциональный драйвер применительно к созданным драйверами дискового класса объектам устройств, соответствующим дискам. Диспетчер разделов использует функцию `IoReadPartitionTableEx` диспетчера ввода-вывода для распознавания разделов и создания соответствующих им объектов устройств. Когда драйверы мини-порта представляют идентифицированные ими на стадии загрузки диски драйверу дискового класса, последний вызывает для каждого диска функцию `IoReadPartitionTableEx`. Эта функция активирует дисковый ввод-вывод на уровне сектора, который предоставляется драйверами класса, порта и мини-порта для чтения *главной загрузочной записи* (Master Boot Record, MBR) диска или *таблицы GUID разделов* (GUID Partition Table, GPT), о которой мы поговорим позже, конструирует внутреннее представление разделов диска и возвращает структуру `PDRIVE_LAYOUT_INFORMATION_EX`. Драйвер диспетчера разделов создает объекты устройств для всех основных разделов (включая логические диски внутри расширенных разделов), информацию о которых он получает от функции `IoReadPartitionTableEx`. Имена этих объектов имеют вид `\Device\HarddiskVolumeY`, где Y – номер раздела.

Диспетчер разделов отвечает также за наличие у всех дисков и разделов уникального идентификатора, каковым для MBR является подпись, а для GPT – GUID (Global Unique Identifier – глобальный уникальный идентификатор). Обнаружив два диска с одинаковым идентификатором, диспетчер пытается определить (путем записи на один диск и чтения с другого), что это такое – два разных диска или один, просматриваемый с разных путей (такое может случиться при отсутствии или некорректной работе MPIO-программ). В случае разных дисков диспетчер разделов делает один из них доступным для использования верхними слоями стека драйверов внешней памяти, переводя его в состояние подключения и отключая все прочие диски с этим идентификатором. Программы управления дисками и прикладные программные интерфейсы хранилища могут подключать отключенные диски, но диспетчер разделов в этом случае изменяет идентификатор для предотвращения конфликтов.

Управляя хранящимися в реестре атрибутами диска (такими, как `read-only` и `offline`), диспетчер разделов может скрывать разделы от диспетчера томов, что препятствует обнаружению томов в системе. Используют эти атрибуты также процедуры создания кластеров и виртуализации Hyper-V. Кроме того, диспетчер разделов направляет соответствующему диспетчеру томов операции записи, посланные непосредственно на диск, но попавшие в пространство разделов. Диспетчер томов разрешает или запрещает запись в зависимости от того, присоединен ли нужный том.

Управление томами

В Windows существует концепция *базовых* (*basic*) и *динамических* (*dynamic*) дисков. Диски, разбитые на разделы по схеме MBR или GPT, называются базовыми. В динамических дисках реализована более гибкая схема разбиения. Их основным отличием от базовых является умение создавать составные тома. В фигурирующем в начале главы списке терминов подчеркивалось, что составные тома превосходят простые по таким параметрам, как производительность, надежность и гибкость в изменении размеров. По умолчанию Windows рассматривает все диски как базовые. Динамический диск можно создать вручную или преобразовать в него базовый (если для этого хватает свободного места). Если функциональность составных томов вам не требуется, Microsoft рекомендует пользоваться базовыми дисками.

ПРИМЕЧАНИЕ

На базовых дисках Windows не поддерживает составные тома. На портативных компьютерах по ряду причин, в том числе потому, что они обычно имеют всего один диск, который невозможно просто переставить с одного компьютера на другой, Windows использует только базовые диски. Кроме того, динамическими могут быть только несъемные диски. Диски, подключенные к шинам IEEE 1394 или USB, а также диски, совместно используемые серверным кластером, всегда являются базовыми.

Базовые диски

В этом разделе описываются две схемы разбиения на разделы, MBR и GPT. С их помощью Windows определяет тома на базовых дисках. Кроме того, мы поговорим о драйвере диспетчера томов, представляющем тома драйверам файловых систем. Windows по умолчанию определяет все диски как базовые.

Схема MBR

Стандартные реализации BIOS, которыми пользуется аппаратное обеспечение, работающее на BIOS старого образца (не EFI) x86 (и x64), диктуют единственное требование к формату разбиения в Windows — первый сектор основного диска должен содержать главную загрузочную запись (MBR). При запуске компьютера на базе процессора x86 BIOS читает главную загрузочную запись и интерпретирует ее часть как исполняемый код. Именно этот код инициирует процесс загрузки операционной системы после того, как BIOS выполнит предварительное конфигурирование оборудования. В операци-

онных системах Microsoft, таких как Windows, в MBR также входит *таблица разделов* (partition table). Она состоит из четырех записей, определяющих местоположение на диске четырех *главных разделов* (primary partitions). Также в ней указываются типы разделов. Существует множество предопределенных типов, определяющих, какую файловую систему будет содержать раздел. К примеру, имеются типы разделов для FAT32 и NTFS.

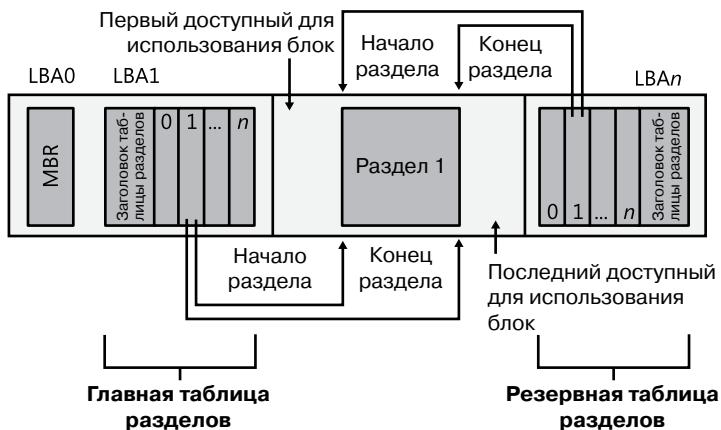
Специальным типом раздела является *расширенный раздел* (extended partition). Он содержит еще одну главную загрузочную запись с собственной таблицей разделов. Эквивалент главного раздела в этом случае называется *логическим диском* (logical drive). Расширенные разделы позволяют операционным системам производства Microsoft обходить ограничение, согласно которому на одном диске может располагаться не более четырех разделов. В общем случае рекурсия, позволяющая создавать расширенные разделы, является бесконечной, что означает отсутствие верхнего предела на количество разделов на диске. Разница между главными разделами и логическими дисками становится очевидной при загрузке Windows. Системе нужно пометить один из главных разделов основного жесткого диска как активный (загрузочный). Код, содержащийся в MBR, загружает в память код из первого сектора активного раздела (системного тома) и передает ему управление. Из-за роли, которую играет в процессе загрузки первый сектор первого раздела, Windows помечает первый сектор любого раздела как загрузочный. Как вы увидите в главе 13, все отформатированные файловой системой разделы имеют загрузочный сектор, в котором хранятся данные о структуре файловой системы в конкретном разделе.

Схема GPT

В рамках инициативы, направленной на создание стандартизованной и расширяющейся платформы микрокода, которую операционные системы могли бы использовать в процессе загрузки, корпорация Intel спроектировала спецификацию EFI (Extensible Firmware Interface). Изначально она была ориентирована на процессор Itanium. Затем компания Intel передала EFI в организацию Unified EFI Forum, которая занялась развитием и продвижением этой спецификации для процессоров x86, x64 и ARM. В UEFI входит среда операционной мини-системы, реализованная в виде микрокода (обычно это флэш-память). Операционные системы изначально применяли эту среду при старте для загрузки диагностических процедур и загрузочного кода. UEFI определяет стандарт, который называется GPT (GUID Partition Table – таблица GUID разделов). Этот стандарт должен был устранить ряд недостатков схемы разбиения MBR. К примеру, адреса секторов, используемые GPT-структурой, из 32-разрядных превратились в 64-разрядные. 32-разрядный адрес сектора обеспечивал доступ только к 2 Тбайт памяти, в то время как схема GPT в обозримом будущем должна повысить этот лимит. Другим преимуществом схемы GPT является использование циклически избыточного кода (Cyclic Redundancy Checksums, CRC), гарантирующего целостность таблицы разделов, а также резервное копирование этой таблицы. Название GPT появилось из-за того, что схема назначает каждому разделу не только 36-разрядное Unicode-имя, но и глобальный уникальный идентификатор (GUID).

Пример простой структуры GPT-раздела показан на рис. 9.6. Как и в схеме MBR, первый сектор GPT-диска содержит главную загрузочную запись, защищающую диск

от доступа из операционных систем, не поддерживающих GPT (это так называемая *защитная главная загрузочная запись*). Но второй и последний секторы диска хранят GPT-заголовки с реальной таблицей разделов, которая располагается после второго и перед последним сектором. Благодаря расширяемому списку разделов в схеме GPT отпадает необходимость во вложенных разделах, присутствующих в схеме MBR.



Примечание: LBA — логический адрес блока

Рис. 9.6. Пример структуры GPT-раздела

ПРИМЕЧАНИЕ

Так как Windows не поддерживает на базовых дисках создание составных томов, новый раздел базового диска эквивалентен тому. Поэтому в оснастке Disk Management консоли MMC для обозначения тома, созданного на базовом диске, используется термин «раздел» (partition).

Диспетчер томов на базовых дисках

Драйвер диспетчера томов (%SystemRoot%\System32\Drivers\Volmgr.sys) создает объекты устройства для дисков, представляющих тома на базовых дисках, и играет центральную роль в управлении всеми этими томами, в том числе простыми. Для каждого тома диспетчера создает объект устройства вида \Device\HarddiskVolumeX, где X — идентификатор тома (начиная с 1).

Диспетчера томов на самом деле является драйвером шины, так как отвечает за пересчет базовых дисков с целью распознавания базовых томов и сообщения о них PnP-диспетчеру. Для реализации этого пересчета диспетчера томов применяет PnP-диспетчера и драйвер диспетчера разделов (Partmgr.sys), чтобы определить существующие разделы базового диска. Диспетчера разделов регистрируется у PnP-диспетчера, чтобы получать от Windows сообщения о том, что драйвер дискового класса создал объект устройства для раздела. При этом сам он через закрытый интерфейс информирует диспетчера томов о новых объектах для разделов и создает объекты устройств для фильтра, которые потом

подключаются к объектам устройств для разделов. Наличие объектов устройств для фильтра побуждает Windows информировать диспетчер разделов обо всех удалениях объектов устройств для разделов, что, в свою очередь, позволяет диспетчеру разделов обновлять диспетчер томов. Драйвер дискового класса удаляет объект устройства для раздела, когда раздел удаляется в оснастке **Disk Management** консоли MMC. Как только диспетчер томов узнает о разделах, на основе данных о конфигурации базового диска он определяет соответствие между разделами и томами, а когда получает сведения о наличии всех разделов в описании тома, создает объект устройства для томов.

Затем Windows создает в папке \Global?? диспетчера объектов символические ссылки, указывающие на объекты устройств для томов, сформированных диспетчером томов. При первом обращении к тому со стороны операционной системы или приложения выполняется монтирование, позволяющее драйверам файловых систем распознать тома, отформатированные управляемой ими файловой системой, и заявить права на их использование. (Процесс монтирования описан чуть позже.)

Динамические диски

Как уже упоминалось, динамические диски в Windows требуются для создания составных томов, таких как зеркальные, чередующиеся и RAID-5 (о них мы поговорим чуть позже). Разбиение динамических дисков осуществляется с помощью диспетчера логических дисков (Logical Disk Manager, LDM), который является частью подсистемы Virtual Disk Service (VDS). Последняя включает в себя компоненты пользовательского режима и драйверов устройств и контролирует динамические диски. В отличие от схем разбиения MBR и GPT, LDM поддерживает одну унифицированную базу данных, хранящую сведения о разделах на всех динамических дисках системы, включая информацию о конфигурации составных томов.

База данных для LDM

В конце каждого динамического диска зарезервировано пространство размером 1 Мбайт, предназначенное для базы данных диспетчера логических дисков (LDM). Именно поэтому Windows требует наличия в конце базового диска свободного места перед преобразованием его в динамический. База данных для LDM состоит из показанных на рис. 9.7 областей: сектора заголовка, называемого закрытым (private header), таблицы содержимого, записей базы данных и журнала транзакций (последняя область на рис. 9.7 представляет собой обычную копию закрытого заголовка). Сектор закрытого заголовка располагается за 1 Мбайт до конца динамического диска и фиксирует положение базы данных. Чем больше вы работаете с Windows, тем более очевидным становится то, что система пользуется GUID для идентификации всего, чего только можно, в том числе дисков. Данный идентификатор представляет собой 128-разрядное значение, с помощью которого различные компоненты в Windows однозначно распознают объекты. LDM присваивает каждому динамическому диску свой глобальный уникальный идентификатор, а сектор закрытого заголовка отмечает у себя GUID диска, на котором он располагается. Данный заголовок именно потому называется закрытым, что содержит закрытую информацию о диске. Также в этом заголовке хранятся указатель на начало таблицы с содержимым базы и имя дисковой группы, которое образуется путем присоединения к имени компьютера символов Dg0

(к примеру, дисковая группа компьютера с именем Daryl будет называться Daryl-Dg0). В целях защиты от сбоев LDM хранит в последнем секторе диска копию закрытого заголовка.

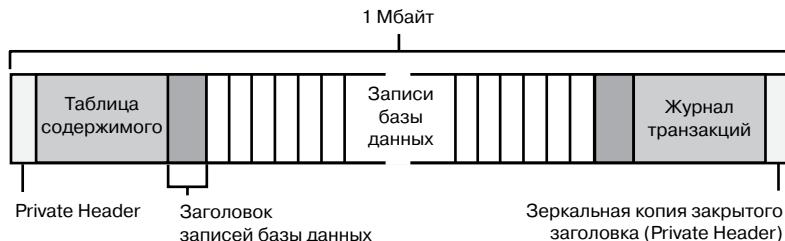


Рис. 9.7. Структура базы данных для LDM

Таблица содержимого состоит из 16 секторов и содержит сведения о структуре базы данных. Область записей начинается сразу после этой таблицы, с сектора, выделенного под заголовок записей базы. Этот сектор хранит информацию об области записей, в том числе о количестве имеющихся записей, имени и GUID дисковой группы, к которой относится база. Здесь же находится порядковый номер, используемый LDM для создания в базе следующей порции данных. За сектором заголовка записей следуют секторы с записями фиксированной длины (128 байт), в которых хранятся данные с описанием разделов и томов дисковой группы.

Данные в базе могут относиться к одному из четырех типов: разделу, диску, компоненту или тому. LDM использует эти типы для идентификации трех уровней описания томов. LDM связывает данные через внутренние идентификаторы объектов. На самом нижнем уровне *данные раздела* (partition entries) описывают мягкие разделы (о жестких разделах мы поговорим в этой главе чуть позже), образующие непрерывные области на диске; идентификаторы, сохраняемые в данных раздела, связывают эти данные с данными компонента и диска. *Данные диска* (disk entry) представляют динамический диск в составе дисковой группы и содержат GUID диска. *Данные компонента* (component entry) служат связующим звеном между данными одного или более разделов и данными тома, с которым ассоциирован каждый раздел. В *данных тома* (volume entry) хранятся GUID тома, его общий размер, состояние и буквенный идентификатор. Данные диска размером более 178 байт размещаются в нескольких записях базы данных. Данные раздела, компонента и тома редко занимают более одной записи.

Для описания простого тома LDM требуется три порции данных: раздела, компонента и тома. Следующий фрагмент кода демонстрирует содержимое простой базы данных для LDM — в нем определяется один том объемом 200 Мбайт, состоящий из одного раздела:

Disk Entry	Volume Entry	Component Entry	Partition Entry
Name: Disk1	Name: Volume1	Name: Volume1-01	Name: Disk1-01
GUID: XXX-XX...	ID: 0x408	ID: 0x409	ID: 0x407
Disk ID: 0x404	State: ACTIVE	Parent ID: 0x408	Parent ID: 0x409
	Size: 200MB		Disk ID: 0x404
	GUID: XXX-XX...		Start: 300MB
	Drive Hint: H:		Size: 200MB

Данные раздела описывают область на диске, выделенную системой для тома, данные компонента связывают данные раздела и тома, а данные тома содержат глобальный уникальный идентификатор (GUID), который Windows использует для внутренней идентификации тома. Для составных томов требуется более трех порций данных. К примеру, чередующийся том (который подробно рассматривается чуть позже) состоит, по меньшей мере, из двух порций данных раздела, а также данных компонента и тома. Единственным томом, который может иметь более одной порции данных компонента, является зеркальный том — он имеет две такие порции данных, причем каждая представляет свою сторону зеркала. LDM использует для зеркала две порции данных компонента, чтобы при удалении зеркала можно было разделить его на уровне компонента, создав два тома с одной порцией данных компонента для каждого.

Последней областью базы данных для LDM является область журнала транзакций, состоящая из нескольких секторов для хранения резервной информации о структуре базы в процессе ее изменения. Данная область может помочь при сбое в базе или отключении электропитания, так как журнал позволяет вернуть базу данных в рабочее состояние.

ЭКСПЕРИМЕНТ: ПРОСМОТР БАЗЫ ДАННЫХ ДЛЯ LDM ПРИ ПОМОЩИ ПРОГРАММЫ LDMDUMP

Утилита LDMDump, разработанная компанией Sysinternals, позволяет получить детальные сведения о содержимом базы данных для LDM. В качестве аргумента командной строки она принимает номер диска, а выводимая ею информация занимает несколько экранов, поэтому ее имеет смысл направлять в файл для последующего просмотра в текстовом редакторе, например с помощью команды ldmdump /d0 > disk.txt. Далее показан фрагмент данных, полученных с помощью утилиты LDMDump. Первым выводится заголовок базы данных для LDM, затем следуют ее записи, описывающие диск объемом 12 Гбайт с тремя динамическими томами по 4 Гбайт. Данные тома обозначены как Volume1. В конце списка LDMDump перечисляет смежные области на диске и присутствующие в базе определения томов.

```
C:\>ldmdump /d0
Logical Disk Manager Configuration Dump v1.03
Copyright (C) 2000-2002 Mark Russinovich

PRIVATE HEAD:
Signature : PRIVHEAD
Version : 2.12
Disk Id : b5f4a801-758d-11dd-b7f0-000c297f0108
Host Id : 1b77da20-c717-11d0-a5be-00a0c91db73c
Disk Group Id : b5f4a7fd-758d-11dd-b7f0-000c297f0108
Disk Group Name : WIN-SL5V78KD01W-Dg0
Logical disk start : 3F
Logical disk size : 7FF7C1 (4094 MB)
Configuration start: 7FF800
Configuration size : 800 (1 MB)
Number of TOCs : 2
TOC size : 7FD (1022 KB)
Number of Configs : 1
Config size : 5C9 (740 KB)
Number of Logs : 1
Log size : E0 (112 KB)
```

продолжение ↗

```
TOC 1:  
Signature : TOCBLOCK  
Sequence : 0x1  
Config bitmap start: 0x11  
Config bitmap size : 0x5C9  
Log bitmap start : 0x5DA  
Log bitmap size : 0xE0  
...  
VBLK DATABASE:  
0x000004: [000001] <DiskGroup>  
Name : WIN-SL5V78KD01W-Dg0  
Object Id : 0x0001  
GUID : b5f4a7fd-758d-11dd-b7f0-000c297f010  
0x000006: [000003] <Disk>  
Name : Disk1  
Object Id : 0x0002  
Disk Id : b5f4a7fe-758d-11dd-b7f0-000c297f010  
0x000007: [000005] <Disk>  
Name : Disk2  
Object Id : 0x0003  
Disk Id : b5f4a801-758d-11dd-b7f0-000c297f010  
0x000008: [000007] <Disk>  
Name : Disk3  
Object Id : 0x0004  
Disk Id : b5f4a804-758d-11dd-b7f0-000c297f010  
0x000009: [000009] <Component>  
Name : Volume1-01  
Object Id : 0x0006  
Parent Id : 0x0005  
0x00000A: [00000A] <Partition>  
Name : Disk1-01  
Object Id : 0x0007  
Parent Id : 0x3157  
Disk Id : 0x0000  
Start : 0x7C100  
Size : 0x0 (0 MB)  
Volume Off : 0x3 (0 MB)  
0x00000B: [00000B] <Partition>  
Name : Disk2-01  
Object Id : 0x0008  
Parent Id : 0x3157  
Disk Id : 0x0000  
Start : 0x7C100  
Size : 0x0 (0 MB)  
Volume Off : 0x7FE80003 (1047808 MB)  
0x00000C: [00000C] <Partition>  
Name : Disk3-01  
Object Id : 0x0009  
Parent Id : 0x3157  
Disk Id : 0x0000  
Start : 0x7C100  
Size : 0x0 (0 MB)  
Volume Off : 0xFFD00003 (2095616 MB)  
0x00000D: [00000F] <Volume>  
Name : Volume1  
Object Id : 0x0005  
Volume state: ACTIVE
```

```
Size : 0x017FB800 (12279 MB)
GUID : b5f4a806-758d-11dd-b7f0-c297f0108
Drive Hint : E:
```

Разбиение на разделы в стиле LDM и GPT или в стиле MBR

Первое, что нужно сделать при установке Windows, — создать раздел на главном физическом диске системы (в BIOS и UEFI он фигурирует как диск, с которого загружается система). Чтобы упростить включение BitLocker, программа Windows Setup создает маленький (размером 100 Мбайт) нешифрованный раздел, известный как *системный том* (system volume) и содержащий диспетчер загрузки (Bootmgr), хранилище конфигурационных данных загрузки (BCD) и другие необходимые на ранних этапах загрузки файлы. (По умолчанию этому тому не присваивается буквенное обозначение, но если вы хотите посмотреть содержимое тома в Проводнике, ему можно присвоить букву с помощью оснастки Disk Management консоли MMC, находящейся в файле %SystemRoot%\System32\Diskmgmt.msc.) Кроме того, Windows Setup требует создать раздел для загрузочного тома, в который запишутся системные файлы Windows и где будет располагаться системная папка (\Windows). Определения системного и загрузочного томов, данные Microsoft, несколько сбиваются с толку. Системным называется том, в который Windows помещает загрузочные файлы, например диспетчер загрузки, а загрузочным считается том, в котором Windows хранит остальные файлы операционной системы, такие как Ntoskrnl.exe (файл ядра операционной системы).

ПРИМЕЧАНИЕ

После подключения BitLocker загрузочный том шифруется, в то время как системный том никогда не шифруется.

Хотя данные о разбиении динамического диска на разделы находятся в базе данных, LDM реализует таблицы разделов в стиле MBR или GPT, чтобы загрузочный код Windows получил возможность найти на динамических дисках системный и загрузочный тома. (К примеру, Winload и микрокод Itanium не имеют представления о LDM-разделах.) Если на диске присутствует системный или загрузочный том, их положение указывается в таблице разделов MBR или GPT. В противном случае один раздел занимает всю доступную для использования область диска. LDM помечает его как раздел типа «LDM». В области, размеченной в стиле MBR или GPT, диспетчер логических дисков создает разделы, формируемые его базой данных. На дисках с разбиением в стиле MBR база данных для LDM располагается в скрытых секторах в самом конце диска, а на дисках в стиле GPT существует раздел с метаданными для LDM, содержащий базу данных. Этот раздел располагается в самом начале диска.

Кроме того, LDM создает таблицы разделов в стиле MBR или GPT, чтобы устаревшие программы обслуживания дисков, в том числе работающие в средах с двухвариантной загрузкой, не решили, что на динамическом диске отсутствуют разделы.

Так как LDM-разделы не описываются в таблицах MBR- и GPT-разделов, их называют *мягкими* (soft partitions); разделы же в стиле MBR и GPT называют *жесткими* (hard partitions). Структура динамического диска в стиле MBR показана на рис. 9.8.

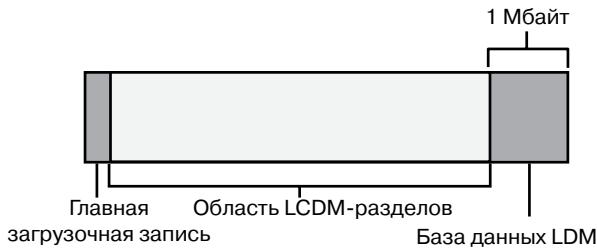


Рис. 9.8. Внутренняя структура динамического диска

Диспетчер томов для динамических дисков

Показанная на рис. 9.9 библиотека оснастки Disk Management консоли MMC (программа DMDiskManager, находящаяся в файле %SystemRoot%\System32\Dmdskmgr.dll) применяется при создании и редактировании базы данных для LDM. При запуске оснастки Disk Management она загружается в память и считывает со всех дисков базу данных для LDM. Полученная информация возвращается пользователю. Обнаружив базу данных из дисковой группы другого компьютера, она отмечает соответствующие тома как чужие, а если вы захотите им воспользоваться, позволяет импортировать их в базу текущего компьютера. При изменении конфигурации динамических дисков DMDiskManager обновляет в памяти свою копию базы данных. После подтверждения изменений DMDiskManager передает обновленную версию базы драйверу VolMgrX (%SystemRoot%\System32\Drivers\Volmgrx.sys). VolMgrX представляет собой динамически подключаемую библиотеку (Dynamic Link Library, DLL) режима ядра, обеспечивающую функциональность динамического диска для VolMgr. То есть она отвечает за доступ к расположенной на диске базе данных и создает объекты устройств, представляющие тома динамических дисков. Закрытие окна оснастки Disk Management останавливает работу программы DMDiskManager.

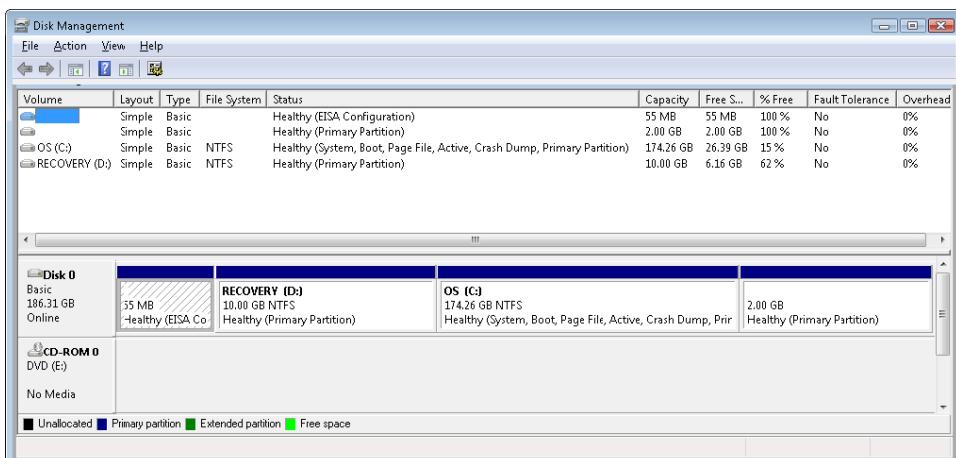


Рис. 9.9. Оснастка Disk Management

Управление составными томами

Драйвер VolMgr отвечает за представление томов, управляемых драйверами файловой системы, и за перенаправление адресованного томам ввода и вывода в нижележащие разделы, которые являются частями тома. В случае простых томов диспетчер томов преобразует смещение в томе в смещение на диске, суммируя смещение в томе со смещением тома от начала диска.

В случае составных томов имеет место более сложный процесс, так как составляющие том разделы могут оказаться несмежными или вообще располагаться на разных дисках. В некоторых типах составных томов имеет место избыточность данных, поэтому для них требуется еще более сложное преобразование. Следовательно, VolMgr использует VolMgrX для обработки всех адресованных составным томам запросов на ввод и вывод, определяя, на какие разделы будет в конечном счете влиять каждая из операций.

В Windows поддерживаются следующие типы составных томов:

- перекрытие;
- зеркальные;
- чередующиеся;
- RAID-5.

Мы начнем с рассмотрения конфигурации разделов составных томов и логических операций для всех типов томов, затем перейдем к способу обработки драйвером VolMgr IRP-пакетов, передаваемых системным драйвером составным томам. В посвященных составным томам материалах термин «диспетчер томов» будет использоваться для обозначения DLL-расширений драйверов VolMgr и VolMgrX.

Перекрытие тома

Перекрытым (spanned volume) называется единый логический том, состоящий из нескольких (до 32) свободных разделов на одном или нескольких дисках. Оснастка Disk Management консоли MMC объединяет разделы в перекрытый том, которым затем можно отформатировать для любой поддерживаемой Windows файловой системы. На рис. 9.10 представлен 100-гигабайтный перекрытый том, обозначенный буквой D и созданный из последней трети первого диска и первой трети второго. В Windows NT 4 перекрытые тома назывались *наборами томов* (volume sets).

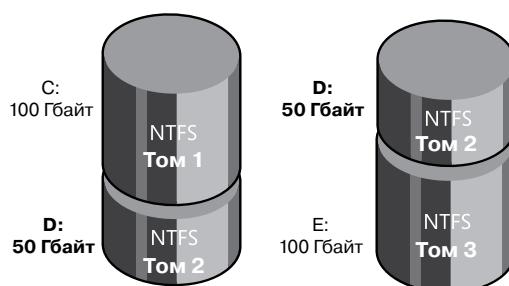


Рис. 9.10. Перекрытый том

Перекрытый том служит для объединения небольших областей свободного дискового пространства в единый том большего объема, а также для создания из двух и более маленьких дисков одного большого тома. Отформатированный для NTFS перекрытый том можно расширять, добавляя к нему свободные области или диски, и это никак не затронет уже хранящиеся на томе данные. Это является одним из важнейших преимуществ описания всех данных на NTFS-тome как некоего единого файла. Файловая система NTFS позволяет динамически увеличивать объем логического тома, так как регистрирующая состояния тома битовая карта является всего лишь очередным файлом — файлом битовой карты. Этот файл ничего не стоит расширить, включив в него добавляемое в том пространство. В то же время динамическое расширение FAT-тому потребовало бы расширения самой файловой системы FAT, что привело бы к смещению всех данных на диске.

Диспетчер томов в Windows скрывает от файловых систем физическую конфигурацию дисков. К примеру, на рис. 9.10 файловая система NTFS видит том D как обычный том объемом в 100 Гбайт. Чтобы определить, сколько места на этом томе можно выделить, NTFS смотрит на свою битовую карту. После преобразования битового смещения в смещение кластера файловая система вызывает диспетчера томов для чтения или записи данных, начиная с определенного смещения кластера на томе. Диспетчер томов видит физические секторы перекрытого тома как последовательно пронумерованные с первой свободной области первого диска до последней свободной области последнего диска. Он определяет, какой физический сектор на каком диске соответствует указанному смещению кластера.

Чередующиеся тома

Чередующимся томом (striped volume) называют группу разделов (их число может доходить до 32), в которой каждый раздел размещается на отдельном диске и которая объединяется в один логический том. Чередующиеся тома также известны как тома RAID-уровня 0 (RAID-0). Рисунок 9.11 демонстрирует такой том, состоящий из трех разделов — по одному на каждом из трех дисков. (Раздел чередующегося тома не обязательно занимает диск целиком; но разделы на всех дисках должны иметь один и тот же размер.)

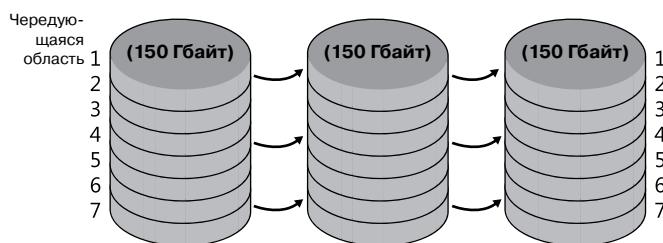


Рис. 9.11. Чередующийся том

Для файловой системы чередующийся том выглядит как единый том размером 450 Гбайт, но диспетчер томов оптимизирует хранение и время поиска данных на таком томе, распределяя данные между физическими дисками. Диспетчер томов обращается

к физическим секторам дисков, как будто они последовательно пронумерованы по чередующимся областям, как показано на рис. 9.12.

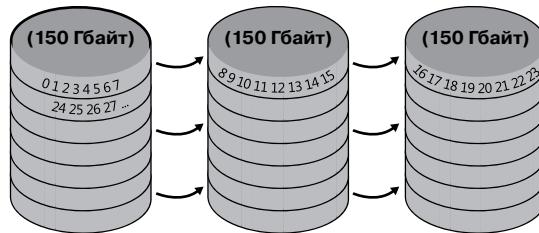


Рис. 9.12. Логическая нумерация физических секторов в чередующихся томах

Так как каждая чередующаяся область занимает всего 64 Кбайт (это значение выбрано для того, чтобы отдельные операции чтения и записи не требовали обращения сразу к двум дискам), данные между дисками распределяются более-менее равномерно. Таким образом, чередование повышает вероятность того, что несколько одновременно ожидающих выполнения операций ввода и вывода потребуют доступа к разным дискам. А так как к данным на всех трех дисках можно обращаться одновременно, время задержки при дисковом вводе-выводе часто снижается, особенно в сильно загруженных системах.

Перекрытие тома упрощают управление томами на диске, а чередующиеся позволяют распределять операции ввода-вывода между дисками. Однако ни те ни другие не могут обеспечить восстановления данных при сбое диска. Для устранения этого недостатка диспетчер томов реализует две избыточные схемы хранения: зеркальные тома и тома RAID-5. В Windows они задаются через инструмент администрирования Disk Management.

Зеркальные тома

В зеркальном томе (mirrored volume) содержимое раздела одного диска дублируется в разделе такого же размера на другом диске, как показано на рис. 9.13. Такие тома иногда называют томами RAID-уровня 1 (RAID-1).

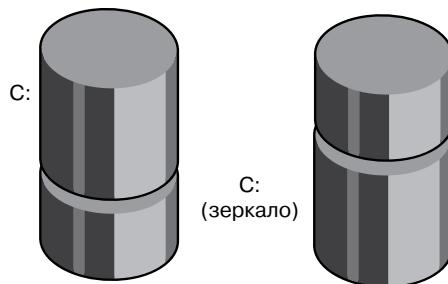


Рис. 9.13. Зеркальный том

Когда программа пишет данные на диск С, диспетчер томов помещает их на аналогичный участок зеркального раздела. Если данные на первом диске или в разделе С оказываются нечитабельными из-за аппаратного или программного сбоя, диспетчер томов автоматически обращается за данными к зеркальному разделу. Зеркальный том можно отформатировать под любую поддерживающую Windows файловую систему. При этом драйверы файловых систем остаются независимыми, на них никак не влияют операции диспетчера томов по созданию зеркальных копий.

Зеркальные тома способствуют увеличению пропускной способности операций чтения-записи в сильно загруженных системах. При высокой интенсивности ввода-вывода диспетчер томов распределяет операции чтения между основным и зеркальным разделами (принимая во внимание количество незавершенных запросов на ввод и вывод от каждого диска). Две операции чтения допускают одновременное выполнение и теоретически завершаются за вдвое меньшее время. При редактировании файла требуется сделать запись в оба раздела зеркального набора, но записи на диск выполняются параллельно, поэтому дополнительное обновление диска в общем случае не влияет на производительность программ в пользовательском режиме.

Зеркальные тома являются единственными из составных томов, которые поддерживаются как системным, так и загрузочным томами. Дело в том, что загрузочный код Windows, в том числе код MBR и Winload, не обладают сложной логикой, необходимой для работы с составными томами. Исключением являются зеркальные тома, так как загрузочный код воспринимает их как простые, считывая данные с той половины зеркального тома, которая в таблице разделов MBR помечена как загрузочный или системный диск. Так как загрузочный код не редактирует метаданные диска и осуществляет чтение и запись на одну и ту же половину зеркального набора, он может просто проигнорировать вторую половину. Однако программа диспетчера загрузки и загрузчик OS обновят файл \Boot\BootStat.dat на системном томе. Этот файл служит только для передачи сведений о состоянии между различными фазами загрузки, именно поэтому его можно не записывать на вторую сторону зеркала.

ЭКСПЕРИМЕНТ: НАБЛЮДЕНИЕ ЗА ОПЕРАЦИЯМИ ВВОДА-ВЫВОДА НА ЗЕРКАЛЬНОМ ТОМЕ

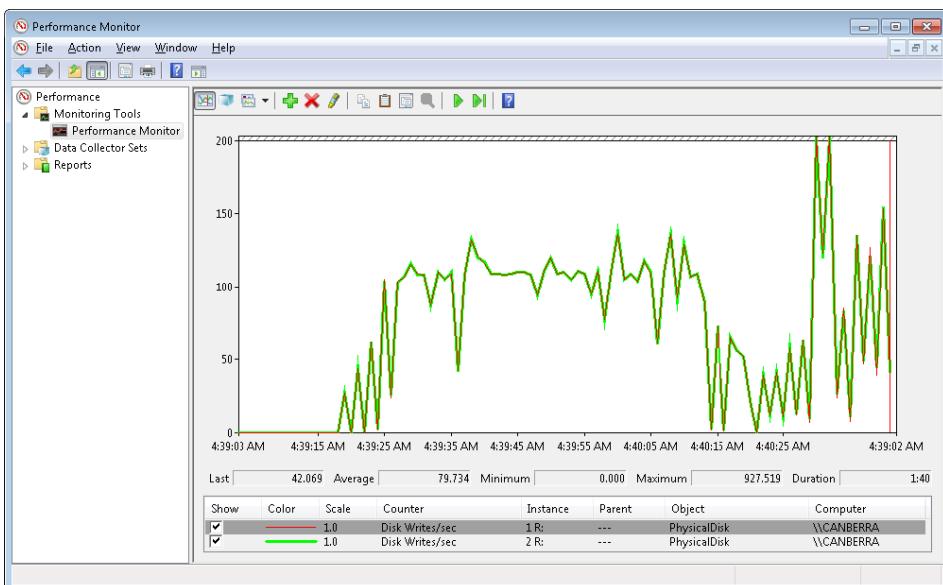
При помощи оснастки Performance Monitor можно убедиться, что при записи на зеркальные тома данные копируются на оба входящих в зеркало диска, в то время как операции чтения, если они происходят не слишком часто, выполняются в основном на одной из половин тома. Для эксперимента потребуется система с тремя жесткими дисками. Если они у вас отсутствуют, пропустите подготовительные инструкции и переходите сразу к снимку экрана, демонстрирующему результаты эксперимента.

Воспользуйтесь оснасткой Disk Management консоли MMC для создания зеркального тома:

1. Запустите программу Computer Management (Управление компьютером), раскройте узел Storage (Запоминающие устройства) и щелчком мыши выберите папку Disk Management (Управление дисками). Также можно открыть Disk Management как оснастку консоли MMC.
2. Щелкните правой кнопкой мыши на свободном пространстве диска и выберите команду New Simple Volume (Создать простой том).

3. Следуйте инструкциям Мастера создания тома (New Simple Volume Wizard). Обязательно предварительно убедитесь, что на другом диске достаточно места для создания тома равного размера.
4. Щелкните правой кнопкой мыши на новом томе и выберите в появившемся меню команду Add Mirror (Добавить зеркало).

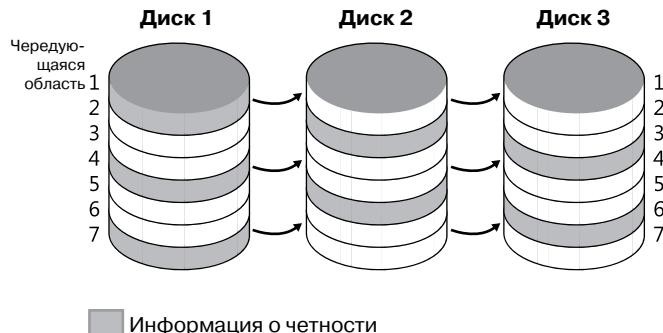
После создания зеркального тома запустите инструмент Performance Monitor (Монитор производительности) и добавьте счетчики к объекту PhysicalDisk (Физический диск) для каждого экземпляра диска, на котором содержится раздел зеркального тома. Выберите счетчики Disk Writes/sec (Обращений записи на диск/с). На третьем диске (который не является частью зеркального тома) выберите большую папку и скопируйте ее на зеркальный том. По мере выполнения копирования окно монитора производительности станет походить на показанное на рисунке.



Две верхние касающиеся линии представляют собой графики изменения параметра Disk Writes/sec для каждого из дисков. Снимок экрана демонстрирует, что диспетчер тома (в данном случае это драйвер VolMgr) записывает данные из копируемого файла в обе половины тома.

RAID-5

Том RAID-5 представляет собой отказоустойчивый вариант обычного чередующегося тома. В таких томах реализуется технология RAID уровня 5, и так как они основаны на уже знакомом нам принципе чередования, их называют *чертежами с записью четности* (striped volumes with rotated parity). Отказоустойчивость достигается за счет выделения одного диска под хранение информации о четности для каждой чередующейся области. Структура тома RAID-5 показана на рис. 9.14.

**Рис. 9.14.** Том RAID-5

Из рисунка видно, что информация о четности чередующейся области 1 хранится на диске 1. Она представляет собой побайтовую логическую сумму (XOR) первых чередующихся областей с дисков 2 и 3. Информация о четности чередующейся области 2 хранится на диске 2, о четности чередующейся области 3 – на диске 3. Подобное распределение данных о четности по дискам позволяет оптимизировать операции ввода и вывода. При каждой записи данных на диск байты четности, соответствующие изменяемым байтам, должны пересчитываться и перезаписываться. Если бы информация о четности записывалась на один диск, он был бы все время занят, что создавало бы препятствие для операций ввода и вывода.

Основой для восстановления диска после сбоя в томе RAID-5 служит простой арифметический принцип: если в уравнении с n переменными известны значения $n - 1$ переменной, недостающее значение можно определить вычитанием. Например, в уравнении $x + y = z$, в котором z представляет собой чередующуюся область четности, диспетчер томов вычисляет $z - y$ для определения x ; а чтобы найти y , он вычисляет $z - x$. Аналогичной логикой пользуется диспетчер томов для восстановления потерянных данных. Если диск на томе RAID-5 выходит из строя или данные на нем становятся нечитабельными, диспетчер томов реконструирует их с помощью операции XOR (побитового логического сложения).

При сбое на диске 1 содержимое его чередующихся областей 2 и 5 вычисляется побитовым логическим сложением соответствующих чередующихся областей на диске 3 с чередующимися областями четности на диске 2. Содержимое чередующихся областей 3 и 6 с диска 1 определяется аналогичным образом, то есть побитовым логическим сложением соответствующих чередующихся областей на диске 2 с чередующимися областями четности на диске 3. Для создания тома RAID-5 требуются по меньшей мере три диска (точнее, три одинаковых по размеру раздела на трех дисках).

Пространство имен томов

Механизм пространства имен томов отвечает за присвоение объектам устройств, представляющим тома, буквенных обозначений, которые дают Windows-приложениям возможность привычным образом обращаться к этим дискам, а также обеспечивают такую функциональность, как монтирование и размонтирование.

Диспетчер монтирования

Драйвер диспетчера монтирования (%SystemRoot%\System32\Drivers\Mountmgr.sys) назначает буквы созданным после установки Windows томам динамических и базовых дисков, устройствам CD-ROM, приводам гибких дисков и съемным устройствам. Windows хранит все назначенные томам буквы в разделе HKLM\SYSTEM\MountedDevices реестра. Заглянув в этот раздел, вы увидите параметры с именами вида \??\Volume{X} (здесь X – GUID) и такие значения, как \DosDevices\C:. Каждый том имеет запись со своим именем, но далеко не всем томам назначаются буквы дисков (к примеру, системный том не имеет буквы). На рис. 9.15 в качестве примера показано содержимое раздела реестра диспетчера монтирования. Обратите внимание, что раздел MountedDevices не входит в набор параметров управления и в связи с этим не восстанавливается при загрузке последней удачной конфигурации. (Подробно о параметрах управления и их восстановлении речь пойдет в главе 13.)

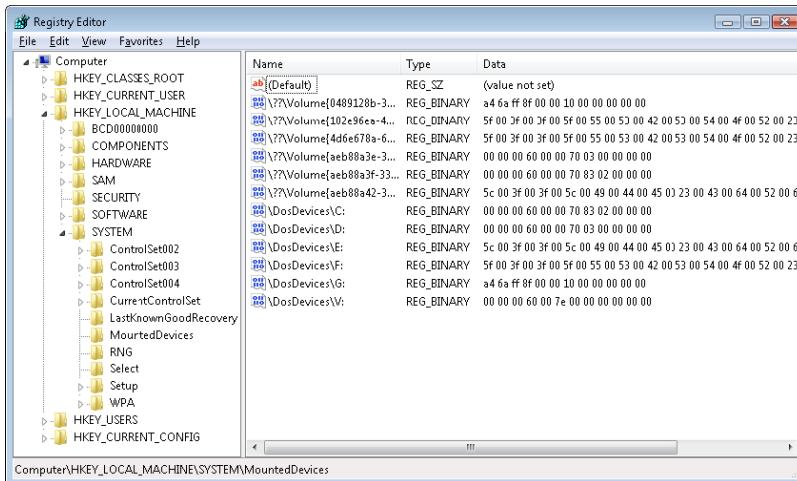


Рис. 9.15. Смонтированные устройства, перечисленные в разделе реестра, принадлежащем диспетчеру монтирования

Данные, которые хранятся в виде параметров реестра для букв и имен томов базовых дисков, представляют собой сигнатуру диска и начальное смещение от первого отображеного на том раздела. Аналогичные данные для томов динамических дисков включают внутренний глобальный уникальный идентификатор тома, используемый драйвером VolMgr. В процессе инициализации диспетчера монтирования при загрузке системы он регистрируется в подсистеме Plug and Play, что в дальнейшем позволяет ему получать уведомления, когда устройство идентифицирует себя как том. При получении такого уведомления диспетчер монтирования определяет GUID или сигнатуру диска нового тома и использует найденное значение как поисковый критерий для своей внутренней базы данных, отражающей содержимое раздела MountedDevices реестра. Затем диспетчер монтирования определяет, присутствуют ли в базе назначенные томам буквы. Если запись для определенного тома отсутствует, диспетчер запрашивает

у драйвера VolMgr букву для идентификации и сохраняет ее в базе. Простым томам VolMgr не предлагает никаких букв, а в случае динамических томов ищет подсказки среди данных тома в базе данных.

Если для динамического тома никаких предложений не поступает, диспетчер мониторинга берет первую свободную букву (если таковая существует), определяет новое присваивание и создает для него символьическую ссылку (например, \Global??\D:). После этого он обновляет раздел MountedDevices реестра. При отсутствии свободных букв буква тому просто не назначается. Одновременно диспетчер мониторинга создает символьическую ссылку (то есть \Global??\Volume{X}), определяющую новый глобальный уникальный идентификатор тома, если таковой пока отсутствует. Этот идентификатор отличается от идентификаторов томов, которыми пользуется драйвер VolMgr.

Точки монтирования

Точки монтирования (mount points) дают возможность связывать тома через NTFS-папки, что позволяет получать доступ даже к тем томам, которым не было назначено букв. Например, NTFS-папка, которой вы присвоили имя C:\Projects, может монтировать другой том (NTFS или FAT), содержащий папки и файлы ваших проектов. Если в томе с проектами есть файл с именем \CurrentProject\Description.txt, доступ к нему осуществляется по пути C:\Projects\CurrentProject\Description.txt. Точки монтирования стали возможны благодаря технологии точек повторной обработки, которая подробно рассматривается в главе 12.

Точкой повторной обработки (reparse point) называют блок произвольных данных с неким фиксированным заголовком, который Windows ассоциирует с NTFS-файлом или NTFS-папкой. Приложение или система определяет формат и поведение точки повторной обработки, в том числе значение уникального тега, который идентифицирует принадлежность этой точки к приложению или системе и говорит о размере и назначении составляющего ее фрагмента данных. (Максимально допустимый размер фрагмента – 16 Кбайт.) Любое приложение, реализующее точку повторной обработки, должно предоставить фильтрующий драйвер файловой системы для наблюдения за кодами возврата связанных с повторной обработкой файловых операций, которые выполняются на NTFS-томах. Обнаружив такой код, драйвер должен предпринять соответствующие действия. NTFS возвращает код состояния повторной обработки каждый раз, когда при выполнении файловой операции обнаруживает связанный с точкой повторной обработки файл (или папку).

Драйвер файловой системы NTFS, диспетчер ввода-вывода и диспетчер объектов частично реализуют функциональность точки повторной обработки. Диспетчер объектов инициирует операции анализа путей к файлам, взаимодействуя с драйверами файловой системы через диспетчер ввода-вывода, и поэтому диспетчер объектов должен повторно инициировать операции, для которых диспетчер ввода-вывода возвращает код состояния повторной обработки. Диспетчер ввода-вывода поддерживает модификацию путей, которая может потребоваться точкам монтирования и другим точкам повторной обработки, а драйвер файловой системы NTFS должен связывать данные этих точек с файлами и папками. Поэтому данный диспетчер можно рассматривать как фильтрующий драйвер файловой системы, поддерживающий функциональность повторной обработки для многих точек, определенных Microsoft.

Одним из распространенных примеров применения точек повторной обработки является функциональность символических ссылок, появившаяся в Windows благодаря NTFS (более подробно символические ссылки в NTFS рассматриваются в главе 12). Если файл или папка, для которой диспетчер ввода-вывода получает от NTFS код состояния повторной обработки, не ассоциирована с одной из предустановленных точек повторной обработки, значит, ее точка не обрабатывается ни одним фильтрующим драйвером. В этом случае диспетчер ввода-вывода сообщает диспетчеру объектов об ошибке «file cannot be accessed by the system» («система не может получить доступ к файлу»), которая передается обратившемуся к файлу или папке приложению.

Точки монтирования представляют собой точки повторной обработки, в которых имя тома (`\Global??\Volume{X}`) хранится в виде данных повторной обработки. Назначая или удаляя пути для томов в оснастке Disk Management консоли MMC, вы создаете точки монтирования. Создавать и показывать точки монтирования можно также с помощью встроенной утилиты командной строки Mountvol.exe (`%SystemRoot%\System32\Mountvol.exe`).

Диспетчер монтирования поддерживает на каждом NTFS-тome удаленную базу данных, в которую записываются все определенные для конкретного тома точки монтирования. Файл этой базы находится в папке `System Volume Information` NTFS-тoma. При перемещении диска из одной системы в другую или в среду с двухвариантной загрузкой (например, с возможностью загружать разные версии Windows) перемещаются и точки монтирования (благодаря наличию удаленной базы данных диспетчера монтирования). Кроме того, NTFS отслеживает точки повторной обработки и в файле метаданных `\$Extend\$Reparse`. (Ни один файл метаданных недоступен приложениям.) Сведения о точках повторной обработки NTFS хранят в файле метаданных, поэтому Windows в состоянии, к примеру, легко перебрать определенные для тома точки монтирования (которые являются точками повторной обработки) при поступлении соответствующего запроса от Windows-приложения, скажем, от Disk Management.

Монтирование томов

Тот факт, что Windows присваивает тому букву диска, еще не означает наличия на томе данных, систематизированных в формате известной Windows файловой системы. Процесс распознавания тома начинается с того, что какая-либо файловая система объявляет раздел своим; первый раз этот процесс запускается при обращении ядра, драйвера устройства или приложения к файлу или папке на данном томе. После того как драйвер файловой системы уведомляет о взятии на себя ответственности за управление разделом, диспетчер ввода-вывода направляет все адресованные этому тому IRP-пакеты указанному драйверу. В операцию монтирования в Windows вовлечено три компонента: драйвер файловой системы, блоки параметров тома (Volume Parameter Blocks, VPB) и запросы на монтирование.

ПРИМЕЧАНИЕ

Диспетчер разделов придерживается SAN-политик, устанавливаемых утилитой DiskPart, которая указывает, будет ли диспетчер томов видеть диски, подключенные к сети хранения данных. По умолчанию в версиях Windows Server 2008 Enterprise и Datacenter они невидимы, что позволяет избежать автоматического монтирования их томов.

Диспетчер ввода-вывода обеспечивает контроль над процессом монтирования и имеет сведения обо всех доступных драйверах файловых систем, так как именно в нем эти драйверы регистрируются в ходе инициализации. Диспетчер ввода-вывода предоставляет для регистрации драйверов файловых систем на локальных (не сетевых) дисках функцию `IoRegisterFileSystem`. При регистрации драйвера диспетчер ввода-вывода сохраняет ссылку на драйвер в списке, который используется в операциях монтирования.

Каждый объект устройства содержит VPB-структуру данных, но диспетчер ввода-вывода учитывает только VPB объектов устройств для томов. VPB служит ссылкой от объекта устройства для тома на тот объект устройства, который драйвер файловой системы создает для представления экземпляра файловой системы, смонтированной для рассматриваемого тома. Пустая VPB-ссылка на файловую систему (`VPB->DeviceObject == NULL`) означает, что на томе нет ни одной смонтированной файловой системы. Диспетчер ввода-вывода проверяет VPB объекта устройства для тома при вызове любого прикладного программного интерфейса, задающего имя открываемого файла или папки на данном объекте.

К примеру, если диспетчер монтирования назначает второму тому системы букву D, он создает символическую ссылку `\Global??\D:`, представляющую объект `\Device\HarddiskVolume2`. Windows-приложение, пытающееся открыть файл `\Temp\Test.txt` на диске D, укажет имя `D:\Temp\Test.txt`, которое подсистема Windows перед вызовом отвечающей за открытие процедуры ядра `NtCreateFile` преобразует в `\Global??\D:\Temp\Test.txt`. Для преобразования имени в понятный машине формат эта процедура использует диспетчер объектов, который обнаруживает объект устройства `\Device\HarddiskVolume2` с еще не разрешенным путем `\Temp\Test.txt`. На этом этапе диспетчер ввода-вывода проверяет, присутствует ли в VPB объекта `\Device\HarddiskVolume2` ссылка на файловую систему. При ее отсутствии диспетчер ввода-вывода отправляет всем зарегистрированным драйверам файловой системы запрос на монтирование, чтобы выяснить, распознает ли конкретный драйвер формат монтируемого тома как формат файловой системы.

ЭКСПЕРИМЕНТ: ПРОСМОТР БЛОКА ПАРАМЕТРОВ ТОМА

Увидеть содержимое VPB позволяет команда `!vpb` отладчика ядра. Так как на VPB указывает объект устройства для тома, начать следует с локализации указанного объекта. Для этого создается дамп объекта драйвера диспетчера томов и ищется объект устройства, представляющий том, в котором присутствует поле `Vpb`.

```
1kd> !drvobj volmgr
Driver object (84905030) is for:
\Driver\volmgr
Driver Extension List: (id , addr)

Device Object list:
84a64780 849d5b28 84a64518 84a64030
84905e00
```

Команда `!drvobj` выводит список адресов, принадлежащих драйверу объектов устройств. В нашем примере таких объектов пять. Один из них представляет программный (управляющий) интерфейс для драйвера устройства, а остальные являются объектами устройств

для томов. Так как порядок перечисления объектов противоположен порядку их создания, а первым драйвер создает объект для управляющего устройства, значит, первым в списке фигурирует объект, представляющий том. Теперь выполним команду !devobj, указав в качестве параметра адрес объекта устройства для тома:

```
1kd> !devobj 84a64780
Device object (84a64780) is for:
HarddiskVolume4 \Driver\volmgr DriverObject 84905030
Current Irp 00000000 RefCount 0 Type 00000007 Flags 00001050
Vpb 84a64228 Dacl 8b1a8674 DevExt 84a64838 DevObjExt 84a64930 Dope
849fd838 DevNode 849d5938
ExtensionFlags (0x00000800)
Unknown flags 0x00000800
AttachedDevice (Upper) 84a66020 \Driver\volsnap
Device queue is not busy
```

Команда !devobj показывает нам поле VPB для этого объекта устройства. (Объект называется HarddiskVolume4.) Теперь можно воспользоваться командой !vpb:

```
1kd> !vpb 84a64228
Vpb at 0x84a64228
Flags: 0x1 mounted
DeviceObject: 0x84a6b020
RealDevice: 0x849d5b28
RefCount: 4311
Volume Label: OS
```

Теперь понятно, что объект устройства для тома смонтирован драйвером файловой системы, который присвоил тому имя OS. Поле RealDevice в блоке параметров тома указывает обратно на объект устройства для тома, а поле DeviceObject — на объект устройства, смонтированный для файловой системы.

Вы можете воспользоваться командой !devobj, указав данный адрес, и получить дополнительные сведения о смонтированной файловой системе — в следующем примере показано, что на том монтировалась файловая система NTFS:

```
1kd> !devobj 0x84a6b020
Device object (84a6b020) is for:
\FileSystem\Ntfs DriverObject 84a02ad0
Current Irp 00000000 RefCount 0 Type 00000008 Flags 00040000
DevExt 84a6b0d8 DevObjExt 84a6bc00
ExtensionFlags (0x00000800)
Unknown flags 0x00000800

AttachedDevice (Upper) 84a63ac0 \FileSystem\FltMgr
Device queue is not busy
```

По правилам драйвер файловой системы для распознавания формата монтируемого тома должен проанализировать загрузочную запись тома (Volume Boot Record, VBR), хранящуюся в первом секторе этого тома. Загрузочные записи файловых систем производства Microsoft содержат поле, хранящее данные о формате файловой системы. Его обычно проверяют драйверы файловой системы и при обнаружении распознаваемого ими формата начинают анализировать остальную информацию из загрузочной записи,

а именно имя файловой системы и данные, необходимые драйверу для обнаружения на томе важных файлов с метаданными. К примеру, NTFS распознает том, только если в MBR-разделе в поле *Type* указано NTFS (0x07), в поле *Name* — "NTFS", а описанные загрузочной записью файлы метаданных находятся в согласованном состоянии.

Если драйвер файловой системы подтверждает распознавание, диспетчер ввода-вывода заполняет VPB и передает запрос на открытие с оставшейся частью пути (то есть \Temp\Test.txt) драйверу файловой системы, который, в свою очередь, завершает запрос, используя для интерпретации хранящихся на томе данных свой формат файловой системы. После заполнения нужной информацией полей VPB объекта устройства для тома диспетчер ввода-вывода передает все последующие адресованные тому запросы драйверу смонтированной файловой системы. Если том не объявляет своим ни один из драйверов файловой системы, это сделает драйвер Raw, встроенный в Ntoskrnl.exe, и пресечет все попытки открыть файл в данном разделе. Впрочем, драйвер Raw допускает ввод-вывод сектора в рамках раздела для приложений с правами администратора, хотя даже администратор не может осуществлять запись в секторы монтированного тома, за исключением загрузочных секторов. На рис. 9.16 представлена упрощенная схема пути, по которому осуществляются ввод и вывод на смонтированном томе (взаимодействие драйвера файловой системы с диспетчерами кэша и памяти в данном случае не показано).

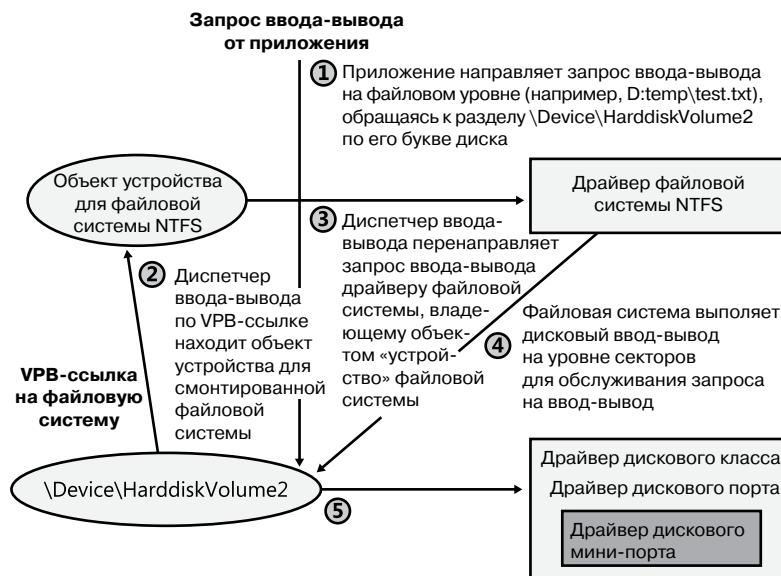


Рис. 9.16. Процесс ввода-вывода на смонтированном томе

Вместо загрузки всех драйверов файловых систем вне зависимости от наличия соответствующих томов Windows пытается минимизировать нагрузку на память, используя для предварительного распознавания файловой системы суррогатный драйвер File System Recognizer (%SystemRoot%\System32\Drivers\Fs_rec.sys). Он знает о формате всех поддерживаемых Windows файловых систем ровно столько, сколько требуется для анализа загрузочной записи и определения ее соответствия какому-либо драйверу

файловой системы Windows. При загрузке системы File System Recognizer регистрируется как драйвер файловой системы, и когда диспетчер ввода-вывода вызывает его в процессе монтирования файловой системы на новом томе, он загружает нужный драйвер файловой системы, если VBR описывает еще не загруженную файловую систему. После этого File System Recognizer направляет IRP монтирования драйверу файловой системы и позволяет ему заявить права на том.

Кроме загрузочного тома, драйвер которого монтируется при инициализации ядра, драйверы файловых систем монтируют большинство томов в момент запуска приложения Chkdsk для проверки целостности файловой системы на этапе загрузки. Загрузочная версия Chkdsk является встроенным приложением (в отличие от Win32-приложений) и называется Autochk.exe (%SystemRoot%\System32\Autochk.exe). Диспетчер сеансов (%SystemRoot%\System32\Smss.exe) запускает его так, как это указано в параметре HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\BootExecute. Приложение Autochk проверяет все буквы дисков, чтобы выяснить, не требуют ли соответствующие им тома проверки целостности.

Единственным случаем, когда монтирование одного диска может осуществляться более одного раза, является сменимый носитель. Драйверы файловых систем в Windows реагируют на смену носителя запросом идентификатора тома диска. Если оказывается, что идентификатор сменился, драйверы демонтируют диск и монтируют его заново.

Ввод и вывод на томах

Драйверы файловых систем управляют хранящимися на томах данными, но для взаимодействия с драйверами устройств внешней памяти при передаче данных им требуется содействие диспетчера томов. Драйверы файловых систем в процессе монтирования получают ссылки на объекты томов этого диспетчера, а затем через эти объекты посылают диспетчеру запросы. Запросы диспетчера томов в обход драйверов файловой системы могут посыпать и приложения, когда возникает необходимость напрямую обратиться к данным тома. К числу таких приложений относятся, например, программы восстановления удаленных файлов.

Каждый раз, когда драйвер файловой системы или приложение посыпает представляемому том объекту устройства запрос на ввод или вывод (поступающий в виде IRP-пакета), диспетчер ввода-вывода направляет его диспетчеру томов, создавшему целевой объект устройства. Соответственно, если приложение (запущенное с правами администратора) хочет прочитать загрузочный сектор второго тома в системе (в данном примере это простой том), оно открывает объект \\.\HarddiskVolume2 и вызывает функцию `ReadFile` для чтения 512 байт, начинающихся с нулевого смещения на устройстве. (При этом и начальное байтовое смещение, и длина должны быть кратны размеру сектора.) Диспетчер ввода-вывода передает запрос приложения в виде IRP-пакета диспетчеру томов, владеющему данным объектом устройства, с уведомлением, что пакет адресован устройству HarddiskVolume2.

Так как тома являются логическими структурами, при помощи которых Windows представляет непрерывные области одного или нескольких физических дисков, диспетчеру томов приходится преобразовывать смещения, указанные относительно тома, в смещения от начала диска. Если том 2 состоит из одного раздела, начинающегося с 4096 секторов на диске, диспетчер раздела перед передачей запроса драйверу дискового класса соответствующим образом корректирует параметры IRP. Драйвер дискового

класса для реализации ввода и вывода на физическом диске и чтения запрошенных данных в указанный в IRP буфер приложения пользуется драйвером мини-порта.

Роль диспетчера томов в обработке запросов к составным томам помогут прояснить следующие примеры. Если чередующийся том состоит из двух разделов (1 и 2), объект устройства VolMgr перехватывает дисковый ввод-вывод файловой системы, нацеленный на объект устройства для тома, а драйвер VolMgr корректирует запрос перед тем, как передать его драйверу дискового класса. В результате этой коррекции запрос начинает ссылаться на нужное смещение относительно начала целевой чередующейся области раздела 1 или 2. Если ввод-вывод затрагивает оба раздела тома, драйвер VolMgr должен выдать два дополнительных запроса на ввод-вывод — по одному к каждому диску. Это иллюстрирует рис. 9.17.

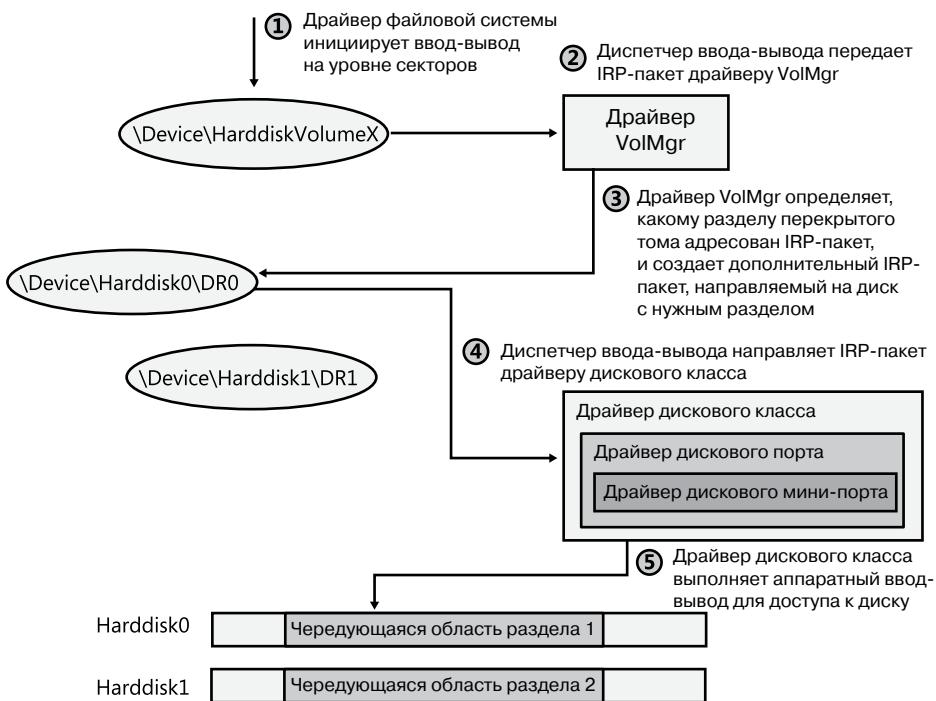


Рис. 9.17. Операции ввода-вывода драйвера VolMgr

При записи на зеркальный том VolMgr делит каждый запрос таким образом, чтобы операция записи выполнялась для каждой половины тома. При запросе на чтение с зеркального тома VolMgr читает только с одной половины, обращаясь к другой, только если выполнить операцию не удается.

Служба виртуальных дисков

Компания, выпускающая продукцию, имеющую отношение к внешней памяти, например RAID-адAPTERы, жесткие диски или массивы хранения данных, вынуждена

создавать собственные приложения для установки этих устройств и управления ими. Необходимость применения разных управляющих приложений для разных устройств внешней памяти является очевидным недостатком с точки зрения системного администрирования. К примеру, приходится осваивать множество интерфейсов, а для управления устройствами внешней памяти сторонних производителей нельзя пользоваться стандартными служебными Windows-программами.

В Windows имеется служба виртуальных дисков (Virtual Disk Service, VDS). Эта служба (%SystemRoot%\System32\Vds.exe) предоставляет системным администраторам унифицированный высокоуровневый интерфейс внешней памяти. Благодаря этой службе устройствами внешней памяти от разных производителей можно управлять через одни и те же пользовательские интерфейсы. Схема VDS показана на рис. 9.18. Служба экспортирует прикладной программный интерфейс на базе COM, позволяющий приложениям создавать и форматировать диски, а также управлять аппаратными RAID-адаптерами. К примеру, служба может воспользоваться VDS API, чтобы запросить список физических дисков, отображенных на номер логического блока (Logical Unit Number, LUN) RAID-массива. Средства управления дисками Windows, включая оснастку Disk Management консоли MMC и утилиты командной строки DiskPart и DiskRAID, также пользуются VDS API.



Рис. 9.18. Архитектура VDS

VDS предоставляет два интерфейса, один для программных, другой — для аппаратных провайдеров:

- ❑ Программные провайдеры (software providers) реализуют интерфейсы к таким высокоуровневым абстракциям устройств внешней памяти, как диски, дисковые разделы и тома. К операциям, поддерживаемым этими интерфейсами, относятся создание, расширение и удаление томов, добавление и отключение зеркалирования, форматирование и назначение букв дисков. Поиск программных провайдеров VDS осуществляется в разделе `HKEY\SYSTEM\CurrentControlSet\Services\Vds\SoftwareProviders` реестра, имена подразделов в котором представляют собой глобальные уникальные идентификаторы (GUID). В каждом подразделе имеется значение с именем `Clslid`, указывающее идентификатор COM-класса. Эти значения перечислены в разделе `HKEY_CLASSES_ROOT\CLSID\<Clslid>`. Также в Windows входят динамический VDS-провайдер (VDS Dynamic Provider), служащий интерфейсом для динамических дисков (%SystemRoot%\System32\Vdsdyn.dll), и базовый VDS-провайдер (VDS Basic Provider), используемый в качестве интерфейса для базовых дисков (%SystemRoot%\System32\Vdsbas.dll).
- ❑ Производители аппаратного обеспечения реализуют аппаратные провайдеры (hardware providers) в виде динамически подключаемых библиотек (DLL), которые регистрируются в разделе `HKEY\SYSTEM\CurrentControlSet\Services\Vds\HardwareProviders` реестра и преобразуют аппаратно-независимые VDS-команды в команды, понятные конкретному оборудованию. Аппаратный провайдер позволяет также управлять подсистемой внешней памяти, например RAID-массивом или платой адаптера, и поддерживает такие операции, как создание, расширение, удаление, маскирование и отмена маскирования LUN.

Если приложение инициирует соединение с VDS API, а служба виртуальных дисков еще не запущена, процесс хост-службы RPC Svchost инициирует процесс загрузки VDS (%SystemRoot%\System32\Vdsldr.exe). Последний запускает VDS и прекращает работу. После закрытия последнего соединения с VDS API VDS-процесс завершается.

Поддержка виртуального жесткого диска

В Windows реализована встроенная поддержка файлов формата VHD (Virtual Hard Disk) — формата Microsoft для виртуального жесткого диска. Средства управления диском позволяют создавать, удалять и объединять виртуальные жесткие диски друг с другом, а также подсоединять их к системе как физические диски. Также Windows поддерживает загрузку установленных копий Windows, хранящихся на томах NTFS внутри VHD.

Существуют три типа виртуальных дисков, обеспечивающих функциональность VHD в Windows:

- ❑ **Динамический** (dynamic) диск. Виртуальный жесткий диск не обязан содержать все возможные блоки и способен увеличиваться до некоторого предельного значения. Другими словами, занимаемое VHD пространство равно размеру записанных на диск данных (плюс небольшой фрагмент для служебной информации).

- **Фиксированный** (fixed) диск. Этот диск не может менять свой размер и содержит все заявленные при его создании блоки (полностью подготовленные).
- **Разностный** (differencing) диск. Аналогичен динамическому виртуальному жесткому диску, но содержит только секторы, измененные относительно родительского образа (который предназначен только для чтения). Родительский виртуальный жесткий диск может относиться к любому из трех типов. Разностные виртуальные жесткие диски обычно применяются для получения снимка состояния родительского диска. Данное состояние впоследствии можно восстановить, просто удалив разностный виртуальный жесткий диск. Это часто требуется при создании контрольных точек виртуальных машин (Virtual Machines, VM), чтобы дать пользователю возможность вернуть виртуальную машину в предшествующее состояние. Следует заметить, что разностный виртуальный жесткий диск должен находиться в той же папке, что и родительский образ.

Когда виртуальный диск появляется в системе, стандартный диспетчер разделов и диспетчер томов распознают и монтируют том, предоставляя прикладным программным интерфейсам файловых систем и служебным Windows-программам доступ к хранящимся на VHD файловым системам.

Один виртуальный диск может содержаться внутри другого, поэтому в Windows число уровней вложенности ограничено двумя. При этом максимальное количество уровней вложенности задается значением HKLM\System\CurrentControlSet\Services\FsDepends\Parameters\VirtualDiskMaxTreeDepth реестра. Запретить монтирование виртуальных дисков можно, присвоив параметру HKLM\System\CurrentControlSet\Services\FsDepends\Parameters\VirtualDiskNoLocalMount реестра значение 1.

Допустима загрузка Windows с виртуального диска. Загрузочный виртуальный жесткий диск можно создать «с нуля» в процессе установки (при загрузке установочного диска Windows) или в уже работающей системе при помощи таких инструментов, как ImageX или Disk2VHD производства Sysinternals. Подобная «система на виртуальном диске» может быть запущена в средствах виртуализации Virtual PC и Hyper-V (на Windows Server), а редакции Windows Ultimate и Enterprise допускают непосредственную загрузку с виртуального диска.

В Windows поддерживаются не только VHD, но и все встроенные программы управления дисками. Создание, монтирование и размонтирование VHD можно осуществить в процессе работы Windows при помощи оснастки Disk Management консоли MMC (%SystemRoot%\System32\Diskmgmt.msc) или утилиты командной строки DiskPart (%SystemRoot%\System32\Diskpart.exe). Для реализации этих инструментов применялись прикладные программные интерфейсы службы виртуальных дисков (VDS), к которым могут прибегать и программы управления VHD сторонних производителей.

Присоединение виртуальных жестких дисков

Драйвер шины перечислителя виртуальных дисков Vdrvroot (%SystemRoot%\System32\Drivers\Vdrvroot.sys) создает объект физического устройства (Physical Device Object, PDO) для каждой подлежащей монтированию вложенной файловой системы. РПР-диспетчер загружает драйвер мини-порта Storport, который называется Vhdmp (%SystemRoot%\System32\Drivers\Vhdmp.sys), как функцию драйвера на этом объекте

физического устройства, транслируя для остальной системы физический диск. Затем диспетчер ввода-вывода помещает остальной стек драйверов внешней памяти (драйвер дискового класса, диспетчер разделов, диспетчер томов и драйвер файловой системы) наверх стека устройств (DevStack), содержащего Vhdmp. Получая запросы на чтение или запись в сектор, драйвер Vhdmp преобразует их в смещение в VHD-файле и затем передает в стек драйверов внешней памяти, в котором находится VHD-файл.

Вложенные файловые системы

Для поддержки вложенных файловых систем создается дерево зависимостей, отслеживающее, какие файловые системы зависят от других файловых систем. Это требуется для корректного функционирования ряда системных операций, например размонтирования тома (сначала требуется размонтировать зависимые файловые системы), завершения работы системы (оно аналогично размонтированию тома) и моментальных снимков томов (зависимые тома должны очищаться раньше родительских). Отслеживанием зависимостей занимается драйвер мини-фильтра файловой системы (%SystemRoot%\System32\Drivers\Fsdepends.sys), расположенный поверх драйвера файловой системы. При этом в расчет берется не соотношение предок–потомок, а соотношение, в соответствии с которым удаляются устройства, так как последние более динамичны и могут запрашиваться в процессе выполнения. (Это крайне важно, ведь вложенные драйверы могут устанавливать дополнительные зависимости, после того как смонтирован виртуальный жесткий диск.)

С точки зрения большинства Windows-компонентов смонтированный VHD-том идентичен тому, расположенному на физическом диске, но на нем не могут располагаться ни файлы подкачки, ни файл спящего режима, ни файл аварийного дампа. Кроме того, размер VHD не может превышать 2 Тбайт.

Шифрование диска BitLocker

Операционная система реализует свои политики безопасности только в активном состоянии, поэтому для защиты данных в ситуациях с нарушениями физической безопасности системы или доступа извне требуется принять дополнительные меры. Для предотвращения несанкционированного доступа, особенно на портативных компьютерах, которые чаще всего становятся объектами кражи, используются две аппаратные технологии: BIOS-пароли и шифрование.

Хотя Windows поддерживает шифрующую файловую систему (Encrypting File System, EFS), вы не сможете ею воспользоваться для защиты уязвимых областей, таких как куст реестра. К примеру, если групповые политики дают возможность авторизоваться на портативном компьютере даже без подсоединения к домену, то средства проверки учетных данных в домене кэшируются в реестр. А значит, воспользовавшись специальными инструментами, злоумышленник может получить хэш пароля вашей доменной учетной записи и попытаться узнать пароль при помощи взломщика паролей. Это даст ему доступ к вашей учетной записи и EFS-файлам (предполагается, что вы не храните EFS-ключ на смарт-карте). Для простоты шифрования всего загрузочного

тoma, включая все системные файлы и данные, в Windows существует специальный инструмент, который называется **BitLocker Drive Encryption** (Шифрование диска BitLocker). Он работает в двух режимах:

- ❑ **Standard** – защита фиксированных дисков;
- ❑ **BitLocker To Go** – защита съемных носителей (в том числе и флэш-дисков с USB-интерфейсом), отформатированных для файловой системы FAT.

В стандартном режиме BitLocker предотвращает неавторизованный доступ к данным потерянного или украденного компьютера, комбинируя две основные процедуры защиты данных:

- ❑ шифрование всего тома с операционной системой Windows на жестком диске;
- ❑ проверка целостности компонентов загрузки на ранних стадиях и конфигурационных загрузочных данных.

Обеспечивающая максимальную безопасность реализация BitLocker использует расширенные возможности спецификации Trusted Platform Module (TPM) версии 1.2. TPM представляет собой криптографический сопроцессор, устанавливаемый производителями на многие новые компьютеры. Он реализует множество функций, в том числе шифрование с открытым ключом. Сведения о том, как функционирует TPM, можно получить на сайте <http://www.TrustedComputingGroup.org/>. С помощью BitLocker TPM защищает данные пользователя и обеспечивает недоступность компьютера с операционной системой Windows после того, как пользователь отключает систему. Если на компьютере отсутствует TPM версии 1.2, BitLocker все равно может зашифровать том с операционной системой Windows. Но в этом случае для включения компьютера или пробуждения его из спящего режима потребуется воспользоваться загрузочным флэш-диском с USB-интерфейсом, кроме того, на таком компьютере отсутствует не зависящая от операционной системы защита, которую предоставляет TPM.

Архитектура BitLocker обеспечивает функциональность и механизмы управления как в режиме ядра, так и в режиме пользователя. Вот основные компоненты BitLocker на верхнем уровне:

- ❑ Драйвер Trusted Platform Module (%SystemRoot%\System32\Drivers\Tpm.sys), который обращается к микросхеме TPM в режиме ядра.
- ❑ Основные службы TPM, к которым относятся пользовательские службы, предоставляющие доступ к TPM в режиме пользователя (%SystemRoot%\System32\Tbssvc.dll), поставщик WMI и оснастка MMC для конфигурации (%SystemRoot%\System32\Tpm.msc).
- ❑ Связанный код BitLocker в диспетчере загрузки (\Bootmgr, на системном томе), который аутентифицирует доступ к жесткому диску, а также позволяет восстанавливать и разблокировать загрузчик.
- ❑ Фильтрующий драйвер BitLocker (%SystemRoot%\System32\Drivers\Fvevol.sys), позволяющий «на лету» шифровать и расшифровывать тома в режиме ядра.
- ❑ Поставщик BitLocker WMI и сценарий управления, позволяющий выбирать конфигурации интерфейса BitLocker и писать для него сценарии.

Далее мы более подробно рассмотрим все эти компоненты и предоставляемую ими функциональность. Схематично архитектура BitLocker представлена на рис. 9.19.

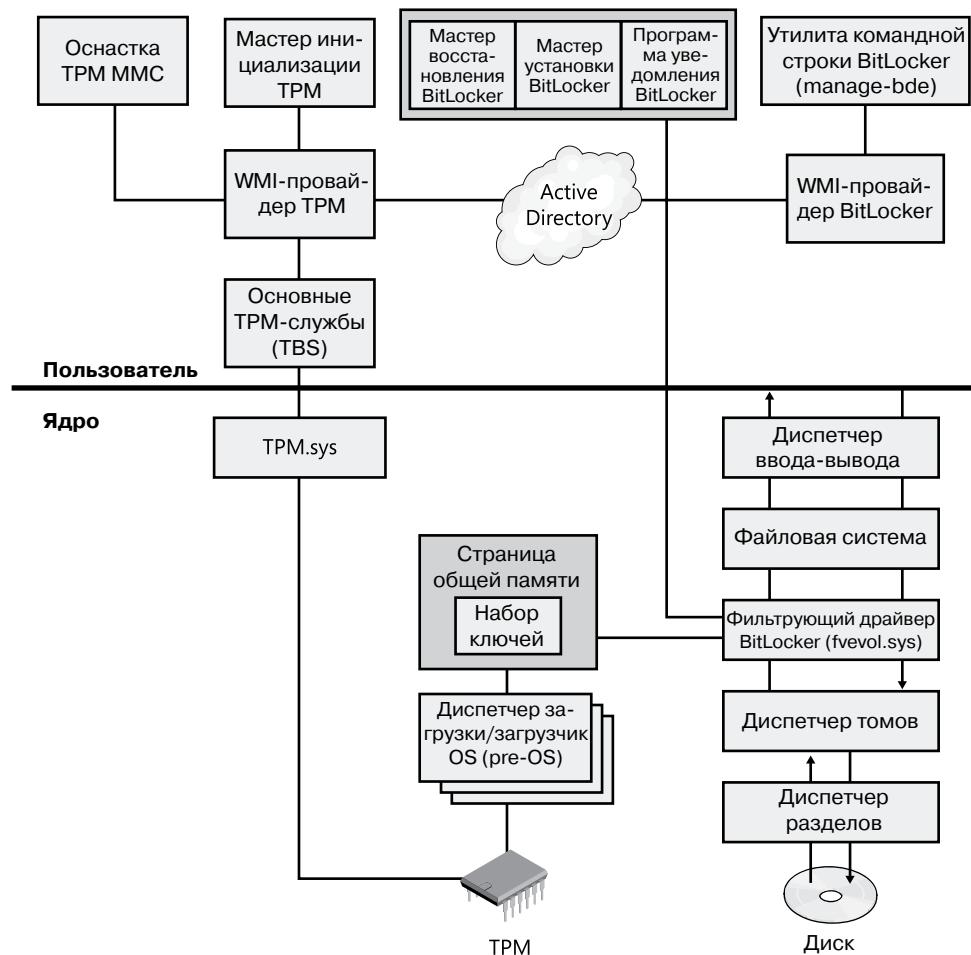


Рис. 9.19. Архитектура BitLocker

Ключи шифрования

BitLocker шифрует содержимое тома с помощью ключа шифрования всего тома (Full-Volume Encryption Key, FVEK) и алгоритма AES128-CBC (по умолчанию) или AES256-CBC с расширением Microsoft, которое называется диффузором. В свою очередь, ключ FVEK шифруется с помощью главного ключа тома (Volume Master Key, VMK) и хранится на томе в области, специально отведенной для метаданных. Защита главного ключа тома является косвенным способом защиты данных на томе: добавление этого ключа позволяет системе воссоздать утерянные или скомпрометированные

ключи, расположенные выше в цепочке доверия, что экономит время и затраты на дешифрирование и повторное шифрование целого тома.

В процессе выбора конфигурации BitLocker вам доступны различные способы защиты VMK в зависимости от аппаратных возможностей системы. При наличии TPM можно сохранить VMK только в TPM, заставить систему зашифровать VMK, сохранив один ключ в TPM, а другой — на флэш-устройстве с USB-интерфейсом, зашифровать VMK с сохранением ключа в TPM и кодом PIN, вводимым при загрузке системы. Также можно воспользоваться комбинацией кода PIN и флэш-устройства с USB-интерфейсом. В системах, не совместимых с технологией TPM, BitLocker предлагает зашифровать VMK с сохранением ключа на внешнем USB-устройстве.

В любом случае вам потребуется нешифрованный системный NTFS-том на 100 Мбайт, на котором будут храниться диспетчер загрузки и BCD, так как MBR и код загрузочного сектора являются устаревшими и работают только в 16-разрядном реальном режиме (как описано в главе 13). Они не в состоянии выполнять дешифрирование «на лету» на том томе, на котором они запущены. Это означает, что данные компоненты должны оставаться на незашифрованном томе, чтобы система BIOS получила к ним доступ, а они запустились и обнаружили Bootmgr.

Как уже упоминалось, системный том создается автоматически в процессе установки Windows вне зависимости от того, пользуетесь вы BitLocker или нет. В результате этот том оказывается в начале диска (в первом разделе), обеспечивая непрерывность его остальной части.

Различные способы генерации VMK обобщены на рис. 9.20 и в табл. 9.1.

Таблица 9.1. Источники VMK

Источник	Определяет	Безопасность	Зависимость от пользователя
Только TPM	Что это такое	Защищает от программных атак, но уязвим к аппаратным атакам	Отсутствует
TPM + PIN	Что это такое + Что вы знаете	Добавляет защиту от аппаратных атак	Нужно вводить PIN-код при каждой загрузке
TPM + USB-ключ	Что это такое + Что у вас есть	Полная защита от аппаратных атак, но уязвим при краже USB-ключа	Нужно вставлять USB-ключ при каждой загрузке
TPM + USB-ключ + PIN	Что это такое + Что у вас есть + Что вы знаете	Максимальный уровень защиты	При каждой загрузке нужно вводить PIN-код и вставлять USB-ключ
Только USB-ключ	Что у вас есть	Минимальный уровень защиты для систем, не имеющих TPM, но есть риск утраты ключа	Нужно вставлять USB-ключ при каждой загрузке

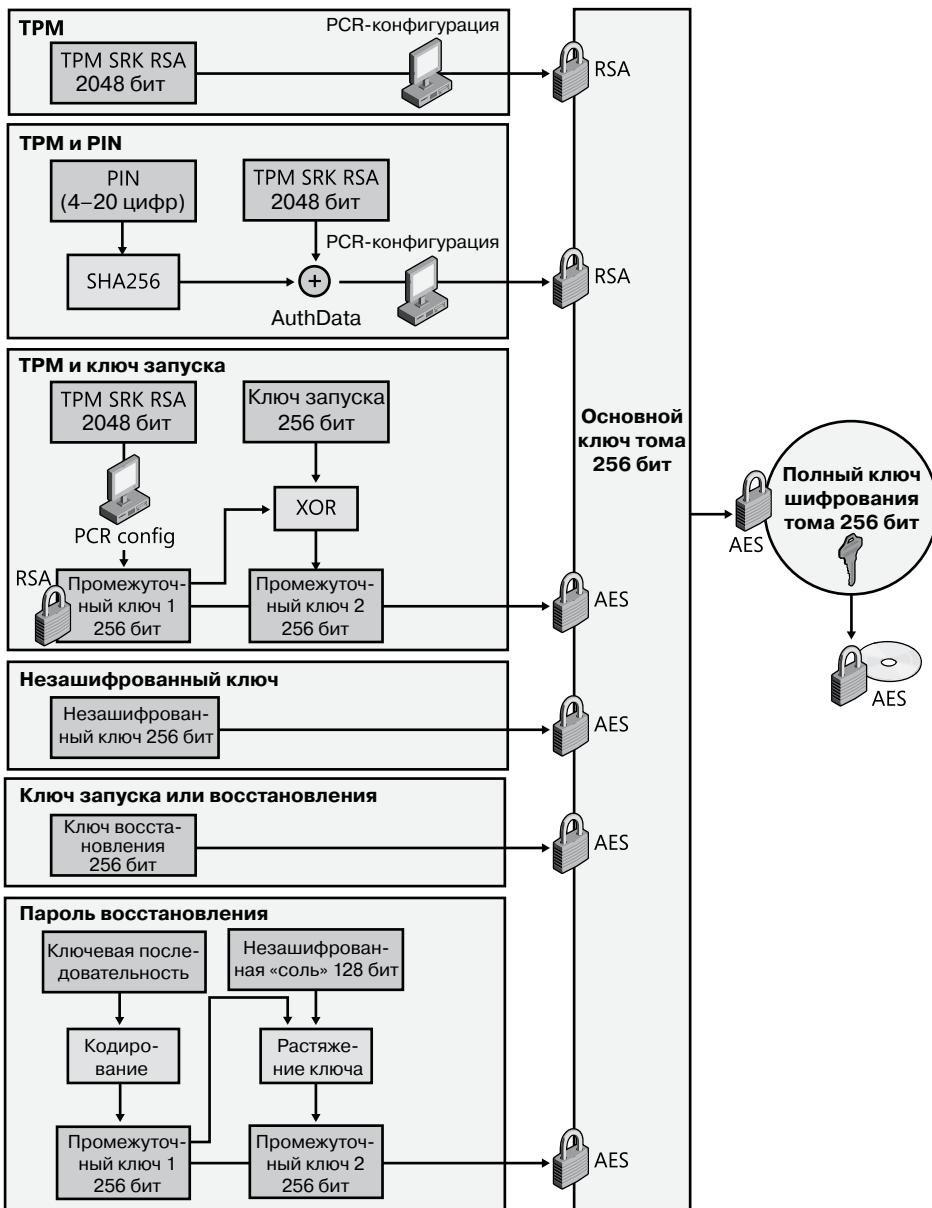


Рис. 9.20. Генерация ключа BitLocker

Также BitLocker предоставляет простую схему проверки подлинности на основе шифрования для обеспечения целостности содержимого диска. В настоящее время алгоритм шифрования AES считается недоступным для взлома методом прямого подбора и является одним из самых распространенных, но он не дает гарантий невозможности путем редактирования преобразовать зашифрованные данные в обычный

текст, которым сможет воспользоваться злоумышленник. К примеру, манипулируя зашифрованными данными, хакер может изменить поведение функции входа таким образом, что станет возможным любой вариант входа.

Для защиты системы против атак данного типа в BitLocker включен алгоритм *диффузора* (diffuser) под названием Elephant. Он гарантирует, что изменение даже одного бита в исходном тексте приведет к полному изменению всего сектора зашифрованных данных. В результате модифицированный исполняемый код просто не сработает. Кроме того, в комбинации с целостностью исходного кода (эта тема рассматривалась в главе 3 части I) диффузор не даст основным системным файлам осуществить проверку сигнатур, что предотвратит загрузку системы.

Доверенный платформенный модуль

Доверенный платформенный модуль (Trusted Platform Module, TPM) представляет собой устойчивый к взлому процессор, вмонтированный в материнскую плату и предоставляющий различные криптографические службы, такие как службы генерации ключа и случайного числа, а также служба надежного хранения. Однако поддержка TPM в Windows выходит за рамки поддержки BitLocker. Через базовые TPM-службы (TPM Base Services, TBS) другие приложения системы также могут воспользоваться преимуществами аппаратно-совместимых TPM-микросхем и применять WMI для администрирования и доступа сценариев к TPM. К примеру, в Windows TPM используется в качестве дополнения к генератору случайных чисел, что позволяет повысить общую защищенность всех тех приложений системы, которые зависят от безопасности системы в целом или алгоритмов хэширования (включая механизмы авторизации).

Даже если на вашем компьютере присутствует доверенный платформенный модуль, Windows не всегда может его использовать. Для этого необходимы два условия:

- на компьютере должен быть установлен доверенный платформенный модуль версии 1.2 или выше;
- компьютер должен иметь совместимую с TCG систему BIOS, которая установит цепочку доверия для среды предварительной загрузки, и поддерживать статический корень измерения доверия (Static Root of Trust Measurement, SRTM).

Проще всего определить наличие на компьютере совместимого модуля TPM, попытавшись запустить TPM-оснастку из консоли MMC (%SystemRoot%\System32\Tpm.msc). При обнаружении совместимого модуля TPM появляется окно, подобное показанному на рис. 9.21. В противном случае появляется сообщение об ошибке.

Как было показано ранее, BitLocker можно настроить на использование TPM для выполнения проверки целостности системы и ее критически важных компонентов на ранних стадиях загрузки. На верхнем уровне TPM собирает и сохраняет результаты измерений множества компонентов ранней загрузки и конфигурационных данных загрузки, создавая на их основе идентификатор системы (подобный отпечатку пальца) для данного компьютера. Каждая часть этого отпечатка сохраняется в виде хэша в 160-разрядном *регистре конфигурации платформы* (Platform Configuration Register, PCR). BitLocker использует данные из этого регистра для генерации главного ключа

тoma (VMK), привязанного к конфигурации компьютера. С его помощью BitLocker защищает другие ключи, в том числе ключи шифрования всего тома (FVEK), применяемые для шифрования томов.

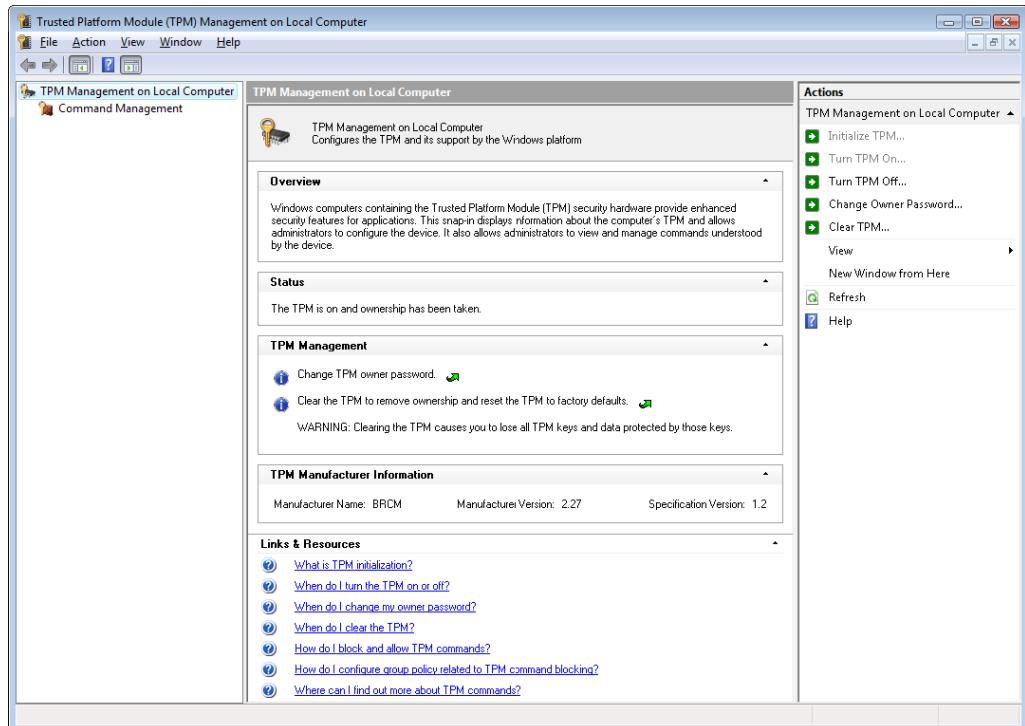


Рис. 9.21. TPM-оснастка консоли MMC после инициализации TPM

При обнаружении несанкционированных изменений на ранних стадиях загрузки, например изменения BIOS или MBR, изменения файла операционной системы или установки жесткого диска на другой компьютер, TPM запрещает системе BitLocker расшифровывать VMK, и Windows входит в режим восстановления ключа (о нем мы поговорим позже). Если PCR-значения совпадают с использовавшимися при шифровании ключа, система считается неповрежденной и ключ расшифровывается, после чего BitLocker получает возможность расшифровать ключи, применяющиеся для шифрования томов. После этого запускается Windows и ответственность за защиту пользователя и системы берет на себя операционная система.

Поддерживаемый TPM профиль проверки платформы состоит из как минимум 16, а как максимум 24 регистров конфигурации платформы, которые содержат дополнительную информацию и сбрасываются только после сброса TPM (что влечет за собой перезагрузку компьютера). Каждый регистр PCR связан с компонентами, запускаемыми при запуске операционной системы. Они перечислены в табл. 9.2.

Таблица 9.2. Регистры конфигурации платформы

Регистр	Значение
0	Ядро корня измерения доверия (Core Root of Trust of Measurement, CRTM), BIOS и платформенные расширения
1	Конфигурация и данные, относящиеся к платформе и материнской плате (данные BIOS и микрокод CPU)
2	BIOS периферийных устройств (Option ROM code)
3	Конфигурация и данные Option ROM
4	Код главной загрузочной записи (MBR)
5	Таблица MBR-разделов
6	События, связанные с системой электропитания
7	Зависит от производителя компьютера
8	Первый загрузочный сектор NTFS (загрузочный сектор тома)
9	Остальные загрузочные секторы NTFS (загрузочная запись тома)
10	Диспетчер загрузки
11	Управление доступом к BitLocker
12	Предназначен для использования статической операционной системой
13	Предназначен для использования статической операционной системой
14	Предназначен для использования статической операционной системой
15	Предназначен для использования статической операционной системой
16	Применяется для отладки
17	Динамическое CRTM
18	Зависит от платформы
19	Используется доверенной операционной системой
20	Используется доверенной операционной системой
21	Используется доверенной операционной системой
22	Используется доверенной операционной системой
23	Поддержка приложения

По умолчанию для генерации ключа VMK, привязанного к конфигурации, BitLocker пользуется регистрами 0, 2, 4, 5, 8, 9, 10 и 11. Этот набор регистров известен как профиль проверки платформы (platform validation profile), настраивается через групповые политики (Computer Configuration\Administrative Templates\Windows Components\BitLocker Drive Encryption\Operating System Drives\Configure TPM platform validation profile) и зависит от требований к безопасности конкретной фирмы, как показано в табл. 9.2. Для включения BitLocker-защиты следует выбрать PCR 11.

ПРИМЕЧАНИЕ

При внесении изменений в область, PCR которой вы указали в профиле проверки платформы, система не сможет загрузиться без ключа или пароля восстановления. К примеру, если требуется обновить BIOS, сначала приостановите работу BitLocker, используя апллет панели управления BitLocker Drive Encryption (Шифрование диска BitLocker).

Процесс загрузки BitLocker

Хранящиеся в регистрах конфигурации платформы TPM данные генерируются самим микроконтроллером TPM, а также TPM BIOS и операционной системой Windows. Загрузка операционной системы сопровождается самотестированием TPM, после которого CRTM в BIOS проверяет значение хэша и загрузочный код PCR и записывает полученный хэш в первый TPM-регистр. Затем CRTM в BIOS проверяет целостность BIOS и также сохраняет полученные результаты в виде хэша в первый регистр PCR. В свою очередь, BIOS проверяет следующий компонент в цепочке загрузки, MBR загрузочного диска, и процесс продолжается, пока не дойдет до загрузчика операционной системы. Каждый последующий запущенный фрагмент кода отвечает за проверку загружаемого им кода и за сохранение результатов этой проверки в подходящий TPM-регистр PCR.

В конце, когда пользователь выбирает, какую операционную систему он хочет загрузить, диспетчер загрузки (Bootmgr) читает с тома зашифрованный ключ VMK и просит TPM его дешифрировать. Как уже упоминалось, только при совпадении всех результатов проверки с данными, зафиксированными при генерации VMK, в том числе с необязательным PIN-кодом (паролем), TPM приступит к расшифровке VMK. Этот процесс не только гарантирует приложениям или операционным системам, которые могут читать с диска, идентичность компьютера и системных файлов, но и проверку уникальности установленной копии операционной системы. Скажем, даже еще одна идентичная операционная система Windows, установленная на ту же машину, не получит доступа к диску, так как Bootmgr активно участвует в защите VMK от попадания в чужую операционную систему (генерируя код аутентичности сообщений из нескольких конфигурационных параметров системы).

Данную схему можно представить в виде цепочки проверок, в которой каждый загрузочный компонент описывает для TPM следующий компонент. В результате TPM превращается в подобие сейфа с 12 наборами комбинаций, каждая из которых содержит 2160 чисел. TPM откроет тайну, только если все PCR-значения совпадут с исходными, зафиксированными в момент включения BitLocker. Поэтому BitLocker в состоянии защитить зашифрованные данные даже в ситуациях, когда диск извлечен из компьютера и помещен в другую систему, когда для загрузки используется другая операционная система или когда скомпрометированы незашифрованные файлы на загрузочном томе. Рисунок 9.22 демонстрирует различные этапы процесса, предшествующего моменту, когда Winload начинает загрузку операционной системы.

Администратору может потребоваться на время приостановить работу BitLocker для внесения изменений в компоненты, указанные в профиле проверки платформы (например, обновить BIOS, отредактировать таблицу разделов диска, установить на тот же самый диск еще одну операционную систему и т. п.). Апплет панели управления BitLocker Drive Encryption (Шифрование диска BitLocker) предоставляет простой механизм для приостановки BitLocker — достаточно щелчком выбрать для нужного тома режим Suspend Protection (Приостановить защиту). После этой операции содержимое тома останется зашифрованным, но основной ключ тома будет зашифрован симметричным открытым ключом, записанным в BitLocker-метаданные тома. При монтировании тома BitLocker автоматически ищет открытый ключ и получает воз-

можность расшифровать содержимое тома. После включения BitLocker открытый ключ из метаданных удаляется.

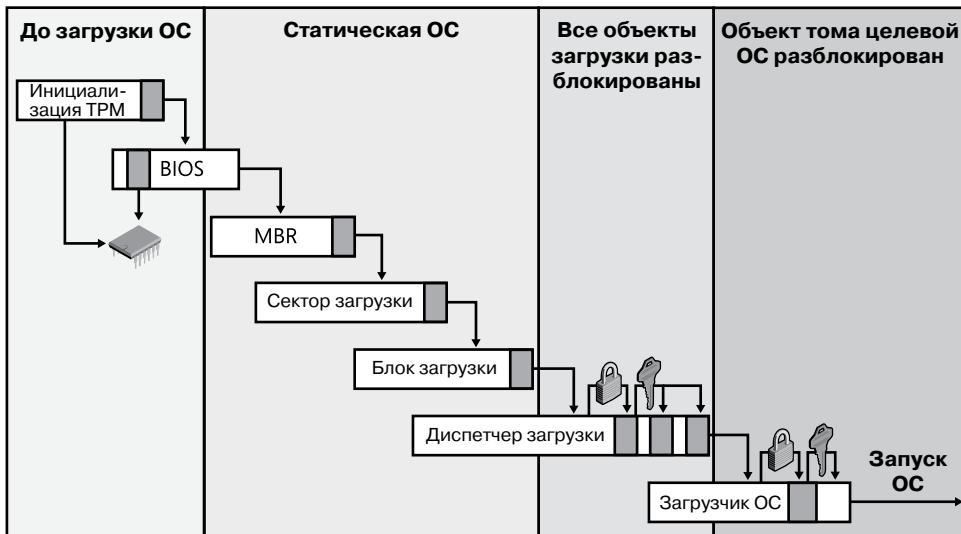


Рис. 9.22. Процесс предварительной загрузки BitLocker

ПРИМЕЧАНИЕ

Открытие основного ключа тома даже на короткий период времени ставит под угрозу безопасность компьютера, так как через открытый ключ злоумышленник может получить доступ к главному ключу тома и к FVEK. Поэтому не приостанавливайте работу BitLocker на более долгое время, чем это нужно.

Восстановление с помощью BitLocker

Для восстановления BitLocker использует ключ (хранящийся на USB-устройстве) или пароль (числовой) восстановления, как было показано на рис. 9.20. Ключ и пароль восстановления BitLocker создает в процессе инициализации. Копия VMK шифруется при помощи 256-разрядного ключа AES-CCM, для вычисления которого требуется пароль восстановления и «соль», хранящаяся в блоке метаданных. Пароль представляет собой состоящее из 48 цифр число, восемь групп по 6 цифр, с тремя свойствами для вычисления контрольной суммы:

- ❑ Каждая группа из 6 цифр должна быть кратной 11. Эта проверка применяется для идентификации групп, при наборе которых пользователь допустил опечатку.
- ❑ Каждая группа из 6 цифр должна быть меньше чем $2^{16} \times 11$. Каждая группа содержит 16 бит данных о ключе. Следовательно, в восьми группах будет содержаться 128 бит ключа.
- ❑ Шестая цифра в каждой группе представляет собой контрольную сумму.

Вставка ключа восстановления или ввод пароля восстановления позволяет авторизованному пользователю снова получить доступ к зашифрованному тому при попытке нарушения безопасности или системном сбое. На рис. 9.23 показано приглашение пользователю на ввод пароля восстановления.



Рис. 9.23. Экран восстановления BitLocker

Ключ или пароль восстановления используется также при изменении частей системы, приводящих к другим характеристикам. Распространенным примером является редактирование BCD, скажем, добавление параметра `debug`. После перезагрузки Bootmgr обнаружит изменение и попросит пользователя подтвердить его, введя ключ восстановления. Именно поэтому данный ключ крайне важно иметь под рукой, ведь он нужен не только для восстановления, но и для подтверждения системных изменений. Еще одной областью применения ключа восстановления являются *чужие тома* (*foreign volumes*). Этим термином называются тома с операционной системой, которые после включения BitLocker были перенесены на другой компьютер с Windows. Администратор может снять с них блокировку, введя пароль восстановления.

Драйвер шифрования всего тома

В отличие от шифрующей файловой системы (EFS), реализованной NTFS-драйвером и функционирующей на уровне файлов, BitLocker выполняет шифрование на уровне томов, используя драйвер шифрования всего тома (%SystemRoot%\System32\Drivers\Fvevol.sys), как показано на рис. 9.24.

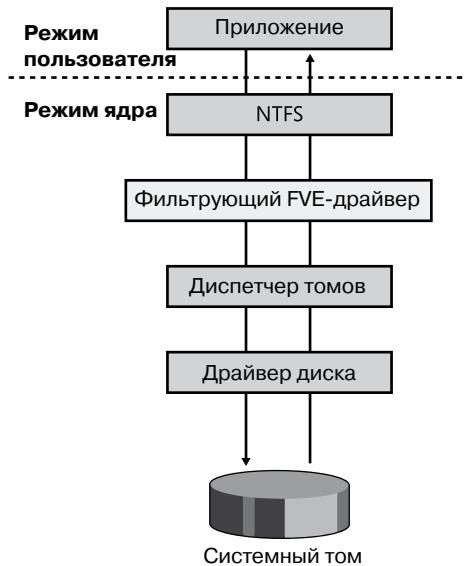


Рис. 9.24. Реализация фильтрующего драйвера в BitLocker

Драйвер шифрования всего тома (Full-Volume Encryption, FVE) является фильтрующим, то есть он автоматически видит все запросы к тому на ввод и вывод, зашифровывая блоки по мере их записи и расшифровывая в процессе чтения при помощи ключа FVEK, назначенного тому в момент включения на нем BitLocker. Так как шифрование и дешифрирование происходят в подсистеме ввода-вывода, находящейся ниже NTFS, для NTFS том выглядит незашифрованным. Более того, файловая система даже не знает о том, что произошло включение BitLocker. Но попытавшись прочитать что-то с тома не из Windows, вы получите случайный набор данных.

Кроме того, BitLocker предпринимает дополнительные меры, чтобы затруднить атаки на основе открытых текстов, когда злоумышленник знает содержимое сектора и с помощью этой информации пытается получить ключ, который использовался при шифровании.

Скомунировав FVEK с номером сектора, чтобы сгенерировать для этого сектора ключ шифрования, и передав зашифрованные данные через диффузор Elephant, BitLocker гарантирует, что ключи к разным секторам будут слегка отличаться друг от друга. Это обеспечит разницу в зашифрованных данных в каждом секторе даже при идентичном исходном содержимом.

BitLocker шифрует все секторы (в том числе неразмеченные) на томе, кроме первого сектора и трех блоков метаданных, содержащих зашифрованный ключ VMK и прочие используемые самим средством BitLocker данные. Эти метаданные отображаются в папке системной информации тома.

Управление системой BitLocker

Средство шифрования BitLocker предоставляет набор административных интерфейсов, каждый из которых соответствует определенной роли или задаче. BitLocker предоставляет интерфейс WMI для программного доступа к собственной функциональности (и работает с базовыми TPM-службами), набор групповых политик, позволяющий администратору задать поведение в сети или на нескольких машинах, интеграцию с Active Directory и утилиту для управления из командной строки (%SystemRoot%\System32\Manage-bde.exe).

Разработчики и системные администраторы, знакомые с написанием сценариев, могут получить доступ к интерфейсам Win32_Tpm и Win32_EncryptableVolume для защиты ключей, задания методов идентификации, указания, какие из регистров PCR будут использоваться как часть профиля проверки платформы BitLocker, и вручную инициировать шифрование или дешифрирование тома. Расположенная в папке %SystemRoot%\System32 программа Manage-bde.exe при помощи этих интерфейсов реализует управление службой BitLocker из командной строки.

В присоединенных к домену системах ключ для каждой машины автоматически подвергается резервному копированию как часть системы депонирования ключей, что позволяет администраторам легко восстанавливать доступ к входящим в корпоративную сеть машинам. Кроме того, можно выбирать конфигурацию связанных с BitLocker групповых политик. Доступ к ним осуществляется через оснастку Local Group Policy Editor (Редактор локальной групповой политики), где нужно последовательно выбрать разделы Computer Configuration (Конфигурация компьютера), Administrative Templates (Административные шаблоны), Windows Components (Компоненты Windows), BitLocker Drive Encryption (Шифрование диска BitLocker). К примеру, на рис. 9.25 показано, как включить режим резервного копирования ключа Active Directory.

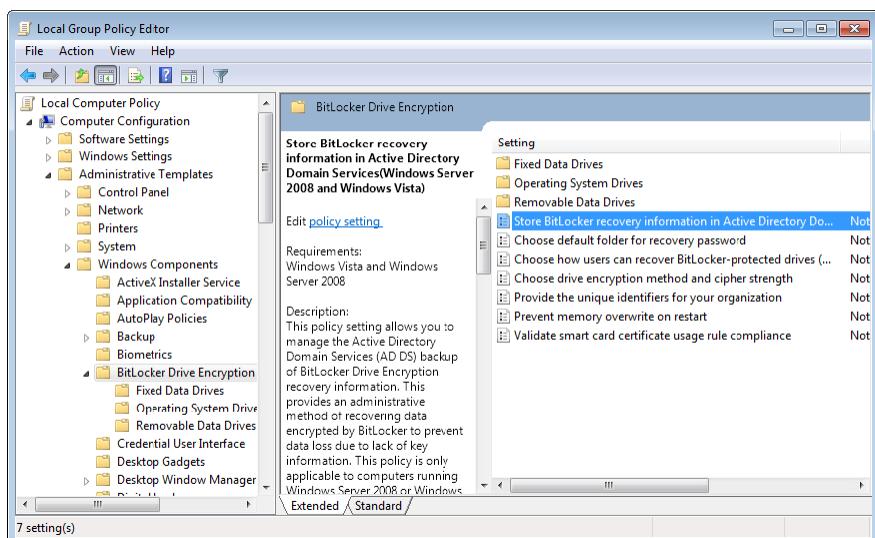


Рис. 9.25. Настройки групповых политик BitLocker

При наличии в системе микросхемы TPM для доступа к дополнительным параметрам, таким как TPM Key Backup (Резервное копирование TPM-ключа), нужно выбрать запись Trusted Platform Module Services (Службы доверенного платформенного модуля) в разделе Windows Components (Компоненты Windows).

Чтобы гарантировать легкий доступ к корпоративным данным, к BitLocker был добавлен агент восстановления данных (Data Recovery Agent, DRA). Он настраивается через групповые политики и позволяет указать сертификат как предохранитель ключа. В результате любой владелец сертификата (или смарт-карты, на которой он находится) получает доступ к тому, защищенному при помощи BitLocker. О конфигурировании DRA рассказывается на странице [http://technet.microsoft.com/en-us/library/dd875560\(WS.10\).aspx](http://technet.microsoft.com/en-us/library/dd875560(WS.10).aspx).

Технология BitLocker To Go

Благодаря своему небольшому размеру, низкой стоимости и большой емкости флэш-память с USB-интерфейсом является популярным средством переноса данных. Но именно эти качества превращают такой накопитель в угрозу безопасности. На устройстве размером со спичку, которое легко может быть потеряно или украдено, хранятся гигабайты конфиденциальной информации. Стандартная система BitLocker работает только с NTFS-томами, в то время как для разметки флэш-памяти с USB-интерфейсом по умолчанию применяется файловая система FAT. Поэтому для таких накопителей была разработана технология BitLocker To Go (BTG). Зашифровать диск с ее помощью можно только в таких редакциях Windows, как Enterprise, Ultimate или Server. Чтение возможно в любой операционной системе семейства Windows, даже в таких старых версиях, как Windows XP и Windows Vista, запись же доступна только в Windows 7 или Windows Server 2008/R2. Чтобы гарантировать применение BTG, можно через групповые политики запретить запись на незащищенные сменные накопители.

Как и стандартная система BitLocker, BTG для шифрования тома использует алгоритм AES, ключ дешифрования шифруется с набором предохранителей, а ключ восстановления можно сохранить в файле или передать через Active Directory. В BTG в отличие от BitLocker не применяется ни TPM, ни шифрование с открытым ключом. В качестве одного из предохранителей ключ может выступать задаваемый пользователем пароль или смарт-карта.

Включить BTG можно в Проводнике, щелкнув на флэш-накопителе правой кнопкой мыши и выбрав в появившемся меню команду Turn On BitLocker (Включить BitLocker), или через апплет Панели управления. После включения BTG создаст *том обнаружения* (discovery volume) с файловой системой FAT32, содержащий файлы, как показано на рис. 9.26. На этот том помещается изолированное приложение BitLockerToGo, его MUI-файлы (строки пользовательского интерфейса на разных языках) и метаданные для операционной системы.

Зашифрованный том реализуется в виде одного или нескольких *криптоконтейнеров* (cover files) с именами от COV 0000. ER до COV 9999. ER. Максимальный размер каждого из них может доходить до 4 Гбайт, как показано на рис. 9.26 и 9.27. Оставшееся свободное пространство заполняется файлами-заглушками (padding files), чтобы на том обнаружения больше ничего нельзя было добавить.

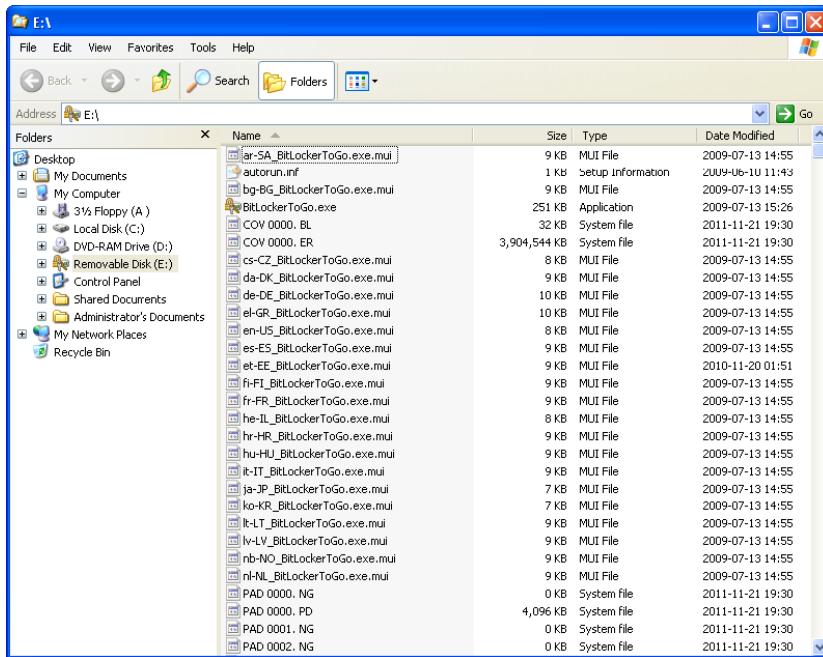


Рис. 9.26. Файлы системы BitLocker To Go

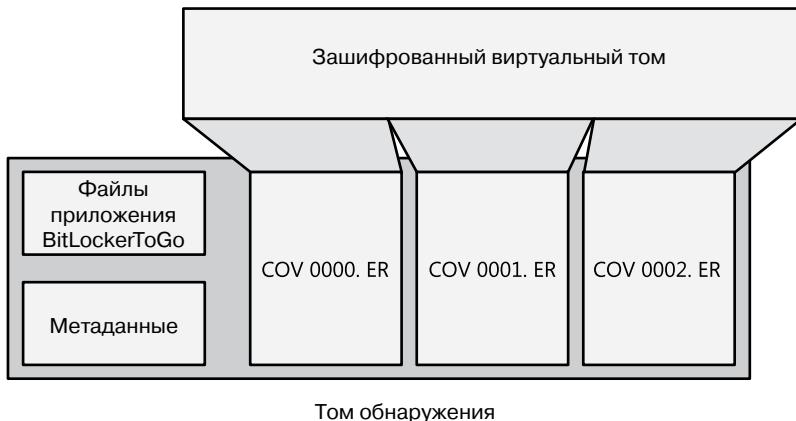


Рис. 9.27. Структура системы BitLocker To Go

После монтирования на зашифрованный виртуальный том приложения BitLocker-ToGo том обнаружения становится невидимым и недоступным. Доступ к виртуальному тому при этом осуществляется в обычном режиме.

Служба теневого копирования томов

Служба теневого копирования томов (Volume Shadow Copy Service, VSS) представляет собой встроенный в Windows механизм, позволяющий создавать согласованные на определенный момент времени копии всех данных, известные как *теневые копии* (shadow copies), или *снимки* (snapshots). Для создания согласованных теневых копий VSS координирует свои действия с приложениями, службами файловой системы, приложениями для резервного копирования, решениями для быстрого восстановления и оборудованием для хранения данных.

Теневые копии

Существует два механизма создания теневых копий — *клонирование* (clone) и *копирование при записи* (copy-on-write). Выбор между ними осуществляется VSS-провайдером (он подробно рассмотрен далее). Провайдеры могут реализовывать снимки наиболее подходящим с их точки зрения способом. К примеру, некоторые провайдеры применяют гибридный подход: начинают с клонирования, а затем прибегают к копированию при записи.

Полные теневые копии

Получаемая путем клонирования *полная теневая копия* (clone shadow copy), также называемая *полным зеркалом* (split mirror), представляет собой полный дубликат оригинальных данных тома, получаемый программным или аппаратным зеркалированием. Клон непрерывно синхронизируется с оригиналом, пока соединение между ними не разрывается для создания теневой копии. После этого оригинал и его клон становятся полностью независимыми друг от друга. На оригинальный том можно записывать новые данные, в то время как теневой том предназначен только для чтения и представляет собой «снимок» исходного тома на момент создания.

Разностные теневые копии

Создаваемая путем копирования при записи так называемая *разностная копия* (differential copy) представляет собой не клон исходных данных, а сведения о внесенных в них изменениях. Эти копии также допускают как программное, так и аппаратное создание. При внесении на диск любых изменений модифицированный блок данных записывается в связанную с теневой копией «область изменений», и только после этого новая версия блока записывается на оригинал тома. Для восстановления состояния системы в определенный момент времени нужно наложить снятый в этот момент модифицированный блок данных на оригинал тома.

ПРИМЕЧАНИЕ

Поставляемый вместе с Windows VSS-провайдер поддерживает только разностные теневые копии.

Архитектура VSS

Служба теневого копирования томов (%SystemRoot%\System32\Vssvc.exe) координирует работу компонентов записи (VSS writers), провайдеров (VSS providers) и инициаторов запросов (VSS requestors). Компонент записи представляет собой программу, включающую для оснащенных средствами работы с теневыми копиями приложений, например для Microsoft SQL Server, Microsoft Exchange Server и Active Directory, возможность получения уведомлений о «замораживании» и «размораживании», что гарантирует внутреннюю целостность резервных копий их данных. Реализация VSS-провайдера позволяет независимым поставщикам аппаратного и программного обеспечения, предлагающим уникальные схемы хранения, использовать службу теневого копирования. К примеру, поставщик зеркалированных накопителей может определить теневую копию как «замороженное» состояние одного из накопителей в зеркале. Инициаторы запросов представляют собой приложения, запрашивающие теневые копии тома и включающие в себя службы резервного копирования и компонент восстановления системы. Взаимоотношения между службой теневого копирования томов (VSS), компонентами записи, провайдерами и инициаторами запросов иллюстрирует рис. 9.28.

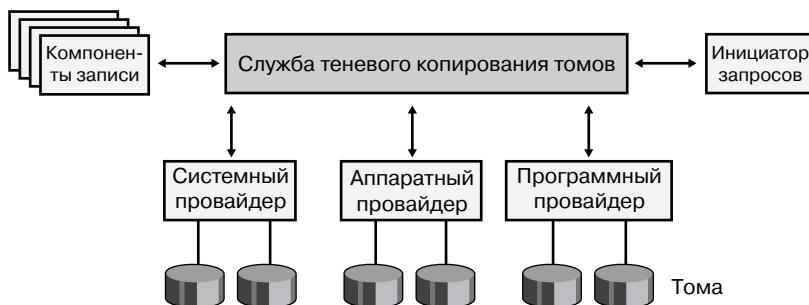


Рис. 9.28. Архитектура VSS

Функционирование VSS

Независимо от назначения теневой копии и приложения, которое ею пользуется, процедура создания теневой копии проходит по схеме, представленной на рис. 9.29. Прежде всего, инициатор запросов отправляет команду в VSS для перечисления компонентов записи, сбора метаданных и подготовки к копированию (1). VSS просит компоненты записи прислать данные о возможности их восстановления и описание компонентов резервного копирования в формате XML (2). Затем каждый компонент записи готовится к копированию по собственной схеме, которая может включать завершение транзакций и сброс кэша. Подготовительная команда также рассыпается всем провайдерам (3).

В этот момент VSS инициирует фазу *подтверждения* (commit) копии (4). Все компоненты записи получают от VSS команду зафиксировать данные и временно приостановить запросы на запись (запросы на чтение пока выполняются). После этого VSS очищает буферы файловой системы тома и отправляет драйверам файловой системы

тота запрос на замораживание всех операций ввода-вывода через команду управления вводом-выводом устройства `IOCTL_VOLSNAP_FLUSH_AND_HOLD_WRITES`. Это гарантирует целостность записи на диск всех метаданных файловой системы (5). Приведя систему в данное состояние, VSS отправляет провайдеру команду создать копию (6). На эту операцию выделяется до 10 секунд. Если процесс за это время не завершается, он прерывается. После создания провайдером теневой копии VSS просит файловую систему «разморозить» операции записи, отправляя им команду `IOCTL_VOLSNAP_RELEASE_WRITES`. Это снимает временную заморозку с компонентов записи, а все стоявшие в очереди запросы на запись начинают обрабатываться (7).

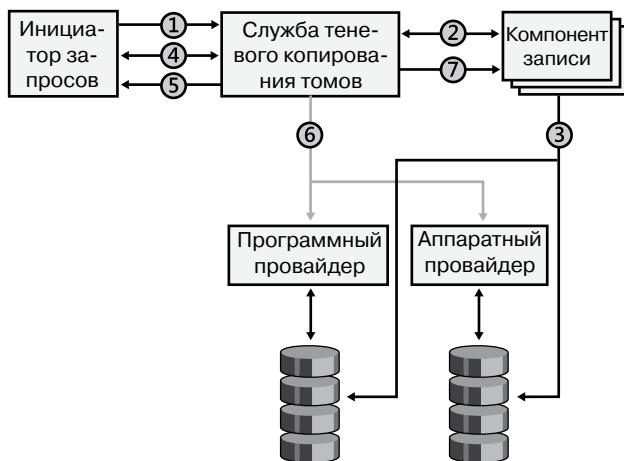


Рис. 9.29. Создание теневой копии в VSS

Затем VSS просит компоненты записи подтвердить, что в процессе создания все операции ввода-вывода были успешно приостановлены, чтобы убедиться в целостности полученной теневой копии. Если целостность копии нарушена в результате повреждения файловой системы, VSS ее удаляет. В остальных случаях сбоя компонентов записи VSS просто уведомляет инициатор запросов, который может либо попробовать вернуться к шагу 1 и повторить процедуру, либо подождать реакции пользователя. Если же создание копии прошло без сбоев, VSS сообщает инициатору запросов ее местоположение.

В качестве дополнительного финального шага можно сделать моментальный снимок тома доступным для записи, чтобы такие модули записи, как TxF (система NTFS с поддержкой транзакций), могли выполнять дополнительные операции восстановления на самом устройстве. После этого этапа восстановления снимок фиксируется в состоянии «только для чтения» и передается инициатору запросов.

ПРИМЕЧАНИЕ

VSS также позволяет переносить снимки на другой сервер — это осуществляется при помощи технологии транспортируемой теневой копии (*transportable shadow copy*).

Провайдер теневого копирования

Провайдер теневого копирования (%SystemRoot%\System32\Drivers\Swprov.dll) реализует разностные копии при помощи драйвера теневого копирования тома (%SystemRoot%\System32\Drivers\Volsnap.sys). Этот фильтрующий драйвер устройств внешней памяти расположен между драйверами файловой системы и драйверами диспетчера томов, поэтому подсистема ввода-вывода адресует операции ввода и вывода непосредственно к тому.

Когда со стороны VSS поступает запрос на создание теневой копии, драйвер теневого копирования ставит адресованные тому операции ввода-вывода в очередь и формирует разностный файл в находящейся на томе папке **System Volume Information**. Именно в этом файле сохраняются постоянно изменяющиеся данные. Кроме того, драйвер теневого копирования создает виртуальный том, через который приложения получают доступ к теневой копии. К примеру, если в пространстве имен диспетчера объектов том называется \Device\HarddiskVolume1, теневой том получит имя вида \Device\HarddiskVolumeShadowCopyN, где N — уникальный идентификатор.

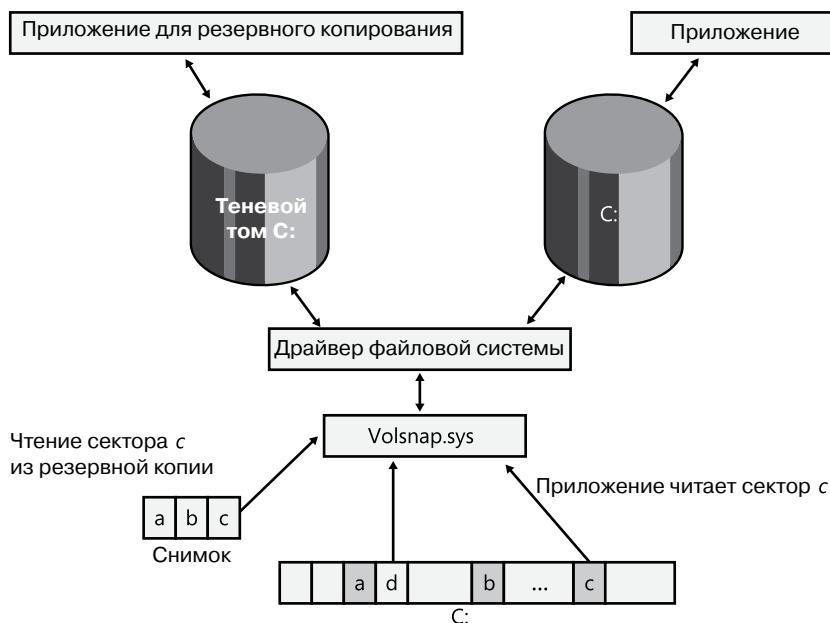


Рис. 9.30. Функционирование драйвера теневого копирования

Каждый раз, когда драйвер теневого копирования обнаруживает операцию записи на рабочий том, он читает копию переписанных секторов в файл подкачки — раздел памяти, связанный с соответствующей теневой копией. Именно отсюда обслуживаются операции чтения, адресованные теневой копии модифицированных секторов. При этом чтение незатронутых редактированием областей осуществляется с рабочего тома. Так как программа резервного копирования не сохраняет ни файл подкачки, ни содержимое расположенной на каждом томе папки **System Volume Information** (в которой находится теневая копия разностных файлов), драйвер теневого копирования с помощью API дефрагментации определяет местоположение файлов и папок и не регистрирует вносимые в них изменения.

На рис. 9.30 показано, как ведут себя приложение, обращающееся к тому, и служба резервного копирования, обращающаяся к теневой копии этого тома. Когда приложение пишет в сектор по истечении времени снятия снимка, драйвер теневого копирования создает резервную копию, как на иллюстрации, где он копирует секторы *a*, *b* и *c* для тома С. Аналогично, когда приложение считывает сектор *c*, драйвер теневого копирования направляет операцию чтения тому С, а когда служба резервного копирования считывает тот же сектор, драйвер теневого копирования получает содержимое этого сектора из снимка. Если операция чтения требует обращения к немодифицированному сектору, например к *d*, драйвер теневого копирования направляет ее исходному тому.

ПРИМЕЧАНИЕ

Драйвер теневого копирования избегает операций копирования при записи для файла подкачки, файла спящего режима и разностных данных, хранящихся в папке System Volume Information. Все остальные файлы получают защиту путем копирования при записи.

ЭКСПЕРИМЕНТ: ОБЪЕКТЫ УСТРОЙСТВ ДЛЯ ДРАЙВЕРА ПРОВАЙДЕРА ТЕНЕВОГО КОПИРОВАНИЯ ОТ MICROSOFT

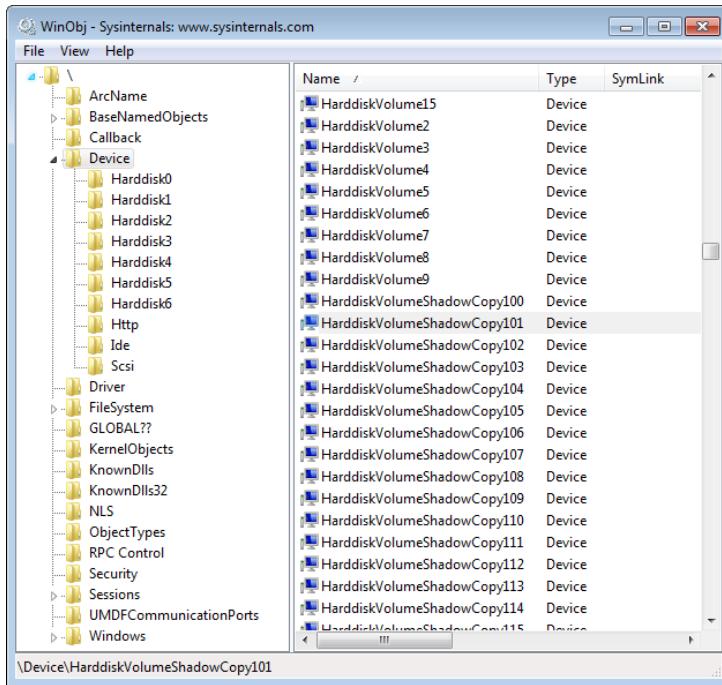
Чтобы посмотреть связанные с каждым томом объекты устройств для драйвера провайдера теневого копирования от Microsoft, воспользуемся отладчиком ядра. В любой системе присутствует хотя бы один том, а следующая команда выводит сведения об объекте устройства для первого тома системы:

```
1: kd> !devobj \device\harddiskvolume1
Device object (88cf908) is for:
HarddiskVolume1 \Driver\volmgr DriverObject 8861a550
Current Irp 00000000 RefCount 3274 Type 00000007 Flags 00201150
Vpb 88cfc3f8 Dacl 8bbcfc7ec DevExt 88cf9c0 DevObjExt 88cfdaa8 Dope
88cfdb38 DevNode 88cfc008
ExtensionFlags (0x00000080) DOE_DEFAULT_SD_PRESENT
Characteristics (0000000000)
AttachedDevice (Upper) 88cf3b8 \Driver\fvevol
Device queue is not busy.
1: kd> !devstack 88cf908
!DevObj !DrvObj !DevExt ObjectName
88d015a0 \Driver\volsnap 88d01658
88cfc478 \Driver\rdyboost 88cfc530
88cf3b8 \Driver\fvevol 88cf470
> 88cf908 \Driver\volmgr 88cf9c0 HarddiskVolume1
!DevNode 88cfc008 :
DeviceInst is "STORAGE\Volume\{53ffaec4-5e9c-11e1-a633-
806e6f6e6963}\#000000000100000"
ServiceName is "volsnap"
```

Адрес объекта устройства HarddiskVolume1 (88cf908) передается команде !devstack, которая выводит вышележащие объекты устройств.

ЭКСПЕРИМЕНТ: ПРОСМОТР ОБЪЕКТОВ УСТРОЙСТВ ТЕНЕВОГО ТОМА

Вы можете убедиться в наличии объектов устройств теневого тома в пространстве имен диспетчера объектов, запустив приложение для резервного копирования. Для этого нужно выбрать в меню Start (Пуск) команду Accessories (Стандартные) ▶ System Tools (Служебные) ▶ WinObj и изучить объекты в папке \Device, как показано на следующем снимке экрана.



Применение в Windows

Службой теневого копирования в Windows пользуется ряд инструментов, в том числе средства резервного копирования, восстановления системы, предыдущих версий и теневого копирования для общих папок. Далее мы рассмотрим некоторые варианты применения службы VSS, поговорим о том, зачем она нужна и какая ее функциональность востребована этими инструментами.

Резервное копирование

Многие службы резервного копирования не работают с открытыми файлами. Если приложение открыло файл для монопольного доступа, доступ к его содержимому для службы резервного копирования будет закрыт. Но даже при наличии доступа к открытому файлу велик риск получить на выходе резервную копию с нарушенной целостностью. Представьте, что приложение обновляет начало, а затем конец файла. Сохраняющая во время этой операции файл служба резервного копирования может записать образ файла, в который начало файла попадет до внесения в него изменений приложением, а конец файла — после их внесения. При восстановлении такого файла приложение может счесть его поврежденным, так как оно, к примеру, умеет обрабатывать файлы, в которых претерпело изменение начало, но остался в исходном виде конец, но не наоборот. Именно поэтому большинство программ резервного копирования предпочитают просто пропускать открытые файлы.

Вместо того чтобы открывать резервируемые файлы на исходном томе, служба резервного копирования открывает их на теневом. Последний отражает представление

тома, привязанное к определенной временной точке. Именно таким способом обходится связанные с открытыми файлами проблема резервного копирования.

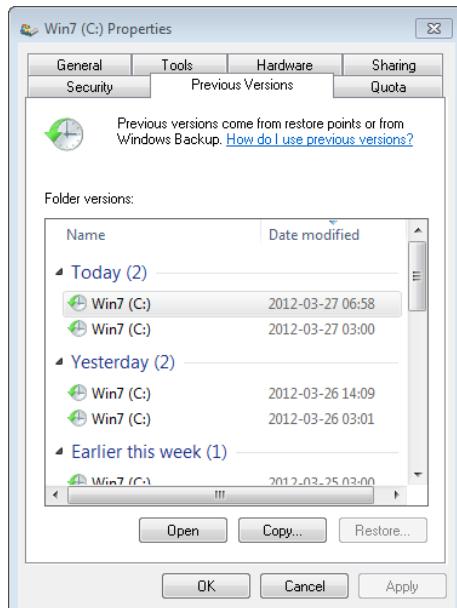
Предыдущие версии и восстановление системы

Служба предыдущих версий в Windows тоже поддерживает автоматическое создание снимков тома. Обычно создается один снимок в день. Доступ к ним осуществляется через Проводник — вам потребуется открыть диалоговое окно *Properties* (Свойства). Тот же интерфейс используется в службе теневых копий для общих папок. Таким способом можно посмотреть, восстановить или скопировать старые версии файлов и папок после случайного удаления или редактирования.

Кроме того, снимки тома позволяют Windows унифицировать механизмы защиты пользовательских и системных данных и избежать повторений в резервируемых данных. Если установка приложения или изменение конфигурации привели к некорректному или нежелательному поведению, функция восстановления системы позволит вернуть системные файлы и данные в состояние на момент создания точки восстановления. Используя для перехода к этой точке интерфейс восстановления системы в Windows 7, вы, по сути, копируете на рабочий том более раннюю версию модифицированных системных файлов из связанного с точкой восстановления снимка.

ЭКСПЕРИМЕНТ: ПРОСМОТР ПРЕДЫДУЩИХ ВЕРСИЙ

Как вы уже видели, каждый раз, когда Windows создает новую точку восстановления системы, появляется теневая копия тома. Проводник позволяет совершить путешествие во времени и увидеть более старые копии каждого подвергшегося теневому копированию диска. Для просмотра предшествующих версий тома следует щелкнуть на имени раздела, например C, правой кнопкой мыши и выбрать команду *Restore Previous Versions* (Восстановить предыдущую версию). Откроется вот такое диалоговое окно.



Выберите любую из предлагаемых версий и щелкните на кнопке Open (Открыть). Откроется новое окно Проводника, показывающее состояние тома на момент создания снимка. Путь к нему будет представлен в форме localhost\С\$\<метка тома> (<диск>:<дата>, <время>). Именно так Проводник осуществляет виртуализацию снятых теневых копий. (Символы С\$ означают локальный скрытый общий ресурс, предлагаемый по умолчанию, который использует сетевая поддержка Windows, — более подробно эта тема рассматривается в главе 7 части I.) Обратите внимание, что в адресной строке Проводника путь отображается в понятной пользователю форме. Чтобы увидеть реальный путь, следует щелкнуть в адресной строке.

ПРИМЕЧАНИЕ

Если на диске практически отсутствует свободное место, пространство под теневые копии выделяться не будет. В этом случае перечня предыдущих версий вы не увидите.

Все представленные теневые копии томов не являются полными аналогами диска. Если бы все содержимое дисков дублировалось, для каждой копии потребовалось бы очень много места. Служба предыдущих версий пользуется механизмом создания разностных копий, о котором мы уже говорили. К примеру, если от времени A до времени B, когда была создана теневая копия, изменился только файл New.txt, в копию попадет только он. Это позволяет применять VSS в клиентских сценариях почти незаметно для пользователя, так как все содержимое диска не дублируется, что позволяет оставаться в рамках ограничений по размеру.

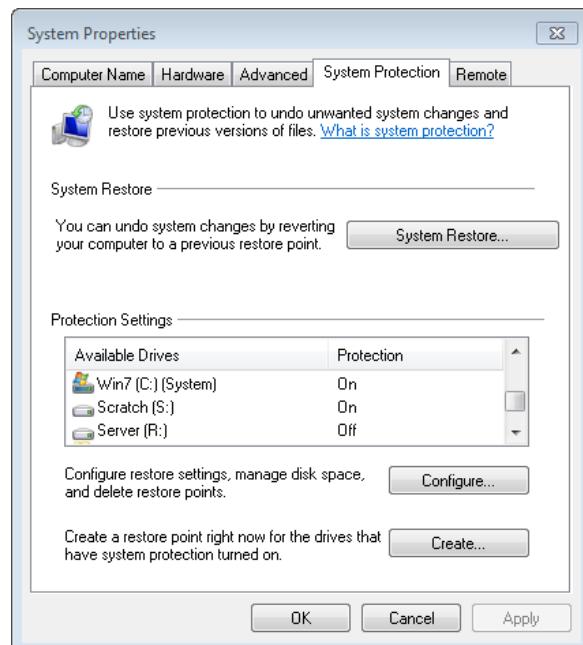


Рис. 9.31. Настройка служб восстановления системы и предыдущих версий

Хотя теневое копирование версий осуществляется ежедневно (или, например, при обновлении Windows и установке программного обеспечения), создать копию можно вручную. Это требуется, например, перед внесением серьезных изменений в систему или после копирования большого набора файлов, который хотелось бы немедленно сохранить, получив содержащий их снимок. Для доступа к соответствующим параметрам нужно щелкнуть правой кнопкой мыши на строке Computer (Компьютер) в меню Start (Пуск) или на Рабочем столе, выбрать команду Properties (Свойства) и в открывшемся окне выбрать вариант System Protection (Защита системы). Кроме того, можно открыть Панель управления и выбрать там раздел System And Maintenance (Система и поддержка), а затем – System (Система). Откроется диалоговое окно, показанное на рис. 9.31, где вы сможете выбрать том, на котором следует включить режим восстановления системы (это окажет влияние на предыдущие версии), немедленно создать точку восстановления и присвоить ей имя.

ЭКСПЕРИМЕНТ: ОТОБРАЖНИЕ ОБЪЕКТОВ УСТРОЙСТВ ДЛЯ ТЕНЕВОГО ТОМА

Хотя предыдущие версии можно посмотреть через Проводник, при этом у вас не будет интерфейса, позволяющего увидеть диск в не зависящей от приложений форме. Встроенная в Windows служебная программа Vssadmin (%SystemRoot%\System32\Vssadmin.exe) позволяет посмотреть все созданные копии и воспользоваться символическими ссылками для их отображения. Вот как это делается:

1. Для вывода списка всех доступных теневых копий применяется команда list shadows:

```
vssadmin list shadows
```

Вы увидите примерно такой результат. Каждая запись представляет собой либо копию предыдущей версии, либо папку общего доступа с включенным режимом теневого копирования.

```
vssadmin 1.1 - Volume Shadow Copy Service administrative command-line tool
(C) Copyright 2001-2005 Microsoft Corp.
Contents of shadow copy set ID: {dfe617b7-ef2b-4280-9f4e-ddf94c2ccfac}
Contained 1 shadow copies at creation time: 8/27/2008 1:59:58 PM
Shadow Copy ID: {f455a794-6b0c-49e4-9ae5-e54647fd1f31}
Original Volume: (C):\?\Volume{f5f9d9c3-7466-11dd-9ba5-806e6f6e6963}\VolumeShadowCopy1
Shadow Copy Volume: \?\GLOBALROOT\Device\HarddiskVolumeShadowCopy1
Originating Machine: WIN-SL5V78KD01W
Service Machine: WIN-SL5V78KD01W
Provider: 'Microsoft Software Shadow Copy provider 1.0'
Type: ClientAccessibleWriters
Attributes: Persistent, Client-accessible, No auto release,
Differential, Auto recovered
Contents of shadow copy set ID: {02dad996-e7b0-4d2d-9fb9-7e692be8fe3c}
Contained 1 shadow copies at creation time: 8/29/2008 1:51:14 AM
Shadow Copy ID: {79c9ee14-ca1f-4e46-b3f0-0dc98f8eb0d4}
Original Volume: (C):\?\Volume{f5f9d9c3-7466-11dd-9ba5-806e6f6e6963}\VolumeShadowCopy2.
Shadow Copy Volume: \?\GLOBALROOT\Device\HarddiskVolumeShadowCopy2.
...
```

Обратите внимание, что показанный в этом листинге идентификатор набора теневых копий совпадает с элементами C\$ в Проводнике, которые вы видели в предыдущем

эксперименте (хотя формат даты/времени может отличаться). Еще данный инструмент отображает том теневой копии, соответствующий объекту устройства для теневой копии, который мы наблюдали при помощи WinObj.

2. Теперь можно воспользоваться служебной программой Mklink.exe и создать символьическую ссылку на папку (подробно символические ссылки рассматриваются в главе 12), которая позволит отобразить теневую копию на реальный адрес. При помощи флага /d создайте ссылку на папку и укажите папку на жестком диске, которая будет отображаться на данный объект устройства. Обязательно укажите путь с обратной косой чертой:

```
mklink /d c:\old \\?\GLOBALROOT\Device\HarddiskVolumeShadowCopy2\
```

3. Ну и наконец, служебная программа Subst.exe позволит отобразить папку c:\old на реальный том с помощью вот этой команды:

```
subst g: c:\old
```

Теперь доступ к старому содержимому вашего диска вы можете получить из любого приложения, указав путь c:\old, или из любой утилиты командной строки, указав путь g:\. Например, команда dir g: выводит содержимое вашего диска.

Заключение

В этой главе мы рассмотрели вопросы организации внешней дисковой памяти в Windows, а также ее компоненты и принципы управления ею. Глава 11 посвящена диспетчеру кэша, который представляет собой исполнительный компонент, неразрывно связанный с драйверами файловой системы, смонтированной на рассмотренных в этой главе типах томов. Но сначала нам предстоит близко познакомиться с неотъемлемым компонентом ядра Windows — диспетчером памяти.

Глава 10. Управление внутренней памятью

Из данной главы вы узнаете, как в Windows реализуется виртуальная память и как Windows управляет подмножеством виртуальной памяти внутри физической памяти. Также в этой главе мы рассмотрим внутреннюю структуру и компоненты, составляющие диспетчер памяти, включая основные структуры данных и алгоритмы. Перед изучением этих механизмов мы познакомимся с основными службами, предоставляемыми диспетчером памяти, и связанными с ним концепциями, например сравним зарезервированную память с подтвержденной и общей памятью.

Знакомство с диспетчером памяти

По умолчанию виртуальный размер процесса для 32-разрядной версии Windows составляет 2 Гбайт. Если же образ специально помечен как подготовленный к большому адресному пространству, а система загружена со специальным параметром (см. далее в этой главе), размер процесса для 32-разрядной версии Windows может возрасти до 3 Гбайт, а для 64-разрядной – до 4 Гбайт. Размер виртуального адресного пространства процесса для 64-разрядной версии Windows на системах IA64 составляет 7,152 Гбайт, а на системах x64 – 8,192 Гбайт. (В будущих выпусках это значение может возрасти.)

Как показано в главе 2 части I (см., в частности, табл. 2.2), максимальный объем физической памяти, поддерживаемой в настоящее время Windows, находится в диапазоне от 2 до 2,048 Гбайт в зависимости от версии и выпуска Windows. Поскольку виртуальное адресное пространство может быть больше или меньше, чем физическая память машины, у диспетчера памяти есть две основные задачи:

- ❑ Перевод, или отображение, виртуального адресного пространства процесса на физическую память, чтобы при выполнении программного потока в контексте данного процесса чтение (и запись) в виртуальном адресном пространстве происходило с соответствующей ссылкой на физический адрес. (Физически резидентное подмножество виртуального адресного пространства процесса называется рабочим набором. Подробно рабочие наборы рассматриваются в данной главе позже.)
- ❑ Подкачка части содержимого памяти на диск при ее избыточном подтверждении (ситуация, когда выполняемые программные потоки или системный код пытаются использовать больше физической памяти, чем ее доступно на данный момент) и возвращение содержимого обратно в физическую память по мере необходимости.

Кроме управления виртуальной памятью, диспетчер памяти предоставляет основной набор служб, на которых построены различные подсистемы Windows. К этим

службам относятся служба отображения на память файлов, которые внутри системы называются *объектами разделов* (section objects), служба копирования при записи в память и служба поддержки приложений, использующих большие разбросанные адресные пространства. Кроме того, диспетчер памяти предоставляет процессам механизм распределения и использования объемов физической памяти, превышающих те объемы, которые могут быть одновременно отображены на виртуальное адресное пространство процесса (например, для 32-разрядных систем, оснащенных физической памятью, превышающей 3 Гбайт). Этот механизм рассматривается далее в разделе «Оконные расширения адресов».

ПРИМЕЧАНИЕ

Управление размером, количеством и размещением файлов подкачки обеспечивается апллетом Панели управления, спецификация которого предполагает, что виртуальная память и файл подкачки (он же страничный файл) — одно и то же, однако это не так. Файл подкачки является всего лишь одним из аспектов виртуальной памяти. На самом деле, если работать вообще без файлов подкачки, Windows все равно будет использовать виртуальную память. Более подробно эта разница рассматривается в данной главе чуть позже.

Компоненты диспетчера памяти

Диспетчер памяти является частью исполнительной системы Windows и поэтому он находится в файле Ntoskrnl.exe. В HAL никаких частей диспетчера памяти нет. Диспетчер памяти состоит из следующих компонентов:

- ❑ Набор служб исполнительной системы, предназначенных для выделения виртуальной памяти, освобождения выделенной памяти и управления памятью; основная часть этих служб доступна через Windows API или через интерфейсы драйверов устройств режима ядра.
- ❑ Обработчик системных прерываний, вызванных неправильным переводом и отказом в доступе для разрешения обнаруженных оборудованием исключений диспетчера памяти и превращения виртуальных страниц в резидентные от имени процесса.
- ❑ Шесть основных высокоуровневых процедур, каждая из которых запускается в одном из шести различных программных потоков режима ядра в процессе **System** (см. эксперимент «Отображение системного потока на драйвер устройства» в главе 2 части I, который показывает, как идентифицировать системные потоки):
 - *Диспетчер настройки баланса* (**KeBalanceSetManager**, приоритет 16) раз в секунду (и в случае уменьшения объема свободной памяти ниже определенного порога) вызывает внутреннюю процедуру, называемую диспетчером рабочих наборов (**MmWorkingSetManager**). Диспетчер рабочих наборов управляет общими политиками диспетчеризации памяти, такими как точная настройка рабочего набора, устаревание данных и запись измененных страниц.
 - *Планировщик подкачки процесса и стека* (**KeSwapProcessOrStack**, приоритет 23) выполняет подкачку в память и из памяти как для процесса, так и для стека

потока ядра. Диспетчер настройки баланса и код планировки потоков в ядре активируют этот программный поток, когда нужна операция подкачки в память или из памяти.

- *Процедура записи измененных страниц* (*MiModifiedPageWriter*, приоритет 17) записывает измененные страницы из списка таких страниц назад в соответствующие файлы подкачки. Этот программный поток активируется, когда размер списка измененных страниц нужно уменьшить.
- *Процедура записи отображаемых страниц* (*MiMappedPageWriter*, приоритет 17) записывает измененные страницы в отображаемые файлы на диске (или в удаленном хранилище). Поток активируется, когда размер списка измененных страниц нужно уменьшить или страницы для отображаемых файлов пробыли в списке измененных страниц более 5 минут. Этот второй поток процедуры записи измененных страниц необходим потому, что он может генерировать ошибки отсутствия страниц, которые приводят к запросам свободных страниц. При отсутствии свободных страниц и существовании только одного программного потока процедуры записи измененных страниц система может войти во взаимную блокировку в ожидании свободных страниц.
- *Поток разыменования сегментов* (*MiDereferenceSegmentThread*, приоритет 18) отвечает за сокращение размера кэша, а также за увеличение и уменьшение файла подкачки. (Например, при отсутствии виртуального адресного пространства для роста нерезидентного пула этот поток урезает кэш страниц, чтобы нерезидентный пул, требуемый для его привязки, можно было освободить для повторного использования.)
- *Поток заполнения страниц нулями* (*MmZeroPageThread*, базовый приоритет 0) заполняет нулями страницы в списке свободных страниц, чтобы кэш заполненных нулями страниц мог удовлетворить в будущем потребности в ошибках обращения к нулевым страницам. В отличие от других рассматриваемых здесь процедур, эта процедура не является функцией высокого уровня потока, но она вызывается процедурой высокого уровня потока *Phase1Initialization*. Процедура *MmZeroPageThread* никогда не возвращает управление вызвавшей ее процедуре, поэтому в результате вызова этой процедуры поток *Phase 1 Initialization* становится потоком заполнения страниц нулями. В некоторых случаях заполнение памяти нулями осуществляется более быстродействующей функцией под названием *MiZeroInParallel*. Обратите внимание на примечание в разделе «Динамика списков страниц» этой главы.

Далее каждый из этих компонентов рассматривается более подробно.

Внутренняя синхронизация

Как и все остальные компоненты исполнительной системы Windows, диспетчер памяти является полностью реenterабельным, поддерживая возможность одновременного выполнения программных потоков на мультипроцессорных системах. То есть он позволяет двум программным потокам получать ресурсы таким образом, чтобы они не разрушали данные друг друга. Для достижения цели полной реenterабельности при управлении

доступом к своим внутренним структурам данных диспетчер памяти использует различные внутренние механизмы синхронизации, такие как спин-блокировки. (Объекты синхронизации рассматриваются в главе 3 части I.)

К общесистемным ресурсам, доступ к которым должен синхронизироваться диспетчером памяти, относятся:

- динамически распределяемые части виртуального адресного пространства системы;
- системные рабочие наборы;
- пулы памяти ядра;
- список загруженных драйверов;
- список файлов подкачки (страничных файлов);
- списки физической памяти;
- структуры случайного размещения базовых адресов образов (на основе механизма ASLR);
- каждая отдельная запись в базе данных номеров страничных блоков (Page Frame Number, PFN).

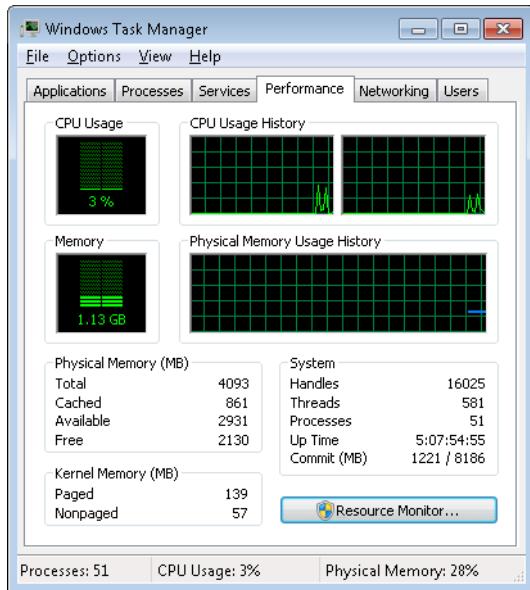
В структуры данных управления памятью каждого процесса, требующие синхронизации, включается блокировка рабочего набора (удерживается, пока вносятся изменения в список рабочего набора) и блокировка адресного пространства (удерживается при изменении адресного пространства). Обе эти блокировки реализуются с использованием механизма пуш-блокировок.

Исследование использования памяти

Доступ к большинству аспектов использования памяти системы и процесса обеспечивается объектами счетчиков производительности памяти и процесса. На протяжении всей данной главы даются ссылки на те или иные счетчики производительности, содержащие информацию, относящуюся к рассматриваемому компоненту. Также предлагаются соответствующие примеры и эксперименты. Тем не менее следует учитывать, что при выводе информации, касающейся памяти, в разных утилитах используются разные, причем не всегда согласованные друг с другом и вносящие путаницу названия. Это показано в следующем эксперименте. (Концепции, упоминаемые в этом примере, рассматриваются в следующих разделах.)

ЭКСПЕРИМЕНТ: ПРОСМОТР ИНФОРМАЦИИ О СИСТЕМНОЙ ПАМЯТИ

На вкладке Performance (Быстродействие) Диспетчера задач выводится основная информация о системной памяти. Эта информация является частью подробной информации о памяти, доступ к которой можно получить через счетчики производительности. Сюда входят данные об использовании как физической, так и виртуальной памяти.



Связанные с памятью показатели и их определения показаны в следующей таблице.

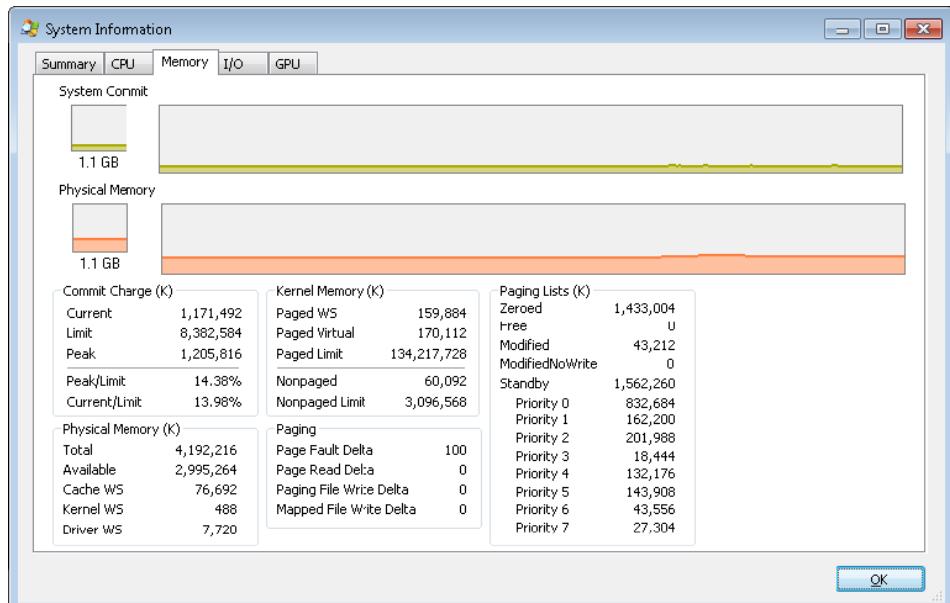
Показатель в диспетчере задач	Определение
Диаграмма Memory (Память)	Высота линий на графике показывает используемую Windows физическую память (которая не показывается в счетчике производительности). Незаполненное пространство графика соответствует показаниям счетчика Available (Доступно) в разделе Physical Memory (Физическая память). Эта диаграмма позволяет увидеть общий объем оперативной памяти, используемый операционной системой, и не включает теневые страницы BIOS, память устройств и т. д.
Раздел Physical Memory (Физическая память), счетчик Total (Всего)	Физическая память, используемая Windows
Раздел Physical Memory (Физическая память), счетчик Cached (Кэшировано)	Суммарный показатель следующих счетчиков производительности в объекте Memory: Cache Bytes, Modified Page List Bytes, Standby Cache Core Bytes, Standby Cache Normal Priority Bytes и Standby Cache Reserve Bytes (все они находятся в объекте Memory)
Раздел Physical Memory (Физическая память), счетчик Available (Доступно)	Объем памяти, непосредственно доступный для использования операционной системой, процессами и драйверами. Равен суммарному размеру списков ожидающих, свободных и нулевых страниц

продолжение ↗

Показатель в диспетчере задач	Определение
Раздел Physical Memory (Физическая память), счетчик Free (Свободно)	Количество байтов в списках свободных и нулевых страниц
Раздел Kernel Memory (Память ядра), счетчик Paged (Выгружаемая)	Количество байтов выгружаемого пула. Это общий размер пула, включая свободную и распределенную область
Раздел Kernel Memory (Память ядра), счетчик Nonpaged (Невыгружаемая)	Количество байтов невыгружаемого пула. Это общий размер пула, включая свободную и распределенную области
Раздел System (Система), счетчик Commit (Выделено) — показаны два числа	Равны показателям счетчиков производительности Committed Bytes и Commit Limit соответственно

Чтобы оценить конкретный показатель выгружаемого и невыгружаемого пулов, нужно воспользоваться утилитой Poolmon (см. раздел «Мониторинг использования пулов»).

Программа Process Explorer из набора Windows Sysinternals (<http://www.microsoft.com/technet/sysinternals>) может показать намного больше данных о физической и виртуальной памяти. Для этого на главном экране нужно выбрать команду View ▶ System Information (Вид ▶ Информация о системе) и перейти на вкладку Memory (Память). Для 32-разрядной версии Windows можно получить следующую картину.



Большинство из показанных здесь дополнительных счетчиков рассматриваются в соответствующих разделах данной главы чуть позже.

Две другие утилиты из набора Sysinternals позволяют увидеть расширенную информацию, касающуюся памяти:

- VMMap — сведения об использовании виртуальной памяти внутри процесса с очень высокой степенью детализации;
- RAMMap — подробные сведения об использовании физической памяти.

Мы еще встретимся с этими утилитами в экспериментах данной главы.

И наконец, команда !vm отладчика ядра позволяет получить основную информацию по управлению памятью, которая доступна из счетчиков производительности, имеющих отношение к памяти. Этой командой можно воспользоваться при рассмотрении аварийного дампа или дампа зависания системы. Вот пример информации, полученной для клиентской системы Windows с памятью объемом 4 Гбайт:

```
1: kd> !vm
*** Virtual Memory Usage ***
Physical Memory: 851757 ( 3407028 Kb)
Page File: \??\C:\pagefile.sys
Current: 3407028 Kb Free Space: 3407024 Kb
Minimum: 3407028 Kb Maximum: 4193280 Kb
Available Pages: 699186 ( 2796744 Kb)
ResAvail Pages: 757454 ( 3029816 Kb)
Locked IO Pages: 0 ( 0 Kb)
Free System PTEs: 370673 ( 1482692 Kb)
Modified Pages: 9799 ( 39196 Kb)
Modified PF Pages: 9798 ( 39192 Kb)
NonPagedPool Usage: 0 ( 0 Kb)
NonPagedPoolNx Usage: 8735 ( 34940 Kb)
NonPagedPool Max: 522368 ( 2089472 Kb)
PagedPool 0 Usage: 17573 ( 70292 Kb)
PagedPool 1 Usage: 2417 ( 9668 Kb)
PagedPool 2 Usage: 0 ( 0 Kb)
PagedPool 3 Usage: 0 ( 0 Kb)
PagedPool 4 Usage: 28 ( 112 Kb)
PagedPool Usage: 20018 ( 80072 Kb)
PagedPool Maximum: 523264 ( 2093056 Kb)
Session Commit: 6218 ( 24872 Kb)
Shared Commit: 18591 ( 74364 Kb)
Special Pool: 0 ( 0 Kb)
Shared Process: 2151 ( 8604 Kb)
PagedPool Commit: 20031 ( 80124 Kb)
Driver Commit: 4531 ( 18124 Kb)
Committed pages: 179178 ( 716712 Kb)
Commit limit: 1702548 ( 6810192 Kb)

Total Private: 66073 ( 264292 Kb)
0a30 CCC.exe 11078 ( 44312 Kb)
0548 dwm.exe 6548 ( 26192 Kb)
091c MOM.exe 6103 ( 24412 Kb)
...
```

Многие позиции, выводимые этой командой, рассматриваются в этой главе далее.

Службы диспетчера памяти

Диспетчер памяти предоставляет группу системных служб, предназначенных для выделения и освобождения виртуальной памяти, распределения памяти между процессами, отображения файлов на память, сброса виртуальных страниц на диск, извлечения информации о диапазоне виртуальных страниц, изменения защиты виртуальных страниц и блокирования виртуальных страниц в памяти.

Как и другие службы исполнительной системы Windows, службы управления памятью позволяют вызывающим их процедурам передавать дескриптор процесса, указывающий на конкретный процесс, с виртуальной памятью которого ведется работа. Таким образом, вызывающая процедура может работать либо со своей собственной памятью, либо (при соответствующих полномочиях) с памятью другого процесса. Например, если процесс создает дочерний процесс, то по умолчанию он имеет право управлять виртуальной памятью дочернего процесса. Соответственно, родительский процесс может выделять, освобождать, читать память и вести запись в память от имени дочернего процесса, вызывая службы виртуальной памяти и передавая им в качестве аргумента дескриптор дочернего процесса. Этой возможностью пользуются подсистемы для управления памятью своих клиентских процессов. Она также необходима для реализации отладчиков, которые должны иметь возможность производить операции чтения из памяти и записи в память отлаживаемого процесса.

Большинство служб управления памятью предоставляют свои возможности через Windows API. Для управления памятью в приложениях Windows API предлагает три группы функций: функции кучи (`Heapxxx` и более старые интерфейсы `Localxxx` и `Globalxxx`, внутренне использующие API-интерфейсы `Heapxxx`), которые могут применяться для распределения пространств памяти меньше страницы; функции виртуальной памяти, работающие со страничными объемами памяти (`Virtualxxx`), и функции, работающие с файлами, отображаемыми на память (`CreateFileMapping`, `CreateFileMappingNuma`, `MapViewOfFile`, `MapViewOfFileEx` и `MapViewOfFileExNuma`). (Диспетчер кучи рассмотрен в данной главе чуть позже.)

Диспетчер памяти также предоставляет ряд служб для других компонентов режима ядра, находящихся внутри исполнительной системы, а также для драйверов устройств. В качестве примера можно привести выделение и освобождение физической памяти и блокировку страниц в физической памяти для передачи данных в режиме прямого доступа к памяти (Direct Memory Access, DMA). Имена этих функций начинаются с префикса `Mm`. Кроме того, хотя, строго говоря, это и не является частью диспетчера памяти, некоторые вспомогательные процедуры исполнительной системы, чьи имена начинаются с префикса `Ex`, служат для выделения и освобождения памяти в системных кучах (выгружаемого и невыгружаемого пула), а также для работы с ассоциативными списками. Эти темы рассмотрены далее в разделе «Кучи режима ядра» данной главы.

Большие и малые страницы

Виртуальное адресное пространство делится на части, называемые страницами. Это связано с тем, что аппаратный диспетчер памяти переводит виртуальные адреса в физические, разбивая их постранично. Следовательно, страница является наиболее мелкой единицей защиты на аппаратном уровне. (Различные варианты защиты

страниц рассмотрены далее в разделе «Защита памяти» этой главы.) Процессоры, на которых работает Windows, поддерживают страницы двух размеров — малые и большие. Фактические размеры варьируются в зависимости от архитектуры процессора (табл. 10.1).

Таблица 10.1. Размеры страниц

Архитектура	Размер малых страниц, Кбайт	Размер больших страниц, Мбайт	Количество малых страниц в большой странице
x86	4	4 (2, если включена поддержка PAE, о чем рассказывается далее в этой главе)	1024 (512 с поддержкой PAE)
x64	4	2	512
IA64	8	16	2048

ПРИМЕЧАНИЕ

Процессоры IA64 поддерживают разнообразные динамически конфигурируемые размеры страниц от 4 Кбайт и до 256 Мбайт. Windows на Itanium использует 8 Кбайт и 16 Мбайт для малых и больших страниц соответственно, поскольку по результатам тестов производительности эти значения были признаны оптимальными. Кроме того, последние процессоры x64 поддерживают для больших страниц размер 1 Гбайт, но Windows этой их особенностью не пользуется.

Главным преимуществом больших страниц является скорость преобразования адресов для ссылок на другие данные внутри большой страницы. Это преимущество основано на том, что первая ссылка на любой байт внутри большой страницы вызывает аппаратный буфер быстрого преобразования адресов (Translation Look-aside Buffer, TLB), о котором рассказывается в одном из следующих разделов, чтобы в его кэшке была информация, необходимая для преобразования ссылок на любой другой байт внутри большой страницы. Если используются малые страницы, для одного и того же диапазона виртуальных адресов понадобится больше TLB-записей, что увеличит степень повторного использования записей, как только потребуется преобразование новых виртуальных адресов. Это, в свою очередь, означает необходимость возвращения к структурам таблицы страниц, где ссылки делаются на виртуальные адреса за пределами области малой страницы, преобразования адресов в которой были кэшированы. TLB является весьма небольшим кэшем, поэтому этот ограниченный ресурс лучше подходит для больших страниц.

Для использования преимуществ больших страниц в системе, обладающей оперативной памятью, превышающей 2 Гбайт, Windows благодаря большим страницам осуществляет отображение на память образов ядра операционной системы (`Ntoskrnl.exe` и `Hal.dll`), а также данных ядра операционной системы (таких, как исходная часть невыгружаемого пула и структуры данных, описывающие состояние каждой физической страницы памяти). Windows также с помощью больших страниц автоматически отображает запросы пространства ввода-вывода (являющиеся обращениями драйверов устройств к `MmMapIoSpace`), если запрос удовлетворяет размеру большой

страницы и выравниванию. Кроме того, Windows позволяет приложениям использовать большие страницы для отображения их образов, собственной памяти и разделов, поддерживаемых страничными файлами. (См. флаг `MEM_LARGE_PAGE` в функциях `VirtualAlloc`, `VirtualAllocEx` и `VirtualAllocExNuma`.) Можно также заставить другие драйверы устройств осуществлять отображение с помощью больших страниц, добавив многострочный параметр реестра к разделу `HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\LargePageDrivers` и указав имена драйверов в виде отдельных строк, оканчивающихся нулевыми байтами.

Попытки выделения больших страниц могут оказаться неудачными, если операционная система проработала уже достаточно длительный период времени, поскольку физическая память для каждой большой страницы может занимать существенное количество последовательно расположенных малых страниц (см. табл. 10.1), и этот объем физических страниц должен к тому же начинаться на границе большой страницы. (Например, физические страницы от 0 по 511 могут использоваться в качестве большой страницы на платформе x64 точно так же, как и страницы от 512 по 1023, но страницы с 10 по 521 использоваться не могут.) При запуске системы свободная физическая память становится фрагментированной. Для выделения малых страниц это не является проблемой, но может привести к отказу в выделении больших страниц.

Указать что-либо для больших страниц, кроме доступа по чтению-записи, невозможно. К тому же память не может быть выгружена ни при каких условиях, поскольку страничная файловая система не поддерживает большие страницы. И поскольку память не может быть выгружена, она не может рассматриваться как часть рабочего набора процесса (см. далее). И при этом распределение больших страниц является причиной ограничений в использовании виртуальной памяти в объеме задания.

Это неприятный побочный эффект больших страниц. Каждая страница (малая или большая) должна быть отображена с единой защитой, применяемой ко всей странице (потому что аппаратная защита памяти осуществляется постранично). Если, к примеру, большая страница содержит как код, предназначенный только для чтения, так и данные, предназначенные для чтения-записи, страница должна быть помечена как доступная для чтения-записи, а стало быть, в код можно будет ввести запись. Следовательно, драйверы устройств или другой код режима ядра могут в результате сбоя модифицировать предназначенный только для чтения код операционной системы или код драйвера, не вызывая нарушений доступа к памяти. Если для отображения кода операционной системы режима ядра используются малые страницы, то предназначенные только для чтения части `Ntoskrnl.exe` и `Hal.dll` могут быть отображены как страницы, предназначенные только для чтения. Применение малых страниц снижает эффективность преобразования адресов, но если драйвер устройства (или другой код режима ядра) попытается модифицировать ту часть операционной системы, которая предназначена только для чтения, это немедленно вызовет системный сбой с выдачей информации исключения, указывающей на проблемную инструкцию в драйвере. Если же запись будет разрешена, то системный сбой, скорее всего, случится чуть позже (и его будет намного труднее диагностировать), когда воспользуется поврежденными данными попытается какой-нибудь другой компонент.

Если есть подозрения о нарушениях кода ядра, нужно включить такое инструментальное средство, как `Driver Verifier` (оно рассматривается далее в этой главе), которое запретит использование больших страниц.

Резервирование и подтверждение страниц

Страницы в виртуальном адресном пространстве процесса могут быть *свободными* (free), *зарезервированными* (reserved), *подтвержденными* (committed) или *общими* (shareable). Подтвержденными и общими являются страницы, при обращении к которым в итоге происходит преобразование с указанием настоящих страниц в физической памяти.

Подтвержденные страницы называются также *закрытыми* (private) – это название отражает тот факт, что они, в отличие от общих страниц, не могут использоваться совместно с другими процессами (а могут, разумеется, использоваться только одним процессом).

Закрытые страницы выделяются с помощью Windows-функций `VirtualAlloc`, `VirtualAllocEx` и `VirtualAllocExNuma`. Эти функции позволяют программному потоку резервировать адресное пространство, а затем подтверждать части зарезервированного пространства. Промежуточное «зарезервированное» состояние позволяет потоку отложить непрерывный диапазон виртуальных адресов для возможного использования в будущем (например, в качестве массива), потребляя при этом незначительные системные ресурсы, а затем выделить подтвержденные части зарезервированного пространства, как только это потребуется для выполнения приложения. Или же, если требуемые объемы известны заранее, поток может выполнить резервирование и подтверждение в одном вызове функции. В любом случае, подтвержденные в итоге страницы затем становятся доступны потоку. Попытки обращения к свободной или зарезервированной памяти приводят к исключению, поскольку страница не отображена ни на одно из хранилищ, способных разрешить ссылку.

Если к подтвержденным (закрытым) страницам до этого еще не было обращений, они создаются во время первого обращения в качестве страниц, подлежащих заполнению нулевыми байтами. Закрытые (подтвержденные) страницы могут быть позже автоматически записаны операционной системой в страничный файл (файл подкачки), если это потребуется для удовлетворения спроса на физическую память. Закрытые страницы обычно недоступны другим процессам.

ПРИМЕЧАНИЕ

Существуют такие функции, как `ReadProcessMemory` и `WriteProcessMemory`, которые, по всей видимости, допускают доступ к памяти между процессами, но они реализованы выполняемым кодом режима ядра в контексте целевого процесса (это называется *при соединением к процессу*). Они также требуют, чтобы либо дескриптор безопасности целевого процесса предоставлял методу доступа право `PROCESS_VM_READ` или `PROCESS_VM_WRITE` соответственно, либо метод доступа владел привилегией `SeDebugPrivilege`, которая по умолчанию предоставляется только участникам группы администраторов.

Общие страницы обычно отображаются на представление *раздела* (section), который, в свою очередь, является частью файла или целым файлом, но может вместо этого представлять часть пространства страничного файла. Все общие страницы потенциально могут использоваться другими процессами. Разделы представлены в Windows API в качестве объектов, отображаемых на файлы.

При первом обращении к общей странице со стороны какого-нибудь процесса она считывается из связанного отображаемого файла (если только раздел не связан с файлом подкачки, тогда страница создается заполненной нулевыми байтами). Позже, если она все еще находится в физической памяти, то есть является *резидентной* (resident),

при повторном и последующих обращениях процессов можно просто использовать то же самое содержимое страницы, которое уже находится в памяти. Общие страницы могут также *заранее* извлекаться из памяти самой системой.

Более подробно общие страницы рассматриваются в разделах «Совместно используемая память и отображаемые файлы» и «Объекты разделов». Страницы записываются на диск с помощью механизма *записи модифицированных страниц* (modified page writing). Эта запись осуществляется при перемещении страниц из рабочего набора процесса в общесистемный список, называемый *списком модифицированных страниц* (modified page list); отсюда они записываются на диск или в удаленное хранилище. (Рабочие наборы и список модифицированных страниц рассматриваются в данной главе чуть позже.) Страницы отображаемых файлов также могут быть записаны назад в свои исходные файлы на диске в результате явного вызова функции `FlushViewOfFile` или процедурой записи отображаемой страницы по мере выставления требований к памяти.

С помощью функции `VirtualFree` или `VirtualFreeEx` вы можете отказаться от подтверждения закрытых страниц и (или) освободить адресное пространство. Разница между отказом от подтверждения и освобождением напоминает разницу между резервированием и подтверждением — память после отказа от подтверждения остается зарезервированной, а освобожденная память становится свободной, не являясь ни подтвержденной, ни зарезервированной.

Использование двухэтапного процесса с резервированием и последующим подтверждением виртуальной памяти приводит к задержке в подтверждении страниц, то есть к задержке, связанной с «показателем подтверждения». Тем не менее резервирование памяти является относительно экономной операцией, поскольку фактически расходуется очень малый объем памяти. Для обновления или построения нужны относительно небольшие внутренние структуры данных, которые представляют состояние адресного пространства процесса. Эти структуры данных, называемые *таблицами страниц* (page tables) и *дескрипторами виртуальных адресов* (Virtual Address Descriptors, VAD), рассматриваются в данной главе чуть позже.

Одним весьма распространенным примером резервирования большого пространства памяти и подтверждения по необходимости его частей является стек пользовательского режима для каждого программного потока. При создании потока стек создается путем резервирования непрерывной части адресного пространства процесса. (По умолчанию предлагается 1 Мбайт; этот размер можно изменить вызовом функций `CreateThread` и `CreateRemoteThread`, а для изменения в масштабах всего образа нужно воспользоваться ключом `/STACK` компоновщика.) По умолчанию исходная страница стека подтверждается, следующая же страница не подтверждается, а помечается как *сторожевая* (guard page), в результате перехватываются все ссылки, выходящие за пределы подтвержденной части стека, и стек расширяется.

ЭКСПЕРИМЕНТ: СРАВНЕНИЕ ЗАРЕЗЕРВИРОВАННЫХ И ПОДТВЕРЖДЕННЫХ СТРАНИЦ

Утилита `TestLimit` (которую можно загрузить с веб-страницы книги) может использоваться для выделения больших объемов зарезервированной или закрытой подтвержденной виртуальной памяти, а разницу можно отследить с помощью программы `Process Explorer`. Сначала нужно открыть два окна командной строки. В одном из них вызовите утилиту `TestLimit`, чтобы создать большой объем зарезервированной памяти:

```
C:\temp>testlimit -r 1 -c 800
```

```
Testlimit v5.2 - test Windows limits
Copyright (C) 2012 Mark Russinovich
Sysinternals - www.sysinternals.com
```

Process ID: 1544

```
Reserving private bytes 1 MB at a time ...
Leaked 800 MB of reserved memory (800 MB total leaked). Lasterror: 0
The operation completed successfully.
```

В другом окне создайте такой же объем подтвержденной памяти:

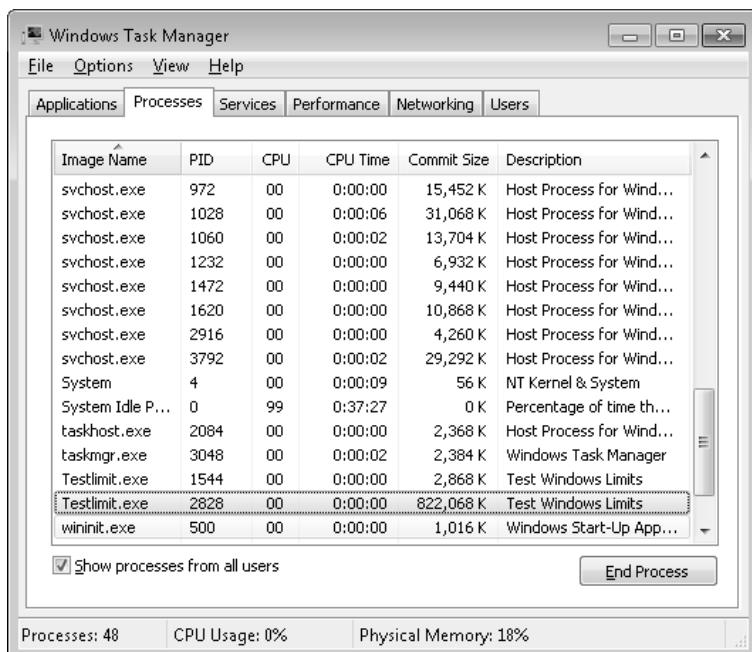
```
C:\temp>testlimit -m 1 -c 800
```

```
Testlimit v5.2 - test Windows limits
Copyright (C) 2012 Mark Russinovich
Sysinternals - www.sysinternals.com
```

Process ID: 2828

```
Leaking private bytes 1 KB at a time ...
Leaked 800 MB of private memory (800 MB total leaked). Lasterror: 0
The operation completed successfully.
```

Далее запустите диспетчер задач, перейдите на вкладку Processes (Процессы) и воспользуйтесь командой Select Columns (Выбрать столбцы) меню View (Вид), чтобы вывести на экран показатель подтвержденной памяти (Memory — Commit Size). Найдите в списке два экземпляра TestLimit. Они должны выглядеть примерно так, как показано на следующем рисунке.



Диспетчер задач показывает объем подтвержденной памяти, но в нем нет счетчиков, которые покажут объем зарезервированной памяти в другом процессе TestLimit.

И наконец, запустите Process Explorer. Выберите команду View ▶ Select Columns (Вид ▶ Выбрать столбцы), перейдите на вкладку Process Memory (Память процесса) и включите счетчики Private Bytes (Закрытые байты) и Virtual Size (Виртуальный размер). Найдите в главном окне два процесса TestLimit.

The screenshot shows the Process Explorer interface with the following data:

Process	PID	CPU	Description	Private Bytes	Virtual Size
svchost.exe	1620		Host Process for Windows Services	10,868 K	55,836 K
svchost.exe	2916		Host Process for Windows Services	4,288 K	43,752 K
svchost.exe	3792		Host Process for Windows Services	29,272 K	124,588 K
System	4	0.08		56 K	7,040 K
System Idle Pr...	0	98.17		0 K	0 K
taskhost.exe	2084		Host Process for Windows Tasks	2,368 K	41,772 K
taskmgr.exe	3048	0.32	Windows Task Manager	2,376 K	69,836 K
Testlimit.exe	1544		Test Windows Limits	2,868 K	844,620 K
Testlimit.exe	2828		Test Windows Limits	822,068 K	844,620 K

At the bottom of the window, status bars show: CPU Usage: 1.83%, Commit Charge: 22.01%, Processes: 48, Physical Usage: 18.11%.

Обратите внимание на то, что виртуальные размеры обоих процессов одинаковы, но только один из них имеет в столбце Private Bytes (Закрытые байты) объем, сопоставимый с объемом в столбце Virtual Size (Виртуальный размер). Большая разница с другим процессом (с идентификатором 1544) получается из-за того, что память является зарезервированной. Такое же сравнение можно сделать в мониторе производительности системного монитора (программа Performance Monitor), сравнив показания счетчиков Process: Virtual Bytes (Процесс: Байт виртуальной памяти) и Process: Private Bytes (Процесс: Байт исключительного пользования).

Лимит подтверждения

На вкладке Performance (Производительность) диспетчера задач в разделе Commit Charge (Выделение памяти) есть два числа. Диспетчер памяти отслеживает использование закрытой подтвержденной памяти с помощью так называемого *показателя подтверждения* (commit charge). Этот показатель, представленный первым из двух чисел, показывает общий объем всей подтвержденной виртуальной памяти в системе.

Существует и общесистемный лимит, называемый *системным лимитом подтверждения* (system commit limit), или просто *лимитом подтверждения* (commit limit). Он выражает объем подтвержденной виртуальной памяти, доступный в любой момент времени. Этот лимит относится к текущему общему объему всех страничных файлов плюс объем оперативной памяти (RAM), доступный операционной системе. Это второе из двух чисел, выводимых в области Commit Charge (Выделение памяти) вкладки Performance (Производительность) диспетчера задач. Диспетчер памяти может автоматически увеличить лимит подтвержденной памяти, расширив один или несколько страничных файлов, если те еще не достигли максимального настраиваемого размера.

Более подробно показатель подтверждения и системный лимит подтверждения рассматриваются в одном из следующих разделов.

Блокирование памяти

В общем случае решение о том, какие страницы оставить в физической памяти, лучше оставить диспетчеру памяти. Но могут быть особые обстоятельства, в которых для приложения или драйвера устройства может возникнуть необходимость заблокировать страницы в физической памяти. Блокирование страниц в памяти можно осуществлять двумя способами:

- ❑ Windows-приложения для блокирования страниц могут в своем рабочем наборе процесса вызвать функцию `VirtualLock`. Страницы, заблокированные с помощью этого механизма, остаются в памяти до тех пор, пока они не будут явным образом разблокированы или пока заблокировавший их процесс не завершится. Количество страниц, которые могут быть заблокированы процессом, не может превышать минимальный размер его рабочего набора минус восемь страниц. Поэтому если процессу нужно заблокировать большее количество страниц, он может увеличить минимум своего рабочего набора с помощью функции `SetProcessWorkingSetSizeEx` (упоминаемой в разделе «Управление рабочими наборами»).
- ❑ Драйверы устройств могут вызвать функцию `MmProbeAndLockPages`, `MmLockPagableCodeSection`, `MmLockPagableDataSection` или `MmLockPagableSectionByHandle` режима ядра. Страницы, заблокированные с помощью этого механизма, остаются в памяти до момента их явного разблокирования. Последние три из этих API-функций не накладывают квоты на количество страниц, блокируемых в памяти, потому что резидентно доступный объем страниц определяется при первой загрузке драйвера, чем гарантируется невозможность краха системы из-за наложения страниц в памяти. Для первой API-функции общая квота должна быть получена, иначе API вернет состояние сбоя.

Гранулярность выделения памяти

Windows устанавливает каждую область зарезервированного адресного пространства процесса на начало целочисленной границы, определяемой значением установленной в системе *гранулярности выделения памяти* (allocation granularity). Это значение может быть извлечено с помощью Windows-функции `GetSystemInfo` или функции `GetNativeSystemInfo`. Оно равно 64 Кбайт, и именно такая гранулярность используется диспетчером памяти для эффективного выделения метаданных (например, дескрипторов виртуальных адресов, двоичных образов и т. д.) для поддержки различных операций процесса. Кроме того, если включена поддержка будущих процессоров с более крупными размерами страниц (например, превышающими 64 Кбайт) или виртуально проиндексированных кэшей, требующих общесистемного выравнивания физических страниц по виртуальным страницам, снизится риск необходимости внесения изменений в приложения, которые строят свою работу на предположении о выравнивании при выделении памяти.

ПРИМЕЧАНИЕ

Код режима ядра в Windows под такие же ограничения не подпадает. Он может резервировать память с гранулярностью, равной объему одной страницы (хотя это не распространяется на драйверы устройств по ранее рассмотренным причинам). Этот уровень гранулярности используется в основном для более плотной упаковки блоков окружения потока (Thread Environment Block, TEB), и поскольку это исключительно внутренний механизм, код может быть легко изменен, если для будущей платформы потребуются другие значения. Также в целях поддержки 16-разрядных приложений и DOS-приложений на системах x86 диспетчер памяти предоставляет API-функции MapViewOfFileEx флаг MEM_DOS_LIM, который служит для установки принудительной гранулярности, равной одной странице.

И наконец, когда резервируется область адресного пространства, Windows гарантирует, что размер и базовый адрес области кратны размеру системной страницы, каким бы он ни был. Например, поскольку системы x86 используют страницы величиной 4 Кбайт, при попытке резервирования области памяти размером 18 Кбайт фактически на системе x86 будет зарезервировано 20 Кбайт. Если для 18-килобайтной области указано базовое значение 3 Кбайт, фактический объем зарезервированной памяти составит 24 Кбайт. Учтите, что дескриптор виртуального адреса (VAD) для выделения будет также округлен до 64-килобайтного выравнивания и длины, делая остаток, получающийся от этого округления, недоступным. (VAD-дескрипторы рассматриваются в этой главе чуть позже.)

Совместно используемая память и отображаемые файлы

Как и большинство современных операционных систем, Windows предоставляет механизм для совместного использования памяти несколькими процессами и операционной системой. *Общую память* (shared memory) в этом случае можно определить как память, видимую более чем одному процессу, или память, которая присутствует в виртуальном адресном пространстве более чем одного процесса. Например, если два процесса задействуют одну и ту же DLL-библиотеку, целесообразно загрузить страницы кода этой библиотеки, из которой делаются ссылки на физическую память, только один раз и совместно использовать эти страницы всеми процессами, как показано на рис. 10.1.

Каждый процесс будет по-прежнему содержать собственные области памяти для хранения закрытых данных, но страницы кода DLL-библиотек и неизменяемых данных могут использоваться совместно без причинения какого-либо вреда. Как показано далее, такой вид совместного использования реализуется автоматически, поскольку страницы кода в исполняемых образах (файлы с расширениями .exe и .dll и некоторые другие типы файлов, такие как экранные заставки с расширением .scr, являющиеся фактически DLL-библиотеками под другими именами) отображаются как страницы, предназначенные только для выполнения, а страницы, предназначенные для записи, отображаются как копируемые при записи — дополнительные сведения можно найти в разделе «Копирование при записи»).

Базовые элементы диспетчера памяти, предназначенные для реализации совместно используемой памяти, называются *объектами разделов* (section objects), а в Windows API они представлены как *объекты отображения файлов* (file mapping objects). Внутренняя структура и реализация объектов разделов рассматривается далее в этой главе в разделе «Объекты разделов».

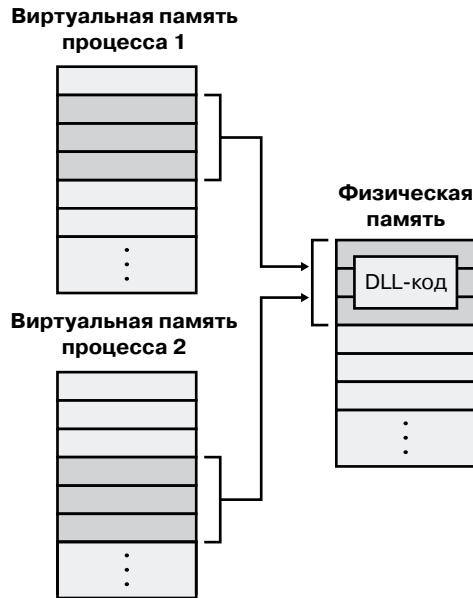


Рис. 10.1. Совместное использование памяти несколькими процессами

Этот базовый элемент диспетчера памяти служит для отображения виртуальных адресов в основной памяти, в страничном файле или в каком-нибудь другом файле, к которому приложению нужно получить доступ при его наличии в памяти. Раздел может быть открыт одним процессом или несколькими процессами, иными словами, объекты разделов не обязательно равнозначны общей памяти.

Объект раздела может быть связан с открытым файлом на диске (который называется отображаемым файлом) или с подтвержденной памятью (для предоставления общей памяти). Разделы, отображаемые на подтвержденную память, поддерживаются страничными файлами, потому что страницы записываются в страничный файл (в отличие от отображаемого файла), если того требуют запросы на физическую память. (Поскольку Windows может работать без страничных файлов, разделы, поддерживаемые страничными файлами, фактически могут «обслуживаться» только физической памятью.) Как и в случае любой другой пустой страницы, видимой коду в пользовательском режиме (например, закрытой подтвержденной страницы), общие подтвержденные страницы при первом к ним обращении всегда заполнены нулевыми байтами, чтобы гарантировать полное отсутствие утечки секретных данных.

Для создания объекта раздела нужно вызвать Windows-функцию `CreateFileMapping` или `CreateFileMappingNuma`, указать дескриптор файла, на который он будет отображен (или `INVALID_HANDLE_VALUE` для раздела, поддерживаемого страничными файлами), и дополнительно указать имя и дескриптор безопасности. Если у раздела есть имя, то с помощью функции `OpenFileMapping` он может быть открыт другими процессами. Или же можно предоставить доступ к объектам разделов либо через наследование дескрипторов (указав возможность наследования дескриптора при его открытии или создании), либо через дублирование дескриптора (используя функцию `DuplicateHandle`). Драйверы устройств также могут работать с объектами разделов с помощью функций `ZwOpenSection`, `ZwMapViewOfSection` и `ZwUnmapViewOfSection`.

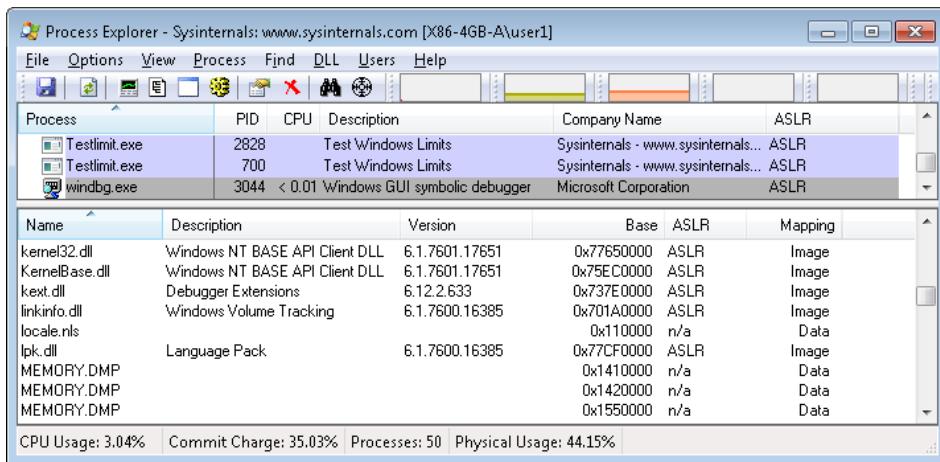
Объекты разделов могут ссылаться на файлы, намного превышающие размеры адресного пространства процесса. (Если объект раздела поддерживается страничным файлом, в страничном файле и (или) в оперативной памяти должно быть пространство, достаточное для размещения этого объекта.) Для обращения к очень большому объекту раздела процесс может отобразить только часть требуемого ему объекта раздела, называемую *представлением раздела* (*view of the section*), вызвав функцию `MapViewOfFile`, `MapViewOfFileEx` или `MapViewOfFileExNuma`, а затем указав диапазон отображения. Отображаемые представления позволяют процессам сохранять адресное пространство, поскольку на память должны отображаться только те представления объекта раздела, которые нужны в данный конкретный момент времени.

Windows-приложения могут использовать отображаемые файлы для файлового ввода-вывода, просто заставив их появиться в своем адресном пространстве. Пользовательские приложения не являются единственными потребителями объектов разделов: загрузчик образов задействует объекты разделов для отображения в памяти исполняемых образов, DLL-библиотек и драйверов устройств, а диспетчер кэша — для доступа к данным в кэшируемых файлах. (Сведения о том, как диспетчер кэша взаимодействует с диспетчером памяти, можно найти в главе 11.) Реализация разделов общей памяти в концепциях преобразования адресов и внутренних структур данных рассматривается в данной главе чуть позже.

ЭКСПЕРИМЕНТ: ПРОСМОТР ОТОБРАЖАЕМЫХ НА ПАМЯТЬ ФАЙЛОВ

Список отображаемых на память файлов процесса можно получить с помощью программы *Process Explorer*, разработанной в *Sysinternals*. Для этого настройте самую нижнюю панель на показ DLL-библиотек, выбрав в меню команду `View ▶ Lower Pane View ▶ DLLs` (Вид ▶ Вид нижней панели ▶ DLL-библиотеки). Учтите, что в выводимом списке указываются не только DLL-библиотеки — в нем представлены все отображаемые на память файлы в адресном пространстве процесса. Некоторые из них являются DLL-библиотеками, один — исполняемым файлом образа (EXE), а остальные элементы списка могут представлять отображаемые на память файлы данных.

Например, на следующем рисунке показано окно *Process Explorer*, демонстрирующее процесс *WinDbg*, который имеет несколько различных вариантов отображения на память для доступа к изучаемому файлу дампа памяти. Подобно большинству Windows-программ, он (или одна из используемых им DLL-библиотек) задействует отображаемый на память файл для доступа к файлу данных с именем *Locale.nls*, который является частью механизма поддержки интернационализации в Windows.



Вести поиск отображаемых на память файлов можно также, выбрав в меню команду Find ▶ DLL. Так можно попытаться определить, какой процесс (или какие процессы) использует DLL-библиотеку или отображеный на память файл, который вы решили заменить.

Защита памяти

Как отмечено в главе 1 части I, Windows обеспечивает защиту памяти, не позволяющую пользовательскому процессу случайно или преднамеренно нанести повреждение адресному пространству другого процесса или операционной системы. Для реализации такой защиты Windows предлагает четыре основных механизма.

Во-первых, все общесистемные структуры данных и пулы памяти, используемые системными компонентами режима ядра, могут быть доступны только в режиме ядра, потоки пользовательского режима к этим страницам обращаться не могут. Если с их стороны будет предпринята такая попытка, возникнет аппаратная ошибка, о которой диспетчер памяти сообщит программному потоку как о нарушении прав доступа.

Во-вторых, у каждого процесса имеется отдельное закрытое адресное пространство, защищенное от доступа со стороны любого программного потока другого процесса. Исключение не делается даже для общей памяти, поскольку каждый процесс обращается к общим областям по адресам, являющимся частью его собственного виртуального адресного пространства. Единственное исключение касается доступа другого процесса к виртуальной памяти объекта процесса по чтению и записи (или наличия у него отладочной привилегии SeDebugPrivilege), что позволяет использовать функцию `ReadProcessMemory` или `WriteProcessMemory`. При каждом обращении потока к адресу оборудование виртуальной памяти во взаимодействии с диспетчером памяти вмешивается и осуществляет преобразование виртуального адреса в физический. Контролируя процесс преобразования виртуальных адресов, Windows гарантирует, что потоки, запущенные в рамках одного процесса, не смогут необоснованно получать доступ к странице, принадлежащей другому процессу.

В-третьих, в дополнение к мерам, характерным для системы защиты преобразования виртуальных адресов в физические, все процессоры, поддерживаемые Windows, предоставляют ту или иную форму аппаратной защиты памяти (например, записи-чтения, только чтения и т. д.), конкретные особенности такой защиты варьируются в зависимости от процессора. Так, кодовая страница в адресном пространстве процесса помечается как доступная только для чтения, получая таким образом защиту от изменений со стороны пользовательских потоков.

Варианты защиты, определенные в Windows API, перечислены в табл. 10.2. (См. описания функций `VirtualProtect`, `VirtualProtectEx`, `VirtualQuery` и `VirtualQueryEx`.)

Таблица 10.2. Варианты защиты памяти, определенные в Windows API

Атрибут	Описание
PAGE_NOACCESS	Любая попытка чтения и выполнения кода из данной области, а также записи в эту область приведет к нарушению прав доступа
PAGE_READONLY	Любая попытка записи в память (и выполнения в памяти кода для процессоров без поддержки выполнения) приведет к нарушению прав доступа, но чтение разрешается
PAGE_READWRITE	Страница открывается для чтения и записи, но не для выполнения кода
PAGE_EXECUTE	Любая попытка записи в код, находящийся в этой области памяти, приведет к нарушению прав доступа, но выполнение кода (и осуществление операций чтения на всех существующих процессорах) разрешается
PAGE_EXECUTE_READ ¹	Любая попытка записи в эту область памяти приведет к нарушению прав доступа, но выполнение и чтение разрешаются
PAGE_EXECUTE_READWRITE ¹	Страница доступна для чтения, записи и выполнения кода, то есть любые попытки доступа будут успешными
PAGE_WRITECOPY	Любая попытка записи в эту область памяти заставит систему предоставить процессу закрытую копию страницы. На процессорах без поддержки выполнения кода попытка выполнить код в этой области памяти приведет к нарушению прав доступа
PAGE_EXECUTE_WRITECOPY	Любая попытка записи в эту область памяти заставит систему предоставить процессу закрытую копию страницы. Разрешается читать и выполнять код в этой области (в таком случае копия не делается)
PAGE_GUARD	Любая попытка чтения сторожевой страницы или записи в нее приведет к исключению <code>EXCEPTION_GUARD_PAGE</code> и потерне статуса сторожевой страницы. Таким образом, сторожевые страницы действуют в качестве одноразовой сигнальной меры. Следует учесть, что этот флаг может быть указан с любым другим видом защиты страниц, перечисленным в этой таблице, за исключением <code>PAGE_NOACCESS</code>

¹ Защита, связанная с запретом на выполнение, доступна только на процессорах, имеющих необходимую для этого аппаратную поддержку (к примеру, это относится ко всем процессорам x64 и IA64, но не к более старым процессорам x86).

Атрибут	Описание
PAGE_NOCACHE	Приводит к использованию некэшируемой физической памяти. Для общего пользования не рекомендуется. Применяется для драйверов устройств, например для отображения буфера кадра без кэширования
PAGE_WRITECOMBINE	Разрешает доступ к памяти в режиме объединенной записи. При таком разрешении процессор не кэширует записи в память (что может привести к существенному росту трафика по сравнению с кэшированием записи в память), но при этом предпринимается попытка объединить запросы на запись для оптимизации производительности. Например, если осуществляется множество записей по одному и тому же адресу, может быть выполнена только самая последняя запись. Отдельные записи по примыкающим адресам также могут быть сведены к одной продолжительной записи. Для обычных приложений это вряд ли пригодится, а вот для драйверов устройств – вполне, например в виде объединенной записи отображаемого буфера видеокадра

И наконец, объекты разделов общей памяти имеют обычные в Windows списки контроля доступа (Access Control Lists, ACL). Эти списки проверяются при попытке открытия разделов общей памяти, ограничивая доступ к общей памяти только теми процессами, у которых имеются соответствующие права. Контроль доступа также осуществляется при создании потоком раздела, содержащего отображаемый файл. Для создания раздела у потока должен быть к намеченному файлу как минимум доступ по чтению, иначе операция закончится неудачей.

После того как программный поток успешно откроет дескриптор раздела, его действия по-прежнему будут контролироваться диспетчером памяти и упомянутым ранее механизмом аппаратной защиты страниц. Поток может изменить защиту на уровне страниц в отношении виртуальных страниц раздела, если изменение не нарушает права доступа, определенные в ACL для данного объекта раздела. Например, диспетчер памяти позволяет потоку внести изменения в страницы раздела, предназначенные только для чтения, разрешив к ним доступ для копирования при записи (copy-on-write), но не разрешив доступ для чтения-записи. Доступ для копирования при записи будет разрешен, потому что он не оказывает влияния на другие процессы, участвующие в совместном использовании данных.

Защита страниц от выполнения

Защита страниц от выполнения, также известная как предотвращение выполнения кода (Data Execution Prevention, DEP), приводит к тому, что попытки передачи управления инструкции, находящейся в странице, помеченной как «неисполняемая», заканчиваются отказом в доступе. Тем самым могут пресекаться случаи использования определенными типами вредоносных программ ошибок в системе, позволяющих выполнять код, помещенный в страницу данных, например в стек. Благодаря DEP можно также выявлять неверно написанные программы, неправильно устанавливающие права доступа к тем страницам, из которых они намереваются выполнять код. Если попытка выполнения кода из страницы, помеченной как неисполняемая,

осуществляется из режима ядра, происходит сбой системы с кодом ошибки ATTEMPTED_EXECUTE_OF_NOEXECUTE_MEMORY. (Подобные коды рассматриваются в главе 14.) Если такое имеет место в пользовательском режиме, то потоку, предпринимающему попытку совершить неправильную ссылку, будет выдано исключение STATUS_ACCESS_VIOLATION (0xc0000005). Если процесс выделяет память, в которой должен быть исполняемый код, он должен явно пометить соответствующие страницы, установив флаг PAGE_EXECUTE, PAGE_EXECUTE_READ, PAGE_EXECUTE_READWRITE или PAGE_EXECUTE_WRITECOPY для функций гранулярности выделения памяти.

Для пометки страницы как неисполнимой на 32-разрядных платформах x86, поддерживающих DEP-защиту, в записи таблицы страниц (Page Table Entry, PTE) устанавливается в единицу разряд 63. Поэтому DEP-защита доступна только при работе процессора в режиме расширения физических адресов (Physical Address Extension, PAE), без которого элементы таблицы страниц имеют ширину только в 32 разряда. (См. раздел «Расширение физических адресов» далее в этой главе.) Следовательно, аппаратная поддержка DEP-защиты на 32-разрядных системах требует загрузки ядра с поддержкой PAE-расширения (%SystemRoot%\System32\Ntkrnlpa.exe), даже если система не нуждается в расширенной физической адресации (например, физических адресов более 4 Гбайт). На 32-разрядных системах с аппаратной поддержкой DEP-защиты загрузчик операционной системы загружает ядро с поддержкой PAE автоматически. Для принудительной загрузки ядра без поддержки PAE на системах, которые имеют аппаратную поддержку DEP-защиты, для BCD-параметра nx должно быть установлено значение AlwaysOff, а для параметра rae — значение ForceDisable.

На 64-разрядных версиях Windows защита от выполнения неизменно применяется ко всем 64-разрядным процессам и драйверам устройств и может быть отключена только путем установки для BCD-параметра nx значения AlwaysOff. Защита от выполнения для 32-разрядных программ зависит от конфигурационных параметров системы (см. далее). В 64-разрядной версии Windows защита от выполнения применяется к стекам программных потоков (как в пользовательском режиме, так и в режиме ядра), к страницам пользовательского режима, не помеченным как исполняемые, к выгружаемому пулу ядра и к пулу сеанса ядра (описание пулов памяти ядра дано в разделе «Кучи режима ядра»). Но в 32-разрядной версии Windows защита от выполнения применяется только к стекам потоков и к страницам пользовательского режима, но не применяется к выгружаемому пулу и пулу сеанса.

Применение защиты от выполнения в 32-разрядных процессах зависит от значения BCD-параметра nx. Как показано на рис. 10.2, изменить параметры можно на вкладке Data Execution Prevention (Предотвращение выполнения данных) в диалоговом окне Performance Options (Параметры быстродействия), выбрав в главном меню команду Computer ▶ Properties ▶ Advanced System Settings (Компьютер ▶ Свойства ▶ Дополнительные параметры системы) и щелкнув на кнопке Settings (Параметры) в области Performance (Быстродействие). При настройке режима защиты от выполнения в диалоговом окне Performance Options (Параметры быстродействия) происходит установка соответствующего значения BCD-параметра nx. Варианты значений и их соответствие вариантам настройки на вкладке Data Execution Prevention (Предотвращение выполнения данных) перечислены в табл. 10.3. Перечень 32-разрядных приложений, не охваченных защитой от выполнения, размещается в реестре в разделе HKLM\SOFTWARE\Microsoft\

Windows NT\CurrentVersion\AppCompatFlags\Layers. Для таких приложений указывается полный путь к исполняемому файлу с параметром DisableNXShowUI.

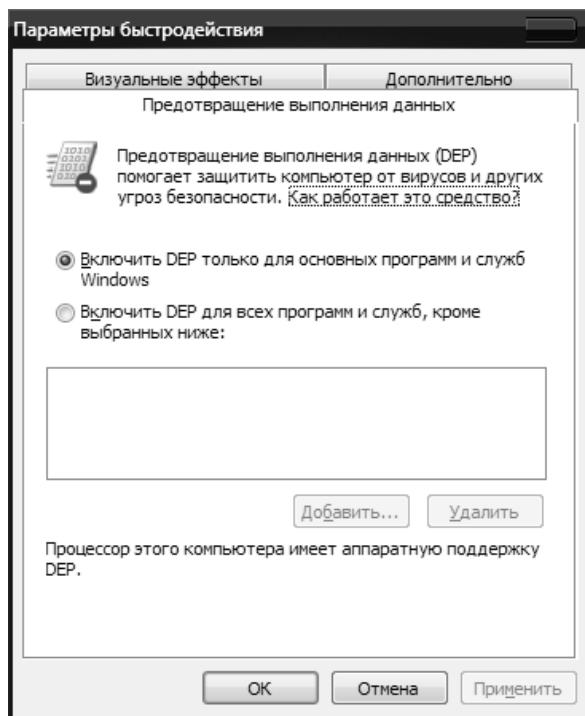


Рис. 10.2. Параметры предотвращения выполнения кода

Таблица 10.3. Значения BCD-параметра px

Значение BCD-параметра px	Параметр предотвращения выполнения кода	Описание
OptIn	Turn on DEP for essential Windows programs and services only (Включить DEP только для основных программ и служб Windows)	DEP-защита включается для основных образов в Windows. Это позволяет 32-разрядным процессам динамически настраивать DEP-защиту на все время их существования
OptOut	Turn on DEP for all programs and services except those I select (Включить DEP для всех программ и служб, кроме выбранных ниже)	DEP-защита включается для всех программ, за исключением указанных в списке. Это позволяет 32-разрядным процессам динамически настраивать DEP-защиту на все время их существования. Для DEP-защиты включается режим исправления системной совместимости

продолжение ↗

Таблица 10.3 (продолжение)

Значение BCD-параметра px	Параметр предотвращения выполнения кода	Описание
AlwaysOn	Соответствующий параметр в диалоговом окне отсутствует	DEP-защита включается для всех компонентов без возможности исключения конкретных приложений. Возможность динамической настройки для 32-разрядных процессов отключается, также отключается режим исправления системной совместимости для DEP-защиты
AlwaysOff	Соответствующий параметр в диалоговом окне отсутствует	DEP-защита отключается (не рекомендуется). Возможность динамической настройки для 32-разрядных процессов отключается

На клиентских версиях Windows (как на 64-разрядных, так и на 32-разрядных) защита от выполнения для 32-разрядных процессов изначально настраивается применительно только к исполняемым компонентам ядра операционной системы (для BCD-параметра px устанавливается значение OptIn), чтобы не нарушать работу 32-разрядных приложений, которая может зависеть от возможности выполнения кода в страницах, не помеченных как исполняемые. Это, к примеру, может касаться самоизвлекающихся или упакованных приложений. На серверных системах Windows защита от выполнения для 32-разрядных приложений изначально настраивается применительно ко всем 32-разрядным программам (для BCD-параметра px устанавливается значение OptOut).

ПРИМЕЧАНИЕ

Для получения полного перечня защищенных программ нужно установить инструментарий Windows Application Compatibility Toolkit (загружаемый с сайта www.microsoft.com) и запустить программу Compatibility Administrator Tool. Затем нужно выбрать команду System Database ▶ Applications ▶ Windows Components (База данных системы ▶ Приложения ▶ Компоненты). Перечень защищенных исполняемых компонентов появится на правой панели.

Даже при принудительном включении DEP-защиты остаются методы, посредством которых приложения могут отключать DEP-защиту своих образов. Например, независимо от заданных параметров защиты от выполнения загрузчик образов (см. главу 3 в части I) будет проверять сигнатуру исполняемости, сравнивая ее с известными механизмами защиты от копирования (например, SafeDisc и SecuROM) и отключая защиту от выполнения с целью обеспечить совместимость со старыми программами защиты от копирования, например для компьютерных игр.

Кроме того, для обеспечения совместимости со старыми версиями (версии 7.1 или ниже) Active Template Library (ATL) ядро Windows предоставляет среду эмуляции ATL-преобразователя (ATL thunk emulation). Эта среда обнаруживает строки кода ATL-преобразователя, что становится причиной исключения DEP-защиты и эмулирования ожидаемых операций. Разработчики приложений могут запросить отказ от

эмуляции ATL-преобразователя, воспользовавшись самой последней версией компилятора Microsoft C++ и указав флаг /NXCOMPAT (который устанавливает в заголовке PE флаг IMAGE_DLLCHARACTERISTICS_NX_COMPAT), сообщающий системе, что исполняемая программа полностью поддерживает DEP-защиту. Учтите, что если установлено значение AlwaysOn, эмуляция ATL-преобразователя постоянно отключена.

И наконец, если система работает в режиме OptIn или OptOut и на ней выполняется 32-разрядный процесс, функция SetProcessDEPPolicy позволяет процессу динамически отключать DEP-защиту или держать ее постоянно включенной. (Когда DEP-защита включается через эту API-функцию, то на все время выполнения процесса программным способом отключить ее будет невозможно.) Эта функция также может быть использована для динамического отключения эмуляции ATL-преобразователя, если образ не был скомпилирован с установленным флагом /NXCOMPAT. При выполнении 64-разрядными процессами или на системах, загруженных с параметром AlwaysOff или AlwaysOn, эта функция всегда возвращает ошибку. Функция GetProcessDEPPolicy возвращает относящееся к конкретному 32-разрядному процессу значение DEP-политики (для 64-разрядных процессов она возвращает ошибку, поскольку для них политика всегда имеет одно и то же значение — защита включена), а функция GetSystemDEPPolicy может использоваться для возвращения значения, соответствующего политикам, перечисленным в табл. 10.3.

ЭКСПЕРИМЕНТ: ПРОСМОТР СТАТУСА DEP-ЗАЩИТЫ ПРОЦЕССОВ

Узнать текущий статус DEP-защиты всех процессов вашей системы, в том числе выяснить, охвачен тот или иной процесс защитой или какова польза от его постоянной защиты, можно с помощью программы Process Explorer. Для просмотра статуса DEP-защиты процессов нужно щелкнуть правой кнопкой мыши на любом столбце дерева процессов, выбрать команду Select Columns (Выбрать столбцы), а затем на вкладке Process Image (Образ процесса) установить один из трех переключателей в группе DEP Status (статус DEP-защиты):

- DEP (permanent) — у процесса включена DEP-защита, поскольку он является «необходимой для Windows программой или службой»;
- DEP — процесс охвачен DEP-защитой по причине, например, общесистемной политики участия всех 32-разрядных процессов, вызова такой API-функции, как SetProcessDEPPolicy, или установки флага компоновщика /NXCOMPAT при создании образа;
- Nothing — если информация для процесса в столбце отсутствует, значит, DEP-защита отключена либо в соответствии с общесистемной политикой, либо в результате явного вызова соответствующей API-функции или прокладки.

В следующем окне программы Process Explorer показан пример системы, на которой для DEP-защиты установлено значение OptOut. Обратите внимание на то, что для двух процессов, запущенных под пользовательской учетной записью, диспетчера звуковой карты стороннего производителя и монитора USB-порта, в столбце DEP в качестве статуса DEP-защиты указано просто DEP. Это означает, что DEP-защита может быть для них отключена с помощью диалогового окна, показанного на рис. 10.2. Для других процессов, соответствующих исполняемым Windows-программам, в столбце DEP в качестве статуса DEP-защиты указано значение DEP (permanent), означающее, что DEP-защиту для них отключить невозможно.

Process	PID	CPU	Description	Company Name	DEP
sposvc.exe	3756	< 0.01	Microsoft Software Protection Plat...	Microsoft Corporation	DEP (permanent)
svchost.exe	3792	< 0.01	Host Process for Windows Services	Microsoft Corporation	DEP (permanent)
lsass.exe	576	< 0.01	Local Security Authority Process	Microsoft Corporation	DEP (permanent)
lsm.exe	584	< 0.01	Local Session Manager Service	Microsoft Corporation	DEP (permanent)
csrss.exe	512	0.06	Client Server Runtime Process	Microsoft Corporation	DEP (permanent)
conhost.exe	2516	< 0.01	Console Window Host	Microsoft Corporation	DEP (permanent)
conhost.exe	2032	< 0.01	Console Window Host	Microsoft Corporation	DEP (permanent)
conhost.exe	2884	< 0.01	Console Window Host	Microsoft Corporation	DEP (permanent)
conhost.exe	3228	< 0.01	Console Window Host	Microsoft Corporation	DEP (permanent)
winlogon.exe	680	< 0.01	Windows Logon Application	Microsoft Corporation	DEP (permanent)
explorer.exe	1420	0.07	Windows Explorer	Microsoft Corporation	DEP (permanent)
RHDV\Cpl.exe	2232	< 0.01	Realtek HD Audio Manager	Realtek Semiconductor	DEP
nusbmon.exe	2248	< 0.01	USB 3.0 Monitor	Renesas Electronics C...	DEP
cmd.exe	2520	< 0.01	Windows Command Processor	Microsoft Corporation	DEP (permanent)
prosexp.exe	2620	0.97	Sysinternals Process Explorer	Sysinternals - www.sys...	DEP (permanent)
RAMMap.exe	1148	< 0.01	RamMap - physical memory analyzer	Sysinternals - www.sys...	DEP (permanent)

CPU Usage: 1.81% | Commit Charge: 35.38% | Processes: 49 | Physical Usage: 44.75%

Программное предотвращение выполнения кода

Для старых процессоров, не поддерживающих аппаратную защиту от выполнения, Windows поддерживает ограниченную *программную защиту от выполнения кода*. Один из аспектов программной DEP-защиты предполагает снижение объема использования механизма обработки исключений в Windows. (Описание структуры обработки исключений дано в главе 3 части I.) Если файлы образов программ созданы с безопасной структурированной обработкой исключений (свойством в компиляторе Microsoft Visual C++, включаемом с помощью флага /SAFESEH), то перед отправкой исключения система проверяет регистрацию обработчика исключений в таблице функций (создаваемой компилятором), размещаемой внутри файла образа.

Предыдущий механизм зависит от файлов образов программ, созданных с безопасной структурированной обработкой исключений. В противном случае программная DEP-защита не дает выполнить перезапись цепочки структурированной обработки исключений в стеке процессов x86 посредством механизма, известного как защита от перезаписи структурированного обработчика исключений (Structured Exception Handler Overwrite Protection, SEHOP). Новая символьическая регистрационная запись исключения добавляется в стек, когда программный поток впервые начинает выполняться в пользовательском режиме. К этой записи приведет нормальная цепочка регистрации исключений. При выдаче исключения диспетчер исключений сначала проходит по перечню записей регистрации обработчиков исключений, чтобы цепочка привела к этой символьической записи. Если этого не происходит, значит, цепочка исключений повреждена (случайно или намеренно) и диспетчер исключений должен просто остановить процесс, не вызывая никаких обработчиков исключений, указанных в стеке. Надежность этого метода повышается за счет механизма случайного размещения схем адресного пространства (Address Space Layout Randomization, ASLR), который усложняет для атакующего кода распознавание месторасположения функции, указывающей на символьическую регистрационную запись исключения, и мешает ему в создании его собственной поддельной символьической записи.

Для дальнейшей проверки структурированного обработчика исключений (Structured Exception Handler, SEH), когда флаг /SAFESEH отсутствует, так называемый механизм *смягчения условий диспетчеризации образа* (image dispatch mitigation) гарантирует, что SEH-обработчик расположен в том же разделе образа, что и функция, выдающая исключение, а это, как правило, характерно для большинства программ (хотя и не обязательно, поскольку у некоторых DLL-библиотек могут быть обработчики исключений, настраиваемые основной исполняемой программой, из-за чего режим смягчения условий диспетчеризации изначально отключен). И наконец, механизм *смягчения условий диспетчеризации исполняемой программы* (executable dispatch mitigation) дает дальнейшую гарантию того, что SEH-обработчик расположен в пределах страницы исполняемой программы — это менее строгое требование, чем предъявляемое к механизму смягчения условий диспетчеризации образа, но оно входит в число тех требований, которые порождают меньше проблем в отношении совместимости.

Двумя другими реализуемыми системой методами программной DEP-защиты являются cookie-значения стека и кодирование указателей. Первый из них основан на том, что компилятор вставляет специальный код в начало и конец каждой потенциально применяемой функции. Этот код сохраняет специальное числовое значение (cookie-значение) на входе в стек и проверяет это cookie-значение перед возвращением управления сохраненному в стеке месту, откуда был сделан вызов (и которое было искажено, чтобы указывать на фрагмент вредоносного кода). Если cookie-значения не совпадают, приложение останавливается, а его дальнейшее выполнение запрещается. Cookie-значение вычисляется для каждой загрузки при выполнении первого программного потока пользовательского режима и сохраняется в структуре KUSER_SHARED_DATA. Загрузчик образа считывает это значение и инициализирует его, когда процесс начинает выполняться в пользовательском режиме. (Дополнительные сведения о совместно используемом разделе данных и о загрузчике образа можно найти в главе 3 части I.)

Вычисленное cookie-значение также сохраняется для использования с API-функциями `EncodeSystemPointer` и `DecodeSystemPointer`, реализующими кодирование указателей. Когда у приложения или у DLL-библиотеки имеются статические указатели, получающие названия в динамическом режиме, возникает риск того, что вредоносный код заменит значения указателей кодом, управляемым вредоносной программой. Кодирование всех указателей с помощью cookie-значений с их последующим декодированием приводит к тому, что когда вредоносный код устанавливает незакодированный указатель, приложение по-прежнему пытается его декодировать, что ведет к порче значения и вызывает аварийное завершение программы. API-функции `EncodePointer` и `DecodePointer` предоставляют такую же защиту, но с cookie-значением, создаваемым по требованию для каждого отдельного процесса, а не для каждой системы.

ПРИМЕЧАНИЕ

Системное cookie-значение представляет собой комбинацию системного времени на момент генерации, стекового значения сохраненного системного времени, количества ошибок обращения к страницам и текущего времени прерывания.

Копирование при записи

Защита страницы путем копирования при записи (copy-on-write) — это оптимизация, используемая диспетчером памяти для сохранения физической памяти. Когда вместо создания закрытой копии процесс отображает копируемое при записи представление объекта раздела, содержащего страницы, доступные для чтения и записи, то при отображении представления диспетчер памяти откладывает создание копии страниц до тех пор, пока не будет произведена запись в страницу. Например, как показано на рис. 10.3, два процесса совместно используют три страницы, каждая из которых имеет пометку копирования при записи, но ни один из двух процессов не пытается изменить какие-либо данные на страницах.

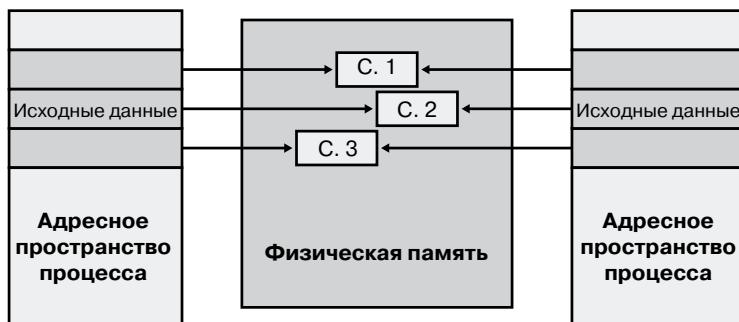


Рис. 10.3. Состояние до копирования при записи

Если поток в любом из процессов осуществляет запись в страницу, система управления памятью выдает ошибку. Диспетчер памяти видит, что запись осуществляется в страницу с копированием при записи, поэтому вместо выдачи ошибки нарушения прав доступа он размещает в физической памяти новую страницу, доступную для чтения и записи, копирует в новую страницу содержимое исходной страницы, обновляет соответствующую информацию об отображении страниц (о том, как это делается, мы поговорим чуть позже) в данном процессе, чтобы указать на новое место страницы, и отклоняет исключение, что приводит к повторному выполнению вызвавшей его инструкции. На этот раз операция записи проходит успешно, но, как показано на рис. 10.4, только что скопированная страница теперь является закрытой страницей того процесса, который осуществлял запись, и не видна другому процессу, по-прежнему пользующемуся общей страницей копирования при записи. Каждый новый процесс, выполняющий запись в ту же самую общую страницу, также будет получать собственную закрытую копию.

Одним из применений механизма копирования при записи является поддержка контрольных точек в отладчиках. Например, изначально страницы с программным кодом предназначаются только для выполнения. Если программист устанавливает при отладке программы контрольную точку, отладчик, конечно же, должен добавить к коду инструкцию остановки в контрольной точке. Он это делает, предварительно

поменяв защиту страницы на PAGE_EXECUTE_READWRITE с последующим изменением череды инструкций. Поскольку страница с программным кодом является частью отображаемого раздела, диспетчер памяти создает для процесса закрытую копию с набором инструкций остановок в контрольных точках, а другие процессы продолжают использовать немодифицированную страницу с программным кодом.

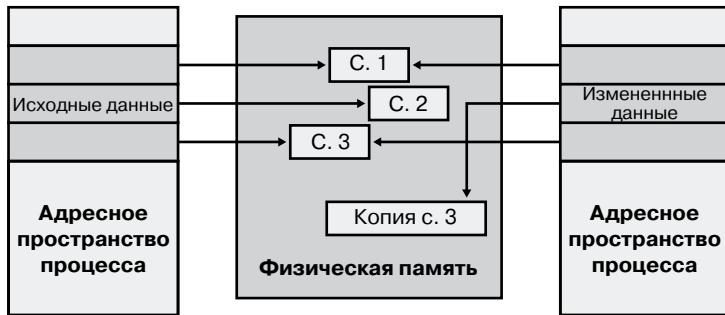


Рис. 10.4. Состояние после копирования при записи

Копирование при записи является одним из примеров технологии, известной как отложенное вычисление (lazy evaluation). Эта технология используется диспетчером памяти максимально часто. Алгоритмы отложенного вычисления избегают выполнять затратные операции до тех пор, пока в них не возникнет абсолютная необходимость. А если такая операция никогда не будет востребована, на нее не придется тратить время зря.

Чтобы узнать уровень отказов в случае копирования при записи, нужно обратить внимание на счетчик производительности Memory: Write Copies/sec (Память: Запись копий страниц/с).

Оконные расширения адресов

Хотя 32-разрядная версия Windows может поддерживать до 64 Гбайт физической памяти (см. табл. 2.2 в главе 2 части I), каждый 32-разрядный пользовательский процесс изначально получает только 2 Гбайт виртуального адресного пространства. (При использовании BCD-параметра increaseuserva, рассматриваемого далее в разделе «Структура пользовательского адресного пространства», можно настроить процесс для адресного пространства вплоть до 3 Гбайт.) Приложение, которому нужен в одном процессе объем легкодоступных данных, превышающий 2 Гбайт (или 3 Гбайт), может добиться желаемого через отображаемые файлы, заново отображая часть своего адресного пространства на различные части большого файла. Но при каждом новом отображении будут возникать существенные потери на страничную подкачку файла.

Для достижения более высокой производительности (а также для более рационального управления) Windows предоставляет набор функций, который называется

ется оконными расширениями адресов (Address Windowing Extensions, AWE). Эти функции позволяют процессу выделять больше физической памяти, чем ее может быть представлено в его виртуальном адресном пространстве. При этом появляется возможность доступа к физической памяти посредством отображения части имеющегося виртуального адресного пространства на выбираемые в разное время части физической памяти.

Выделение и использование памяти через AWE-функции осуществляется в три этапа:

1. Выделение используемой физической памяти. Приложение использует Windows-функцию `AllocateUserPhysicalPages` или `AllocateUserPhysicalPagesNuma`. (Для этого требуется, чтобы у пользователя было право блокирования страниц в памяти.)
2. Создание одной или нескольких областей виртуального адресного пространства с целью их применения в качестве окон для отображения представлений физической памяти. Приложение использует Win32-функцию `VirtualAlloc`, `VirtualAllocEx` или `VirtualAllocExNuma` с флагом `MEM_PHYSICAL`.
3. Предшествующие этапы, в общем-то, можно считать подготовительными. Для фактического использования памяти приложение задействует функцию `MapUserPhysicalPages` или `MapUserPhysicalPagesScatter`, добиваясь отображения части физической области памяти, выделенной на первом этапе, на одну из виртуальных областей, или окон, созданных на втором этапе.

Пример показан на рис. 10.5. Приложение создало в своем адресном пространстве окно размером 256 Мбайт и выделило 4 Гбайт физической памяти (в системе, имеющей более 4 Гбайт физической памяти). Теперь оно может использовать функцию `MapUserPhysicalPages` или `MapUserPhysicalPagesScatter`, чтобы получить доступ к любой части физической памяти, отобразив желаемую часть памяти на окно размером 256 Мбайт. Размер имеющегося у приложения виртуального адресного пространства окна определяется размером физической памяти, к которому приложение может получить доступ с помощью любого заданного отображения. Для доступа к другой части выделенной оперативной памяти приложение может просто заново отобразить требуемую область памяти.

AWE-функции имеются во всех выпусках Windows и доступны независимо от объема имеющейся в системе физической памяти. Но наибольшую пользу из технологии AWE можно извлечь на 32-разрядных системах с объемом физической памяти, превышающим 2 Гбайт, поскольку эта технология позволяет 32-разрядным процессам получить доступ к большему объему оперативной памяти, чем было бы доступно их виртуальному адресному пространству без ее применения. Еще одно применение этой технологии касается вопросов безопасности: поскольку AWE-память никогда не выгружается, данные из AWE-памяти никогда не окажутся в файле подкачки, который злоумышленник мог бы изучить путем перезагрузки в альтернативной операционной системе. (В общем, те же гарантии для страниц предоставляет и функция `VirtualLock`.)

И наконец, следует упомянуть о некоторых ограничениях, накладываемых на выделение памяти и отображение, реализуемые с помощью AWE-функций:

- ❑ страницы не могут совместно использоваться несколькими процессами;
- ❑ одна и та же физическая страница не может быть отображена на более чем один виртуальный адрес одного процесса;
- ❑ защита страниц ограничена вариантами доступа по чтению-записи, только по чтению и отсутствием доступа.

AWE-технология менее полезна для Windows-платформ x64 или IA64, поскольку они поддерживают соответственно 8 и 7 Гбайт виртуального адресного пространства для каждого процесса, при этом разрешая использовать максимум 2 Гбайт оперативной памяти. То есть нет необходимости прибегать к AWE, чтобы разрешить приложениям использовать тот объем оперативной памяти, который превышает имеющееся у них виртуальное адресное пространство, — объем оперативной памяти в системе будет всегда меньше объема виртуального адресного пространства процесса. Но польза от AWE все же остается в отношении конфигурирования невыгруженных областей адресного пространства процесса. Эта технология обеспечивает лучшую гранулярность, чем API-функции отображения файлов (размер системной страницы 4 или 8 Кбайт против 64 Кбайт).

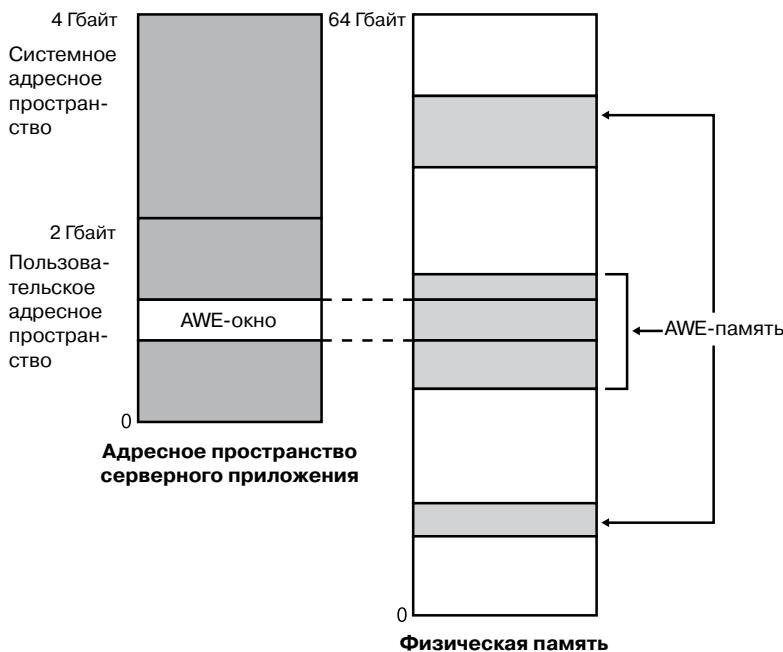


Рис. 10.5. Использование технологии AWE для отображения физической памяти

Описание структур данных таблицы страниц, используемой для отображения памяти в системах, имеющих объем физической памяти, превышающий 4 Гбайт, можно найти в разделе «Расширение физических адресов».

Кучи режима ядра

При инициализации системы диспетчер памяти создает пулы, или кучи, системной памяти, меняющие свой размер в динамическом режиме. Кучи используются для выделения системной памяти многими компонентами, работающими в режиме ядра.

- ❑ **Невыгружаемый пул** состоит из диапазонов системных виртуальных адресов, которые гарантированно находятся в физической памяти в любое время и, следовательно, могут быть доступны в любое время без выдачи ошибки обращения к отсутствующей странице, поэтому доступ к ним можно получить из любого IRQL-уровня. Одной из причин востребованности невыгружаемого пула является правило (см. главу 3 части I), согласно которому ошибки обращения к отсутствующей странице не могут быть разрешены на уровне DPC/dispatch или выше. Поэтому любой исполняемый код или данные, которые должны быть доступны на уровне DPC/dispatch или выше, должны находиться в невыгружаемой памяти.
- ❑ **Выгружаемый пул** представляет собой область виртуальной памяти в системном пространстве, которая может загружаться в систему или выгружаться из нее. Выгружаемый пул может использоваться драйверами устройств, которым не нужен доступ к памяти с уровня DPC/dispatch или выше. Этот пул доступен из контекста любого процесса.

Оба пула памяти размещаются в системной части адресного пространства и отображаются в виртуальном адресном пространстве каждого процесса. Исполнительная система предоставляет процедуры для выделения и освобождения пулов, информация об этих процедурах дана в документации WDK в описании функций, имена которых начинаются с префиксов `ExAllocatePool` и `ExFreePool`.

Система начинает работу, имея четыре выгружаемых пула (объединенных для создания системного выгружаемого пула) и один невыгружаемый пул. Дополнительно могут создаваться максимум до 64 пулов, что зависит от количества имеющихся в системе NUMA-узлов. Наличие более одного выгружаемого пула снижает частоту блокировки системного кода при одновременных вызовах процедур пула. Кроме того, различные созданные пулы отображаются на различные диапазоны виртуального адресного пространства, которые соответствуют различным NUMA-узлам, имеющимся в системе. (Разные структуры данных, такие как большие ассоциативные списки страниц, описывающие распределения пула, также отображаются в различных NUMA-узлах. Об оптимизации NUMA см. далее.)

В дополнение к выгружаемым и невыгружаемым пулам имеется ряд других пулов со специальными атрибутами или вариантами использования. Например, есть область пула в пространстве сеанса, предназначенная для данных, общих для всех процессов сеанса. (Сеансы рассматриваются в главе 1 части I.) Есть еще пул, который буквально называется *особым* (special pool). Память, выделенная из особого пула, окружена страницами, помеченными как недоступные, — это позволяет упростить поиск проблем в коде, который обращается к памяти до или после выделенной ему области пула. Особый пул рассматривается в главе 14.

Размеры пулов

Начальный размер невыгружаемого пула определяется на основе объема физической памяти, имеющегося в системе, а затем увеличивается по мере необходимости. Начальный размер невыгружаемого пула составляет 3 % от объема оперативной памяти системы. Если этот размер составляет менее 40 Мбайт, система будет использовать 40 Мбайт, но только если 10 % от объема оперативной памяти не составят более 40 Мбайт, в противном случае в качестве минимума выбирается 10 % от оперативной памяти.

Максимальный размер пулов Windows выбирает в динамическом режиме, позволяя заданному пулу расти от его минимального размера до максимума (табл. 10.4).

Таблица 10.4. Максимальные размеры пулов

Тип пула	Максимум для 32-разрядных систем	Максимум для 64-разрядных систем
Невыгружаемый	75 % от физической памяти или 2 Гбайт в зависимости от того, что меньше	75 % от физической памяти или 128 Гбайт в зависимости от того, что меньше
Выгружаемый	2 Гбайт	128 Гбайт

Четыре из этих вычисленных размеров хранятся в переменных ядра, три из них выставляются в виде счетчиков производительности, а один вычисляется только как значение счетчика производительности. Все эти переменные и счетчики перечислены в табл. 10.5.

Таблица 10.5. Переменные и счетчики производительности, определяющие размер системного пула

Переменная ядра	Счетчик производительности	Описание
MmSizeOfNonPagedPoolInBytes	Memory: Pool Nonpaged Bytes (Память: Байтов в невыгружаемом страничном пуле)	Начальный размер невыгружаемого пула. Он может быть уменьшен или увеличен системой в автоматическом режиме, если это продиктовано потребностями в памяти. Переменная ядра этих изменений не покажет, но это сделает счетчик производительности
MmMaximumNonPagedPoolInBytes	Недоступен	Максимальный размер невыгружаемого пула
Недоступна	Memory: Pool Paged Bytes (Память: Байтов в выгружаемом страничном пуле)	Текущий общий виртуальный размер выгружаемого пула

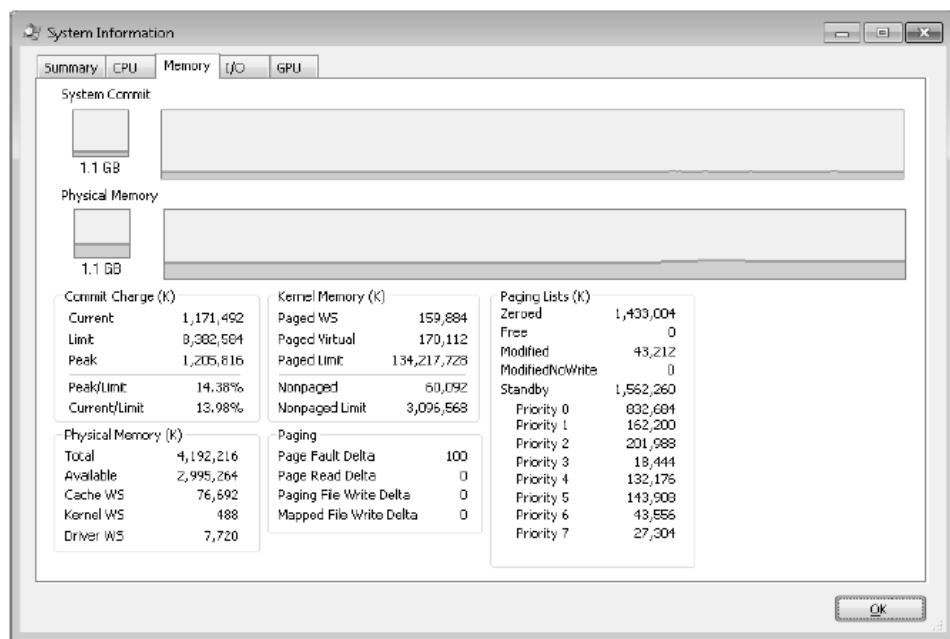
продолжение ↴

Таблица 10.5 (продолжение)

Переменная ядра	Счетчик производительности	Описание
WorkingSetSize (количество страниц) в структуре MmPagedPoolWs (типа _MMSUPPORT)	Memory: Pool Paged Resident Bytes (Память: Байтов в резидентном страничном пуле)	Текущий физический (резидентный) размер выгружаемого пула
MmSizeOfPagedPoolInBytes	Недоступен	Максимальный (виртуальный) размер выгружаемого пула

ЭКСПЕРИМЕНТ: ОПРЕДЕЛЕНИЕ МАКСИМАЛЬНЫХ РАЗМЕРОВ ПУЛОВ

Максимальные размеры могут быть получены либо с помощью Process Explorer, либо путем непосредственной отладки ядра (см. главу 1 части I). Для просмотра максимальных значений с помощью Process Explorer нужно выбрать команду **View ▶ System Information** (Вид ▶ Системная информация) и перейти на вкладку **Memory** (Память). Предельные размеры пулов выводятся в средней области **Kernel Memory** (Память ядра).



Учтите, что для извлечения этой информации у Process Explorer должен быть доступ к символам ядра, запущенного на вашей системе. (Описание вариантов настройки Process Explorer на использование символов можно найти в эксперименте «Просмотр подробностей процесса с помощью Process Explorer» в главе 1 части I.)

Для получения аналогичной информации с помощью отладчика ядра можно воспользоваться командой !vm:

```
kd> !vm
1: kd> !vm
*** Virtual Memory Usage ***
Physical Memory: 851757 ( 3407028 Kb)
Page File: \??\C:\pagefile.sys
Current: 3407028 Kb Free Space: 3407024 Kb
Minimum: 3407028 Kb Maximum: 4193280 Kb
Available Pages: 699186 ( 2796744 Kb)
ResAvail Pages: 757454 ( 3029816 Kb)
Locked IO Pages: 0 ( 0 Kb)
Free System PTEs: 370673 ( 1482692 Kb)
Modified Pages: 9799 ( 39196 Kb)
Modified PF Pages: 9798 ( 39192 Kb)
NonPagedPool Usage: 0 ( 0 Kb)
NonPagedPoolNx Usage: 8735 ( 34940 Kb)
NonPagedPool Max: 522368 ( 2089472 Kb)
PagedPool 0 Usage: 17573 ( 70292 Kb)
PagedPool 1 Usage: 2417 ( 9668 Kb)
PagedPool 2 Usage: 0 ( 0 Kb)
PagedPool 3 Usage: 0 ( 0 Kb)
PagedPool 4 Usage: 28 ( 112 Kb)
PagedPool Usage: 20018 ( 80072 Kb)
PagedPool Maximum: 523264 ( 2093056 Kb)
...
```

На этой 32-разрядной системе с 4 Гбайт оперативной памяти невыгружаемый и выгружаемый пулы далеки от своих максимумов.

Вы также можете исследовать значения переменных ядра, перечисленных в табл. 10.5. Следующие значения были получены на 32-разрядной системе:

```
lkd> ? poi(MmMaximumNonPagedPoolInBytes)
Evaluate expression: 2139619328 = 7f880000
```

```
lkd> ? poi(MmSizeOfPagedPoolInBytes)
Evaluate expression: 2143289344 = 7fc00000
```

Этот пример свидетельствует о том, что максимальный размер как невыгружаемого, так и выгружаемого пулов равен приблизительно 2 Гбайт, что является обычными значениями для 32-разрядных систем с большим объемом оперативной памяти. На системах, использованных для этого примера, текущий невыгружаемый пул имел размер 35 Мбайт, а под выгружаемый пул было выделено 80 Мбайт, то есть оба пула были далеки до своего максимума.

Мониторинг использования пулов

Объект `Memory` счетчика производительности памяти имеет отдельные счетчики для размеров невыгружаемого и выгружаемого пулов (как виртуального, так и физического). Кроме того, детали, касающиеся использования невыгружаемого и выгружаемого пулов, можно отследить с помощью утилиты `Poolmon` (в WDK). При запуске

Poolmon информация, выводимая этой утилитой, должна выглядеть так, как показано на рис. 10.6.

		Memory: 3405552K Avail: 2081680K PageFlts: 39	InRam Krl: 6068K P:63312K	Pool N:42608K P:63648K		
		System pool information				
Tag	Type	Allocs	Frees	Diff	Bytes	Per Alloc
OPM	Paged	9 < 0>	5 < 0>	4	192 < 0>	48
1394	Paged	10 < 0>	10 < 0>	0	0 < 0>	0
1394	Nonp	16 < 0>	3 < 0>	13	17504 < 0>	1346
139c	Paged	1 < 0>	1 < 0>	0	0 < 0>	0
8042	Paged	12 < 0>	12 < 0>	0	0 < 0>	0
8042	Nonp	10 < 0>	3 < 0>	7	4016 < 0>	573
ACPI	Nonp	4 < 0>	4 < 0>	0	0 < 0>	0
ADAP	Nonp	8 < 0>	6 < 0>	2	544 < 0>	272
AFGp	Nonp	1 < 0>	1 < 0>	0	0 < 0>	0
AFGp	Paged	1 < 0>	0 < 0>	1	32 < 0>	32
ALPC	Nonp	7898 < 0>	6984 < 0>	914	257280 < 0>	281
ARFT	Paged	5 < 0>	2 < 0>	3	96 < 0>	32
AcdN	Nonp	2 < 0>	0 < 0>	2	1072 < 0>	536
AcpA	Paged	52 < 0>	52 < 0>	0	0 < 0>	0
AcpA	Nonp	5 < 0>	3 < 0>	2	80 < 0>	40
AcpB	Paged	33 < 0>	32 < 0>	1	16 < 0>	16
AcpD	Nonp	459 < 0>	350 < 0>	109	44872 < 0>	411
AcpF	Nonp	664 < 0>	646 < 0>	18	720 < 0>	40
AcpI	Paged	1311 < 0>	1251 < 0>	60	7264 < 0>	121
AcpI	Nonp	309 < 0>	309 < 0>	0	0 < 0>	0

Рис. 10.6. Информация, выводимая утилитой Poolmon

Подсвеченные строки отражают изменения в выводимой информации. (Режим подсветки можно отключить, введя слэш (/) при запуске Poolmon. Для повторного включения режима подсветки нужно набрать символ / еще раз.) Для вывода экрана справки нужно при работающей утилите Poolmon ввести символ вопроса (?). Отслеживаемый пул (выгружаемый, невыгружаемый или тот и другой), а также порядок сортировки можно задать. Например, нажав клавишу Р при выводе невыгруженных пулов с последующим нажатием клавиши D для сортировки по столбцу Diff (различия), можно выяснить, какие структуры являются наиболее многочисленными в невыгруженном пуле. Кроме того, имеются ключи командной строки, позволяющие отслеживать конкретные признаки (или один конкретный признак). Например, команда poolmon -iCM приведет к мониторингу только CM-признаков, то есть мест выделения памяти от диспетчера конфигурации (configuration manager), который управляет реестром. Значения столбцов описаны в табл. 10.6.

Таблица 10.6. Описание столбцов, выводимых утилитой Poolmon

Столбец	Значение
Tag	Четырехбайтовый признак, свойственный выделению пула
Type	Тип пула: выгружаемый (paged) или невыгружаемый (nonpaged)
Allocs	Подсчет всех выделений памяти (число в круглых скобках показывает разницу в столбце Allocs со временем последнего обновления)
Frees	Подсчет всех освобождений памяти (число в круглых скобках показывает разницу в столбце Frees со временем последнего обновления)
Diff	Подсчет значения Allocs за вычетом значения Frees

Столбец	Значение
Bytes	Общее количество байтов, потребленных этим признаком (число в круглых скобках показывает разницу в столбце Bytes со времени последнего обновления)
Per Alloc	Размер в байтах отдельного экземпляра этого признака

Описание признаков пула, используемых Windows, можно найти в файле \Program Files\Debugging Tools for Windows\Triage\Pooltag.txt. (Этот файл устанавливается как часть отладочного инструментария Debugging Tools for Windows, рассмотренного в главе 1 части I.) Поскольку признаки пула драйверов устройств сторонних производителей в этом файле не перечислены, для генерации локального файла признаков пула (Local-tag.txt) с 32-разрядной версией Poolmon, поставляемой с WDK, можно использовать ключ -с. В этом файле будут содержаться признаки пула, используемые драйверами, найденными на вашей системе, включая и драйверы сторонних производителей. (Учтите, что если исполняемый файл драйвера устройства после загрузки самого драйвера был удален, признаки пула драйвера не распознаются.)

В качестве альтернативы можно искать признаки пула драйверов устройств на вашей системе с помощью программы Strings.exe из набора Sysinternals. Например, командная строка %SYSTEMROOT%\system32\drivers*.sys | findstr /i "abcd" приведет к выводу драйверов, содержащих строку «abcd». Учтите, что драйверы устройств не обязательно находятся в папке %SystemRoot%\System32\Drivers, они могут быть в любой папке. Для получения перечня полных путей ко всем загруженным драйверам нужно из стартового меню открыть диалоговое окно Run (Выполнить) и набрать в нем команду Msinfo32. После этого нужно раскрыть узел Software Environment (Программная среда) и щелкнуть на пункте System Drivers (Системные драйверы). Как уже отмечалось, если драйвер устройства был загружен, а затем удален из системы, в выводимый на экран перечень он не попадет.

Альтернативным вариантом просмотра пула, используемого драйвером устройства, является включение режима отслеживания пула в программе Driver Verifier, которая рассматривается далее в этой главе. Хотя отображать признак пула на драйвер устройств не потребуется, для этого варианта понадобится перезагрузка системы (для подключения Driver Verifier к требуемым драйверам). После перезагрузки с включением режима отслеживания пулов можно будет либо запустить Driver Verifier Manager (%SystemRoot%\System32\Verifier.exe), либо воспользоваться командой Verifier /Log для отправки информации об использовании пулов в файл.

И наконец, информацию об использовании пулов можно получить с помощью команды !poolused отладчика ядра. Команда !poolused 2 выведет информацию для не-выгружаемого пула с сортировкой по признаку пула с наибольшим размером. Команда !poolused 4 покажет список выгружаемых пулов также с сортировкой по признаку пула с наибольшим размером. В следующем примере частично показан результат выполнения этих двух команд:

```
1kd> !poolused 2
      Sorting by NonPaged Pool Consumed
      Pool Used:
      NonPaged          Paged
      Tag    Allocs     Used   Allocs   Used

```

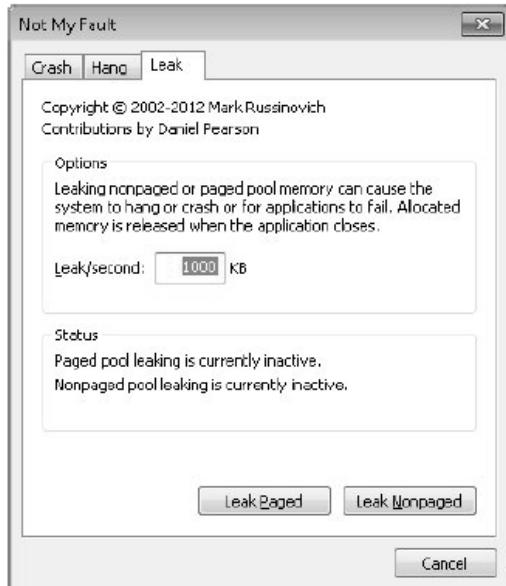
продолжение ↴

Cont	1669	15801344	0	0	Contiguous physical memory allocations for device drivers
Int2	414	5760072	0	0	UNKNOWN pooltag 'Int2', please update pooltag.txt
LSwi	1	2623568	0	0	initial work context
EtwB	117	2327832	10	409600	Etw Buffer , Binary: nt!etw
Pool	5	1171880	0	0	Pool tables, etc.

```
lkd> !poolused 4
Sorting by Paged Pool Consumed
Pool Used:
      NonPaged          Paged
Tag   Allocs    Used   Allocs    Used
CM25      0        0     3921  16777216 Internal Configuration manager
allocations, Binary: nt!cm
MmRe      0        0      720  13508136 UNKNOWN pooltag 'MmRe', please
update pooltag.txt
MmSt      0        0     5369  10827440 Mm section object prototype
ptes, Binary: nt!mm
Ntff      9      2232    4210  3738480 FCB_DATA, Binary: ntfs.sys
AlMs      0        0     212  2450448 ALPC message, Binary: nt!alpc
ViMm    469    440584     608  14688888 Video memory manager, Binary:
dxgkrnl.sys
```

ЭКСПЕРИМЕНТ: ПОИСК И УСТРАНЕНИЕ УТЕЧЕК ПУЛА

В этом эксперименте будет устранена реальная утечка выгружаемого пула на вашей системе, чтобы вы могли использовать для отслеживания утечки технологию, рассмотренную в предыдущем разделе. Утечка будет создана программой Notmyfault, разработанной в Sysinternals. При запуске Notmyfault.exe загружается драйвер устройства Myfault.sys и выводит следующее диалоговое окно.



1. Щелкните на вкладке Leak (Утечка), введите в поле Leak/Second (Утечек в секунду) значение 1000 Кбайт и щелкните на кнопке Leak Paged (Устроить утечку в выгружаемом пуле). В результате Notmyfault начнет отправлять запросы к драйверу устройства Myfault на выделение выгружаемого пула, продолжая это делать до тех пор, пока не будет сделан щелчок на кнопке Stop Paged (Остановить утечку в выгружаемом пуле). Обратите внимание, что выгружаемый пул обычно не освобождается даже при закрытии программы, вызвавшей его появление (при взаимодействии с содержащим ошибки драйвером устройства), причем утечка пула не исчезнет вплоть до перезагрузки системы. Однако чтобы упростить тестирование, драйвер устройства Myfault обнаруживает, какой процесс был открыт, и освобождает выделенную ему память.
2. В ходе утечки пула сначала нужно открыть диспетчер задач и перейти на вкладку Performance (Быстродействие). Обратите внимание на возрастание значения параметра Kernel Memory (MB): Paged (Память ядра (Мб): Выгружаемая). Это возрастание можно также проверить в окне System Information (Системная информация) программы Process Explorer. Для этого выберите команду View ▶ System Information (Вид ▶ Системная информация) и перейдите на вкладку Memory (Память).
3. Для определения признака пула, связанного с утечкой, нужно запустить программу Poolmon и нажать клавишу B для сортировки по количеству байтов. Затем нужно дважды нажать клавишу P, чтобы программа Poolmon показывала только выгружаемый пул. Тогда будет видно, что признак пула Leak (Утечка) окажется в верхней части списка. (Poolmon показывает изменения в выделении памяти пула, подсвечивая изменившиеся строки.)
4. Теперь щелкните на кнопке Stop Paged (Остановить утечку в выгружаемом пуле), чтобы память в вашей системе не была исчерпана.
5. Для просмотра исполняемого файла драйвера, в котором содержится признак Leak (Утечка), воспользуйтесь технологией, рассмотренной в предыдущем разделе, запустив программу Strings, разработанную в Sysinternals:

```
Strings %SystemRoot%\system32\drivers\*.sys | findstr Leak
```

В результате должно быть найдено соответствие в файле Myfault.sys, подтверждающее факт использования драйвером признака пула Leak (Утечка).

Ассоциативные списки

Помимо пулов, Windows предоставляет быстрый механизм выделения памяти, который называется *ассоциативными списками* (look-aside lists). Основное различие пулов и ассоциативных списков заключается в том, что выделение памяти для обычных пулов может варьироваться в размерах, а ассоциативный список содержит только блоки фиксированного размера. Хотя обычные пулы являются более гибкими с точки зрения предоставления памяти, работа с ассоциативными списками происходит быстрее, поскольку они не требуют спин-блокировки.

Компоненты исполнительной системы и драйверы устройств могут создавать ассоциативные списки, соответствующие размерам часто выделяемых структур данных, с помощью функций ExInitializeNPagedLookasideList и ExInitializePagedLookasideList (их описания есть в WDK). Для сведения к минимуму издержек, связанных с синхронизацией мультипроцессорной системы, ряд исполнительных подсистем (таких, как диспетчеры ввода-вывода, кэша и объектов) создают для своих структур данных,

к которым идут частые обращения, отдельные ассоциативные списки для каждого процессора. Для выделения памяти небольших объемов (256 байт и меньше) исполнительная система создает также обычные выгружаемый и невыгружаемый ассоциативные списки для каждого процессора.

Если ассоциативный список пуст (как при его первоначальном создании), система должна выделить память из выгружаемого или невыгружаемого пула. Но если он содержит освобожденный блок, выделение должно быть осуществлено очень быстро. (Список растет по мере того, как ему возвращаются блоки.) Процедуры выделения пулов автоматически настраивают количество освобожденных буферов, хранящихся в ассоциативных списках, в соответствии с тем, насколько часто драйвер устройства или исполнительная подсистема выделяют память из списка — чем чаще осуществляется выделение, тем больше блоков хранится в списке. Ассоциативные списки автоматически уменьшаются в размере, если выделение из них не осуществляется. (Проверка проводится раз в секунду, когда происходит возобновление выполнения системного потока диспетчера настройки баланса и вызов функции `ExAdjustLookasideDepth`.)

ЭКСПЕРИМЕНТ: ПРОСМОТР СИСТЕМНЫХ АССОЦИАТИВНЫХ СПИСКОВ

Содержимое и размеры различных ассоциативных списков можно получить с помощью команды `!lookaside` отладчика ядра. Приведенная далее выборка является частью результата выполнения этой команды:

```
1kd> !lookaside

Lookaside "nt!IopSmallIrpLookasideList" @ 81f47c00 "Irps"
Type = 0000 NonPagedPool
Current Depth = 3 Max Depth = 4
Size = 148 Max Alloc = 592
AllocateMisses = 930 FreeMisses = 780
TotalAllocates = 13748 TotalFrees = 13601
Hit Rate = 93% Hit Rate = 94%

Lookaside "nt!IopLargeIrpLookasideList" @ 81f47c80 "Irpl"
Type = 0000 NonPagedPool
Current Depth = 4 Max Depth = 4
Size = 472 Max Alloc = 1888
AllocateMisses = 16555 FreeMisses = 15636
TotalAllocates = 59287 TotalFrees = 58372
Hit Rate = 72% Hit Rate = 73%

Lookaside "nt!IopMdlLookasideList" @ 81f47b80 "Md1 "
Type = 0000 NonPagedPool
Current Depth = 4 Max Depth = 4
Size = 96 Max Alloc = 384
AllocateMisses = 16287 FreeMisses = 15474
TotalAllocates = 72835 TotalFrees = 72026
Hit Rate = 77% Hit Rate = 78%
...

Total NonPaged currently allocated for above lists = 0
Total NonPaged potential for above lists = 3280
Total Paged currently allocated for above lists = 744
Total Paged potential for above lists = 1536
```

Диспетчер кучи

Большинство приложений выделяют память блоками меньшими по размеру, чем минимально возможная гранулярность выделения, составляющая 64 Кбайт при использовании таких функций страничной гранулярности, как `VirtualAlloc` и `VirtualAllocExNuma`. С точки зрения использования памяти и производительности выделение такой большой области при относительно небольших потребностях в памяти нельзя признать оптимальным. Для удовлетворения таких потребностей Windows предоставляет специальный компонент, который называется *диспетчером кучи* (heap manager) и управляет выделением памяти внутри больших областей памяти, зарезервированных с помощью функций, выделяющих память в соответствии со страничной гранулярностью. Гранулярность выделения в диспетчере кучи относительно невелика: 8 байт для 32-разрядных систем и 16 байт для 64-разрядных. Диспетчер кучи был разработан для оптимизации использования памяти и производительности при таких небольших объемах выделяемой памяти.

Диспетчер кучи находится в двух файлах: `Ntdll.dll` и `Ntoskrnl.exe`. API-функции подсистем (такие, как API-функции кучи в Windows) вызывают функции из файла `Ntdll.dll`, а различные компоненты исполнительной системы и драйверы устройств — из файла `Ntoskrnl.exe`. Исходные интерфейсы `Ntoskrnl` (с префиксами `Rt1`) доступны только для использования во внутренних Windows-компонентах или в драйверах устройств, работающих в режиме ядра. Документированные API-интерфейсы Windows для работы с кучей (с префиксами `Heap`) передают управление родным функциям, находящимся в файле `Ntdll.dll`. В дополнение к этому, для поддержки старых Windows-приложений предоставляются старые API-функции (с префиксом `Local` или `Global`), которые также приводят к внутреннему вызову диспетчера кучи с использованием для поддержки устаревших механизмов работы некоторых из его специализированных интерфейсов. С-среда времени выполнения (C RunTime, CRT) также задействует диспетчер кучи при использовании функций `malloc` и `free` и оператора `new` языка C++. Наиболее часто применяются следующие Windows-функции кучи:

- ❑ `HeapCreate` или `HeapDestroy` соответственно создает или удаляет кучу. Изначально зарезервированный и подтвержденный размер может быть указан при создании.
- ❑ `HeapAlloc` выделяет блок кучи.
- ❑ `HeapFree` освобождает блок, ранее выделенный с помощью функции `HeapAlloc`.
- ❑ `HeapReAlloc` изменяет размер существующего выделения (увеличивает или уменьшает размер существующего блока).
- ❑ `HeapLock` или `HeapUnlock` управляет взаимными исключениями при операциях с кучами.
- ❑ `HeapWalk` перечисляет записи и области в куче.

Типы куч

У каждого процесса имеется как минимум одна куча — куча процесса, предлагаемая по умолчанию. Эта куча создается при запуске процесса и не удаляется, пока сущ-

стывает процесс. Ее размер по умолчанию составляет 1 Мбайт, но ее можно сделать больше, указав начальный размер в файле образа с помощью флага /HEAP компоновщика. Однако этот размер является всего лишь начальным, и по мере необходимости он автоматически увеличивается. (В файле образа можно также указать изначально подтвержденный размер.)

Куча, предлагаемая по умолчанию, может быть явно использована программой или неявно какой-нибудь из внутренних Windows-функций. Приложение может послать запрос к куче процесса, предлагаемой по умолчанию, вызвав Windows-функцию `GetProcessHeap`. Процессы могут также создавать дополнительные закрытые кучи, используя для этого функцию `HeapCreate`. Когда закрытая куча становится процессу ненужной, он может вернуть виртуальное адресное пространство, вызвав функцию `HeapDestroy`. Каждым процессом обслуживается массив со сведениями обо всех кучах, и программный поток может запросить их с помощью Windows-функции `GetProcessHeaps`.

Управление выделениями памяти для кучи может осуществляться либо в больших областях памяти, зарезервированных из диспетчера памяти с помощью функции `VirtualAlloc`, либо из объектов отображаемых на память файлов в адресном пространстве процесса. Последний подход используется на практике довольно редко, но хорошо подходит для сценариев, согласно которым содержимое блоков должно совместно использоваться двумя процессами или компонентами, работающими в режиме ядра и в пользовательском режиме. Драйвер подсистемы Win32 GUI (`Win32k.sys`) задействует такую кучу для совместного использования объектов GDI и User с компонентами пользовательского режима. Если куча создана в верхней части области файла, отображаемого на память, то на компонент, способный вызывать функции кучи, накладываются определенные ограничения. Во-первых, в структурах внутренних куч применяются указатели, поэтому запрещаются перераспределения на другие адреса в других процессах. Во-вторых, функциями кучи не поддерживается синхронизация между несколькими процессами или между компонентом ядра и пользовательским процессом. Кроме того, в случае совместного использования кучи кодом пользовательского режима и кодом режима ядра отображение в пользовательском режиме должно быть только для чтения, чтобы не дать коду пользовательского режима возможности испортить внутренние структуры кучи, вызвав полный крах системы. Драйвер режима ядра также отвечает за то, чтобы конфиденциальные данные не попадали в общую кучу и не создавались условия их утечки в программу, выполняемую в пользовательском режиме.

Структура диспетчера кучи

Как показано на рис. 10.7, диспетчер кучи имеет двухуровневую структуру, состоящую из необязательного внешнего уровня и собственно кучи. На уровне кучи осуществляется управление базовой функциональностью, которая в основном распространяется на реализацию куч в пользовательском режиме и режиме ядра. Базовая функциональность включает в себя управление блоками внутри сегментов, управление сегментами, реализацию политики увеличения кучи, подтверждение и отмену подтверждения памяти, управление большими блоками.

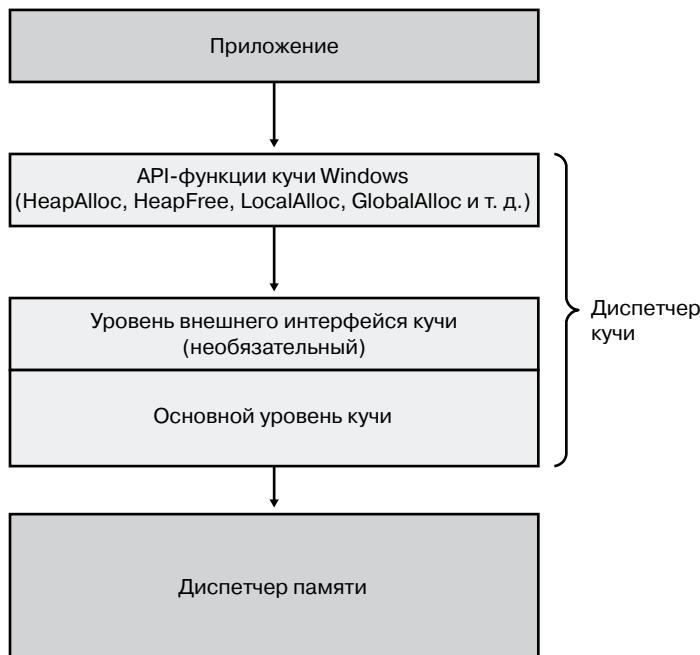


Рис. 10.7. Уровни диспетчера кучи

Необязательный внешний уровень, являющийся надстройкой над имеющейся базовой функциональностью, может существовать только для куч пользовательского режима. Единственным внешним уровнем, поддерживаемым в Windows, является слабо фрагментированная куча (Low Fragmentation Heap, LFH). Одновременно для одной кучи может использоваться только один внешний уровень.

Синхронизация кучи

Изначально диспетчер кучи поддерживает параллельное обращение из нескольких программных потоков. Но если процесс является однопоточным или использует внешний механизм для синхронизации, он может сообщить диспетчеру кучи о необходимости отказаться от синхронизации, установив флаг `HEAP_NO_SERIALIZE` либо при создании кучи, либо при каждом выделении памяти.

Процесс также может заблокировать всю кучу, не давая другим потокам выполнять те операции с кучей, которые требуют согласованных состояний между несколькими обращениями к куче. Например, если несколько потоков могут одновременно выполнять операции с кучей, то подсчет блоков кучи с помощью Windows-функции `HeapWalk` требует блокировки.

Если режим синхронизации кучи включен, то все внутренние структуры кучи защищаются в рамках одной блокировки на каждую кучу. В приложениях, интенсивно использующих множество программных потоков (особенно при их выполнении на мультипроцессорных системах), блокировка кучи может привести к серьезной кон-

фликтной ситуации. В таком случае производительность может быть повышена за счет включения внешнего уровня кучи, рассматриваемого в одном из следующих разделов.

Слабо фрагментированная куча

Многие выполняемые в Windows приложения используют относительно небольшой объем памяти кучи (обычно менее 1 Мбайт). Для этого класса приложений сохранять малый объем памяти для каждого процесса помогает оптимальная политика диспетчера кучи. Но эта стратегия для больших процессов и мультипроцессорных машин не масштабируется. В таких случаях память, доступная для использования в куче, может быть уменьшена в результате фрагментации кучи. В сценариях, где параллельно выполняются разные потоки на разных процессорах, производительность может снижаться. Причина в том, что нескольким процессорам одновременно нужно модифицировать один и тот же участок памяти (например, начало ассоциативного списка для блока конкретного размера), создавая тем самым большую конкуренцию при доступе к соответствующей строке кэша.

Технология слабо фрагментированной кучи (LFH) позволяет избегать фрагментации путем управления выделенными блоками в заранее заданных диапазонах размеров блоков, которые называются *корзинами* (buckets). Когда процесс выделяет память из кучи, LFH предлагает корзину, отображаемую на наименьший блок, подходящий под требуемый размер. (Наименьший блок имеет размер 8 байт.) Первая корзина служит для выделения в диапазоне от 1 до 8 байт, вторая — в диапазоне от 9 до 16 байт и т. д., вплоть до тридцати второй корзины, предназначеннной для выделения в диапазоне от 249 до 256 байт, за которой следует тридцать третья корзина, служащая для выделения в диапазоне от 257 до 272 байт и т. д. И наконец, очередь доходит до сто двадцать восьмой корзины, которая в итоге используется для выделения в диапазоне от 15 873 до 16 384 байт. Все это известно как система *двоичного дружелюбия* (binary buddy). Сводка по различным корзинам, их гранулярности и диапазонам размеров, на которые они отображаются, представлена в табл. 10.7.

Таблица 10.7. Корзины

Корзины	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	4097–8194
113–128	512	8195–16384

В LFH эти проблемы решаются путем использования диспетчера основной кучи и ассоциативных списков. Имеющийся в Windows диспетчер кучи реализует алгоритм автоматической настройки, который может по умолчанию подключить LFH

при определенных условиях, таких как конфликты блокировки или наличие широко распространенных размеров выделения памяти, при работе с которыми подключение LFH способствует более высокой производительности системы. Для больших куч существенный процент операций выделения памяти часто касается относительно небольшого количества корзин определенных размеров. Стратегия выделения памяти, принятая в LFH, заключается в оптимизации использования таких моделей путем эффективной работы с блоками одинакового размера.

В целях обеспечения масштабируемости LFH расширяет часто используемые внутренние структуры на ряд слотов, количество которых вдвое превышает текущее количество процессоров, имеющихся на машине. Назначение потоков этим слотам осуществляется LFH-компонентом, который называется *диспетчером родственности* (affinity manager). Изначально LFH задействует для выделения кучи первый слот, но если при обращении к каким-либо внутренним данным обнаруживается конфликтная ситуация, LFH переключает текущий программный поток на другой слот. Последующие конфликтные ситуации ведут к «расползанию» потоков на дополнительные слоты. Для лучшей локальности и минимизации общего потребления памяти управление этими слотами осуществляется для корзин каждого размера.

Даже если слабо фрагментированная куча подключена к куче в качестве внешнего уровня, в случаях наименее востребованных размеров для выделения памяти могут по-прежнему использоваться соответствующие функции основной кучи, а для наиболее востребованных размеров выделение памяти будет осуществляться из LFH. Отключить LFH можно с помощью API-функции `HeapSetInformation` с классом `HeapCompatibilityInformation`.

Механизмы безопасности куч

По мере совершенствования диспетчера кучи повышалось значение раннего обнаружения ошибок при использовании кучи и смягчения последствий потенциальных уязвимостей куч. Предпринимаемые меры направлены на снижение угрозы безопасности из-за возможных слабых мест приложений. Метаданные, используемые в куче для внешнего управления, упаковываются в значительной степени случайно, чтобы усложнить задачу вредоносному коду, пытающемуся внести изменения во внутренние структуры и предотвратить отказы или скрыть следы атаки. Заголовки блоков также контролируются механизмом проверки целостности, чтобы можно было обнаружить простые повреждения, например выход за пределы размеров буфера. И наконец, в куче имеет место определенная рандомизация базового адреса (или идентификатора). Используя API-функцию `HeapSetInformation` с классом `HeapEnableTerminationOnCorruption`, процессы могут автоматически завершаться в случае обнаружения противоречивых ситуаций, избегая выполнения неизвестного кода.

Из-за рандомизации метаданных блока применение отладчика для получения простого дампа заголовка блока особой пользы не приносит. Например, узнать из обычного дампа, каков размер блока и занят ли он, не так-то просто. То же самое касается и LFH-блоков. В заголовке таких блоков хранится другой тип метаданных, которые также частично рандомизированы. Для получения дампа с этими деталями в отладчике служит команда `!heap -i`, обеспечивающая извлечение из блока полей метаданных,

нахождение контрольной суммы или свободного списка несоответствий, если таковые есть. Эта команда подходит как для LFH-блоков, так и для обычных блоков кучи. Как показано в следующем примере, в результатах, полученных с помощью этой команды, присутствуют общий размер блоков, размер, запрошенный кодом пользовательского режима, сегмент, которому принадлежит блок, а также частично контрольная сумма заголовка. Поскольку в алгоритме рандомизации учитывается гранулярность кучи, команда `!heap -i` должна использоваться только в нужном контексте той кучи, в которой содержится блок. В приведенном примере дескриптор кучи находится по адресу `0x001a0000`. Если текущий контекст кучи был другим, заголовок декодируется неправильно. Для установки правильного контекста сначала нужно выполнить команду `!heap -i` с адресом описателя кучи в качестве аргумента:

```
0:000> !heap -i 001a0000
Heap context set to the heap 0x001a0000
0:000> !heap -i 1e2570
Detailed information for block entry 001e2570
Assumed heap      : 0x001a0000 (Use !heap -i NewHeapHandle to change)
Header content    : 0x1570F4EC 0x0C0015BE (decoded : 0x07010006 0x0C00000D)
Owning segment    : 0x001a0000 (offset 0)
Block flags       : 0x1 (busy )
Total block size  : 0x6 units (0x30 bytes)
Requested size    : 0x24 bytes (unused 0xc bytes)
Previous block size: 0xd units (0x68 bytes)
Block CRC         : OK - 0x7
Previous block    : 0x001e2508
Next block        : 0x001e25a0
```

Средства отладки куч

Диспетчер кучи задействует 8 байт для хранения внутренних метаданных в качестве последовательности контрольных точек, что делает более очевидными потенциальные ошибки использования кучи, а также включает ряд инструментов, помогающих обнаруживать ошибки с помощью следующих функций кучи:

- ❑ **Включение режима проверки окончания блока.** В конце каждого блока находится сигнатура, проверяемая при освобождении блока. Если при переполнении буфера сигнатура будет частично или полностью повреждена, куча сообщит об этой ошибке.
- ❑ **Включение режима проверки свободных блоков.** Свободный блок заполняется некоторыми эталонными данными, проверяемыми в разных точках, когда диспетчеру кучи нужно обратиться к блоку (например, при удалении из списка свободных блоков для удовлетворения запроса на выделение памяти). Если процесс продолжает вести запись в блок после его освобождения, диспетчер кучи обнаружит изменения в эталоне и сообщит об ошибке.
- ❑ **Проверка параметров.** Эта функция подразумевает всестороннюю проверку параметров, передаваемых функциям кучи.
- ❑ **Проверка кучи.** Куча проверяется при каждом обращении к ней.

- **Поддержка маркирования тегами кучи и трассировки стека.** Эта функция поддерживает определенные признаки (теги) выделения памяти и (или) осуществляет сбор данных трассировки стека пользовательского режима при обращениях к куче для сужения диапазона возможных причин ошибок кучи.

Первые три функции включаются по умолчанию, если загрузчик обнаруживает, что процесс запускается под управлением отладчика. (Отладчик может изменить ситуацию и отключить эти функции.) Функции отладки кучи могут указываться для исполняемого образа путем установки различных отладочных флагов в заголовке образа с помощью программы Gflags. (См. раздел «Глобальные флаги Windows» в главе 3 части I.) Отладочные параметры кучи можно также включить с помощью команды `!heap` в стандартных отладчиках Windows. (Дополнительные сведения можно найти в справочной системе отладчика.)

Включение отладочных параметров кучи влияет на все кучи процесса. Кроме того, при включении какого-нибудь отладочного параметра кучи LFH отключается автоматически, в результате используется только основная куча (с включением требуемых отладочных параметров). Кроме того, LFH не применяется для нерасширяемых куч (из-за дополнительных служебных данных, добавляемых к существующим структурам кучи) и куч, не поддерживающих сериализацию.

Инструмент pageheap

Поскольку настройка на проверку окончаний блоков и свободных блоков, о чём рассказывалось в предыдущем разделе, может приводить к обнаружению повреждений, произошедших задолго до выявления проблемы, предоставляется дополнительное средство отладки куч, которое называется *pageheap*. При подключении этого инструмента все или часть обращений к куче направляются другому диспетчеру кучи. Инструмент pageheap подключается с помощью программы Gflags (которая является частью инструментария Debugging Tools for Windows). Когда инструмент pageheap подключен, диспетчер кучи помещает выделенную память в конец страницы, а зарезервированную память — сразу же за страницей. Поскольку зарезервированные страницы недоступны, переполнение буфера вызывает нарушение прав доступа, что облегчает обнаружение проблемного кода. В дополнение pageheap позволяет помещать блоки в начале страниц с резервированием предыдущей страницы — это даёт возможность выявлять проблемы неполного использования буфера (что случается крайне редко). Кроме того, pageheap позволяет защитить освобожденные страницы от любых обращений, обнаруживая ссылки на блоки кучи после их освобождения.

Следует учесть, что применение инструмента pageheap может привести к нехватке адресного пространства из-за существенных издержек на дополнительную память даже при выделении совсем маленьких блоков памяти. Кроме того, в результате увеличения количества ссылок на обнуленные страницы, потери локальности и дополнительных издержек из-за частых проверок структур кучи может пострадать производительность. Процесс может уменьшить негативные последствия, указав, что инструмент pageheap должен применяться только к блокам определенных размеров, диапазонам адресов и (или) исходным DLL-библиотекам.

Дополнительные сведения о pageheap можно найти в справочном файле Debugging Tools for Windows Help.

Отказоустойчивая куча

Как считают в Microsoft, одной из наиболее частых причин сбоев приложений является повреждение метаданных кучи. В Windows имеется средство, которое называется *отказоустойчивой кучей* (Fault Tolerant Heap, FTH). С помощью FTH делается попытка смягчить существующие проблемы и предоставить разработчикам приложений более эффективные ресурсы разрешения проблем. Отказоустойчивая куча реализуется с помощью двух основных компонентов: компонента *обнаружения* (detection), или FTH-сервера, и компонента *смягчения* (mitigation), или FTH-клиента.

Компонент обнаружения является DLL-библиотекой (`Fthsvc.dll`), загружаемой службой Центра обеспечения безопасности Windows (`Wscsvc.dll`), которая, в свою очередь, запускается в одном из общих процессов служб под управлением учетной записи локальной службы. Этот компонент уведомляется о сбоях приложения службой регистрации ошибок (Windows Error Reporting).

Когда сбой приложения происходит в `Ntdll.dll`, а статус ошибки свидетельствует либо о нарушении прав доступа, либо об исключении, связанном с повреждением кучи, то если приложение еще не входит в список «наблюдаемых» приложений FTH-службы, она создает «ярлык» для приложения с целью хранения FTH-данных. Если в приложении за час происходит более четырех сбоев подряд, FTH-служба настраивает приложение на использование будущем FTH-клиента.

FTH-клиент является для приложения своеобразной «прокладкой» обеспечения совместимости». Этот механизм использовался начиная с Windows XP, чтобы разрешить приложениям, которые зависят от специфического поведения прежних систем Windows, выполняться на новых системах. В данном случае прокладка перехватывает вызовы к процедурам кучи и направляет их в собственный код. В FTH-коде реализован ряд «смягчений», с помощью которых делается попытка помочь приложению сохранить работоспособность, несмотря на разнообразные ошибки, связанные с применением куч.

Например, для защиты от ошибок небольшого переполнения буфера FTH вводит 8-байтовое дополнение и резервирует область для каждой операции выделения памяти. Чтобы отвечать самому распространенному сценарию, при котором к блоку кучи происходит обращение после его освобождения, вызовы `HeapFree` выполняются только после задержки: «освобожденные» блоки помещаются в список и реально освобождаются, только когда общий размер блоков в списке превысит 4 Мбайт. Попытки освободить области, не являющиеся ни частью «настоящей» кучи, ни частью кучи, заданной аргументом дескриптора кучи как `HeapFree`, просто игнорируются. Кроме того, при вызове функции `exit` или `RtlExitUserProcess` блоки не освобождаются.

FTH-сервер продолжает отслеживать показатель сбоев приложения и после ввода смягчений. Если показатель сбоев не растет, смягчения удаляются.

Работу отказоустойчивой кучи можно отследить с помощью программы просмотра событий. Наберите в диалоговом окне `Run` (Выполнить) команду `eventvwr.msc`, а затем переместите указатель мыши на левую панель программы просмотра событий и поочередно выберите узел `Applications And Services Logs` (Журналы приложений и служб) ▶ `Microsoft` ▶ `Windows` ▶ `Fault-Tolerant-Heap` ▶ `Operational log` (Работает). Эту службу можно полностью отключить в реестре: для параметра `Enabled` в разделе `HKEY_LOCAL_MACHINE\Software\Microsoft\FTH` нужно установить значение 0.

Обычно FTH-служба не используется со службами, а только с приложениями, кроме того, она отключается на Windows-серверах, чтобы не снижать производительность. Системный администратор может сам установить прокладку в исполняемый файл приложения или службы, воспользовавшись инструментарием Application Compatibility Toolkit.

Структуры виртуального адресного пространства

В данном разделе сначала рассматриваются компоненты пользовательского и системного адресных пространств, а затем — конкретные структуры данных 32- и 64-разрядных систем. Эта информация поможет вам разобраться с ограничениями, накладываемыми 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, код и данные, принадлежащие всему сеансу работы, а также код и данные, принадлежащие всей системе.

В главе 1 части I 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. Поэтому 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 процесс и поэтому доступны каждому такому процессу по его адресам. Аналогичным образом функции памяти, работающие для нескольких процессов (`ReadProcessMemory` и `WriteProcessMemory`), действуют за счет запуска кода режима ядра в контексте целиового процесса.

Информация, описывающая виртуальное адресное пространство процесса и называемая *таблицами страниц* (page tables), рассматривается в разделе, посвященном преобразованию адресов. У каждого процесса есть 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.

Пространство сеанса (session space) содержит информацию, общую для каждого сеанса. (Сеансы рассматриваются в главе 2 части I.) Сеанс состоит из процессов и других системных объектов (таких, как рабочие столы и окна), представляющих сеанс входа в систему отдельного пользователя. У каждого сеанса есть относящаяся только к нему область выгружаемого пула, используемая той частью подсистемы Windows, которая работает в режиме 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 предполагает загрузку принадлежащей сеансу отдельной

копии Win32k.sys, создание пространства имен принадлежащего сеансу диспетчера объектов и создание принадлежащих сеансу экземпляров процессов Csrss и Winlogon. Для виртуализации сеансов все структуры данных, относящиеся ко всему сеансу, 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.

И наконец, системное пространство содержит глобальный код операционной системы и структуры данных, которые видны коду, выполняемому в режиме ядра, причем 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** содержит невыгружаемую кучу системной памяти.
- ❑ **Выгружаемый пул** содержит выгружаемую кучу 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 используемых этой подсистемой графических драйверов, работающих в режиме ядра. (Дополнительные сведения о Win32k.sys можно найти в главе 2 части I.)
- ❑ **Гиперпространство** — это специальная область, используемая 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 послужить аннулирование записей таблицы страниц в таблицах страниц процесса, не являющегося текущим (например, при удалении страницы из списка ожидания).
- ❑ **Данные аварийного дампа** — пространство, зарезервированное для записи информации о состоянии системы на момент ее отказа.
- ❑ **Область, используемая HAL**, — системная память, зарезервированная для структур, относящихся к уровню аппаратных абстракций (HAL).

А теперь, после описания основных компонентов имеющегося в Windows виртуального адресного пространства, давайте перейдем к рассмотрению конкретных структур на платформах x86, IA64 и x64.

Структура адресных пространств на платформе x86

По умолчанию каждый пользовательский процесс на 32-разрядной версии Windows имеет закрытое адресное пространство размером 2 Гбайт, а операционная система забирает оставшиеся 2 Гбайт. Однако система с помощью загрузочного BCD-параметра `increaseusersva` может быть настроена на размер пользовательского адресного пространства вплоть до 3 Гбайт. На рис. 10.8 показаны два возможных варианта структур адресного пространства.

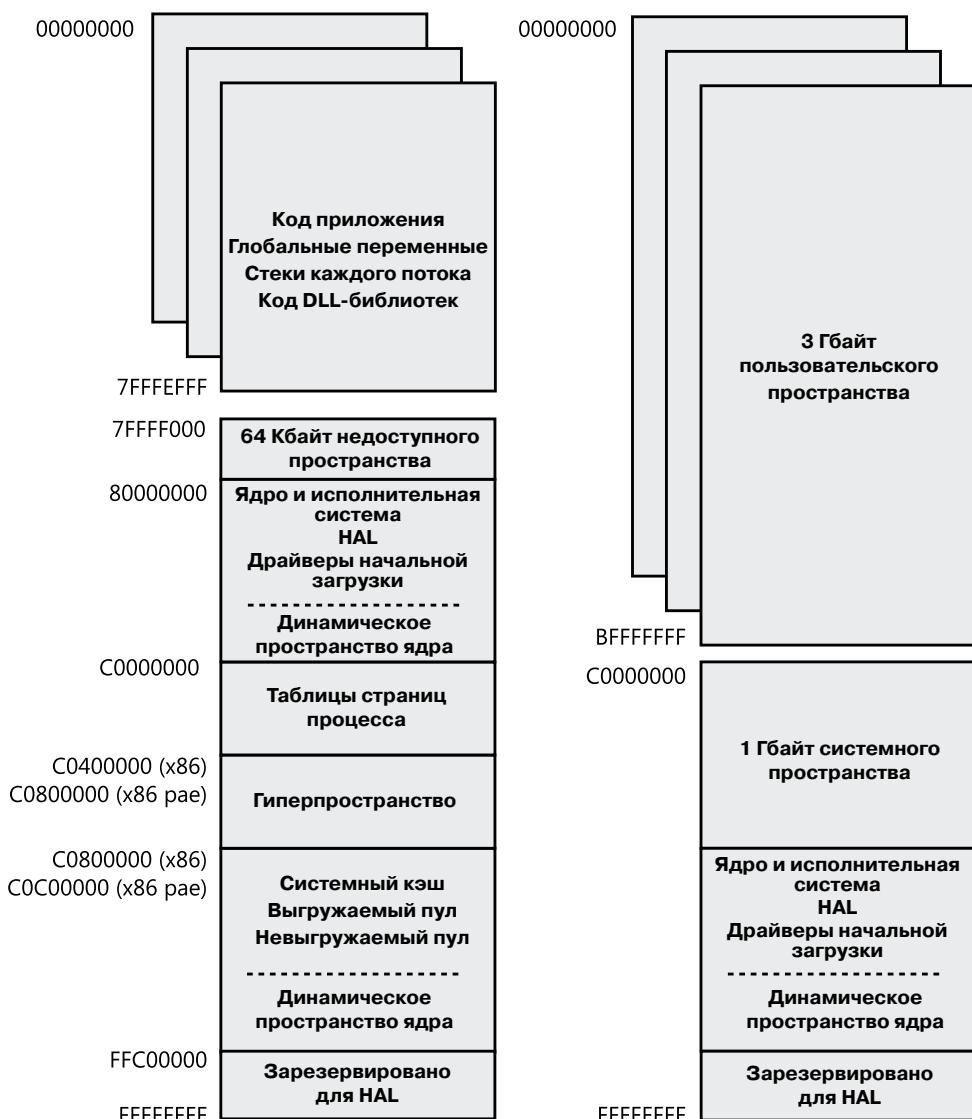


Рис. 10.8. Структура виртуальных адресных пространств на платформе x86

Возможность разрастания 32-разрядного процесса за пределы 2 Гбайт была введена с целью 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`. 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 находится в сброшенном состоянии, такие приложения будут использовать этот старший разряд в указателях как флаг для своих данных, конечно же, сбрасывая его перед ссылкой на данные. При работе в адресном пространстве размером 3 Гбайт они будут непреднамеренно урезать указатели, имеющие значения, превышающие 2 Гбайт, вызывая тем самым ошибки приложения, включая и возможное повреждение данных. Вы можете установить этот флаг при создании исполняемого файла, указав ключ компоновщика `/LARGEADDRESSAWARE`. При запуске приложения на системе с пользовательским адресным пространством размером 2 Гбайт этот флаг игнорируется.

Воспользоваться преимуществами систем, запущенных с большим адресным пространством процесса, может ряд системных образов, промаркованных как осведомленные о большом адресном пространстве:

- `Lsass.exe` — подсистема проверки подлинности локальной системы безопасности;
- `Inetinfo.exe` — информационный интернет-сервер;
- `Chkdsk.exe` — утилита проверки диска;
- `Smss.exe` — диспетчер сеансов;
- `Dllhst3g.exe` — специальная версия `Dllhost.exe` (для приложений, поддерживающих технологию COM+);
- `Dispdiag.exe` — утилита дампа диагностики дисплея;
- `Esentutl.exe` — программа Active Directory Database Utility.

И 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 у него не будет слишком фрагментировано виртуальное адресное пространство, 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` или добавив DWORD-параметр реестра в раздел `HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\AllocationPreference` и установив для него значение `0x100000`.

На рис. 10.9 показаны две копии экрана с результатами работы утилиты TestLimit (рассмотренной в предыдущих экспериментах), реализующей утечку памяти на 32-разрядной машине с Windows, загруженной с установленным и неустановленным на использование памяти в 3 Гбайт значением параметра increaseuserva.

```
C:\temp>testlimit -r
Testlimit v5.1 - test Windows limits
Copyright (C) 2012 Mark Russinovich
Sysinternals - www.sysinternals.com

Reserving private bytes (MB)... .
Leaked 2016 MB of reserved memory (2016 MB total leaked). Lasterror: 8
Not enough storage is available to process this command.

C:\temp>testlimit -r
Testlimit v5.1 - test Windows limits
Copyright (C) 2012 Mark Russinovich
Sysinternals - www.sysinternals.com

Reserving private bytes (MB)... .
Leaked 3038 MB of reserved memory (3038 MB total leaked). Lasterror: 8
Not enough storage is available to process this command.
```

Рис. 10.9. Утечка памяти, реализованная утилитой TestLimit на 32-разрядном компьютере с Windows, загруженным с установленным и неустановленным на использование памяти в 3 Гбайт значением параметра increaseuserva

Обратите внимание на то, что на второй копии экрана утилита TestLimit смогла организовать, как и ожидалось, утечку памяти, равную почти 3 Гбайт. Это стало возможным только потому, что утилита TestLimit была скомпонована с ключом /LARGEADDRESSAWARE. Если бы этого сделано не было, результат был бы точно таким же, как и на системе, загруженной без установки соответствующего значения параметра increaseuserva.

ЭКСПЕРИМЕНТ: ПРОВЕРКА ОСВЕДОМЛЕННОСТИ ПРИЛОЖЕНИЯ О БОЛЬШОМ АДРЕСНОМ ПРОСТРАНСТВЕ

Чтобы выяснить, поддерживают ли другие приложения большие адресные пространства, можно воспользоваться утилитой Dumpbin из инструментария Windows SDK. Для вывода результатов нужно указать ключ /HEADERS. Пример выводимых утилитой Dumpbin данных для диспетчера сеансов выглядит следующим образом:

```
C:\Program Files\Microsoft SDKs\Windows\v7.1>dumpbin
=headers c:\windows\system32\smss.exe
Microsoft (R) COFF/PE Dumper Version 10.00.40219.01
Copyright (C) Microsoft Corporation. All rights reserved.

Dump of file c:\windows\system32\smss.exe
PE signature found
```

продолжение ↴

```

File Type: EXECUTABLE IMAGE

FILE HEADER VALUES
8664 machine (x64)
5 number of sections
4A5BC116 time date stamp Mon Jul 13 16:19:50 2009
0 file pointer to symbol table
0 number of symbols
F0 size of optional header
22 characteristics
Executable
Application can handle large (>2GB) addresses

```

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. 10.8, хотя есть еще несколько специально зарезервированных областей, для многих структур режима ядра адресное пространство выделяется динамически. Поэтому эти структуры не обязательно виртуально непрерывны. 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;
- особый пул;
- выгружаемый пул;
- записи системной таблицы страниц (PTE-записи);
- системные отображаемые представления;
- кэш файловой системы;
- структуры файловой системы (метаданные);
- пространство сеанса.

Пространство сеанса на платформе x86

Для систем с несколькими сессиями 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. 10.10.

Размеры компонентов пространства сеанса, как и всего остального системного адресного пространства ядра, конфигурируются в динамическом режиме, а их размеры меняются диспетчером памяти по запросу.



Рис. 10.10. Структура пространства сеанса на платформе x86
(без соблюдения пропорций)

ЭКСПЕРИМЕНТ: ПРОСМОТР СЕАНСОВ

О принадлежности процессов тому или иному сеансу можно судить по идентификатору сеанса. Чтобы увидеть нужную информацию, используются такие средства, как диспетчер задач, Process Explorer или отладчик ядра. В отладчике ядра можно с помощью команды !session вывести список активных сеансов:

```
1kd> !session
Sessions on machine: 3
Valid Sessions: 0 1 3
Current Session 1
```

Затем можно настроиться на активный сеанс, используя команду !session -s, и вывести список структур данных сеанса и процессов в сеансе с помощью команды !sprocess:

```
1kd> !session -s 3
Sessions on machine: 3
Implicit process is now 84173500
Using session 3
```

```
1kd> !sprocess
Dumping Session 3
```

```
_MM_SESSION_SPACE 9a83c000
_MMSESSION 9a83cd00
PROCESS 84173500 SessionId: 3 Cid: 0d78 Peb: 7ffde000 ParentCid: 0e80
DirBase: 3ef53500 ObjectTable: 8588d820 HandleCount: 76.
Image: csrss.exe
```

```
PROCESS 841a6030 SessionId: 3 Cid: 0c6c Peb: 7ffdcc000 ParentCid: 0e80
```

продолжение ↗

```
DirBase: 3ef53520 ObjectTable: 85897208 HandleCount: 94.
Image: winlogon.exe

PROCESS 841d9cf0 SessionId: 3 Cid: 0d38 Peb: 7ffd6000 ParentCid: 0c6c
DirBase: 3ef53540 ObjectTable: 8589d248 HandleCount: 165.
Image: LogonUI.exe
...
```

Для просмотра деталей сеанса нужно вывести дамп структуры MM_SESSION_SPACE с помощью команды dt:

```
1kd> dt nt!_MM_SESSION_SPACE 9a83c000
+0x000 ReferenceCount : 0n3
+0x004 u : <unnamed-tag>
+0x008 SessionId : 3
+0x00c ProcessReferenceToSession : 0n4
+0x010 ProcessList : _LIST_ENTRY [ 0x841735e4 - 0x841d9dd4 ]
+0x018 LastProcessSwappedOutTime : _LARGE_INTEGER 0x0
+0x020 SessionPageDirectoryIndex : 0x31fa3
+0x024 NonPagablePages : 0x19
+0x028 CommittedPages : 0x867
+0x02c PagedPoolStart : 0x80000000 Void
+0x030 PagedPoolEnd : 0xffffbfffff Void
+0x034 SessionObject : 0x854e2040 Void
+0x038 SessionObjectHandle : 0x80000020c Void
+0x03c ResidentProcessCount : 0n3
+0x040 SessionPoolAllocationFailures : [4] 0
+0x050 ImageList : _LIST_ENTRY [ 0x8519bef8 - 0x85296370 ]
+0x058 LocaleId : 0x409
+0x05c AttachCount : 0
+0x060 AttachGate : _KGATE
+0x070 WsListEntry : _LIST_ENTRY [ 0x82772408 - 0x97044070 ]
+0x080 Lookaside : [25] _GENERAL_LOOKASIDE
...
```

ЭКСПЕРИМЕНТ: ПРОСМОТР СТЕПЕНИ ИСПОЛЬЗОВАНИЯ ПРОСТРАНСТВА СЕАНСА

Степень использования пространства памяти сеанса можно просмотреть с помощью команды !vm 4 отладчика ядра. Например, показанные далее данные были получены на 32-разрядной клиентской системе Windows с созданными по умолчанию при запуске системы двумя сеансами:

```
1kd> !vm 4
.
.
Terminal Server Memory Usage By Session:

Session ID 0 @ 9a8c7000:
Paged Pool Usage: 2372K
Commit Usage: 4832K

Session ID 1 @ 9a881000:
Paged Pool Usage: 14120K
Commit Usage: 16704K
```

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. 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. На 64-разрядных версиях Windows системные PTE-записи позволяют описать до 128 Гбайт последовательного системного виртуального адресного пространства.

ЭКСПЕРИМЕНТ: ПРОСМОТР ИНФОРМАЦИИ О СИСТЕМНЫХ РТЕ-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, связанной с глобальной переменной MiSystemPteInfo. 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.

```
0: kd> !sysptes

System PTE Information
Total System Ptes 307168

starting PTE: c0200000

free blocks: 32 total free: 3856 largest free block: 542
Kernel Stack PTE Information
Unable to get syspte index array - skipping bins

starting PTE: c0200000
free blocks: 165 total free: 1503 largest free block: 75

0: kd> ? nt!MiSystemPteInfo
Evaluate expression: -2100014016 = 82d45440

0: kd> dt _MI_SYSTEM_PTE_TYPE 82d45440
nt!_MI_SYSTEM_PTE_TYPE
+0x000 Bitmap : _RTL_BITMAP
+0x008 Flags : 3
+0x00c Hint : 0x2271f
+0x010 BasePte : 0xc0200000 _MMPTPE
+0x014 FailureCount : 0x82d45468 -> 0
+0x018 Vm : 0x82d67300 _MMSUPPORT
+0x01c TotalSystemPtes : 0n7136
+0x020 TotalFreeSystemPtes : 0n4113
+0x024 CachedPteCount : 0n0
+0x028 PteFailures : 0
+0x02c SpinLock : 0
+0x02c GlobalMutex : (null)
```

продолжение ↗

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:

```
1kd>!sysptes 4
0x1ca2 System PTEs allocated to mapping locked pages

VA MDL PageCount Caller/CallersCaller
ecbfdee8 f0ed0958 2
netbt!DispatchIoctl+0x56a/netbt!NbtDispatchDevCtrl+0xcd
f0a8d050 f0ed0510 1
netbt!DispatchIoctl+0x64e/netbt!NbtDispatchDevCtrl+0xcd
ecef5000 1 20 nt!MiFindContiguousMemory+0xcd
ed447000 0 2
Ntfs!NtfsInitializeVcb+0x30e/Ntfs!NtfsInitializeDevice+0x95
ee1ce000 0 2
Ntfs!NtfsInitializeVcb+0x30e/Ntfs!NtfsInitializeDevice+0x95
ed9c4000 1 ca nt!MiFindContiguousMemory+0x63
eda8e000 1 ca nt!MiFindContiguousMemory+0x63
efb23d68 f8067888 2 mrxsmb!BowserMapUsersBuffer+0x28
efac5af4 f8b15b98 2 ndisui0!NdisuioRead+0x54/nt!NtReadFile+0x566
f0ac688c f848ff88 1 ndisui0!NdisuioRead+0x54/nt!NtReadFile+0x566
efac7b7c f82fc2a8 2 ndisui0!NdisuioRead+0x54/nt!NtReadFile+0x566
ee4d1000 1 38 nt!MiFindContiguousMemory+0x63
efa4f000 0 2
Ntfs!NtfsInitializeVcb+0x30e/Ntfs!NtfsInitializeDevice+0x95
efa53000 0 2
Ntfs!NtfsInitializeVcb+0x30e/Ntfs!NtfsInitializeDevice+0x95
eea89000 0 1
TDI!D11Initialize+0x4f/nt!MiResolveImageReferences+0x4bc
ee798000 1 20 VIDEOPR!pVideoPortGetDeviceBase+0x1f1
f0676000 1 10
hal!HalpGrowMapBuffers+0x134/hal!HalpAllocateAdapterEx+0x1ff

f0b75000 1 1 cpqasm2+0x2af67/cpqasm2+0x7847
f0afa000 1 1 cpqasm2+0x2af67/cpqasm2+0x6d82
```

Структура адресных пространств 64-разрядных систем

Теоретически 64-разрядное виртуальное адресное пространство составляет 16 экзабайтов (18 446 744 073 709 551 616 байтов, или приблизительно 18,44 миллиарда миллиардов байтов). В отличие от систем на платформе x86, где исходное адресное пространство поделено на две части (для процессов и для системы), 64-разрядное адресное пространство поделено на ряд областей разного размера, чьи компоненты концептуально соответствуют частям пользовательского, системного или сеансового пространства. Различные размеры этих областей, перечисленные в табл. 10.8, отражают текущую реализацию ограничений, которые могут быть беспрепятственно расширены в будущих выпусках операционной системы. Несомненно, 64 разряда — это огромный скачок в размерах адресного пространства.

Таблица 10.8. Размеры адресных пространств 64-разрядных систем

Область	IA64	x64
Адресное пространство процесса	7152 Гбайт	8192 Гбайт
Пространство системных РТЕ-записей	128 Гбайт	128 Гбайт
Системный кэш	1 Тбайт	1 Тбайт
Выгружаемый пул	128 Гбайт	128 Гбайт
Невыгружаемый пул	75 % физической памяти	75 % физической памяти

Также на 64-разрядных версиях Windows есть еще одна полезная особенность, касающаяся образов, осведомленных о большом адресном пространстве. Эта особенность заключается в том, что будучи запущенным на 64-разрядной версии Windows (Wow64), такой образ фактически получает все 4 Гбайт доступного пользовательского адресного пространства — в конечном счете, если образ может поддерживать указатели для памяти размером 3 Гбайт, то указатели для памяти размером 4 Гбайт не должны иметь какой-то другой вид, потому что в отличие от переключения с 2 на 3 Гбайт для них не должен задействоваться дополнительный разряд. На рис. 10.11 показано окно программы TestLimit, запущенной как 32-разрядное приложение и резервирующей адресное пространство на 64-разрядной машине с Windows, а затем показана 64-разрядная версия TestLimit, реализующая утечку памяти на той же самой машине.

```
C:\temp>testlimit -r
Testlimit v5.1 - test Windows limits
Copyright (C) 2012 Mark Russinovich
Sysinternals - www.sysinternals.com

Reserving private bytes (MB)...
Leaked 4031 MB of reserved memory (4031 MB total leaked). Lasterror: 8
Not enough storage is available to process this command.

C:\temp>testlimit64 -r
Testlimit v5.1 - test Windows limits
Copyright (C) 2012 Mark Russinovich
Sysinternals - www.sysinternals.com

Reserving private bytes (MB)...
Leaked 8388548 MB of reserved memory (8388548 MB total leaked). Lasterror: 8
Not enough storage is available to process this command.
```

Рис. 10.11. 32- и 64-разрядная версии TestLimit, резервирующие адресное пространство на 64-разрядном компьютере под управлением Windows

Следует заметить, что эти результаты зависят от применения двух версий TestLimit, скомпонованных с ключом /LARGEADDRESSAWARE. Если они не будут скомпонованы с этим ключом, для обеих версий результаты составят около 2 Гбайт. 64-разрядные приложения, скомпонованные без ключа /LARGEADDRESSAWARE, ограничены первыми двумя гигабайтами виртуального адресного пространства процесса, точно так же, как и 32-разрядные приложения.

При подробном рассмотрении структуры адресных пространств для платформ IA64 и x64 немного различаются. Структура адресного пространства для платформы IA64 показана на рис. 10.12, а для платформы x64 — на рис. 10.13.

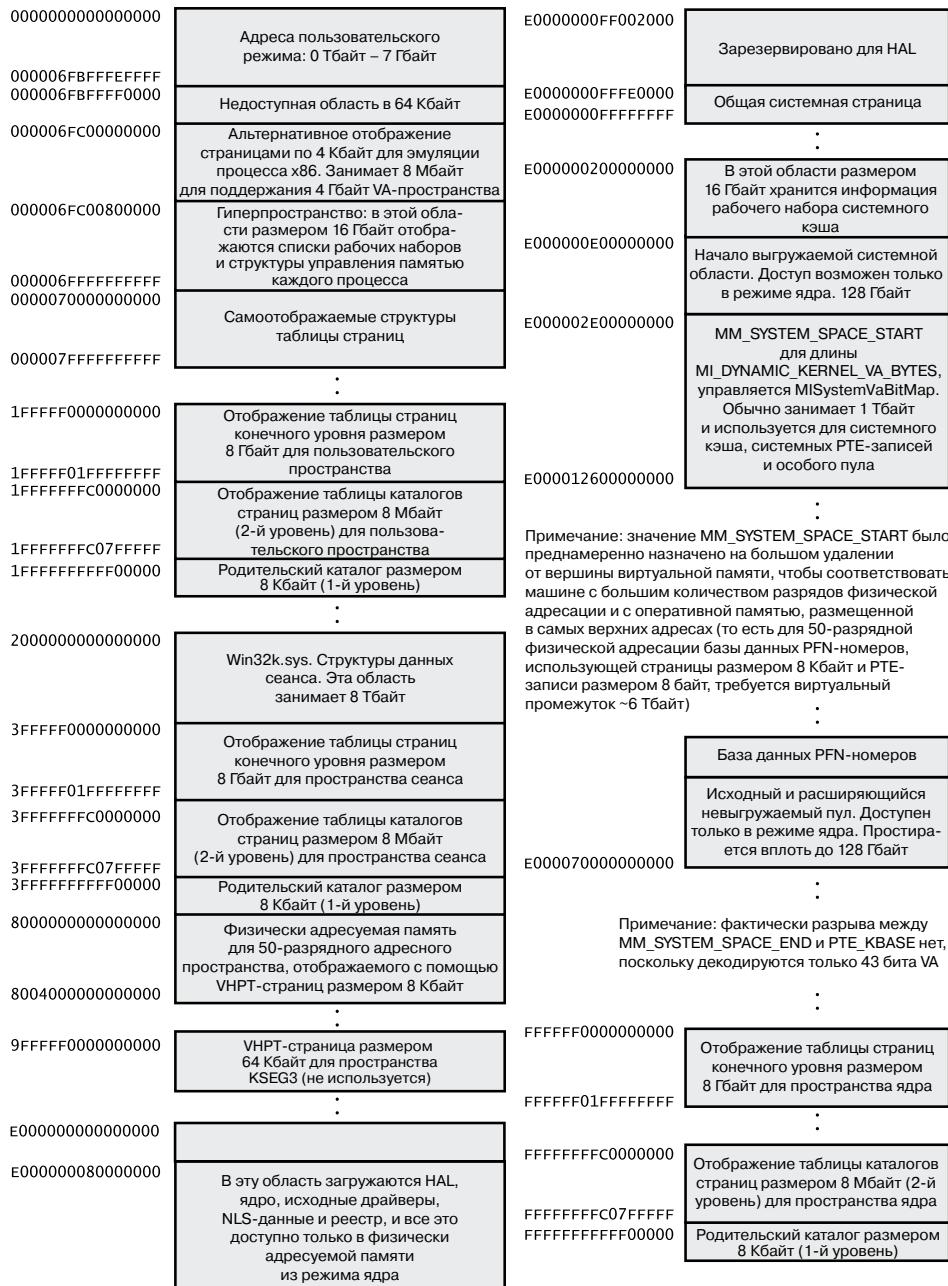


Рис. 10.12. Структура адресного пространства для платформы IA64

0000000000000000	Адреса пользовательского режима: 8 Тбайт минус 64 Кбайт
000007FFFFFFFFF	Недоступная область размером 64 Кбайт
000007FFFFFFF0000	
000007FFFFFFFFF	
	:
FFFFF080000000000	Начало системного пространства
FFFFF680000000000	Отображение четырехуровневой таблицы страниц размером 512 Гбайт
FFFFF700000000000	Гиперпространство: в этой области размером 512 Гбайт отображаются списки рабочих наборов и структуры управления памятью каждого процесса
FFFFF780000000000	Общая системная страница
FFFFF780000010000	В этой области, занимающей 512 Гбайт – 4 Кбайт, размещается информация рабочего набора системного кэша
	:
FFFFF800000000000	Отображения, инициализируемые загрузчиком. Эта область занимает 512 Гбайт
FFFFF880000000000	Начало области системных PTE-записей. Доступна только в режиме ядра. 128 Гбайт
FFFFF8A0000000000	Начало выгружаемой системной области. Доступна только в режиме ядра. 128 Гбайт
FFFFF900000000000	Win32k.sys. Структуры данных сеанса. Эта область занимает 512 Гбайт
FFFFF980000000000	MM_SYSTEM_SPACE_START для длины MI_DYNAMIC_KERNEL_VA_BYTES, управляемся MiSystemVaBitMap. Обычно занимает 1 Тбайт и используется для системного кэша, системных PTE-записей и особого пула
FFFFFA80000000000	
	:
Примечание: здесь преднамеренно зарезервирован VA-диапазон для поддержки машин с большим количеством разрядов физической адресации, с оперативной памятью, размещенной в самых верхних адресах (то есть для 49-разрядной физической адресации базы данных PFN-номеров, использующей страницы размером 4 Кбайт и PTE-записи размером 8 байт, требуется виртуальный промежуток ~6 Тбайт)	
	:
	База данных PFN-номеров
	Исходный и расширяющий невыгружаемый пул. Доступен только в режиме ядра. До 128 Гбайт
	:
FFFFFFFFFF00C00000	Минимальный объем 4 Мбайт, зарезервированный для HAL. Загрузчик или HAL могут задействовать дополнительные адреса виртуальной памяти, оставляя их отображенными при загрузке ядра.
FFFFFFFFFFFFFFFF	

Рис. 10.13. Структура адресного пространства платформы x64

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), что является весьма существенным ростом 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. То есть 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 0x00007FFFFFFFFF. Старшая половина адресного пространства начинается с адреса 0xFFFF800000000000 и заканчивается на адресе 0xFFFFFFFFFFFFFF. Каждая «каноническая» часть составляет 128 Тбайт. Поскольку на самых новых процессорах реализовано больше адресных разрядов, младшая половина памяти будет расширена вверх, к адресу 0x7FFFFFFFFFFFFF, а старшая половина памяти — вниз, к адресу 0x8000000000000000 (разбиение, аналогичное сегодняшнему пространству памяти, но с дополнительными 32 разрядами).

16-терабайтное ограничение для Windows на платформе x64

У версии Windows на платформе x64 есть дополнительное ограничение: из 256 Тбайт виртуального адресного пространства, доступного на процессорах x64, в настоящее время Windows позволяет использовать только чуть больше 16 Тбайт. Это пространство разбито на две области по 8 Тбайт. Они предназначены для пользовательского режима, где область каждого процесса начинается с 0 и простирается до старших адресов (заканчиваясь на адресе 0x000007FFFFFFFFF), и для режима ядра, где общесистемная область начинается со «всех F» и простирается в направлении младших адресов, заканчиваясь для большинства приложений на адресе 0xFFFFFFFF800000000000. В этом разделе мы поговорим о причине возникновения этого 16-терабайтного ограничения.

Неиспользуемые разряды в адресах уже заняты и продолжают применяться различными механизмами Windows. В качестве наиболее распространенных примеров структур данных, использующих разряды внутри указателя для целей, не связанных с адресацией, можно привести пуш-блокировки, быстрые ссылки, контекст Patchguard DPC и списки с одиночными связями. За это ограничение адресации памяти в Windows для платформы x64 несут ответственность списки с одиночными связями, поскольку в исходных центральных процессорах x64 отсутствует инструкция, необходимая для «портирования» структур данных в 64-разрядных версиях Windows.

Рассмотрим структуру `SLIST_HEADER`, используемую в Windows для представления записи внутри списка:

```
typedef union _SLIST_HEADER {
    ULL Alignment;
    struct {
        SLIST_ENTRY Next;
        USHORT Depth;
        USHORT Sequence;
    } DUMMYSTRUCTNAME;
} SLIST_HEADER, *PSLIST_HEADER;
```

Обратите внимание на то, что это 8-байтовая структура, которая будет гарантированно выровнена и которая состоит из трех элементов: указателя на следующую запись (32 бита, или 4 байта), глубины и порядковых номеров, каждый элемент по 16 битов (или по 2 байта). Для создания неблокируемых операций по помещению данных и по их извлечению при реализации механизма используется инструкция `CMPXCHG8B` (сравнить и переставить местами 8 байт), доступная в процессорах от Pentium и выше. Она позволяет проводить атомарные изменения 8 байт данных. За счет использования этой исходной инструкции, которая также поддерживает префикс `LOCK` (гарантируя тем самым атомарность операции на мультипроцессорной системе), надобность в спин-блокировке с целью объединения двух 32-разрядных операций доступа отпадает, и все операции со списком становятся неблокируемыми (что повышает скорость работы и масштабируемость).

На 64-разрядных компьютерах при адресации используются 64 разряда, следовательно, указатель на следующую запись по логике должен иметь 64 разряда. Если глубина и порядковые номера остаются в тех же самых пределах, система должна предоставить способ изменения как минимум $64 + 32$ бит данных, или же, что еще лучше, 128 бит, чтобы повысить возможность изменения глубины и порядковых номеров. Но на первых образцах процессоров x64 инструкция `CMPXCHG16B`, позволяющая сделать это, реализована не была. Поэтому реализация была написана таким образом, чтобы во всего лишь 64 битах уместилось как можно больше информации, что определялось максимальным количеством битов, которое могло быть атомарно изменено за одну операцию. Поэтому структура `SLIST_HEADER` для 64-разрядных систем имеет следующий вид:

```
struct { // 8-байтовый заголовок
    ULL Depth:16;
    ULL Sequence:9;
    ULL NextEntry:39;
} Header8;
```

Первым изменением стало сокращение пространства под порядковый номер с 16 до 9 бит, тем самым уменьшился максимальный порядковый номер в списке. На указатель осталось только 39 бит, которым, однако, еще далеко до 64 бит. Но принудительно задав 16-байтовое выравнивание при размещении, можно использовать дополнительные 4 бита, поскольку можно предположить, что младшие разряды всегда будут нулевыми. Тем самым на адреса выделялось 43 бита, но оставалось еще одно возможное допущение. Поскольку реализация связанных списков применяется либо в режиме ядра, либо в пользовательском режиме, но не может применяться при доступе из одних адресных

пространств в другие адресные пространства, самый верхний разряд может быть проигнорирован, точно так же, как на 32-разрядных машинах. Код при этом предполагает, что адрес относится к пространству режима ядра, если этот код вызывается в режиме ядра, и наоборот. Это позволяет увеличить в указателе `NextEntry` ширину адреса до 44 бит памяти, чем и определяется лимит адресации в Windows.

Иметь сорок четыре разряда намного лучше, чем 32. С их помощью можно описать 16 Тбайт виртуальной памяти и разбить тем самым память в Windows на две равные части по 8 Тбайт для памяти пользовательского режима и режима ядра. И все же это в 16 раз меньше собственного лимита центрального процессора (48 разрядов позволяют адресовать 256 Тбайт памяти) и совсем уж далеко от того максимума, который можно описать, имея 64 разряда. Итак, с прицелом на масштабируемость, в `SLIST_HEADER` имеются и другие биты, определяющие тип заголовка, с которым придется иметь дело. Это означает, что с приходом времени, когда все центральные процессоры x64 станут поддерживать 128-разрядную операцию сравнения и перестановки, Windows сможет ими воспользоваться (а преждевременная реализация будет означать появление двух разных образов ядра).

Давайте рассмотрим полный 8-байтовый заголовок:

```
struct { // 8-байтовый заголовок
    UONGLONG Depth:16;
    UONGLONG Sequence:9;
    UONGLONG NextEntry:39;
    UONGLONG HeaderType:1; // 0: 8-байтовый; 1: 16-байтовый
    UONGLONG Init:1; // 0: неинициализированный;
                      // 1: инициализированный
    UONGLONG Reserved:59;
    UONGLONG Region:3;
} Header8;
```

Обратите внимание на то, как бит `HeaderType` перекрываетяется битами `Depth` и позволяет реализации механизма начать работать с 16-байтовыми заголовками, как только появится их реализация. Для полноты картины рассмотрим определение 16-байтового заголовка:

```
struct { // 16-байтовый заголовок
    UONGLONG Depth:16;
    UONGLONG Sequence:48;
    UONGLONG HeaderType:1; // 0: 8-byte; 1: 16-byte
    UONGLONG Init:1; // 0: неинициализированный;
                      // 1: инициализированный
    UONGLONG Reserved:2;
    UONGLONG NextEntry:60; // последние 4 разряда всегда нулевые
} Header16;
```

Обратите внимание на то, что указатель `NextEntry` теперь становится 60-разрядным, а поскольку структура по-прежнему имеет 16-байтовое выравнивание с четырьмя свободными битами, это позволяет иметь полную 64-разрядную адресацию.

В то же время структуры данных, не содержащие SLIST-списки, не ограничиваются диапазоном адресного пространства в 8 Тбайт. Записи системных таблиц страниц, гиперпространство и рабочий набор кэша в совокупности занимают виртуальные адреса ниже 0xFFFFF80000000000, поскольку эти структуры не используют SLIST-списки.

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 (см. табл. 10.8).

Основные динамические диапазоны (поддерживаемые в настоящее время диапазоны указаны в табл. 10.9) и виртуальные адреса, доступные всему пространству ядра, устанавливаются функцией `MiInitializeDynamicVa` при инициализации системы. Затем с помощью функции `MiInitializeSystemVaRange`, которая используется для установки жестко заданных диапазонов адресов, инициализируются диапазоны адресного пространства для образов начального загрузчика, пространства процесса (гиперпространства) и HAL. Позже, при инициализации невыгруженого пула, функция `MiInitializeSystemVaRange` 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, невыгруженного пула, выгруженного 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 array`, в котором содержатся типы виртуальных адресов для только что распределенного диапазона. Значения, которые могут появляться в массиве `MiSystemVaType`, перечислены в табл. 10.9.

Таблица 10.9. Типы системного виртуального адресного пространства

Область	Описание	Возможность ограничения
<code>MiVaSessionSpace (0x1)</code>	Адреса для пространства сеанса	Да
<code>MiVaProcessSpace (0x2)</code>	Адреса для адресного пространства процесса	Нет
<code>MiVaBootLoaded (0x3)</code>	Адреса для образов, загруженных с помощью загрузчика операционной системы	Нет
<code>MiVaPfnDatabase (0x4)</code>	Адреса для базы данных PFN-номеров	Нет

продолжение ↗

Таблица 10.9 (продолжение)

Область	Описание	Возможность ограничения
MiVaNonPagedPool (0x5)	Адреса для невыгружаемого пула	Да
MiVaPagedPool (0x6)	Адреса для выгружаемого пула	Да
MiVaSpecialPool (0x7)	Адреса для особого пула	Нет
MiVaSystemCache (0x8)	Адреса для системного кэша	Да
MiVaSystemPtes (0x9)	Адреса для системных РТЕ-записей	Да
MiVaHal (0xA)	Адреса для HAL	Нет
MiVaSessionGlobalSpace (0xB)	Адреса для глобального пространства сеанса	Нет
MiVaDriverImages (0xC)	Адреса для загруженных образов драйверов	Нет

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 потребовать память обратно с помощью функции `MiReclaimSystemVa`, которая запрашивает для отчуждения виртуальные адреса, связанные с системным кэшем (через поток разыменования области), если доступное адресное пространство становится меньше 128 Мбайт. (Восстановление также может быть выполнено, если был освобожден исходный невыгружаемый пул.)

Кроме того, когда дело доходит до урезания объемов памяти, динамический распределитель виртуальных адресов имеет преимущества, улучшая дозирование и управление виртуальными адресами, предназначенными для различных имеющихся в ядре потребителей памяти. Вместо предварительного ручного выделения записей для статических таблиц страниц и самих таблиц страниц, структуры, связанные с подкачкой, выделяются по запросу. Как на 32-разрядных, так и на 64-разрядных системах тем самым сокращается использование памяти во время загрузки, поскольку отсутствует выделение таблиц страниц для неиспользуемых адресов. Это также означает, что на 64-разрядных системах для зарезервированных областей больших адресных пространств не нужны таблицы страниц, отображаемых в памяти, что позволяет им иметь сколь угодно большие размеры, особенно в системах с небольшим объемом оперативной памяти для резервного хранения получающихся структур подкачки.

ЭКСПЕРИМЕНТ: ЗАПРОС НА ИСПОЛЬЗОВАНИЕ СИСТЕМНЫХ ВИРТУАЛЬНЫХ АДРЕСОВ

Показатели текущего и пикового использования каждого типа системных виртуальных адресов можно узнать с помощью отладчика ядра. Для каждого типа системных виртуальных адресов, представленного в табл. 10.9, в имеющихся в ядре массивах MiSystemVaTypeCount, MiSystemVaTypeCountFailures и MiSystemVaTypeCountPeak содержатся размеры, показатели счетчика отказов и пиковые размеры. Вывести соответствующие системный дамп и дамп пикового использования можно следующим образом (аналогичный прием применяется и для вывода значения счетчика отказов):

```
1kd> dd /c 1 MiSystemVaTypeCount 1 c
81f4f880 00000000
81f4f884 00000028
81f4f888 00000008
81f4f88c 0000000c
81f4f890 0000000b
81f4f894 0000001a
81f4f898 0000002f
81f4f89c 00000000
81f4f8a0 000001b6
81f4f8a4 00000030
81f4f8a8 00000002
81f4f8ac 00000006
1kd> dd /c 1 MiSystemVaTypeCountPeak 1 c
81f4f840 00000000
81f4f844 00000038
81f4f848 00000000
81f4f84c 00000000
81f4f850 0000003d
81f4f854 0000001e
81f4f858 00000032
81f4f85c 00000000
81f4f860 00000238
81f4f864 00000031
81f4f868 00000000
81f4f86c 00000006
```

Теоретически различные диапазоны виртуальных адресов, выделенные компонентам, могут произвольно возрастать, пока для этого имеется достаточно доступного виртуального адресного пространства. Но на практике на 32-разрядных системах имеющийся в ядре распределитель для надежности и стабильности устанавливает ограничение на каждый тип виртуального адреса. (На 64-разрядных системах истощение адресного пространства ядра на данный момент не грозит.) Хотя изначально 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. 10.9).

Если текущий запрос при вызове функции MiObtainSystemVa превышает доступный лимит, делается пометка об отказе (см. предыдущий эксперимент) и, в зависимости от объема доступной памяти, запрашивается операция возвращения предыдущего лимита. Это должно упростить загрузку памяти и может позволить распределителю виртуальной памяти сработать при следующей попытке. (Но при этом следует помнить, что возврат касается только области системного кэша и невыгружаемого пула.)

ЭКСПЕРИМЕНТ: УСТАНОВКА ОГРАНИЧЕНИЙ СИСТЕМНЫХ ВИРТУАЛЬНЫХ АДРЕСОВ

Ограничения на использование системных виртуальных адресов, которые могут быть установлены для каждого типа таких адресов, находятся в массиве MiSystemVaTypeCountLimit. В настоящее время диспетчер памяти позволяет ограничивать только конкретно указанные типы виртуальных адресов и предоставляет возможность использовать недокументированные системные вызовы при установке ограничений для системы в динамическом режиме во время ее работы. (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\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/bb870880(VS.85).aspx).) Ограничения могут устанавливаться для тех типов системных виртуальных адресов, для которых это допустимо (см. табл. 10.9).

Для запроса и установки различных ограничений для этих типов, а также для просмотра текущего и пикового показателей использования виртуального адресного пространства можно задействовать утилиту MemLimit (<http://www.winsiderss.com/tools/memlimit.html>) из инструментария Winsider Seminars & Solutions. Текущие ограничения можно запросить с помощью ключа -q:

```
C:\ >memlimit.exe -q
```

```
MemLimit v1.00 - Query and set hard limits on system VA space consumption
Copyright (C) 2008 Alex Ionescu
www.alex-ionescu.com
```

System Va Consumption:

```
Type Current Peak Limit
Non Paged Pool 102400 KB 0 KB 0 KB
Paged Pool 59392 KB 83968 KB 0 KB
System Cache 534528 KB 536576 KB 0 KB
System PTEs 73728 KB 75776 KB 0 KB
Session Space 75776 KB 90112 KB 0 KB
```

В целях эксперимента воспользуйтесь для установки ограничения выгружаемого пула в 100 Мбайт следующей командой:

```
memlimit.exe -p 100M
```

А теперь попробуйте повторить эксперимент из главы 3 части I с командой testlimit -h, в котором предпринимается попытка создания 16 миллионов дескрипторов. Вместо получения 16 миллионов дескрипторов процесс аварийно завершится, поскольку система выйдет за пределы адресного пространства, доступного для выгружаемого пула.

Квоты системного виртуального адресного пространства

Ограничения системного виртуального пространства, рассмотренные в предыдущем разделе, позволяют управлять использованием общесистемного виртуального адресного пространства конкретными компонентами ядра, но они работают только на 32-разрядных системах, когда применяются к системе в целом. Чтобы реализовать более конкретные требования, которые могут иметься у системных администраторов относительно выделения квот, диспетчер памяти совместно с диспетчером процесса

устанавливает для каждого процесса общесистемные или индивидуальные для каждого пользователя квоты.

Значения параметров `PagedPoolQuota`, `NonPagedPoolQuota`, `PagingFileQuota` и `WorkingSetPagesQuota` в разделе `HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management` реестра могут быть настроены на установку объемов памяти каждого типа, доступных для использования заданным процессом. Эта информация считывается при инициализации, генерируя исходный блок системных квот, который затем присваивается всем системным процессам (если, как указано далее, квоты для каждого пользователя не заданы, пользовательские процессы получают копию исходных системных квот).

Чтобы включить квоты, настраиваемые под каждого пользователя, можно в разделе `HKLM\SYSTEM\CurrentControlSet\Session Manager\Quota System` реестра создать по одному подразделу, представляющему SID заданного пользователя. Тогда в этих конкретных SID-подразделах можно создать ранее упомянутые параметры, вводя ограничения только для тех процессов, которые созданы заданным пользователем. Порядок настройки этих параметров, которая может выполняться и в ходе выполнения, а также необходимые для этого привилегии указаны в табл. 10.10.

Таблица 10.10. Типы квот процесса

Имя значения	Описание	Тип значения	Динамический режим	Привилегия
<code>PagedPool-Quota</code>	Максимальный размер выгружаемого пула, который может быть выделен данным процессом	Размер (Мбайт)	Только для процессов, запущенных с системным маркером	<code>SeIncrease-QuotaPrivilege</code>
<code>NonPaged-PoolQuota</code>	Максимальный размер невыгружаемого пула, который может быть выделен данным процессом	Размер (Мбайт)	Только для процессов, запущенных с системным маркером	<code>SeIncrease-QuotaPrivilege</code>
<code>PagingFile-Quota</code>	Максимальное количество страниц, поддерживаемых процессом с помощью файла подкачки	Страницы	Только для процессов, запущенных с системным маркером	<code>SeIncrease-QuotaPrivilege</code>
<code>WorkingSet-PagesQuota</code>	Максимальное количество страниц, которое может быть у процесса в его рабочем наборе (в физической памяти)	Страницы	Да	<code>SeIncreaseBasePriorityPrivilege</code> , если операция не запрашивает очистку

Структура пользовательского адресного пространства

По аналогии с динамическим управлением адресным пространством в ядре пользовательское адресное пространство также создается в динамическом режиме — динамическое вычисление адресов стеков для программных потоков, куч процессов и загруженных образов (например, DLL-библиотек и исполняемых образов приложений, если приложение и его образы поддерживают такой режим) реализуется с помощью механизма случайного размещения схем адресного пространства (Address Space Layout Randomization, ASLR).

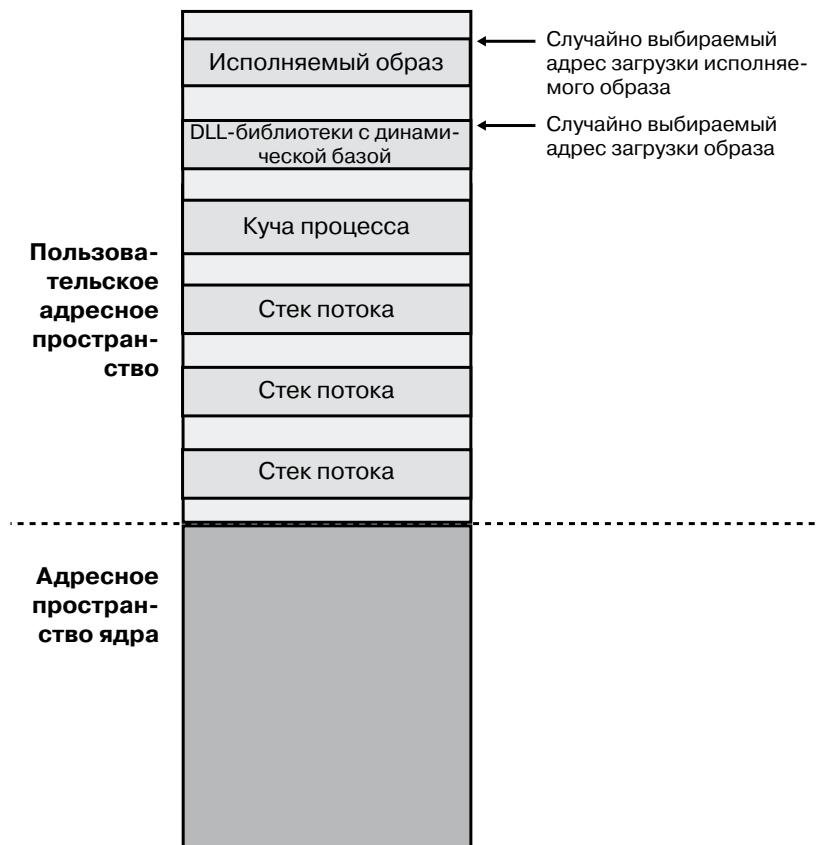


Рис. 10.14. Структура пользовательского адресного пространства с включенным механизмом ASLR

На уровне операционной системы пользовательское адресное пространство разбито на несколько четко определенных областей памяти, показанных на рис. 10.14. Сами

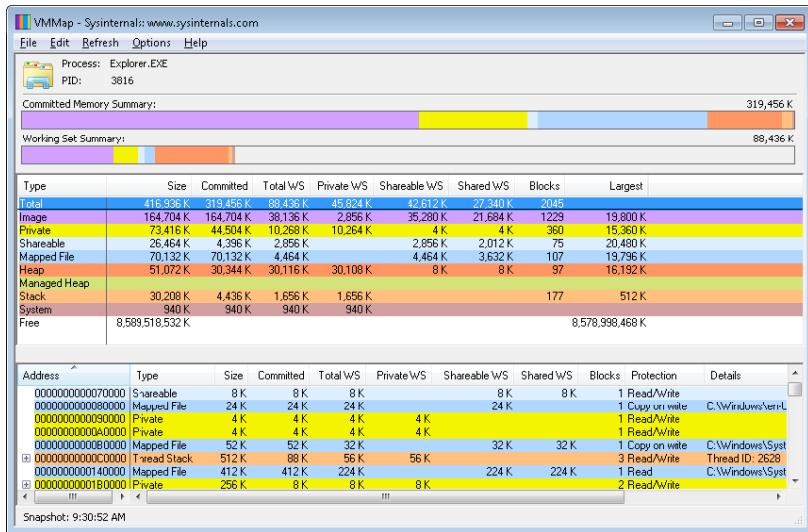
исполняемые образы и DLL-библиотеки представлены в виде отображаемых в памяти файлов образов, за которыми следует куча (или кучи) процесса и стек (или стеки) его потока (или потоков). Кроме этих областей (и ряда зарезервированных системных структур, таких как ТЕВ-блоки и РЕВ-блок), все остальные операции 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 работы с памятью. Поскольку код и данные в Windows помещаются в динамически определяемые места, атакующая программа, как правило, не может жестко задать какое-либо значимое смещение программе или предоставляемой системой DLL-библиотеке.

ЭКСПЕРИМЕНТ: АНАЛИЗ ПОЛЬЗОВАТЕЛЬСКОГО ВИРТУАЛЬНОГО АДРЕСНОГО ПРОСТРАНСТВА

Подробное представление виртуальной памяти, используемой любым процессом на вашей машине, разбитое по категориям для каждого типа выделения, может показать утилита VMMap, разработанная в Sysinternals, предоставив при этом следующие категории:

- **Image** (Под образ). Показатели выделения памяти, используемой для отображения исполняемых образов и всего, от чего зависит их работа (например, динамических библиотек), а также любых других отображаемых на память файлов образов (в переносимом исполняемом формате).
- **Private** (Под закрытую область). Показатели выделения памяти, помеченной как закрытая, например внутренние структуры данных, не относящиеся к стеку или куче.
- **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**).
- **Mapped File** (Под отображаемый файл). Показатели выделения памяти для отображаемых на память файлов данных.
- **Heap** (Под кучу). Память, выделенная под кучу (или кучи), которой владеет процесс.
- **Stack** (Под стек). Память, выделенная под стек каждого программного потока данного процесса.
- **System** (Под систему). Память ядра, выделенная под процесс (например, объекту процесса).

На следующей копии экрана показан обычный вид программы Explorer, полученный с помощью утилиты VMMap.



В зависимости от типа выделения памяти, VMMap может показать дополнительную информацию, например имена отображаемых файлов, идентификаторы куч (при выделении памяти под кучи) и идентификаторы потоков (при выделении памяти под стеки). Более того, 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 тех DLL-библиотек, от которых зависит его работа. 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`, и который содержит раздел перемещения, будет обрабатываться с помощью ASLR. При обнаружении такого образа система глобально выбирает подходящее смещение для текущей загрузки. Это смещение выбирается из корзины, содержащей 256 значений, каждое из которых имеет 64-килобайтное выравнивание.

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. Прибавлением 1 диспетчер памяти обеспечивает такое значение, которое никогда не станет равным нулю, и исполняемые образы при использовании ASLR никогда не будут загружаться по адресу в PE-заголовке. Эта дельта прибавляется к предпочтительному адресу загруз-

ки, 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 `MiImageBias`. 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 замаскированное в 8-разрядную величину, предоставляющую 256 возможных значений. В отличие от значения для исполняемых образов, это значение вычисляется только один раз при начальной загрузке и используется во всей системе, позволяя сохранять возможность совместного использования DLL-библиотек в физической памяти и перемещаться только один раз. Если DLL-библиотеки внутри различных процессов были перемещены в другое место, их код нельзя будет использовать совместно. Загрузчику придется зафиксировать 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, придется иметь свою собственную закрытую копию DLL-библиотеки в физической памяти.

После вычисления смещения диспетчер памяти инициализирует битовое отображение под названием `MiImageBitmap`. Это отображение служит для представления диапазонов от 0x50000000 до 0x78000000 (хранищихся в `MiImageBitmapHighVa`), и каждый бит представляет одну единицу размещения (64 Кбайт, как уже упоминалось). Когда диспетчер загружает DLL-библиотеку, устанавливается соответствующий разряд, чтобы отметить ее размещение в системе; когда та же самая DLL-библиотека загружается опять, диспетчер памяти совместно использует ее объект раздела с уже перемещенной информацией.

После загрузки каждой DLL-библиотеки система сканирует битовое отображение сверху вниз в поиске свободных разрядов. Ранее вычисленное значение `MiImageBias` используется в качестве начального индекса, начиная с верхнего значения для рандомизации загрузки при различных вариантах начальной загрузки, как это и предлагалось. Поскольку двоичное отображение при загрузке первой DLL-библиотеки (которой всегда является `Ntdll.dll`) полностью пустое, адрес ее загрузки может быть легко вычислен: $0x78000000 - \text{MiImageBias} \times 0x10000$. 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`.

И наконец, если места в двоичном отображении 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 им базового адреса.

Рандомизация стека

Следующим этапом работы механизма ASLR является рандомизация размещения стека исходного программного потока (и впоследствии каждого нового потока). Рандомизация выполняется до тех пор, пока для процесса не будет установлен флаг `StackRandomizationDisabled`, и состоит она из первоначального выбора 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 извлеченное из TSC-счетчика значение, на этот раз длиной в 9 разрядов. Затем с целью выравнивания данное значение умножается на 4, а это означает, что оно может быть размером до 2048 байт (до половины страницы). Полученное значение складывается с базовым адресом для получения окончательной базы 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, извлекаемое из 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 принудительно вызвать нарушение прав доступа, если атака грубо ведется по всему возможному диапазону адресов куч.

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.

Управление средствами смягчения уровня опасности

Мы уже видели, что ASLR и многие другие средства смягчения уровня опасности 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 исполнения (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, в Microsoft выпустили улучшенный инструментарий

смягчения уровня опасности (Enhanced Mitigation Experience Toolkit, EMET). EMET предлагает централизованное управление средствами смягчения уровня опасности, 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 смягчения уровня опасности. И наконец, EMET позволяет вручную отключать конкретные приложения, которые могут создать проблемы совместимости в конкретных средах окружения, даже если эти приложения выбирались разработчиком.

ЭКСПЕРИМЕНТ: ПРОСМОТР ASLR-ЗАЩИТЫ ПРОЦЕССОВ

Для просмотра процессов (а также, что не менее важно, загружаемых ими DLL-библиотек) можно воспользоваться программой Process Explorer, разработанной в Sysinternals. Это позволит понять, поддерживают ли процессы механизм ASLR. Следует учесть, что если всего одна DLL-библиотека, загруженная процессом, не поддерживает ASLR, она может сделать процесс более уязвимым при атаках.

Для просмотра ASLR-состояния процессов нужно щелкнуть правой кнопкой мыши на любом столбце в дереве процессов, выбрать команду Select Columns (Выбрать столбцы), а затем на вкладке Process Image (Образ процесса) установить флажок ASLR Enabled (механизм ASLR включен). Заметьте, что не все входящие в Windows программы и службы запускаются с включенным механизмом ASLR; кроме того, есть также один яркий пример — приложения стороннего разработчика, у которого механизм ASLR не включен.

В нашем примере мы выделили процесс Notepad.exe (всем известный Блокнот). В данном случае его адрес загрузки 0xFE0000. 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.

The screenshot shows the Process Explorer interface with two main tables. The top table lists processes by name, PID, CPU usage, description, company name, and ASLR status. The bottom table lists DLL components by name, description, version, base address, ASLR status, and mapping type (Image or Data). The 'notepad.exe' process is selected in both tables, showing its ASLR status as 'ASLR' and base address as '0xFE0000'. Other visible DLLs include kernel32.dll, KernelBase.dll, locale.nls, lpk.dll, msctf.dll, msvcr.dll, and ntdll.dll.

Process	PID	CPU	Description	Company Name	ASLR
notepad.exe	2956	< 0.01	Notepad	Microsoft Corporation	ASLR
taskmgr.exe	3048	0.21	Windows Task Manager	Microsoft Corporation	ASLR

Name	Description	Version	Base	ASLR	Mapping
kernel32.dll	Windows NT BASE API Client DLL	6.1.7601.17651	0x77650000	ASLR	Image
KernelBase.dll	Windows NT BASE API Client DLL	6.1.7601.17651	0x75EC0000	ASLR	Image
locale.nls			0x1E0000	n/a	Data
lpk.dll	Language Pack	6.1.7600.16385	0x77CF0000	ASLR	Image
msctf.dll	MSCTF Server DLL	6.1.7600.16385	0x762E0000	ASLR	Image
msvcr.dll	Windows NT CRT DLL	7.0.7601.17744	0x771E0000	ASLR	Image
notepad.exe	Notepad	6.1.7600.16385	0xFE0000	ASLR	Image
notepad.exe.mui	Notepad	6.1.7600.16385	0x70000	n/a	Data
ntdll.dll	NT Layer DLL	6.1.7601.17725	0x77AD0000	ASLR	Image

CPU Usage: 1.57% | Commit Charge: 34.48% | Processes: 49 | Physical Usage: 44.13%

Преобразование адресов

После изучения порядка структурирования в Windows виртуальных адресных пространств давайте посмотрим, как эта операционная система отображает эти адресные пространства на реально существующие физические страницы. Пользовательские приложения и системный код ссылаются на виртуальные адреса. А начнем мы с подробного изучения преобразования адресов на 32-разрядных системах x86 (без поддержки и с поддержкой PAE-расширения) и продолжим кратким описанием разницы между 64-разрядными платформами IA64 и x64. В следующем разделе мы поговорим о последствиях тех случаев, когда такое преобразование не ведет к получению адреса физической памяти (речь пойдет о подкачке), и узнаем, как Windows управляет физической памятью через рабочие наборы и базу данных страничных блоков.

Преобразование виртуальных адресов на платформе x86

Центральный процессор преобразует виртуальные адреса в физические, используя структуры данных, называемые 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. 10.15 показано, как на платформе x86 три последовательные виртуальные страницы могут быть отображены 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 отсутствия страницы.

На рис. 10.15 пунктириная линия, соединяющая виртуальные страницы с PTE-записями, отражает косвенную связь виртуальных страниц и физической памяти.

ПРИМЕЧАНИЕ

Даже код, выполняемый в режиме ядра (например, драйверы устройств), не может непосредственно ссылаться на физические адреса памяти, но может делать это опосредованно, создав сначала отображаемые на них виртуальные адреса. Для получения дополнительных сведений следует просмотреть процедуры поддержки списков описателей памяти (memory descriptor list, MDL), описание которых есть в документации WDK.

Как уже упоминалось, Windows на платформе x86 может использовать для преобразования адреса любую из двух схем: с расширением физических адресов (PAE) и без него. Сначала мы рассмотрим схему без расширения, а в следующих разделах — с расширением. Поскольку для понимания механизма расширения физических адресов нужно понять, как проходит преобразование адресов без такого расширения, нужно сначала изучить данный раздел, даже если основной интерес у вас вызывает имен-

но PAE. К тому же описание механизма преобразования адресов на платформе x64 строится на информации, касающейся PAE.

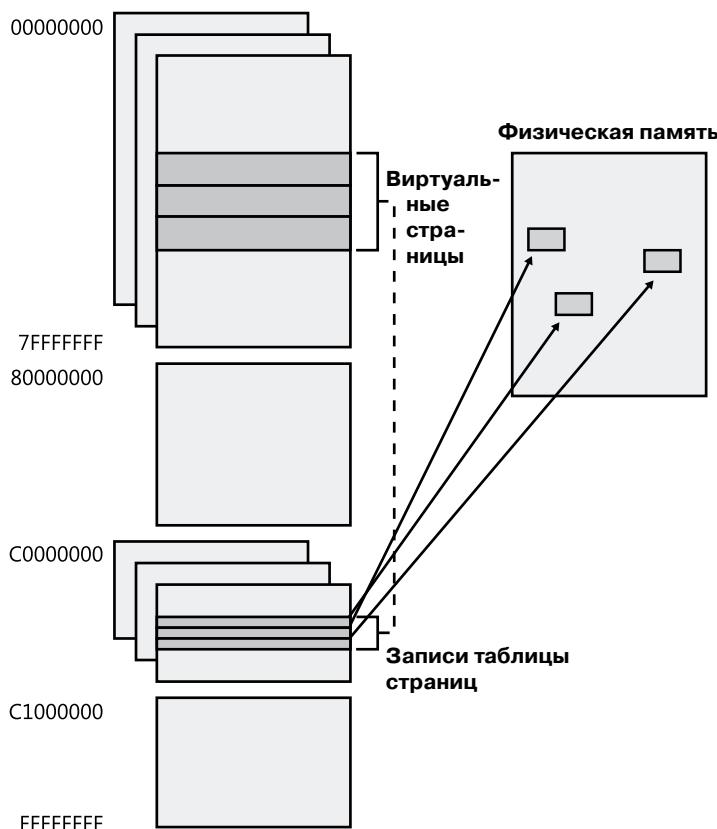


Рис. 10.15. Отображение виртуальных адресов на физическую память (x86)

Системы на платформе x86 без поддержки PAE для преобразования виртуальных адресов в физические используют двухуровневую структуру таблицы страниц. 32-разрядный виртуальный адрес, отображаемый с помощью обычной страницы размером 4 Кбайт, представляется в виде двух полей: поля *номера виртуальной страницы* (virtual page number) и поля байта внутри страницы, называемого *байтовым смещением* (byte offset). Затем номер виртуальной страницы разбивается на два подполя, которые, как показано на рис. 10.16, называются *индексом каталога страниц* (page directory index) и *индексом таблицы страниц* (page table index). Эти два поля служат для локализации записей в каталоге страниц и в таблице страниц.

Размер этих битовых полей задается структурами, на которые они ссылаются. Например, байтовое смещение имеет размер 12 бит, потому что оно указывает на байт внутри страницы размером 4096 байт ($2^{12} = 4096$). Остальные индексы имеют размер

10 бит, поскольку индексируемые с их помощью структуры содержат 1024 записи ($2^{10} = 1024$).



Рис. 10.16. Компоненты 32-разрядного виртуального адреса на платформе x86

Преобразование виртуальных адресов проводится с целью их превращения в физические адреса, то есть в адреса оперативной памяти. Формат физического адреса на платформе x86 без поддержки PAE показан на рис. 10.17.



Рис. 10.17. Компоненты физического адреса на платформе x86 без поддержки PAE

Как видите, формат очень похож на формат виртуального адреса. Более того, значение байтового смещения из виртуального адреса будет таким же и в получающемся физическом адресе. После этого можно сказать, что преобразование адресов приводит к превращению номеров виртуальных страниц в номера физических страниц, также называемых *номерами страницных блоков* (Page Frame Numbers, PFN). Байтовое смещение в этом не участвует, в результате преобразования адреса не меняется. Оно просто копируется из виртуального адреса в физический. Связь между этими тремя значениями и порядок их использования при преобразовании адреса иллюстрирует рис. 10.18.

Преобразование виртуального адреса выполняется в несколько основных этапов:

- Для получения физического адреса каталога страниц диспетчер управления памятью (Memory Management Unit, MMU) использует привилегированный регистр центрального процессора CR3.
- Часть виртуального адреса, называемая индексом каталога, применяется в качестве индекса каталога страниц, с помощью которого находится запись каталога страниц (Page Directory Entry, PDE), содержащая то место в таблице страниц, которое необходимо для отображения виртуального адреса. В свою очередь, PDE содержит

номер физической страницы, называемый также номером страничного блока (Page Frame Number, PFN), искомой таблицы страниц, при условии, что таблица страниц уже находится в оперативной памяти — таблицы страниц могут быть выгружены или еще не созданы, в таких случаях перед обработкой таблица страниц сначала помещается в оперативную память. Если флаг в PDE-записи показывает, что описание дается для большой страницы, значит, в ней просто содержится PFN-номер целевой большой страницы, а весь остальной виртуальный адрес рассматривается как байтовое смещение внутри большой страницы.

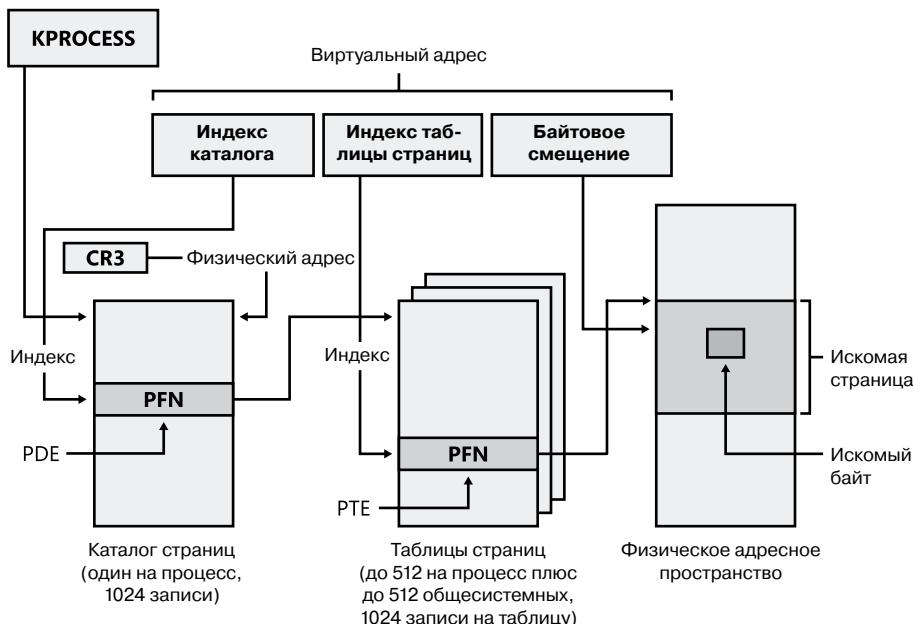


Рис. 10.18. Преобразование допустимого виртуального адреса
(на платформе x86 без поддержки PAE)

3. Индекс таблицы страниц используется в качестве индекса в таблице страниц для определения той РТЕ-записи, которая описывает данную виртуальную страницу.
4. Если бит достоверности РТЕ-записи сброшен, выдается ошибка отсутствия страницы (ошибка управления памятью). Имеющийся в операционной системе обработчик ошибки управления памятью определяет местонахождение страницы и пытается сделать ее достоверной. В случае успешной попытки происходит переход к шагу 5. (См. раздел «Обработка ошибок отсутствия страниц».) Если страница не может быть или не должна быть сделана достоверной (например, из-за отказа защиты), обработчик ошибок генерирует нарушение прав доступа или инициирует проверку ошибок (bug check).

- Когда PTE-запись дает описание достоверной страницы (сразу же или после устранения ошибки), искомый физический адрес составляется из PFN- поля PTE- записи с последующим полем байтового смещения, которое берется из исходного виртуального адреса.

Теперь, получив общее представление, давайте рассмотрим детали, касающиеся структуры каталогов страниц, таблиц страниц и PTE-записей.

Каталоги страниц

На платформе x86 без поддержки PAE у каждого процесса имеется один каталог страниц, то есть страница, которую создает диспетчер памяти для отображения местоположения всех таблиц страниц этого процесса. Физический адрес каталога страниц процесса хранится в блоке процесса ядра (KPROCESS), но на платформе x86 без поддержки PAE он также виртуально отображается на адрес 0xC0300000. (Более подробно KPROCESS и другие структуры данных процесса рассматриваются в главе 5 части I.)

Центральный процессор получает местоположение каталога страниц из своего привилегированного регистра, который называется CR3. В нем содержится номер страничного блока каталога страниц. (Поскольку сам каталог страниц всегда выровнен по границе страницы, младшие 12 бит его адреса всегда нулевые, то есть передавать их регистру CR3 не нужно.) При каждом переключении контекста на поток, принадлежащий другому процессу, а не тому процессу, которому принадлежит текущий выполняемый поток, процедура переключения контекста, находящаяся в ядре, загружает значение этого регистра из поля нового процесса, находящегося в блоке KPROCESS. Переключения контекста между потоками одного и того же процесса не приводит к перезагрузке физического адреса каталога страниц, поскольку все потоки в рамках одного процесса совместно используют одно и то же адресное пространство процесса, и следовательно, тот же каталог страниц и те же таблицы страниц.

Каталог страниц состоит из записей каталога страниц (PDE-записей), каждая из которых имеет длину 4 байта. PDE-записи в каталоге страниц описывают состояние и местоположение всех возможных таблиц страниц процесса. Далее в этой главе показано, что таблицы страниц создаются по мере необходимости, поэтому каталог страниц для большинства процессов указывает только на небольшой набор таблиц страниц. (Если таблица страниц еще не создана, то для принятия решения о необходимости ее материализации при обращении к ней делается запрос к дереву VAD-дескрипторов.) Формат PDE-записи здесь не приводится, чтобы не повторять практически такой же формат аппаратной PTE-записи, который рассматривается далее.

Для описания всего виртуального адресного пространства размером 4 Гбайт требуется 1024 таблицы страниц. Каталог страниц процесса, отображающий эти таблицы страниц, содержит 1024 PDE-записи. Поэтому индекс каталога страниц должен иметь размер 10 бит ($2^{10} = 1024$).

ЭКСПЕРИМЕНТ: ИССЛЕДОВАНИЕ КАТАЛОГА СТРАНИЦ И PDE-ЗАПИСЕЙ

Физический адрес каталога страниц текущего выполняемого процесса можно узнать, изучив поле DirBase в результатах выполнения команды !process отладчика ядра:

```
1kd> !process -1 0
PROCESS 857b3528 SessionId: 1 Cid: 0f70 Peb: 7ffdf000 ParentCid: 0818
DirBase: 47c9b000 ObjectTable: b4c56c48 HandleCount: 226.
Image: windbg.exe
```

Виртуальный адрес каталога страниц можно увидеть, исследовав результат работы отладчика ядра для PTE-записи конкретного виртуального адреса:

```
1kd> !pte 10004
VA 00010004
PDE at C0300000 PTE at C0000040
contains 6F06B867 contains 3EF8C847
pfn 6f06b ---DA--UWEV pfn 3ef8c ---D---UWEV
```

PTE-часть результатов работы отладчика ядра описывается в следующем разделе, а сами эти результаты — в разделе, посвященном преобразованию адреса на платформе x86 с поддержкой PAE.

Поскольку для каждого процесса Windows предоставляет закрытое адресное пространство, у каждого процесса для отображения его закрытого адресного пространства есть собственные каталог страниц и таблицы страниц. Но таблицы страниц, описывающие системное пространство, совместно используются всеми процессами (а пространство сеанса совместно используется только процессами, имеющими отношение к данному сеансу). Чтобы избежать ситуации с несколькими таблицами страниц, описывающими одну и ту же виртуальную память, при создании процесса записи каталога страницы, описывающие системное пространство, инициализируются с целью указания на уже существующие системные таблицы страниц. Если процесс является частью сеанса, таблицы страниц пространства сеанса также совместно используются путем указания в записях каталога страниц на уже 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. Поле индекса таблицы страниц виртуальных адресов (как показано на рис. 10.18) указывает, какая PTE-запись в таблице страниц соответствует искомой странице данных и содержит ее описание. Индекс таблицы страниц имеет ширину 10 бит, позволяя ссылаться на 1024 PTE-записи, каждая размером 4 байта. Разумеется, поскольку система x86 предоставляет 4 Гбайт виртуального адресного пространства, для отображения всего адресного пространства требуется более одной таблицы страниц. Для вычисления количества таблиц страниц, требуемых для отображения всего виртуального адресного пространства, следует разделить 4 Гбайт на объем виртуальной памяти, отображаемый одной таблицей страниц. Вспомним, что каждая таблица страниц на платформе x86 отображает 4 Мбайт страниц данных. Следовательно, для отображения всего адресного пространства в 4 Гбайт потребуется 1024 таблицы страниц (4 Гбайт/4 Мбайт). Это количество соответствует 1024 записям в каталоге страниц.

Для исследования PTE-записей можно в отладчике ядра воспользоваться командой `!pte`. (См. далее эксперимент «Преобразование адресов».) Достоверные PTE-записи мы рассмотрим в этом разделе, а недостоверные — в одном из следующих. У досто-

верных PTE-записей есть два основных поля: номер страничного блока (PFN) той физической страницы, которая содержит данные или физический адрес страницы в памяти, и несколько флагов, описывающих, как показано на рис. 10.19, состояние страницы.

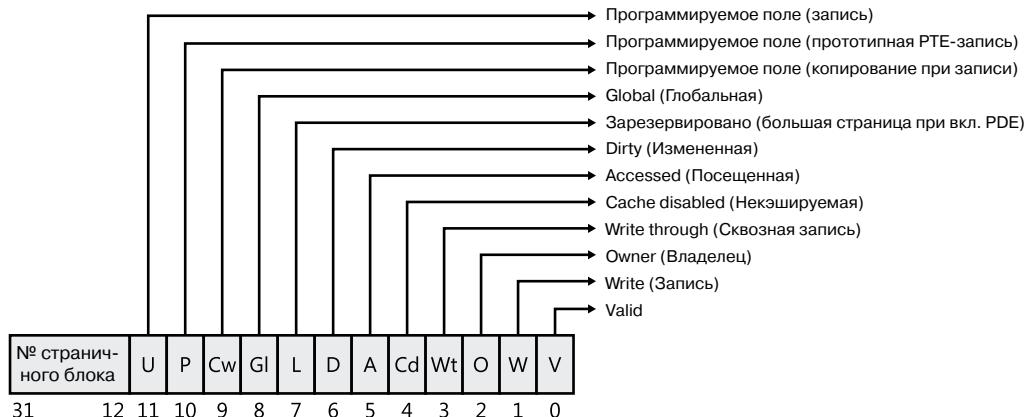


Рис. 10.19. Достоверные аппаратные PTE-записи на платформе x86

Позже вы увидите, что показанные на рис. 10.19 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. 10.11.

Таблица 10.11. Биты состояния и защиты в PTE-записи

Название	Значение
Посещенная	К странице было обращение
Некэшируемая	Выключение кэширования этой страницы центральным процессором
Копирование при записи	Для страницы используется копирование при записи (см. ранее)
Измененная	В страницу велась запись
Глобальная	Преобразование касается всех процессов (например, сброс буфера преобразования эту РТЕ-запись не затрагивает)
Большая страница	Показывает, что РТЕ-запись является отображением страницы размером 4 Мбайт или 2 Мбайт на системах с поддержкой PAE (см. ранее раздел «Большие и малые страницы»)
Владелец	Показывает, может ли код пользовательского режима обращаться к странице или обращение к ней ограничено только кодом режима ядра
Прототипная	РТЕ-запись является прототипной и используется в качестве шаблона для описания общей памяти, связанной с объектами разделов

Название	Значение
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	Показывает MMU доступность страницы для записи

На платформе x86 аппаратная PTE-запись содержит два бита, которые могут быть изменены MMU, — это биты изменения и посещения. MMU устанавливает бит (при условии, что он еще не установлен) посещения при каждом чтении страницы или записи в нее. А бит изменения MMU устанавливает 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 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. Позже бит изменения сообщит диспетчеру памяти о том, что содержимое физической страницы, перед тем как использоваться для чего-либо другого, должно быть записано во внешнюю память.

Фактически, в мультипроцессорных системах это может привести к весьма затратным условиям гонок. MMU-диспетчеры различных 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 пуш-блокировки. Однако на мультипроцессорных системах, даже если блокировка удерживается одним процессором, состояние бита изменения может меняться MMU-диспетчерами других центральных процессоров. При этом возникает возможность пропустить обновление бита изменения.

Чтобы избежать подобного развития событий, диспетчер памяти в Windows изначально устанавливает в PTE-записях, у которых аппаратный бит записи (бит 1)

нулевой, программный бит 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 предназначеннной только для чтения. Но в таком случае диспетчер памяти знает, что на самом деле страница предназначена для записи (благодаря программному биту записи), запрашивает пуш-блокировку рабочего набора, устанавливает в РТЕ-записи бит изменения и аппаратный бит записи, обновляет список рабочего набора, чтобы сделать пометку об изменении страницы, снимает пуш-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 записях в страницу исключения не выдаются, поскольку аппаратный бит записи уже установлен. ММУ 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 слишком большой издержкой, 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, будучи в недостоверном состоянии (при сброшенном нулевом бите в РТЕ-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 выполнитьброс буфера быстрого преобразования адресов (см. далее) без удержания блокировки для каждой сбрасываемой страницы.

Байт внутри страницы

После того как диспетчер памяти выяснит номер физической страницы, он должен найти внутри этой страницы запрошенные данные. Именно для этого и используется поле байтового смещения. Это байтовое смещение просто копируется из исходного виртуального адреса в соответствующее поле физического адреса. На платформе x86 используется байтовое смещение шириной 12 бит, что позволяет ссылаться на 4096 байт данных (таков размер используемой страницы).

Можно все это описать и по-другому: байтовое смещение из виртуального адреса объединяется с номером физической страницы, извлеченным из РТЕ-записи. На этом преобразование виртуального адреса в физический завершается.

Буфер быстрого преобразования адресов

Вы уже знаете, что каждое аппаратное преобразование адреса требует проведения двух поисков: одного для нахождения нужной записи в каталоге страниц (что позволяет узнать местоположение таблицы страниц) и второго для нахождения нужной записи в таблице страниц. Поскольку выполнение двух дополнительных поисковых

операций в памяти при каждой ссылке на виртуальный адрес утраивает требуемое число обращений к памяти и негативно сказывается на производительности, все центральные процессоры кэшируют преобразования адресов, что исключает необходимость повторного преобразования при частом обращении к одним и тем же адресам. Этот кэш представляет собой массив ассоциативной памяти, которая называется *буфером быстрого преобразования адресов* (Translation Look-aside Buffer, TLB). 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. 10.20, 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.

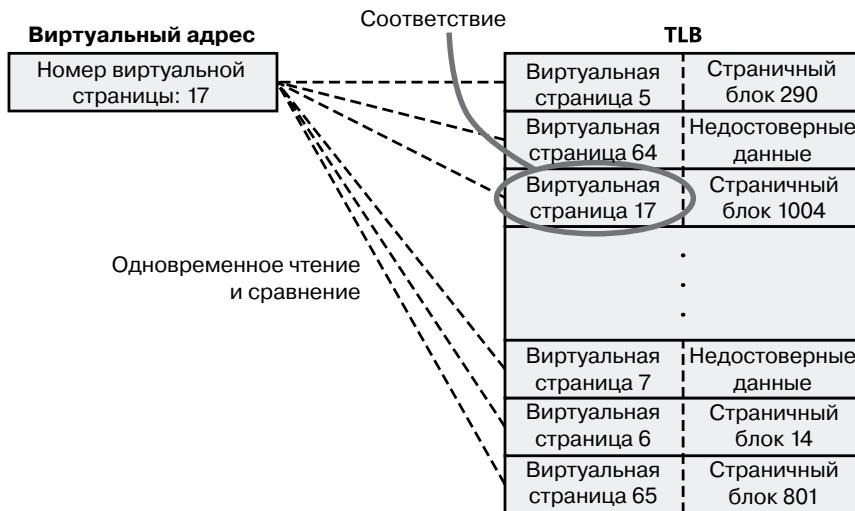


Рис. 10.20. Подключение буфера быстрого преобразования адресов

Скорее всего, в TLB окажутся наиболее часто используемые виртуальные адреса, благодаря чему обеспечивается очень быстрое преобразование виртуального адреса в физический и, стало быть, 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 немножко 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 повторит свое обращение, произойдет ошибка отсутствия страницы, и диспетчер памяти вернет страницу в память (если нужно) и воссоздаст ее PTE-запись (что затем повлечет за собой появление в TLB записи, относящейся к этой странице).

Расширение физических адресов

Режим отображения памяти, который называли *расширением физических адресов* (Physical Address Extension, PAE), впервые был реализован в процессоре Intel x86 Pentium Pro. В сочетании с подходящим микропроцессорным набором PAE позволяет 32-разрядным операционным системам получать на ныне существующих процессорах Intel x86 (которые без расширения физических адресов имеют доступ к памяти размером до 4 Гбайт) получать доступ к физической памяти объемом вплоть до 64 Гбайт, а при запуске на процессорах x64 в унаследованном режиме — вплоть до 1024 Гбайт физической памяти (хотя Windows в настоящее время ограничивает ее объем до 64 Гбайт из-за размера базы данных PFN-номеров, необходимого для описания такого большого объема памяти). Когда процессор работает с расширением физических адресов, диспетчер управления памятью (MMU) делит виртуальные адреса, отображаемые на обычные страницы, на четыре поля (рис. 10.21). MMU по-прежнему реализует каталоги страниц и таблицы страниц, но под третьим уровнем, уровнем PAE, поверх которого находится таблица указателей каталогов страниц.

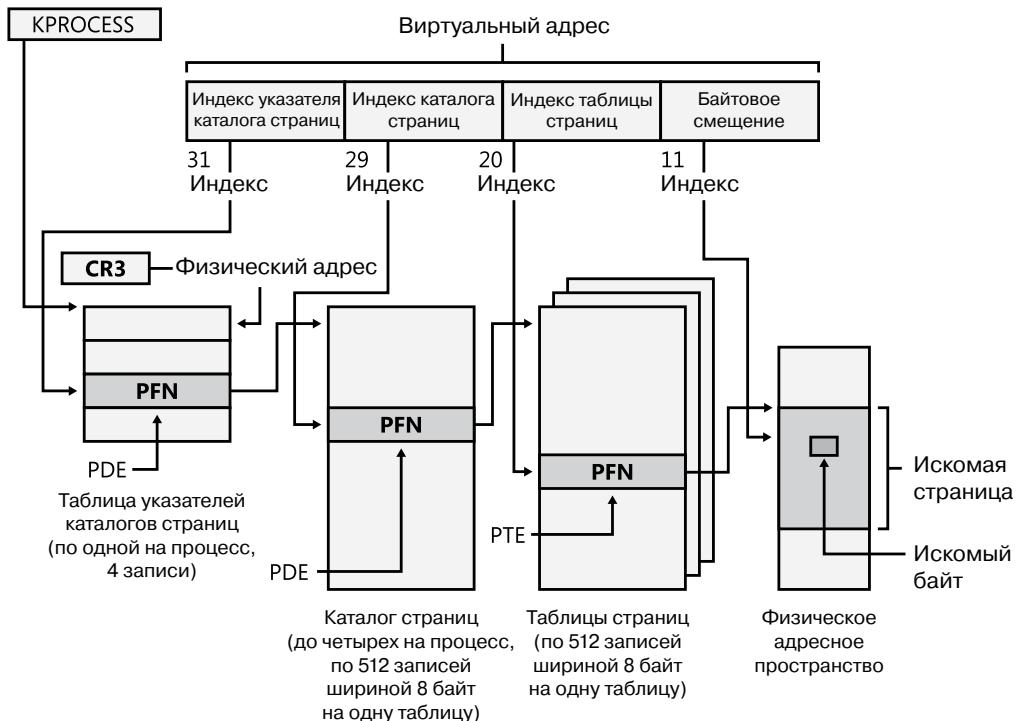


Рис. 10.21. Отображение страниц посредством расширения физических адресов

О том, какие выгоды могут получать 32-разрядные приложения от таких больших объемов памяти, рассказывается в разделе «Оконные расширения адресов». Но даже если приложение не использует эти возможности, диспетчер памяти задействует всю доступную физическую память для рабочих наборов нескольких процессов, файлового кэша и обработанных закрытых данных посредством списков системного кэша, списков ожидающих использования и измененных страниц (см. раздел «База данных номеров страничных блоков»).

Режим PAE выбирается в ходе начальной загрузки и без перезагрузки изменен быть не может. Как уже объяснялось в главе 2 части I, есть специальная версия 32-разрядного ядра Windows с поддержкой PAE, которая называется `Ntkrnlpa.exe`. 32-разрядные системы, имеющие аппаратную поддержку защиты от выполнения неисполняемых данных памяти (см. ранее раздел «Защита страниц от выполнения»), по умолчанию производят начальную загрузку, используя ядро с поддержкой PAE, потому что режим PAE необходим для реализации функции неисполнения. Чтобы принудительно загрузить ядро с поддержкой PAE, можно для BCD-параметра `rae` установить значение `ForceEnable`.

Следует учесть, что ядро с поддержкой PAE устанавливается на диск на всех 32-разрядных системах Windows, даже на системах с небольшим объемом памяти и без аппаратной поддержки защиты от выполнения неисполняемых данных памяти. Это делается для предоставления возможности тестирования кода, зависящего от поддержки PAE, даже на системах с небольшим объемом памяти, и для исключения необходимости переустановки Windows при последующем добавлении оперативной памяти. Еще одним BCD-параметром, касающимся PAE, является `noLowmem`. С его помощью игнорируется память ниже 4 Гбайт (предполагается, что имеется как минимум 5 Гбайт физической памяти) и драйверы устройств перемещаются выше этого диапазона. Тем самым гарантируется, что драйверы будут установлены на физических адресах выше 32 бит, благодаря чему будет легче обнаруживать любые возможные ошибки драйверов, связанные с расширением знака.

Чтобы разобраться в работе PAE, нужно понять, откуда берутся размеры различных структур и битовых полей. Следует напомнить, что целью PAE является адресация оперативной памяти выше 4 Гбайт. Это ограничение для адресов оперативной памяти возникло из-за 12-битного смещения и 20-битных полей номера страничного блока физических адресов: $12 + 20 = 32$ бита физического адреса, а 2^{32} байта составляет 4 Гбайт. (Имейте в виду, что все это получается из-за ограничения формата физического адреса и количества битов, выделенных на PFN в записи таблицы страниц. Тот факт, что на платформе x86 с поддержкой PAE или без таковой виртуальное адресное пространство имеет ширину 32 разряда, физическое адресное пространство никак не ограничивает.)

С применением PAE PFN-номер расширяется до 24 бит. В сочетании с 12-битным байтовым смещением это позволяет адресовать 2^{24+12} байт, или 64 Гбайт, памяти.

Для поддержки 24-разрядных PFN-номеров PAE расширяет поля PFN записей таблиц страниц и каталогов страниц с 20 до 24 бит. Чтобы выделить место под такое расширение, записи таблиц страниц и каталогов страниц получают ширину 8 байт вместо 4. (Казалось бы, можно было расширить поле PFN в PTE- и PDE-записях до 32 бит, а не на 4 бита, но на процессорах x86 PFN-номера ограничены 24 битами. Из-за этого большое количество битов в PDE-записи остались незадействованными или, вернее, оставлены для будущих расширений.)

Поскольку и таблицы страниц, и каталоги страниц должны умещаться на одной странице, в этих таблицах должно быть только 512, а не 1024 записи. Поэтому соответствующие поля индексов виртуального адреса урезаются с 10 до 9 бит.

После всего этого остаются неучтеными два старших бита виртуального адреса. Используя их, PAE расширяет количество каталогов страниц с одного до четырех и добавляет таблицу преобразования адресов, относящуюся к третьему уровню, — она называется *таблицей указателей каталогов страниц* (Page Directory Pointer Table, PDPT). В этой таблице содержится всего четыре записи по 8 байт, предоставляющие PFN-номера четырех каталогов страниц. Два старших бита виртуального адреса, предназначенные для индексирования PDPT-таблицы, называются *индексом указателя каталога страниц* (page directory pointer index).

Как и прежде, CR3 предоставляет местоположение таблицы верхнего уровня, но теперь это PDPT, а не каталог страниц. PDPT-таблица должна быть выровнена по 32-байтовой границе и к тому же должна размещаться в первых 4 Гбайт оперативной памяти (потому что CR3 на процессорах семейства x86 является всего лишь 32-разрядным регистром, даже с поддержкой PAE).

Следует заметить, что с поддержкой расширения физических адресов можно адресовать больше памяти, чем при стандартном преобразовании, но не благодаря дополнительному уровню преобразования, а за счет расширения формата физического адреса. Дополнительный уровень преобразования понадобился для того, чтобы обеспечить обработку всех 32 разрядов виртуального адреса.

ЭКСПЕРИМЕНТ: ПРЕОБРАЗОВАНИЕ АДРЕСОВ

Чтобы стало понятнее, как работает механизм преобразования адресов, в этом эксперименте представлен реальный пример преобразования виртуального адреса на платформе x86 с поддержкой PAE. Мы используем доступные инструментальные средства в отладчике ядра с целью изучения PDPT, каталогов страниц, таблиц страниц и PTE-записей. (На сегодняшних процессорах семейства x86 даже при объеме памяти меньше 4 Гбайт Windows обычно запускается в режиме поддержки PAE, поскольку этот режим нужен для включения защиты при обращении к памяти, не содержащей исполняемого кода.) В данном примере работа ведется с процессом, виртуальный адрес которого равен 0x30004 и который отображается на достоверный физический адрес. В последующих примерах будет показано, как с помощью отладчика ядра отслеживать преобразование недостоверных адресов.

Сначала давайте приведем число 0x30004 в двоичный вид и разобьем его на три поля, используемые для преобразования адреса. В двоичном виде число 0x30004 выглядит как 11.0000.0000.0000.0100. Его разбиение на поля компонентов даст следующую картину.

31 30 29	21 20	12 11	0
00	00.0000.000	0.0011.0000	0000.0000.0100
Индекс указателя каталога страниц	Индекс каталога страниц (0)	Индекс таблицы страниц (0x30 или 48 в десятичном виде)	Байтовое смещение (4)

Для инициирования процесса преобразования центральному процессору нужен физический адрес таблицы указателей каталогов страниц, который при запуске потока в этом процессе находится в регистре CR3. Этот адрес можно вывести путем поиска поля DirBase в выводимых командой !process данных:

```
1kd> !process -1 0
PROCESS 852d1030 SessionId: 1 Cid: 0dec Peb: 7ffdf000 ParentCid: 05e8
DirBase: ced25440 ObjectTable: a2014a08 HandleCount: 221.
Image: windbg.exe
```

Поле DirBase показывает, что таблица указателей каталогов страниц имеет физический адрес 0xced25440. Как видно на предыдущей иллюстрации, значение поля индекса таблицы указателей каталогов страниц виртуального адреса, используемого в нашем примере, нулевое. Следовательно, запись PDPT-таблицы, содержащая физический адрес соответствующего каталога страниц, является ее первой записью и находится по физическому адресу 0xced25440.

Как и при выполнении на системах x86 без поддержки PAE, команда !pte отладчика ядра выводит PDE- и PTE-записи, которые описывают виртуальный адрес:

```
1kd> !pte 30004
VA 00030004
PDE at C0600000 PTE at C0000180
contains 00000002EBF3867 contains 800000005AF4D025
pfn 2ebf3 ---DA--UWEV pfn 5af4d ----A--UR-V
```

Отладчик не показывает таблицу указателей каталогов страниц, но ее нетрудно вывести, задав ее физический адрес:

```
1kd> !dq ced25440 L 4
#ced25440 00000000`2e8ff801 00000000`2c9d8801
#ced25450 00000000`2e6b1801 00000000`2e73a801
```

Здесь мы воспользовались командой !dq расширения отладчика. Она похожа на команду dq (которая выводится в виде четырех слов — это название 64-разрядного поля,вшедшее с тех времен, когда «слова» зачастую состояли из 16 разрядов), но при этом позволяет исследовать память по физическим, а не по виртуальным адресам. Так как известно, что длина PDPT-таблицы составляет всего четыре записи, чтобы не засорять результат, мы добавили к команде аргумент L 4, задающий длину выводимых данных.

Как уже было показано, индекс PDPT-таблицы (два самых старших бита) из нашего примера виртуального адреса равен 0, следовательно, нужная нам запись PDPT-таблицы является первым выведенным четвертым словом. Запись PDPT-таблицы имеет формат, аналогичный записям PD- и PT-таблиц, следовательно, в результате исследования можно увидеть, что данная запись содержит значение PFN-номера, равное 0x2e8ff для физического адреса 2e8ff000. Это физический адрес каталога страниц.

В выводимых командой !pte данных адрес в PDE-записи показан как виртуальный, а не физический. На системах x86 с поддержкой PAE первый процесс каталога страниц начинается с виртуального адреса 0xC0600000. Виртуальный адрес поля индекса каталога страниц в нашем примере равен 0, стало быть, мы смотрим на первую PDE-запись в каталоге страниц. Поэтому в данном случае адрес в PDE-записи совпадает с адресом каталога страниц.

Как и без поддержки PAE, запись в каталоге страниц предоставляет PFN-номер нужной таблицы страниц, в данном примере PFN-номер равен 0x2ebf3. Следовательно, таблица страниц начинается с физического адреса 0x2ebf3000. К этому MMU добавляет поле индекса таблицы страниц (0x30) из виртуального адреса, умноженное на 8 (размер PTE в байтах; на системах без поддержки PAE он равен 4). Тогда физический адрес PTE равен 0x2ebf3180.

Отладчик показывает, что эта PTE-запись находится по виртуальному адресу 0xC0000180. Следует заметить, что та часть, которая относится к байтовому смещению (0x180), равна такой же части в физическом адресе, и это правило в преобразовании адресов остается неизменным. Поскольку диспетчер памяти отображает таблицы страниц, начиная с адреса 0xC0000000, добавление 0x180 к 0xC0000000 дает виртуальный адрес, находящийся в выводимых данных отладчика ядра: 0xC0000180. Отладчик показывает, что поле PFN этой PTE-записи содержит значение 0x5af4d.

И наконец, можно учесть байтовое смещение исходного адреса. Как уже ранее упоминалось, MMU проводит объединение байтового смещения со значением PFN-номера из PTE-записи, что дает физический адрес 0x5af4d004. Это физический адрес, соответствующий на данный момент исходному виртуальному адресу 0x30004.

Битовые флаги из PTE интерпретируются как права, относящиеся к PFN-номеру. Например, PTE-запись, описывающая страницу, на которую делается ссылка, имеет флаги --A--UR-V. Здесь A означает accessed (посещенная страница), U относится к доступности в пользовательском режиме (в отличие от доступности только в режиме ядра), R означает, что страница предназначена только для чтения (в отличие от страницы, в которую можно вести записи), а V означает, что страница является достоверной (то есть PTE-запись представляет достоверную страницу в физической памяти).

Для подтверждения наших вычислений физического адреса можно взглянуть на исследуемую память как по виртуальному, так и по физическому адресу. Сначала, воспользовавшись командой dd отладчика (dd означает display dwords — показать двойные слова) с виртуальным адресом, мы увидим следующий результат:

```
1kd> dd 30004
00030004 00000020 00000001 00003020 000000dc
00030014 00000000 00000020 00000000 00000014
00030024 00000001 00000007 00000034 0000017c
00030034 00000001 00000000 00000000 00000000
00030044 00000000 00000000 00000002 1a26ef4e
00030054 00000298 00000044 000002e0 00000260
00030064 00000000 f33271ba 00000540 0000004a
00030074 0000058c 0000031e 00000000 2d59495b
```

А при выполнении команды !dd с только что вычисленным физическим адресом мы увидим точно такой же результат:

```
1kd> !dd 5af4d004
#5af4d004 00000020 00000001 00003020 000000dc
#5af4d014 00000000 00000020 00000000 00000014
#5af4d024 00000001 00000007 00000034 0000017c
#5af4d034 00000001 00000000 00000000 00000000
#5af4d044 00000000 00000000 00000002 1a26ef4e
#5af4d054 00000298 00000044 000002e0 00000260
#5af4d064 00000000 f33271ba 00000540 0000004a
#5af4d074 0000058c 0000031e 00000000 2d59495b
```

Аналогично можно сравнить полученные данные с виртуальных и физических адресов PTE- и PDE-записей.

Преобразование виртуальных адресов на платформе x64

Преобразование адреса на платформе x64 производится так же, как на платформе x86 с поддержкой PAE, но при этом появляется четвертый уровень преобразования. У каж-

дого процесса имеется расширенный каталог страниц четвертого уровня, называемый *таблицей отображения 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 таблице указателей каталогов страниц в системах x86 с поддержкой РАЕ, 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, каждый из которых, в свою очередь, 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 страницных блоков (PFN).

Нынешние варианты реализации архитектуры x64 ограничивают виртуальные адреса 48 разрядами. Компоненты, из которых состоит этот 48-разрядный виртуальный адрес, представлены на рис. 10.22. Связи между этими структурами показаны на рис. 10.23. И наконец, формат аппаратной записи таблицы страниц на платформе x64 можно видеть на рис. 10.24.

64-разрядный адрес на платформе x64 (на сегодняшних процессорах 48-разрядный)



Рис. 10.22. Виртуальный адрес на платформе x64

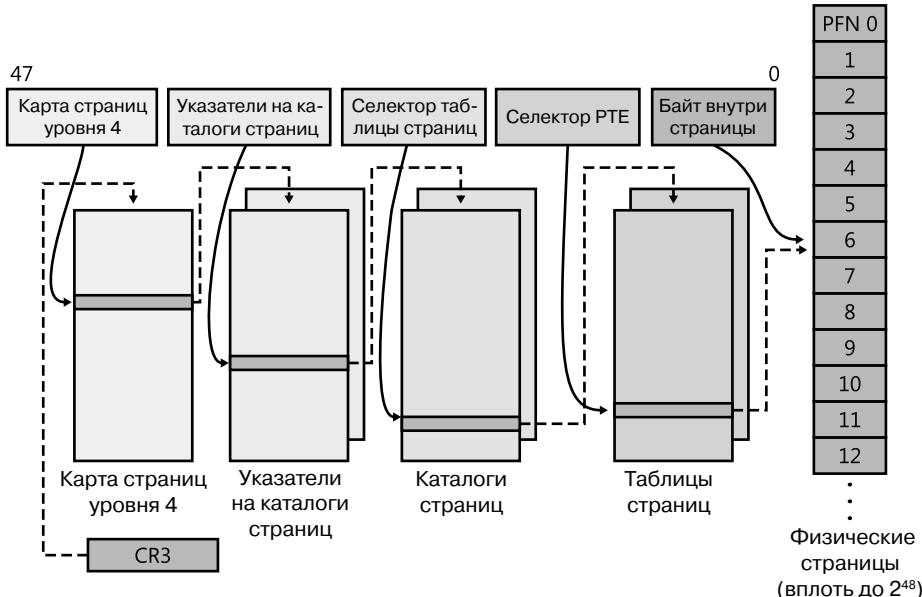


Рис. 10.23. Структуры преобразования адресов на платформе x64

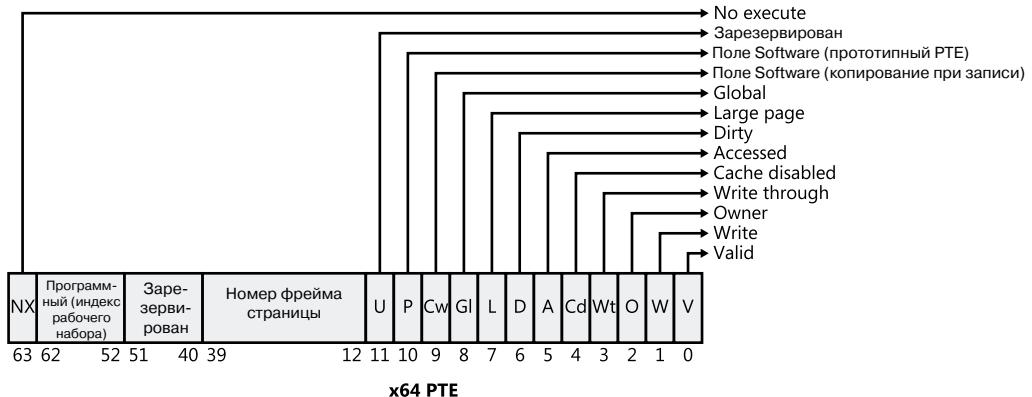


Рис. 10.24. Аппаратная запись таблицы страниц на платформе x64

Преобразование виртуальных адресов на платформе IA64

Виртуальное адресное пространство на платформе IA64 аппаратно разбито на восемь областей. У каждой области может быть собственный набор таблиц страниц. Из этих восьми областей Windows использует пять, у трех из которых есть таблицы страниц. Области и порядок их применения перечислены в табл. 10.12.

Таблица 10.12. Области на платформе IA64

Область	Применение
0	Пользовательский код и данные
1	Код и данные пространства сеанса
2	Не используется
3	Не используется
4	Кэшируемое пространство Kseg3, которое представляет собой точное отображение физической памяти. Таблица страниц для этой области не требуется, поскольку необходимые вставки в TLB делаются непосредственно диспетчером памяти
5	Некэшируемое пространство Kseg4, которое представляет собой точное отображение физической памяти. Используется только в отдельных местах для доступа к местам ввода-вывода, например к диапазону портов ввода-вывода. Таблицы страниц для этой области не нужны
6	Не используется
7	Код и данные ядра

При преобразовании адресов 64-разрядной системой Windows на платформе IA64 используется трехуровневая схема таблиц страниц. У каждого процесса имеется структура указателей каталога страниц, содержащая 1024 указателя на каталоги страниц.

Каждый каталог страниц содержит 1024 указателя на таблицы страниц, которые, в свою очередь, указывают на физические страницы. Формат аппаратной РТЕ-записи на платформе IA64 показан на рис. 10.25.

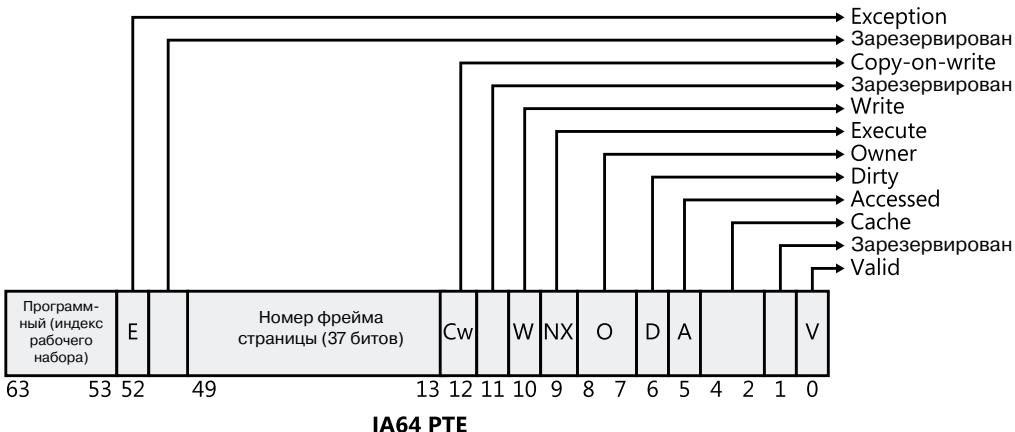


Рис. 10.25. Аппаратная запись таблицы страниц на платформе IA64

Обработка ошибок отсутствия страниц

Ранее было показано, как осуществляется преобразование при достоверной РТЕ-записи. Если в РТЕ-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 РТЕ-записей и порядок разрешения ссылок на такие записи.

ПРИМЕЧАНИЕ

В этом разделе подробно описываются форматы РТЕ-записей, используемые только на 32-разрядной платформе x86. РТЕ-записи для 64-разрядных платформ содержат такую же информацию, но их подробная структура здесь не рассматривается.

Ссылка на недостоверную страницу называется *ошибкой отсутствия страницы* (page fault). Обработчик системных прерываний ядра (см. раздел «Диспетчеризация системных прерываний» в главе 3 части I) направляет 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. 10.13).

Таблица 10.13. Причины ошибок отсутствия страниц

Причина ошибки	Результат
Обращение к странице, отсутствующей в памяти, но имеющейся на диске в файле подкачки или в отображаемом файле	Выделение физической страницы и чтение исключимой 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 неподтвержденной странице (например, в зарезервированном адресном пространстве или в адресном пространстве, которое еще не выделено)	Нарушение прав доступа
Обращение из пользовательского режима к той странице, которая может быть доступна 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
Запись в сторожевую страницу	Нарушение доступа к сторожевой странице (если это ссылка на стек пользовательского режима, 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 РТЕ-записи
Выполнение кода в странице, помеченной как неисполняемая	Нарушение прав доступа (поддерживается только на тех аппаратных платформах, которые поддерживают защиту от исполнения)

В следующем разделе дается описание четырех основных типов недостоверных РТЕ-записей, с которыми имеет дело обработчик ошибок отсутствия страниц. Далее рассказывается о специальном случае недостоверных РТЕ-записей — прототипных РТЕ-записей, служащих для реализации страниц, предназначенных для совместного использования.

РТЕ-записи

Если бит достоверности РТЕ-записи, встреченный в ходе преобразования адреса, имеет нулевое значение, это означает, что РТЕ-запись представляет недостоверную страницу, одну из тех, при ссылке на которую диспетчер памяти выдает исключи-

ние 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.

В следующем списке 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. 10.11, другие битовые поля имеют такое же или похожее значение, что и у соответствующих полей аппаратной PTE-записи.

- **Файл подкачки.** Искомая страница находится в файле подкачки. Как показано на рис. 10.26, 4 бита в PTE-записи показывают, в какой из 16 возможных страниц подкачки находится искомая страница, а 20 бит (на системах x86 без поддержки PAE; в других режимах их больше) предоставляют номер страницы в файле. Обработчик ошибок управления памятью инициирует страничную операцию по помещению страницы в память и придаанию ей статуса достоверной. Смещение в файле подкачки никогда не бывает нулевым и никогда не содержит все единичные биты (то есть самая первая и самая последняя страницы в файле подкачки для самой подкачки не используются), что позволяет существовать другим форматам (см. далее).



Рис. 10.26. Запись таблицы страниц, представляющая страницу в файле подкачки

- **Заполнение нулями.** Этот формат PTE-записи аналогичен формату ранее рассмотренной записи для файла подкачки, но поле смещения в файле подкачки 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 файла подкачки, но на этот раз поле смещения в файле подкачки заполнено единицами. Это служит признаком страницы, чье 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 далее.)

- **Переходное состояние.** Искомая страница находится в памяти в списке ожидающих использования, измененных или измененных, но не записываемых страниц либо не относится ни к какому списку. Как показано на рис. 10.27, РТЕ-запись содержит номер страничного блока искомой страницы. Обработчик ошибок управления памятью удаляет страницу из списка (если она состоит в каком-либо списке) и добавляет ее к рабочему набору процесса.

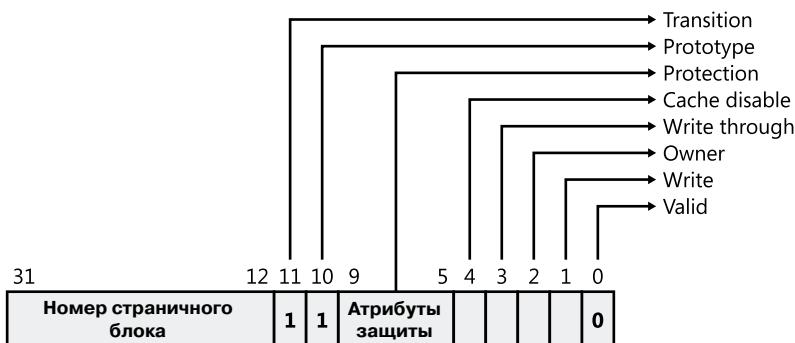


Рис. 10.27. Переходная запись таблицы страниц

- **Неизвестная страница.** РТЕ-запись содержит нули, или таблица страниц еще не существует (запись каталога страниц, 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 подтвержден этот виртуальный адрес. Если он был подтвержден, создаются таблицы страниц, представляющие только что подтвержденное адресное пространство. (См. далее обсуждение VAD-дескрипторов.) Если адрес не был подтвержден (поскольку страница зарезервирована или вообще не определена), ошибка отсутствия страницы вызывает исключение нарушения доступа.

Прототипные РТЕ-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 РТЕ-записями. Для разделов, поддерживаемых

файлом подкачки, 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 конце данной 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, указывают 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. 10.28, при превращении страницы, предназначенной для совместного использования, в недостоверную, PTE-запись в таблице страниц процесса заполняется специальными значениями, указывающими на прототипную PTE-запись, содержащую описание страницы.



Рис. 10.28. Структура недостоверной PTE-записи, указывающей на прототипную PTE-запись

Поэтому при последующем обращении к странице диспетчер памяти, используя информацию, закодированную в исходной PTE-записи, может определить местоположение прототипной PTE-записи, содержащей, в свою очередь, описание страницы, на которую делается ссылка. 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 (табл. 10.19).

- ❑ **Заполнение нулями.** В качестве искомой должна быть предоставлена страница, заполненная нулями.
- ❑ **Страницочный файл.** Искомая страница находится в файле подкачки.
- ❑ **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 страницочных 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 извлекает страницу с диска, то для указания ее нового физического местоположения ему нужно лишь обновить прототипную PTE-запись, а 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 ссылаются на страницу, настоящая PTE-запись обновляется.

На рис. 10.29 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 файле подкачки — 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.

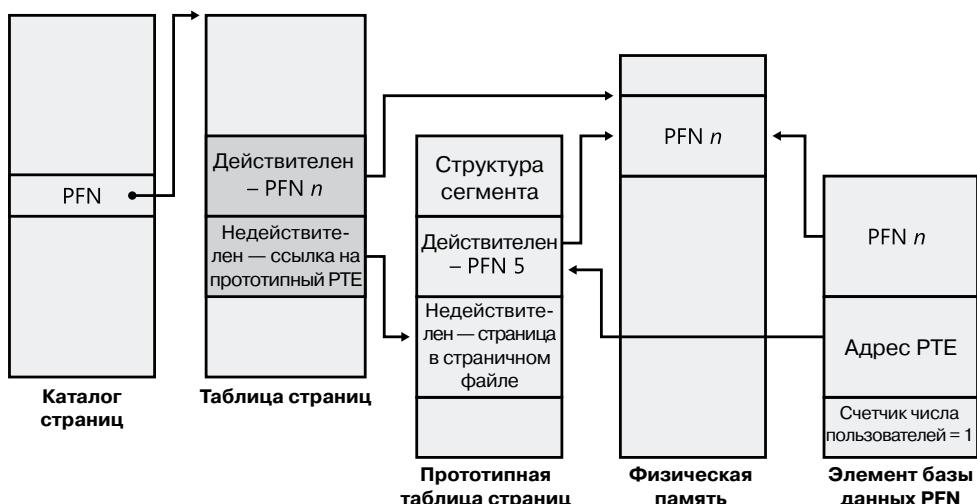


Рис. 10.29. Прототипные записи таблицы страниц

Страницочный ввод-вывод

Страницочный ввод-вывод (In-paging I/O) осуществляется при выполнении операции чтения файла (страницного или отображаемого) для разрешения ошибки отсутствия страницы. Кроме того, поскольку таблицы страниц могут выгружаться на диск, 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, на которую сделана ссылка.

Страницочные операции ввода-вывода проводятся в синхронном режиме, то есть 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 страницочных операций ввода-вывода использует в функции запроса ввода-вывода специальный модификатор. После завершения страницочного ввода-вывода система ввода-вывода запускает событие, которое активирует обработчик ошибок управления памятью и позволяет ему продолжить страницочную обработку.

В ходе страницочного ввода-вывода программный поток, в котором произошла ошибка отсутствия страницы, не может завладеть ни одним из важных объектов синхронизации диспетчера памяти. Другие потоки, принадлежащие тому же процессу, в ходе страницочного ввода-вывода могут вызывать функции виртуальной памяти и обрабатывать ошибки отсутствия страниц. Но при этом при завершении ввода-вывода появляется ряд важных условий, которые должны распознаваться обработчиком ошибок управления памятью:

- ❑ Ошибка обращения к той же самой странице может возникнуть в другом потоке того же самого процесса или другого процесса (эта ситуация, называемая конфликтной ошибкой отсутствия 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 страницочный ввод-вывод. Таким образом, при завершении запроса он может обнаружить возникновение этих условий, и если это необходимо, проигнорировать ошибку отсутствия страницы, не делая страницу достоверной. Если же инструкция, приведшая к ошибке, запускается повторно, обработчик ошибок управления памятью вызывается еще раз, и PTE-запись переоценивается 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 отсутствия страницы* (*collided page fault*). Обработчик ошибок управления памятью обнаруживает и обрабатывает конфликтную ошибку отсутствия страницы оптимальным образом, поскольку на мультипоточных системах подобные ошибки возникают довольно часто. Если у другого потока или процесса возникает ошибка отсутствия той же самой страницы, обработчик ошибок управления памятью обнаруживает конфликтную ошибку отсутствия страницы, отмечая, что страница находится в переходном состоянии и подвергается чтению. (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 заблокировал базу данных PFN-номеров. Эти операции состоят в проверке состояния ввода-вывода, позволяющей убедиться в успешном завершении ввода-вывода, снятия бита выполнения чтения в базе данных PFN-номеров и обновления РТЕ-записи.

Когда последующие потоки заблокируют базу данных PFN-номеров, чтобы разрешить конфликтную ошибку отсутствия страницы, обработчик ошибок управления памятью выяснит, что начальное обновление произошло, поскольку бит выполнения чтения сброшен, и проверит 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, РТЕ-запись не обновляется, и поток, вызвавший ошибку, запускает исключение ошибки страничного ввода-вывода.

Кластерные ошибки отсутствия страниц

Для разрешения ошибки отсутствия страницы и заполнения системного кэша диспетчер памяти производит предвыборку больших 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 проводить весьма затратное межпроцессорное прерывание для очистки TLB.) Предварительно извлеченные страницы помещаются в список ожидания и помечаются в РТЕ-записи как находящиеся в переходном состоянии. Если впоследствии на предварительно извлеченную страницу будет сделана ссылка, диспетчер памяти добавит ее в рабочий набор. Но если ссылка на нее 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, как показано на рис. 10.30, он воспользуется фиктивной страницей, чтобы можно было все-таки провести эффективную единую объемную операцию ввода-вывода.

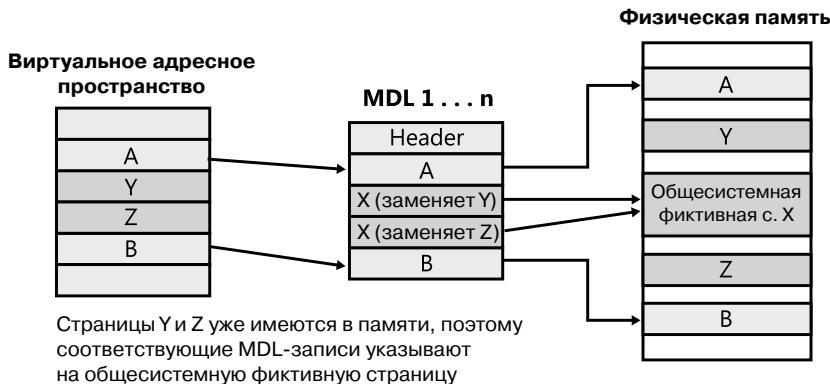


Рис. 10.30. Использование фиктивной страницы при отображении в MDL виртуального адреса на физический

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 присутствуют в памяти, поэтому считывать их не нужно. (Фактически, со временем 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 дает больший эффект, чем считывание страницы *A* и последующее считывание 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, сообразуясь с объемом доступной памяти, текущим использованием системы 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 в один и тот же MDL-список несколько раз или даже в несколько одновременно

задействованных MDL-списков, используемых разными драйверами. Следовательно, данные о местоположении игнорируемых страниц могут измениться в любое время.

Страницочные файлы

Страницочные файлы, или файлы подкачки, служат для хранения измененных страниц, которые все еще нужны некоторым процессам, но должны быть записаны на диск (поскольку их отображение утрачено или от них приходится избавляться из-за нехватки памяти). Пространство в страницочном файле резервируется при начальном подтверждении страниц, но фактически оптимально выбрать сгруппированные места в страницочном файле до выгрузки страниц на диск невозможно.

При начальной загрузке системы процесс диспетчера сеансов (см. главу 13) считывает список открываемых страницочных файлов, изучая параметр реестра `HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\PagingFiles`. Этот многострочный параметр реестра содержит имя, минимальный и максимальный размеры каждого страницочного файла. Windows поддерживает до 16 страницочных файлов. На платформе x86 с обычным ядром каждый страницочный файл может иметь максимальный размер 4095 Мбайт. На платформе x86 с ядром PAE и на платформе x64 каждый страницочный файл может быть объемом 16 Тбайт (терабайт), а на платформе IA64 – 32 Тбайт. Открытые страницочные файлы не могут быть удалены при работе системы, потому что открытый дескриптор каждого страницочного файла поддерживается процессом System (см. главу 2 части I). Тот факт, что страницочные файлы открыты, дает ответ на вопрос, почему встроенное средство дефрагментации не может устранить фрагментацию страницочного файла, пока система работает. Для дефragmentации страницочного файла нужно воспользоваться свободно распространяемой программой Pagedefrag, созданной в Sysinternals. В ней используется тот же подход, что и в других программах дефragmentации сторонних разработчиков, – процесс дефragmentации начинается на ранней стадии начальной загрузки, еще до того, как страницочные файлы открыты процессом диспетчера сеансов.

Поскольку в страницочном файле содержатся части виртуальной памяти процесса и ядра, в целях безопасности система может быть настроена на очистку страницочного файла при завершении своей работы. Чтобы включить этот режим нужно установить для параметра `HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\ClearPageFileAtShutdown` реестра значение 1. В противном случае после завершения работы системы в страницочном 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 страницочного файла имеют нулевые значения, это является признаком страницочных файлов, управляемых системой. Система выбирает размер страницочного файла исходя из следующих принципов:

- Минимальный размер устанавливается равным размеру оперативной памяти или для него устанавливается значение 1 Гбайт, в зависимости от того, что больше.
- Максимальный размер устанавливается равным тройному размеру оперативной памяти или для него устанавливается значение 4 Гбайт, в зависимости от того, что больше.

Как видите, по умолчанию исходный размер страничного файла пропорционален объему оперативной памяти. Такая политика основана на предположении, что машины с более объемной оперативной памятью используются более интенсивно, для чего требуются большие подтвержденные объемы виртуальной памяти.

ЭКСПЕРИМЕНТ: ПРОСМОТР ПАРАМЕТРОВ СТРАНИЧНЫХ ФАЙЛОВ (ФАЙЛОВ ПОДКАЧКИ)

Для просмотра списка страничных файлов нужно заглянуть в параметр HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\PagingFiles реестра. В нем содержатся варианты настройки конфигурации страничного файла, изменяемые через диалоговое окно Advanced System Settings (Дополнительные параметры системы). Откройте Панель управления, щелкните на апплете System And Security (Система и безопасность), а затем — на пункте System (Система). Диалоговое окно System Properties (Свойства системы) также можно открыть, щелкнув правой кнопкой мыши на значке Computer (Компьютер) в Проводнике и выбрав команду Properties (Свойства). Находясь в этом окне, щелкните на пункте Advanced System Settings (Дополнительные параметры системы), затем в области Performance (Быстродействие) щелкните на кнопке Settings (Параметры). В диалоговом окне Performance Options (Параметры быстродействия) перейдите на вкладку Advanced (Дополнительно) и в области Virtual Memory (Виртуальная память) щелкните на кнопке Change (Изменить).

Для добавления нового страничного файла панель управления использует (только для себя) системную службу `NtCreatePagingFile`, которая определена в файле `Ntdll.dll`. Страницочные файлы всегда создаются несжимаемыми, даже если каталог, в котором они находятся, подвергается сжатию. Чтобы уберечь новые файлы подкачки от удаления, дескриптор дублируется в процессе `System`, поэтому даже после того, как создававший файлы процесс закроет дескриптор нового страницочного файла, дескриптор всегда будет для него открыт.

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 страницочный файл) файлу, связанному с отображаемым объектом, на который делается ссылка 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 страницочный 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, то есть 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 встречаются ссылки на страничные файлы. Но в принципе можно, хотя и не рекомендуется, запустить Windows вообще без страничных файлов (файлов подкачки). Встречая в данном разделе любое упоминание страничного файла, всегда нужно мысленно добавлять «если такой файл (или файлы) имеется».

Концептуально системный лимит подтверждения представляет собой все виртуальное адресное пространство, которое может быть создано в дополнение к вариантам выделения виртуальной памяти, связанным с принадлежащим самой системе резервным хранилищем, то есть в дополнение к разделам, отображаемым на файлы. Его числовое значение просто равно объему оперативной памяти, доступному Windows, плюс текущие размеры любых страничных файлов. Если страничный файл увеличивается в размере или создается новый страничный файл, соответственно увеличивается и лимит подтверждения. Если страничные файлы отсутствуют, то системный лимит подтверждения просто равен тому объему оперативной памяти, который доступен Windows.

Показатель подтверждения — это общесистемный совокупный объем всех «подтвержденных» фрагментов выделенной памяти, которые должны находиться либо в оперативной памяти, либо в страничном файле. Из названия должно быть понятно, что одним из вкладчиков в показатель подтверждения является закрытое подтвержденное виртуальное адресное пространство процесса. Но есть множество других вкладчиков в это значение, некоторые из них не столь очевидны.

Windows также обслуживает 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 все эти варианты выделения должны в нем сохраняться.

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 путем вызова функции `VirtualAlloc` с параметром `COMMIT`. Это наиболее распространенный тип вложения 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 быть сохранена 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).
- ❑ 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. Позже будет показано, что существуют большие области виртуального пространства, которые были определены, но на которые еще не было ссылок (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, осуществляемые через API-функции оконного расширения адресов (Address Windowing Extension, AWE).

Для многих из этих позиций показатель подтверждения может быть показателем потенциального, но не фактического использования хранилища. Например, страница закрытой подтвержденной памяти фактически не занимает ни физическую страницу оперативной памяти, ни эквивалентное ей пространство страничного файла до тех пор, пока на нее не будет сделана хотя бы одна ссылка. Пока этого не случится, страница останется страницей, подлежащей заполнению нулями (см. далее). Тем не менее учет таких страниц в показателе подтверждения при первоначальном создании виртуального пространства ведется. Тем самым обеспечивается доступность для таких страниц физического пространства хранилища на тот случай, если позже на них будет сделана ссылка.

К разделу отображаемого файла, копируемого при записи, предъявляются такие же 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 трактуются 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 впоследствии наличие закрытого хранилища на тот случай, если к ним будут обращения, связанные 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. Но он учитывается для любых новых страниц с таблицами страниц, которые потребуются для хранения описания раздела, хотя этого раздела пока еще может и не быть. 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 осуществляет «подтверждение», чтобы нужное хранилище было доступно при возникновении в нем надобности, даже если на данный момент такой надобности нет. Таким образом, последующая ссылка на память в выделенном разделе никогда не получит отказа из-за нехватки пространства хранилища. (Отказ может произойти по другим причинам, например из-за защиты страницы, из-за освобождения раздела и т. д.) Сохранять такой порядок подтверждения позволяет механизм подсчета показателя подтверждения.

Показатель подтверждения фигурирует среди счетчиков монитора производительности под названием `Memory: Committed Bytes` (Память: Байт выделенной виртуальной памяти). Он также представлен первым из двух чисел на вкладке `Performance` (Быстродействие) диспетчера задач рядом с надписью `Commit` (Выделено) — второе число показывает лимит подтверждения. В программе `Process Explorer` показатель подтверждения выводится на вкладке `Memory` (Память) окна `System Information` (Системная информация) в области `Commit Charge` (Общий объем выделенной памяти) рядом с надписью `Current` (Текущее значение).

Квота процесса на страничный файл фигурирует в составе счетчиков монитора производительности под названием `Process: Page File Bytes` (Процесс: Байт файла подкачки). Те же самые данные выводятся в счетчике производительности `Process: Private Bytes` (Процесс: Байт исключительного пользования). (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. 10.14, позволяют исследовать показатель использования закрытой подтвержденной памяти, относящийся ко всей системе, к отдельному процессу или к отдельному страничному файлу.

Таблица 10.14. Счетчики производительности, относящиеся к подтвержденной памяти и страничным файлам

Счетчик производительности	Описание
Memory: Committed Bytes (Память: Байт выделенной виртуальной памяти)	Количество подтвержденных байтов виртуальной 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
Memory: Commit Limit (Память: Предел выделенной виртуальной памяти)	Количество байтов виртуальной памяти, которое может быть подтверждено без увеличения размера страничных файлов; если страничные файлы могут увеличиваться в размерах, этот лимит может меняться
Process: Page File Quota (Процесс: Байт файла подкачки)	Вклад процесса в показание счетчика Memory: Committed Bytes (Память: Байт выделенной виртуальной памяти)
Process: Private Bytes (Процесс: Байт исключительного пользования)	То же самое, что счетчик Process: Page File Quota (Процесс: Байт файла подкачки)
Process: Working Set—Private (Процесс: Рабочий набор (частный))	Составляющая показания счетчика Process: Page File Quota (Процесс: Байт файла подкачки), которая в данный момент находится в оперативной памяти и на которую можно сослаться без получения ошибки отсутствия страницы. Также является составляющей показания счетчика Process: Working Set (Процесс: Рабочий набор)
Process: Working Set (Процесс: Рабочий набор)	Составляющая показания счетчика Process: Virtual Bytes (Процесс: Байт виртуальной памяти), которая в данный момент находится в оперативной памяти и на которую можно сослаться без получения ошибки отсутствия страницы
Process: Virtual Bytes (Процесс: Байт виртуальной памяти)	Общий объем виртуальной памяти, выделенной процессу, включая отображаемые области, закрытые подтвержденные области и закрытые зарезервированные области
Paging File: % Usage (Файл подкачки: % использования)	Процент текущего используемого пространства страничного файла
Paging File: % Usage Peak (Файл подкачки: % использования (пик))	Наивысшее зафиксированное значение счетчика Paging File: % Usage (Файл подкачки: % использования)

Показатель подтверждения и размер страничного файла

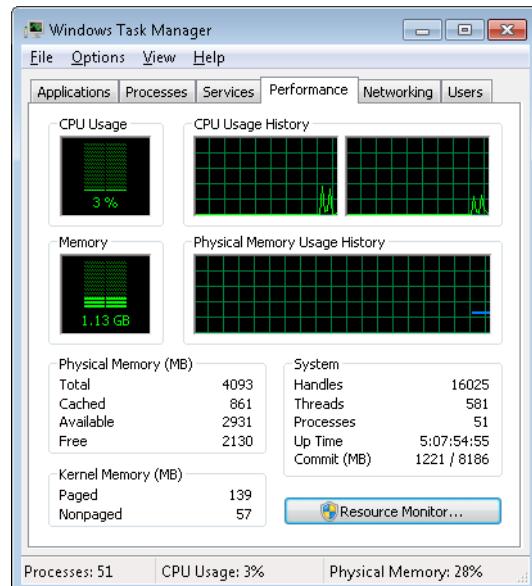
Счетчики, перечисленные в табл. 10.14, могут помочь в выборе нестандартного размера страничного файла. Политика, предлагаемая по умолчанию, основывается на объеме рабочей оперативной памяти, доступной большинству машин, но зависит от рабочей нагрузки при работе со страничным файлом, размер которого может быть неоправданно большим или малым.

Чтобы узнать, какой размер страничного файла действительно нужен 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 (Память) окна System Information (Системная информация). Это число представляет собой 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 на 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 Process: Private Bytes (Процесс: Байт исключительного пользования). 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».)

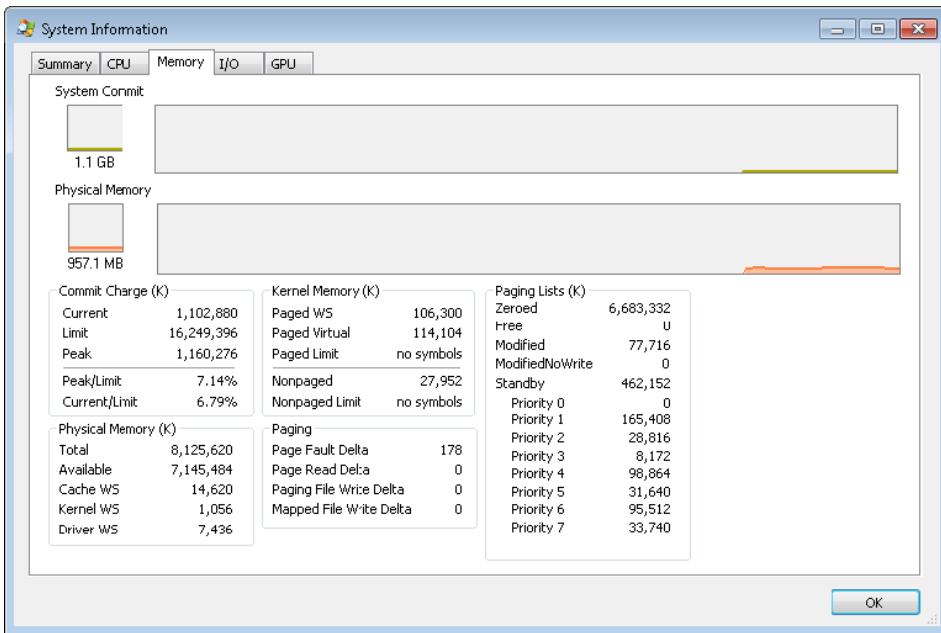
ЭКСПЕРИМЕНТ: ПРОСМОТР ПОКАЗАТЕЛЯ ИСПОЛЬЗОВАНИЯ СТРАНИЧНОГО ФАЙЛА С ПОМОЩЬЮ ДИСПЕТЧЕРА ЗАДАЧ

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 в нижней правой области под названием **System** (Система) 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, представляющий собой 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** (Системная информация) 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

У программных потоков при запуске должен быть доступ к временному хранилищу, в котором можно было бы хранить параметры функций, локальные переменные и адрес 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 книге уже описывался порядок использования стека при трассировке стека, а также порядок хранения в стеке структур данных исключений и прерываний, кроме того, уже рассказывалось, как системные

вызовы, системные прерывания и обычные прерывания заставляют программный поток переключаться с пользовательского стека на стек ядра. Теперь мы рассмотрим ряд дополнительных служб, предоставляемых диспетчером памяти для рационального использования пространства 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` 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 разрастается, достигая сторожевой страницы, выдается исключение, которое приводит к попытке выделения еще одной сторожевой страницы. Благодаря этому механизму пользовательский стек не занимает одновременно всю подтвержденную память объемом 1 Мбайт, а растет по мере необходимости. (Но обратно стек никогда не сжимается.)

ЭКСПЕРИМЕНТ: СОЗДАНИЕ МАКСИМАЛЬНОГО КОЛИЧЕСТВА ПОТОКОВ

При имеющемся пользовательском адресном пространстве размером лишь 2 Гбайт, доступном каждому 32-разрядному процессу, относительно большой объем, резервируемый для стека каждого программного потока, позволяет легко вычислить максимальное количество 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 «знает» о большом адресном пространстве). Если заставить каждый новый поток довольствоваться минимально возможным резервируемым значением стека в 64 Кбайт, этот предел может возрасти до почти 30 400 потоков, что можно самостоятельно проверить, воспользовавшись утилитой `TestLimit`, разработанной в Sysinternals. Пример выводимых этой утилитой данных:

```
C:\>testlimit -t
Testlimit - tests Windows limits
By Mark Russinovich

Creating threads ...
Created 30399 threads. Lasterror: 8
```

Если провести этот эксперимент для 64-разрядной версии Windows (где доступно 8 Тбайт пользовательского адресного пространства), можно рассчитывать на появление информации о потенциально возможных сотнях тысяч создаваемых потоков (если, конечно, для этого хватит памяти). Но на самом деле, что удивительно, 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 3 части I.) В результате у каждого потока будет иметься 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 пра-

вильные 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 с помощью программы 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, предназначающейся стеку ядра, значительно меньше: 12 Кбайт для платформы x86 и 16 Кбайт для платформы x64, далее следует сторожевая PTE-запись (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 содержит рекурсий, 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. Таким образом, Windows предоставляет механизм для динамического расширения и сужения стека ядра от его начального размера 16 Кбайт. 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 из стека в стек при приближении к сторожевой странице). Как показано на рис. 10.31, когда каждый вызов возвращает управление вызывавшему коду (разматывая клубок вызовов), диспетчер памяти освобождает дополнительно выделенный стек ядра.



Рис. 10.31. Переходы между стеками 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.

ЭКСПЕРИМЕНТ: ПРОСМОТР ПОКАЗАТЕЛЯ ИСПОЛЬЗОВАНИЯ СТЕКА ЯДРА

Для вывода на экран сведений о текущей занятости физической памяти стеками ядра можно воспользоваться программой MemInfo, разработанной в Winsider Seminars & Solutions. Ключ –и приведет к выводу сведений об использовании физической памяти для каждого компонента:

```
C:\>MemInfo.exe -u | findstr /i "Kernel Stack"  
Kernel Stack: 980 ( 3920 kb)
```

Обратите внимание на стек ядра после повторения предыдущего эксперимента с использованием утилиты TestLimit:

```
C:\>MemInfo.exe -u | findstr /i "Kernel Stack"  
Kernel Stack: 92169 ( 368676 kb)
```

Запуск утилиты TestLimit еще несколько раз приведет на 32-разрядной системе к быстрому истощению физической памяти, и это ограничение ведет к одному из основных общесистемных ограничений на количество 32-разрядных потоков.

DPC-стек

И наконец, для каждого процесса Windows содержит DPC-стек, доступный для использования системой при выполнении отложенных вызовов процедур (DPC). Благодаря этому подходу DPC-код изолируется от стека ядра текущего программного потока (не имеющего отношения к фактической операции DPC-вызова, поскольку DPC-вызовы выполняются в произвольном потоковом контексте). DPC-стек конфигурируется в качестве исходного стека для обслуживания в ходе системного вызова инструкции SYSENTER или SYSCALL. За переключение стека при выполнении инструкции SYSENTER или SYSCALL отвечает центральный процессор, основываясь на состоянии одного из зависящих от модели регистров (Model-Specific Register, MSR), но 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 оно потребуется.

Диспетчер памяти выполняет отложенное вычисление не только при доставке 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 превратится в напрасный труд. Вместо этого диспетчер памяти откладывает создание таблицы страниц до тех пор, пока поток не получит ошибку отсутствия 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* (Virtual Address Descriptors, VAD). Место для VAD-дескрипторов выделяется в невыгружаемом пуле.

Дескрипторы виртуальных адресов процесса

Для каждого процесса диспетчер памяти содержит набор VAD-дескрипторов с описаниями состояния 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. 10.32.

Когда процесс резервирует адресное пространство или отображает представление раздела, диспетчер памяти создает VAD-дескриптор для хранения любой информации, предоставленной запросом на выделение памяти, например диапазона зарезервированных адресов, 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 подтверждённых адресов, диспетчер памяти узнаёт, что поток не выделил память перед попыткой её использования, и поэтому генерирует нарушение прав доступа.



Рис. 10.32. Дескрипторы виртуальных адресов

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, как показано в следующем примере просмотра VAD-дерева для процесса, в котором запущено приложение Notepad.exe (Блокнот), указать этот адрес в команде !vad:

```

1kd> !process 0 1 notepad.exe
PROCESS 8718ed90 SessionId: 1 Cid: 1ea68 Peb:
7ffdf000 ParentCid: 0680
DirBase: ce2aa880 ObjectTable: ee6e01b0 HandleCount: 48.
Image: notepad.exe
VadRoot 865f10e0 VadCount 51 Clone 0 Private 210. Modified 0. Locked 0.

1kd> !vad 865f10e0
VAD level start end commit
8a05bf88 ( 6) 10 1f 0 Mapped READWRITE
88390ad8 ( 5) 20 20 1 Private READWRITE
87333740 ( 6) 30 33 0 Mapped READONLY
86d09d10 ( 4) 40 41 0 Mapped READONLY
882b49a0 ( 6) 50 50 1 Private READWRITE
...
Total VADs: 51 average level: 5 maximum depth: 6
  
```

Чередующиеся дескрипторы 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 кэша, диспетчер памяти реализует *челедующиеся дескрипторы виртуальных адресов* (rotate virtual address descriptors), позволяющие видеодрайверам осуществлять непосредственный перенос данных с помощью графического процессора (Graphical Processing Unit, GPU) и по мере надобности менять для страниц представления процесса не- нужную память на нужную. Пример того, как один и тот же виртуальный адрес может поочередно переходить между оперативной видеопамятью и виртуальной памятью, показан на рис. 10.33.

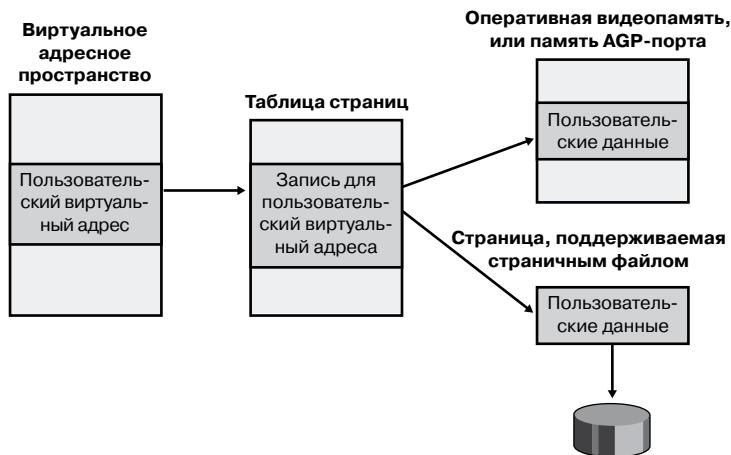


Рис. 10.33. Чередующиеся дескрипторы виртуальных адресов

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 получать преимущества от возможностей, предоставляемых технологией 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. Основываясь на этой информации, диспетчер памяти создает граф издержек доступа к узлу (к каковыим относится расстояние между узлом и другими узлами системы). Когда системе требуются страницы для заданной операции, она обращается к графу, чтобы выбрать наиболее оптимальный узел (то есть ближайший). Если на этом узле нет доступной памяти, выбирается следующий ближайший узел и т. д.

Хотя диспетчер памяти гарантирует, что, по возможности, память 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 выбрать собственный узел.

Идеальный узел используется, не только когда приложения выделяют память, но и в ходе операций ядра и ошибок отсутствия 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. Но точно так же, как и пользовательские приложения, указать собственный узел могут драйверы, используя API-функцию `MmAllocatePagesForMd1Ex` или `MmAllocateContiguousMemorySpecifyCacheNode`.

Оптимизацию с целью получения преимуществ NUMA-узлов проходят также различные пулы и структуры данных диспетчера памяти. 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 отсутствия 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 11.) Объекты разделов можно также использовать для отображения файла внутри адресного пространства процесса. После этого обращаться к файлу можно будет 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 отсутствия 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. 10.34.

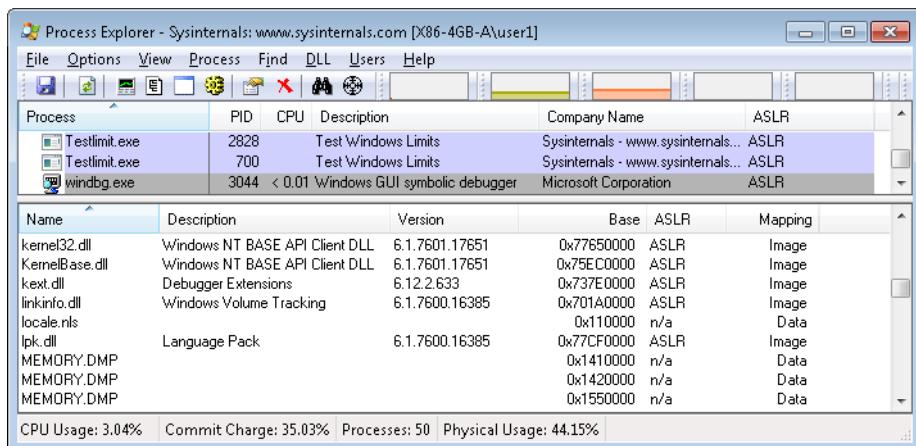


Рис. 10.34. Объект раздела

ЭКСПЕРИМЕНТ: ПРОСМОТР ОБЪЕКТОВ РАЗДЕЛОВ

С помощью программы Object Viewer (Winobj.exe производства Sysinternals) можно увидеть список именованных разделов. Получить список открытых дескрипторов объектов разделов можно с помощью любой программы, упомянутой в разделе «Диспетчер объектов» главы 3 части I. (Как отмечено в главе 3, эти имена хранятся в каталоге \Sessions\x\BaseNamedObjects диспетчера объектов, где x означает соответствующий каталог сеанса.) Неименованные объекты разделов остаются невидимыми.

Как вы знаете, для просмотра файлов, отображаемых на процесс, можно воспользоваться программой Process Explorer, созданной в Sysinternals. Выберите в меню команду View ▶ Lower Pane View (Вид ▶ Вид нижней панели) и установите флашок DLLs (DLL-библиотеки), затем выберите команду View ▶ Select Columns (Вид ▶ Выбрать столбцы) и установите флашок Mapping Type (Отображаемый тип). Файлы с пометкой «Data» в столбце Mapping (Отображение) являются отображаемыми (а не DLL-библиотеками или другими файлами, которые загрузчик образов загружает в виде модулей). Ранее мы уже видели этот пример.



Уникальные атрибуты, хранящиеся в объектах разделов, перечислены в табл. 10.15.

Таблица 10.15. Атрибуты тела объекта раздела

Атрибут	Назначение
Максимальный размер (Maximum size)	Наибольший размер, до которого может дорастти раздел, указанный в байтах; если объект используется для отображаемого файла, максимальный размер является размером этого файла
Защита страниц (Page protection)	Страницчная защита памяти, назначаемая всем страницам раздела при его создании
Страницочный или отображаемый файл (Paging file/Mapped file)	Показывает, создан ли раздел пустым (будучи поддерживаемым страниценным файлом – как уже отмечалось, разделы, поддерживаемые страницными файлами, используют ресурсы этого файла, только когда страницы нужно записать на диск) или загружен с файлом (будучи поддерживаемым отображаемым файлом)

Атрибут	Назначение
Базовый или небазовый (Based/Not based)	Показывает, является ли раздел базовым, который должен появляться в одном и том же виртуальном адресном пространстве для всех совместно использующих его процессов, или небазовым, который может появляться по разным виртуальным адресам для разных процессов

Структуры данных, поддерживаемые диспетчером памяти и описывающие отображаемые разделы, показаны на рис. 10.35. Эти структуры гарантируют согласованность данных, считываемых из отображаемых файлов, независимо от типа доступа (открытый файл, отображаемый файл и т. д.).



Рис. 10.35. Внутренние структуры разделов

Для каждого открытого файла (представленного файловым объектом) существует отдельная структура *указателей объекта раздела* (section object pointers). Эта структура является ключевой в плане поддержания согласованности данных для всех типов доступа к файлам, а также для предоставления возможности 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), выделенного 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, на которые была сделана ссылка.

Хотя Windows гарантирует, что любой процесс, обращающийся к файлу (по чтению или записи), всегда будет видеть одни и те же согласованные данные, есть один случай, когда 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 затем, 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. Поскольку 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 осуществилаброс 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 файлового объекта. Этот адрес можно получить с помощью отладчика ядра путем вывода дампа таблицы дескрипторов процессов, запустив команду !handle и обозначив адрес файлового объекта. Еще одна команда !file отладчика ядра приводит к выводу основной информации о файловом объекте, но не показывает указатель на структуру указателей объекта раздела. Затем, используя команду dt, нужно отформатировать файловый объект, 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 11) 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. (Сведения об использовании команды !handle можно найти в разделе «Диспетчер объектов» главы 3 части I.)

```
1kd> !handle 1 f 86f57d90 File
.
.
0324: Object: 865d2768 GrantedAccess: 00120089 Entry: c848e648
Object: 865d2768 Type: (8475a2c0) File
ObjectHeader: 865d2750 (old version)
HandleCount: 1 PointerCount: 1
Directory Object: 00000000 Name:
\Users\Administrator\Documents\Downloads\
SVR-T331_WH07 (1).pptx {HarddiskVolume3}
```

Взяв адрес файлового объекта (865d2768) и отформатировав его командой dt, можно получить следующий результат:

```
1kd> dt nt!_FILE_OBJECT 865d2768
+0x000 Type : 5
+0x002 Size : 128
+0x004 DeviceObject : 0x84a62320 _DEVICE_OBJECT
+0x008 Vpb : 0x84a60590 _VPB
+0x00c FsContext : 0x8cee4390
+0x010 FsContext2 : 0xbff910c80
+0x014 SectionObjectPointer : 0x86c45584 _SECTION_OBJECT_POINTERS
```

Затем, взяв адрес структуры указателей объекта раздела (0x86c45584) и отформатировав его командой dt, можно получить такой результат:

```
1kd> dt 0x86c45584 nt!_SECTION_OBJECT_POINTERS
+0x000 DataSectionObject : 0x863d3b00
+0x004 SharedCacheMap : 0x86f10ec0
+0x008 ImageSectionObject : (null)
```

И наконец, воспользовавшись адресом, можно запустить команду !ca и вывести на экран область управления:

```
1kd> !ca 0x863d3b00

ControlArea @ 863d3b00
Segment b1de9d48 Flink 00000000 Blink 8731f80c
Section Ref 1 Pfn Ref 48 Mapped Views 2
User Ref 0 WaitForDel 0 Flush Count 0
File Object 86cf6188 ModWriteCount 0 System Views 2
WritableRefs 0
Flags (c080) File WasPurged Accessed

No name for file

Segment @ b1de9d48
ControlArea 863d3b00 ExtendInfo 00000000
Total Ptes 100
Segment Size 100000 Committed 0
Flags (c0000) ProtectionMask

Subsection 1 @ 863d3b48
ControlArea 863d3b00 Starting Sector 0 Number Of Sectors 100
Base Pte bf85e008 Ptes In Subsect 100 Unused Ptes 0
Flags d Sector Offset 0 Protection 6
Accessed
Flink 00000000 Blink 8731f87c MappedViews 2
```

Еще один прием вывода списка всех областей управления предполагает использование команды !memusage. Следующий фрагмент взят из выводимых этой командой данных:

```
1kd> !memusage
loading PFN database
loading (100% complete)
Compiling memory usage data (99% Complete).
Zeroed: 2654 ( 10616 kb)
Free: 584 ( 2336 kb)
Standby: 402938 (1611752 kb)
Modified: 12732 ( 50928 kb)
ModifiedNowrite: 3 ( 12 kb)
Active/Valid: 431478 (1725912 kb)
Transition: 1186 ( 4744 kb)
Bad: 0 ( 0 kb)
Unknown: 0 ( 0 kb)
TOTAL: 851575 (3406300 kb)
Building kernel map
Finished building kernel map
Scanning PFN database - (100% complete)

Usage Summary (in Kb):
Control Valid Standby Dirty Shared Locked PageTables name
86d75f18 0 64 0 0 0 0 mapped_file(
netcfgx.dll )
8a124ef8 0 4 0 0 0 0 No Name for File
8747af80 0 52 0 0 0 0 mapped_file(
iebrshim.dll )
883a2e58 24 8 0 0 0 0 mapped_file(
WINWORD.EXE )
86d6ea0 0 16 0 0 0 0 mapped_file(
oem13.CAT )
```

```
84b19af8 8 0 0 0 0 0 No Name for File
b1672ab0 4 0 0 0 0 0 No Name for File
88319da8 0 20 0 0 0 0 mapped_file(
Microsoft-Windows-MediaPlayer
Package~31bf3856ad364e35~x86~en-US~6.0.6001.18000.cat )
8a04db00 0 48 0 0 0 0 mapped_file(
eapahost.dll )
```

В столбце Control (Управление) содержится указатель на структуру области управления, описывающую отображаемый файл. С помощью команды !ca отладчика ядра можно вывести на экран области управления, сегменты и подразделы. Например, для вывода дампа области управления отображаемого файла Winword.exe в этом примере введите команду !ca с числом, указанным в столбце Control (Управление):

```
lkd> !ca 883a2e58

ControlArea @ 883a2e58
Segment ee613998 Flink 00000000 Blink 88a985a4
Section Ref 1 Pfn Ref 8 Mapped Views 1
User Ref 2 WaitForDel 0 Flush Count 0
File Object 88b45180 ModWriteCount 0 System Views ffff
WritableRefs 80000006
Flags (40a0) Image File Accessed

File: \PROGRA~1\MICROS~1\Office12\WINWORD.EXE
Segment @ ee613998
ControlArea 883a2e58 BasedAddress 2f510000
Total Ptes 57
Segment Size 57000 Committed 0
Image Commit 1 Image Info ee613c80
ProtoPtes ee6139c8
Flags (2000) ProtectionMask

Subsection 1 @ 883a2ea0
ControlArea 883a2e58 Starting Sector 0 Number Of Sectors 2
Base Pte ee6139c8 Ptes In Subsect 1 Unused Ptes 0
Flags 2 Sector Offset 0 Protection 1

Subsection 2 @ 883a2ec0
ControlArea 883a2e58 Starting Sector 2 Number Of Sectors a
Base Pte ee6139d0 Ptes In Subsect 2 Unused Ptes 0
Flags 6 Sector Offset 0 Protection 3

Subsection 3 @ 883a2ee0
ControlArea 883a2e58 Starting Sector c Number Of Sectors 1
Base Pte ee6139e0 Ptes In Subsect 1 Unused Ptes 0
Flags a Sector Offset 0 Protection 5

Subsection 4 @ 883a2f00
ControlArea 883a2e58 Starting Sector d Number Of Sectors 28b
Base Pte ee6139e8 Ptes In Subsect 52 Unused Ptes 0
Flags 2 Sector Offset 0 Protection 1

Subsection 5 @ 883a2f20
ControlArea 883a2e58 Starting Sector 298 Number Of Sectors 1
Base Pte ee613c78 Ptes In Subsect 1 Unused Ptes 0
Flags 2 Sector Offset 0 Protection 1
```

Программа проверки драйверов

Представленная в главе 8 программа Driver Verifier, предназначенная для проверки драйверов, может использоваться для облегчения поиска и изоляции наиболее часто встречающихся ошибок в драйвере устройства или в другом системном коде режима ядра. В данном разделе рассматриваются возможности программы Driver Verifier, относящиеся к управлению памятью (возможности, относящиеся к драйверам устройств, рассматриваются в главе 8).

Параметры верификации хранятся в разделе `HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management` реестра. В параметре `VerifyDriverLevel` содержится поразрядная маска, представляющая подключенные варианты верификации. В параметре `VerifyDrivers` содержатся имена проверяемых драйверов. (Этого параметра не будет в реестре, пока в диспетчере проверки драйверов не выбраны проверяемые драйверы.) Для проверки всех драйверов значением параметра `VerifyDrivers` должен быть символ «звездочка» (*). В зависимости от заданных параметров для выбранных вариантов проверки может потребоваться перезагрузка системы.

На ранней стадии начальной загрузки диспетчер памяти считывает параметры реестра, относящиеся к программе проверки драйверов, чтобы определить, какие драйверы нужно проверять и какие инструменты этой программы включены. (Следует учесть, что при перезагрузке в безопасном режиме все параметры программы проверки драйверов игнорируются.) Впоследствии, если для проверки будет выбран хотя бы один драйвер, ядро проверит имя каждого драйвера устройства, загружаемого им в память, и сравнил его со списком драйверов, выбранных для проверки. Для каждого драйвера устройства, появляющегося в обоих местах, ядро вызывает функцию `VfLoadDriver`, которая вызывает другие функции вида `Vf*` для замены принадлежащих драйверам ссылок на ряд функций ядра ссылками на эквиваленты этих функций в программе проверки драйверов. Например, вызов функции `ExAllocatePool` заменяется вызовом функции `VerifierAllocatePool`. Драйвер системы управления окнами (`Win32k.sys`) также делает такие замены для использования эквивалентных функций из программы проверки драйверов.

После обзора порядка настройки драйвера рассмотрим шесть возможных параметров проверки, имеющих отношение к памяти, которые могут быть применены к драйверам устройств: **Special Pool** (Особый пул), **Pool Tracking** (Слежение за пулом), **Force IRQL Checking** (Обязательная проверка IRQL), **Low Resources Simulation** (Имитация нехватки ресурсов), **Miscellaneous Checks** (Прочие проверки) и **Automatic Checks** (Автоматические проверки).

Особый пул. Выбор этого параметра заставляет процедуры выделения пулза ключать выделенные пулы в недостоверные страницы, чтобы ссылки до и после выделенного в памяти места вызывали нарушение прав доступа в режиме ядра, приводя тем самым к отказу системы с указанием на сбойный драйвер. Выбор параметра **Special Pool** (Особый пул) также становится причиной проведения ряда дополнительных проверок, когда драйвер выделяет или освобождает память.

Когда установлен параметр **Special Pool** (Особый пул), процедуры выделения пулза выделяют область памяти ядра программе Driver Verifier. Программа Driver Verifier,

в свою очередь, перенаправляет запросы на выделение памяти, которые выдают проверяемые драйверы, в область особого пула, а не в область стандартных пулов памяти режима ядра. Когда драйвер устройства выделяет память из особого пула, Driver Verifier выравнивает область выделения по границе страницы. Поскольку Driver Verifier заключает выделенную страницу в недостоверные страницы, при попытке драйвера устройства провести чтение или запись с переходом границы буфера, драйвер обращается к недостоверной странице, и диспетчер памяти выдаст ошибку нарушения прав доступа в режиме ядра.

На рис. 10.36 показан пример буфера особого пула, который программа Driver Verifier выделяет драйверу устройства, когда она проверяет наличие ошибок выхода за границы пула.



Рис. 10.36. Схема выделения особого пула

По умолчанию программа Driver Verifier настроена на обнаружение выхода за верхнюю границу выделенной памяти. Это делается путем размещения буфера, используемого драйвером устройства, в конце выделенной страницы и заполнения начала страницы данными, выбранными случайным образом. Хотя диспетчер проверки драйверов не позволяет задать режим обнаружения выхода за нижнюю границу, это можно сделать самостоятельно, добавив DWORD-параметр реестра в раздел `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\PoolTagOverruns` и назначив ему значение 0 (или запустив утилиту GFlags и установив переключатель Verify Start вместо устанавливаемого по умолчанию переключателя Verify End). Когда Windows заставляют обнаружить выход за нижнюю границу, Driver Verifier выделяет буфер драйвера в начале страницы, а не в ее конце.

Настройка на обнаружение выхода за верхнюю границу в какой-то степени подразумевает и обнаружение выхода за нижнюю границу. Когда драйвер освобождает свой буфер для возвращения памяти программе Driver Verifier, она убеждается в том, что данные, выбранные случайным образом и предшествующие буферу, не изменились. Если эти данные изменились, значит, драйвер устройства перешел нижнюю границу буфера и произвел запись в память за его пределами.

При выделении особого пула проверяется также допустимость процессорного IRQL-уровня при выделении и освобождении памяти. Эта проверка позволяет выявить ошибку, допускаемую некоторыми драйверами устройств: выделение выгружаемой памяти из IRQL-уровня DPC/dispatch или выше.

Режим проверки Special Pool (Особый пул) можно настроить самостоятельно, добавив DWORD-параметр реестра в раздел HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\PoolTag. Он представляет теги выделения, используемые системой для особого пула. Таким образом, даже если программа Driver Verifier не настроена на проверку конкретного драйвера устройства, но тег, который связывает драйвер с выделяемой им памятью, совпадает с указанным в параметре PoolTag реестра, процедура выделения пула выделит память из особого пула. Если установить для параметра PoolTag значение `0x00000002a` или групповой символ (*), то вся память, которую выделяет этот драйвер, будет выделена из особого пула при условии достаточного пространства виртуальной и физической памяти. (При дефиците свободных страниц драйверы вернутся к выделению памяти из обычного пула — при существующих ограничениях в каждой операции выделения участвуют две страницы.)

Следжение за пулом. Если установлен параметр Pool Tracking (Следжение за пулом), диспетчер памяти проверяет время выгрузки драйвера и факт освобождения всей выделенной им памяти. Если этого не произойдет, он выполнит аварийную остановку системы, указывая на проблемный драйвер. Программа Driver Verifier показывает также общую статистику на вкладке Pool Tracking (Следжение за пулом). Можно также воспользоваться командой `!Verifier` отладчика ядра. Эта команда показывает больше информации, чем программа Driver Verifier, что может быть полезно разработчикам драйверов.

В режимах слежения за пулом и особого пула учитываются не только явные вызовы, связанные с выделением памяти, например `ExAllocatePoolWithTag`, но и вызовы других API-функций ядра, которые выделяют пул неявным образом: `IoAllocateMdl`, `IoAllocateIrp` и другие IRP-вызовы, связанные с выделением памяти; различные API-функции, в названиях которых есть строка `Rtl`; а также вызовы функции `IoSetCompletionRoutineEx`.

Еще одна функция проверки драйвера, используемая при установке параметра Pool Tracking (Следование за пулом), имеет отношение к сбору квот пула. Вызов функции `ExAllocatePoolWithTagQuota` приводит к сбору квот пула текущего процесса в виде количества выделенных байтов. Если подобный вызов осуществляется из DPC-процедуры, исследуемый процесс предсказать невозможно, потому что DPC-процедуры могут выполняться в контексте любого процесса. При установленном параметре Pool Tracking (Следование за пулом) проводится проверка вызовов этой процедуры из контекста DPC-процедуры.

Программа Driver Verifier может также отслеживать заблокированные страницы памяти, что дополнительно приводит к проверке страниц, оставшихся заблокированными после операции ввода-вывода. При этом генерируется аварийный код `DRIVER_LEFT_LOCKED_PAGES_IN_PROCESS` (драйвер оставил в процессе заблокированные страницы) вместо кода `PROCESS_HAS_LOCKED_PAGES` (процесс заблокировал страницы) — первый указывает на ответственность драйвера за ошибку, а также на ответственность функции за блокировку страниц.

Обязательная проверка IRQL-уровня. Одна из наиболее распространенных ошибок драйвера устройства происходит, если драйвер обращается к выгружаемым данным или коду в условиях, когда процессор, на котором выполняется код драйвера устройства, имеет повышенный IRQL-уровень. Как упоминается в главе 3 части I, диспетчер памяти не может обработать ошибку отсутствия страницы для IRQL-уровня DPC/dispatch или выше. Система зачастую не обнаруживает экземпляры драйвера устройств, обращающиеся к выгружаемым данным, когда процессор выполняет код на высоком IRQL-уровне, поскольку может случиться так, что обращение к выгружаемым данным осуществляется в то время, когда они находятся в физической памяти. Однако в другие моменты времени данные могут быть выгружены, что приводит к аварии системы с кодом останова IRQL_NOT_LESS_OR_EQUAL (то есть IRQL-уровень не был ниже или равен тому уровню, который требуется для проведения операции — в данном случае операции обращения к выгружаемой памяти).

Хотя проверка драйверов устройств на подобный дефект обычно дается с трудом, программа Driver Verifier упрощает задачу. При установке параметра Force IRQL Checking (Обязательная проверка IRQL) Driver Verifier заставляет весь выгружаемый код и данные режима ядра, не входящие в системный рабочий набор, поднимать IRQL-уровень в ходе проверки драйвера устройства. Внутренней функцией, решающей эту задачу, является MiTrimAllSystemPagableMemory. При установке данного параметра каждое обращение проверяемого драйвера устройства к выгружаемой памяти при повышенном IRQL-уровне ведет к тому, что система немедленно обнаруживает нарушение прав доступа, а получающаяся в итоге авария системы указывает на дефектный драйвер.

Еще одна распространенная авария драйвера из-за использования неправильного IRQL-уровня имеет место, когда объекты синхронизации являются частью структуры данных, которая была выгружена и в отношении которой осуществляется ожидание. Объекты синхронизации никогда не должны выгружаться, потому что диспетчеру нужен доступ к ним при повышенном IRQL-уровне, а невыполнение этого требования ведет к аварии. Программа Driver Verifier проверяет наличие в выгружаемой памяти структур KTIMER, KMUTEX, KSPIN_LOCK, KEVENT, KSEMAPHORE, ERESOURCE и FAST_MUTEX.

Имитация нехватки ресурсов. Установка параметра Low Resources Simulation (Имитация нехватки ресурсов) заставляет программу Driver Verifier случайным образом выдавать отказ на выделение памяти, проверяя тем самым работу драйверов устройств. В прежние времена разработчики создавали многочисленные драйверы устройств, предполагая, что память ядра будет доступна всегда, а если память закончится, драйверу устройства не придется заниматься этой проблемой, поскольку система все равно потерпит крах. Но поскольку условия дефицита памяти время от времени все же могут возникать, важно, чтобы драйверы устройств умели справляться с ошибками выделения памяти, свидетельствующими о дефиците памяти ядра.

В вызовы драйвера, в которые внедряются случайные отказы, включаются такие функции, как ExAllocatePool*, MmProbeAndLockPages, MmMapLockedPagesSpecifyCache, MmMapIoSpace, MmAllocateContiguousMemory, MmAllocatePagesForMdl, IoAllocateIrp, IoAllocateMdl, IoAllocateWorkItem, IoAllocateErrorLogEntry и IOSetCompletionRoutineEx, а также различные API-функции, в названиях которых есть строка Rt1 и которые занимаются выделением пула. Кроме того, можно указать на возможность того, что выделение памяти завершится отказом (по умолчанию это 6 % случаев),

какие приложения должны стать субъектом моделирования (по умолчанию — все), какие признаки пула должны быть затронуты (по умолчанию — все) и какая задержка должна быть использована перед инициированием отказа (по умолчанию — 7 минут после начальной загрузки системы, чего должно хватить на прохождение критического инициализационного периода, когда дефицит памяти может помешать загрузке драйвера устройства).

По истечении задержки программа Driver Verifier приступает к произвольному инициированию отказов в выделении памяти для проверяемых драйверов устройств. Если драйвер не может правильно обработать отказ в выделении памяти, то это, скорее всего, выразится в отказе системы.

Прочие проверки. Некоторые виды проверок, которые в программе Driver Verifier называются прочими, позволяют ей обнаруживать освобождение определенных системных структур в пуле, который остается активным.

- ❑ Наличие в освобожденной памяти активных рабочих элементов (драйвер вызывает функцию `ExFreePool` для освобождения блока пула, в котором имеются один и более рабочих элементов, поставленных в очередь функцией `ToQueueWorkItem`).
- ❑ Наличие в освобожденной памяти активных ресурсов (драйвер вызывает функцию `ExFreePool` раньше функции `ExDeleteResource` для уничтожения объекта `ERESOURCE`).
- ❑ Наличие в освобожденной памяти активных ассоциативных списков (драйвер вызывает функцию `ExFreePool` раньше функции `ExDeleteNPagedLookasideList` или функции `ExDeletePagedLookasideList` для удаления ассоциативного списка).

И наконец, когда включен режим проверки, программа Driver Verifier выполняет определенные виды проверок в автоматическом режиме. Эти виды проверок невозможно отдельно включать или выключать, они выполняются всегда.

- ❑ Вызов для списка дескрипторов памяти (MDL), имеющего неправильные флаги, функции `MmProbeAndLockPages` или `MmProbeAndLockProcessPages`. Например, неверно вызывать функцию `MmProbeAndLockPages` для MDL-списка, установленного вызовом функции `MmBuildMdlForNonPagedPool`.
- ❑ Вызов в отношении MDL-списка, имеющего неправильные флаги, функции `MmMapLockedPages`. Например, неверно вызывать функцию `MmMapLockedPages` для MDL-списка, который уже отображен на системный адрес. Другим примером неправильного поведения драйвера является вызов функции `MmMapLockedPages` для MDL-списка, который не был заблокирован.
- ❑ Вызов в отношении неполного MDL-списка (созданного вызовом функции `IoBuildPartialMdl`) функции `MmUnlockPages` или `MmUnmapLockedPage`.
- ❑ Вызов в отношении MDL-списка, не отображенного на системный адрес, функции `MmUnmapLockedPages`.
- ❑ Выделение мест из невыгружаемого пула сеанса для таких объектов синхронизации, как события и мьютексы.

Программа Driver Verifier является весьма ценным дополнением к арсеналу средств проверки и отладки, доступных создателям драйверов. Многие драйверы устройств

имеют дефекты, которые Driver Verifier в состоянии выявить, поэтому программа Driver Verifier способствует общему повышению качества всего кода режима ядра, выполняемого в Windows.

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. Состояния страниц перечислены в табл. 10.16.

Таблица 10.16. 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 фигурирует 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)
Ожидаящая 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

продолжение ↗

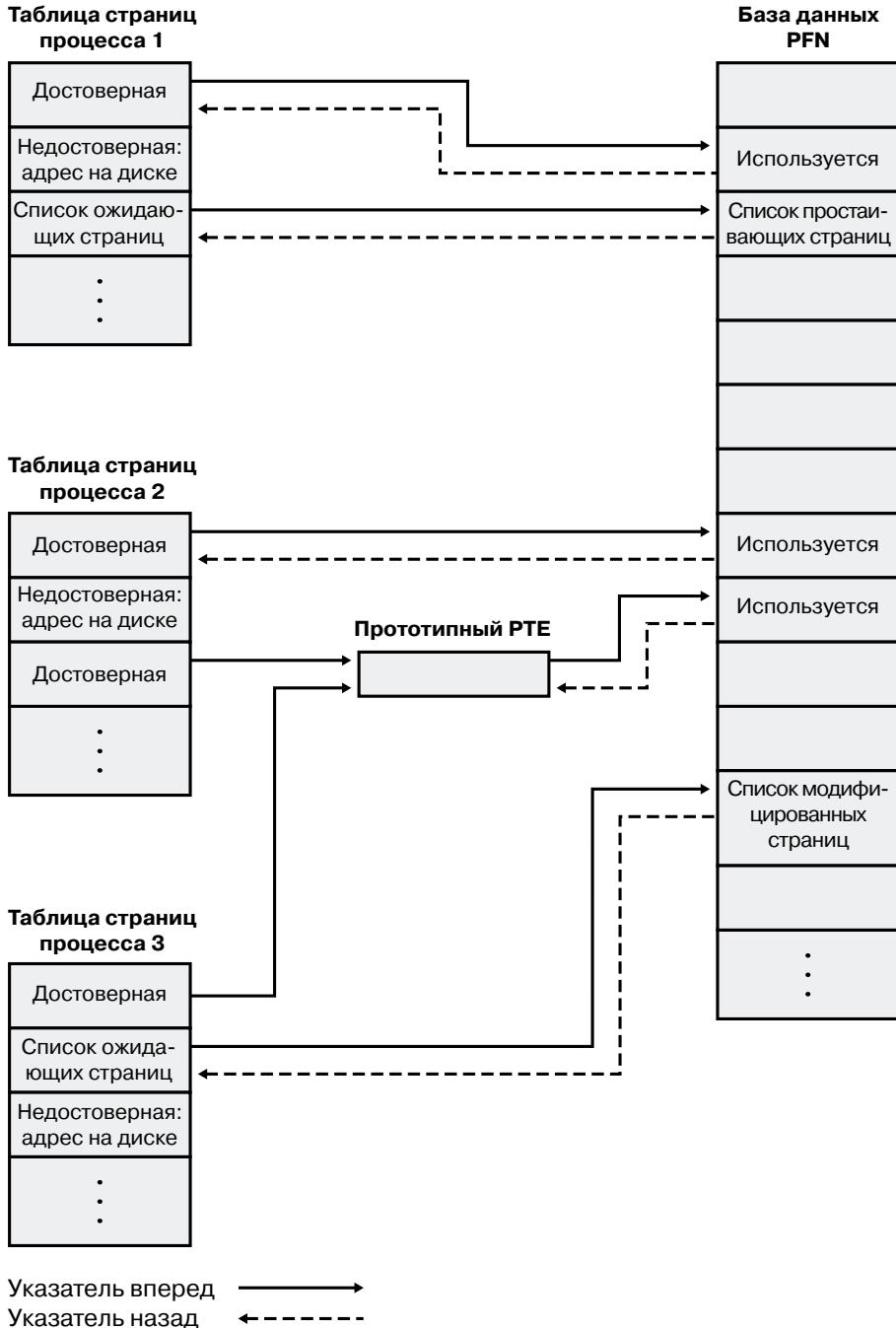
Таблица 10.16 (продолжение)

Состояние	Описание
Измененная, но не подлежащая записи страница (modified no-write)	То же самое, что и измененная страница, но эта страница была помечена 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 12)
Свободная страница (free)	Страница свободна, но имеет неопределенные 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, предназначеннная 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

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. 10.37. 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.

Из тех состояний страниц, которые перечислены в табл. 10.16, шесть состояний организованы в связанные (однонаправленные) 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 ожидания фактически ассоциируется с восемью различными списками, выстроенными по приоритетам (о приоритетах страниц речь пойдет в этом разделе чуть позже). На рис. 10.38 показан пример того, как эти списки связываются друг с другом.

Из следующего раздела вы узнаете, как эти связанные списки используются для разрешения ошибок отсутствия страниц и как страницы заносятся в различные списки и удаляются из них.

**Рис. 10.37.** Таблицы страниц и база данных номеров страничных блоков

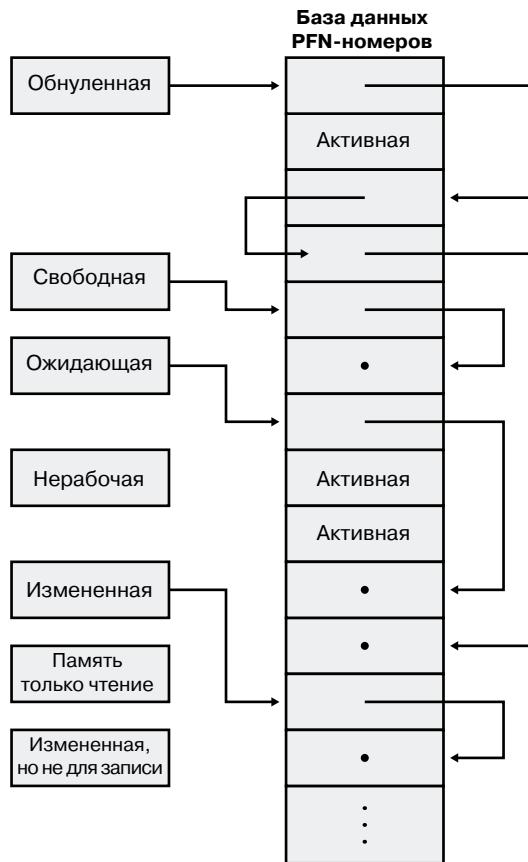


Рис. 10.38. Списки страниц в базе данных PFN-номеров

ЭКСПЕРИМЕНТ: ПРОСМОТР БАЗЫ ДАННЫХ PFN-НОМЕРОВ

Для вывода дампа с размерами различных страничных списков можно воспользоваться программой MemInfo, разработанной в Winsider Seminars & Solutions, с ключом -s. Выводимые данные выглядят следующим образом:

```
C:\>MemInfo.exe -s
```

```
MemInfo v2.10 - Show PFN database information
Copyright (C) 2007-2009 Alex Ionescu
www.alex-ionescu.com
```

```
Initializing PFN Database... Done
```

```
PFN Database List Statistics
Zeroed: 487 ( 1948 kb)
Free: 0 ( 0 kb)
Standby: 379745 (1518980 kb)
```

Modified: 1052 (4208 kb)
ModifiedNoWrite: 0 (0 kb)
Active/Valid: 142703 (570812 kb)
Transition: 184 (736 kb)
Bad: 0 (0 kb)
Unknown: 2 (8 kb)
TOTAL: 524173 (2096692 kb)

Используя команду `!memusage` отладчика ядра, можно получить такую же информацию, хотя это займет существенно больше времени и потребует начальной загрузки в режиме отладки.

Динамика списков страниц

На рис. 10.39 показана диаграмма состояний переходов страницных блоков. Чтобы ее не усложнять, список измененных, но не подлежащих записи страниц не показан.

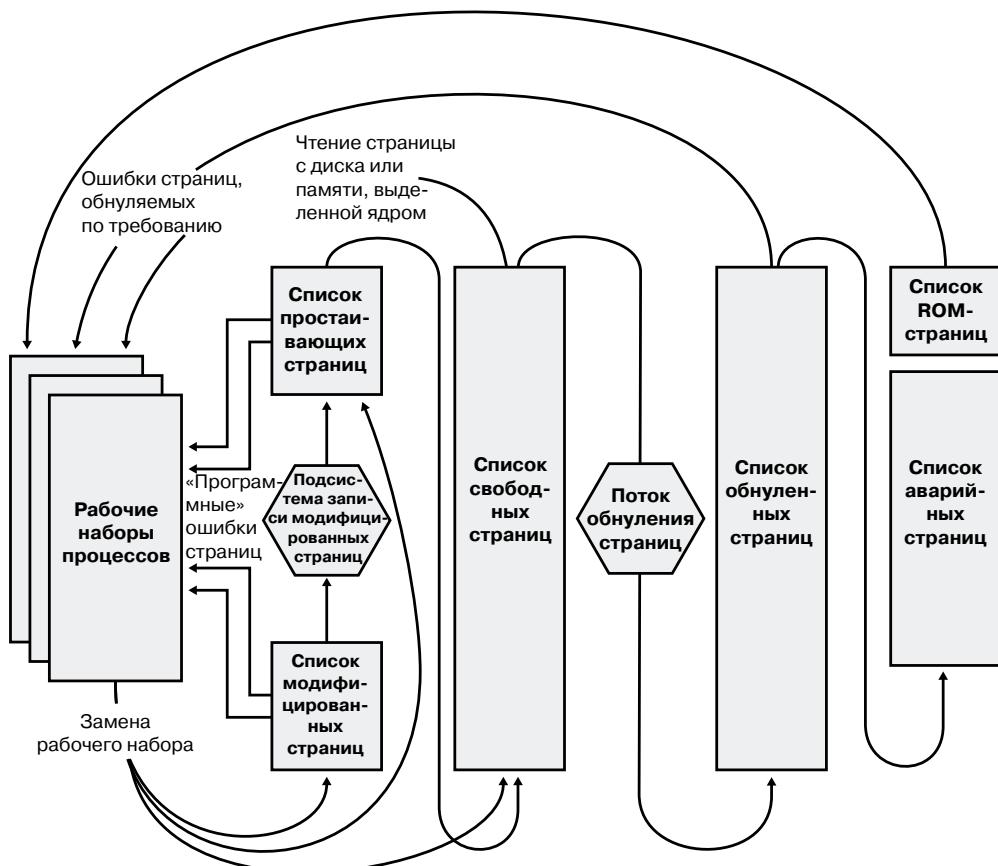


Рис. 10.39. Диаграмма состояний страничных блоков

Страницевые блоки перемещаются между страницевыми списками следующими путями:

- ❑ Когда диспетчеру памяти для обслуживания ошибки отсутствия страницы, связанной с требованием обнуленной страницы (ссылка на страницу, которая определена как полностью заполненная нулями, или на закрытую подтвержденную страницу пользователяского режима, к которой еще не было обращений), нужна страница, заполненная нулями, сначала предпринимается попытка получить такую страницу из списка таких страниц. Если список пуст, берется страница 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, касающихся общих критериев (Common Criteria). Большинство положений общих критериев указывают на то, что процессы пользовательского режима должны получать обнуленные страницевые блоки, чтобы не дать им прочитать содержимое памяти предыдущих процессов. Поэтому диспетчер памяти предоставляет процессам пользовательского режима обнуленные страницевые блоки, если только страница не была считана из резервного хранилища. В таком случае диспетчер памяти использует необнуленные страницевые 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 наименьшим приоритетом, который может быть установлен для пользовательского потока, является 1.

ПРИМЕЧАНИЕ

Поскольку потоку обнуления страниц приходится ждать объекта диспетчера событий, его приоритет повышается (см. раздел «Повышение приоритета» в главе 5 части I), что приводит к его выполнению с приоритетом 1, по крайней мере, на какое-то время. Это ошибка текущей реализации.

ПРИМЕЧАНИЕ

Когда память в результате выделения физической страницы драйвером, вызывающим функцию MmAllocatePagesForMdl или MmAllocatePagesForMdlEx, должна быть заполнена нулями Wndows-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 РТЕ-записи (или из прототипной РТЕ-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 РТЕ-записи для общих страниц), диспетчер памяти может быстро найти РТЕ-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 резидентно в физической памяти.
- ❑ Когда процесс завершает работу, все закрытые страницы переходят в список свободных страниц. Кроме того, если закрыта последняя ссылка на раздел, поддерживаемый страничным файлом, и в разделе не осталось отображенных представлений, страницы этого раздела также попадают в список свободных страниц.

ЭКСПЕРИМЕНТ: ПРОСМОТР СПИСКОВ СВОБОДНЫХ И ОБНУЛЕННЫХ СТРАНИЦ

Освобождение 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:\temp>testlimit -d 1 -c 800

Testlimit v5.1 - test Windows limits
Copyright (C) 2012 Mark Russinovich
Sysinternals - www.sysinternals.com

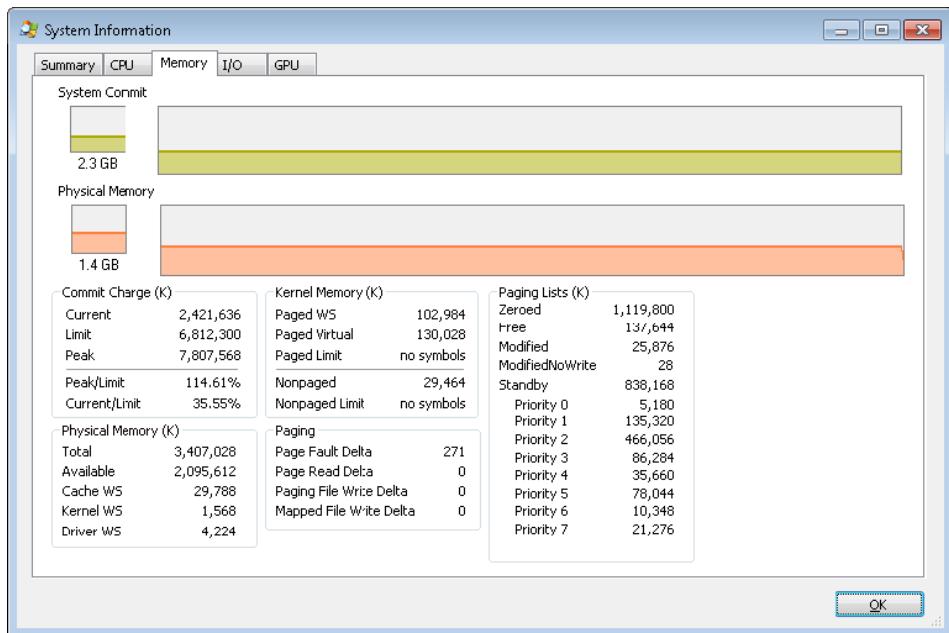
Leaking private bytes 1 MB at a time ...
Leaked 800 MB of private memory (800 MB total leaked). Lasterror: 0
The operation completed successfully.
```

Ключ -d заставляет TestLimit не только выделить память в качестве закрытой и подтверждённой, но и «прикоснуться» к ней, то есть обратиться к этой памяти. Это приводит к выделению физической памяти и присвоению ее процессу, чтобы освободить область закрытой подтверждённой виртуальной памяти. Если в системе имеется достаточный объем доступной оперативной памяти, для процесса в оперативной памяти выделяется целых 800 Мбайт.

Теперь этот процесс будет ждать, пока вы не заставите его совершить выход или завершить работу (возможно, с помощью комбинации клавиш Ctrl+C в его командном окне). Откройте Process Explorer и выберите команду View ▶ System Information (Вид ▶ Системная

информация). Обратите внимание на размеры списков Free (Свободные) и Zeroed (Обнуленные).

Теперь прекратите работу процесса TestLimit или заставьте его совершить выход. Возможно, вам удастся увидеть, что список свободных страниц кратковременно увеличился в размере.



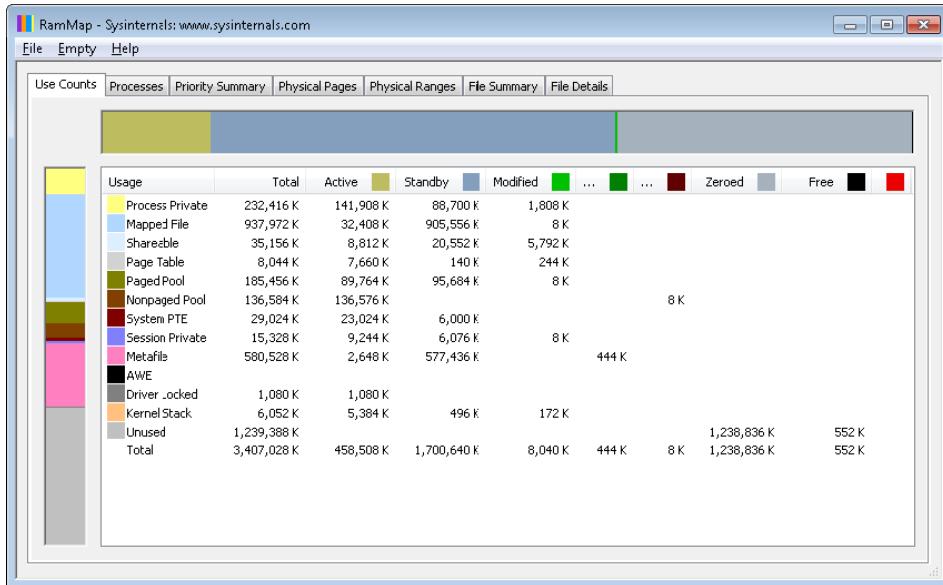
Слово «возможно» применено потому, что поток обнуления страниц «проснется», как только в списке обнуленных страниц останется всего восемь записей, и заработает очень быстро. Обратите внимание на то, что в данном примере мы освободили 800 Мбайт закрытой памяти, но только около 138 Мбайт появилось в списке свободных страниц. Process Explorer обновляет это окно только раз в секунду, и похоже, что остальные страницы уже успевают обнулиться и попасть в список обнуленных страниц, пока нам удалось «поймать» это состояние.

Если вам удалось увидеть временное увеличение списка свободных страниц, то вслед за этим вы увидите, что его размер упадет до нуля, а соответствующее увеличение коснется списка обнуленных страниц. Если же момент будет упущен, вы просто увидите увеличение списка обнуленных страниц.

ЭКСПЕРИМЕНТ: ПРОСМОТР СПИСКОВ ИЗМЕНЕННЫХ И ОЖИДАЮЩИХ СТРАНИЦ

Перемещение страниц из рабочего набора процесса в список измененных страниц и далее — в список ожидающих страниц при запущенном отладчике ядра также можно наблюдать с помощью инstrumentальных средств, разработанных в Sysinternals, к которым относятся программы VMMap и RAMMap.

Сначала нужно запустить программу RAMMap и взглянуть на систему в спокойном состоянии.



В данном случае это система x86, имеющая около 3,4 Гбайт оперативной памяти, используемой Windows. Столбцы в окне отражают различные состояния страниц (см. рис. 10.39). Некоторые столбцы, которые не имеют значения для данного эксперимента, для удобства были скрыты.

У системы имеется около 1,2 Гбайт свободной оперативной памяти (слагаемой из списка свободных и обнуленных страниц). Около 1700 Мбайт фигурирует в списке ожидающих страниц (следовательно, часть из них «доступна», но, скорее всего, содержит данные, ранее утраченные процессами или используемые при супервыборке). Около 448 Мбайт активны, будучи отображенными непосредственно на виртуальные адреса через доверительные записи таблицы страниц.

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 448 Мбайт около 138 Мбайт обусловлено выделением закрытых страниц процесса.

Теперь, как и в предыдущем эксперименте, воспользуйтесь утилитой TestLimit, чтобы создать процесс с большим количеством страниц в его рабочем наборе. Здесь опять мы используем ключ -d, чтобы заставить TestLimit сделать запись в каждую страницу, но на этот раз мы задействуем его без ограничения, чтобы было создано как можно больше закрытых измененных страниц:

```
C:\Users\user1>testlimit -d
```

```
Testlimit v5.21 - test Windows limits
```

продолжение ↗

Copyright (C) 2012 Mark Russinovich
 Sysinternals - www.sysinternals.com

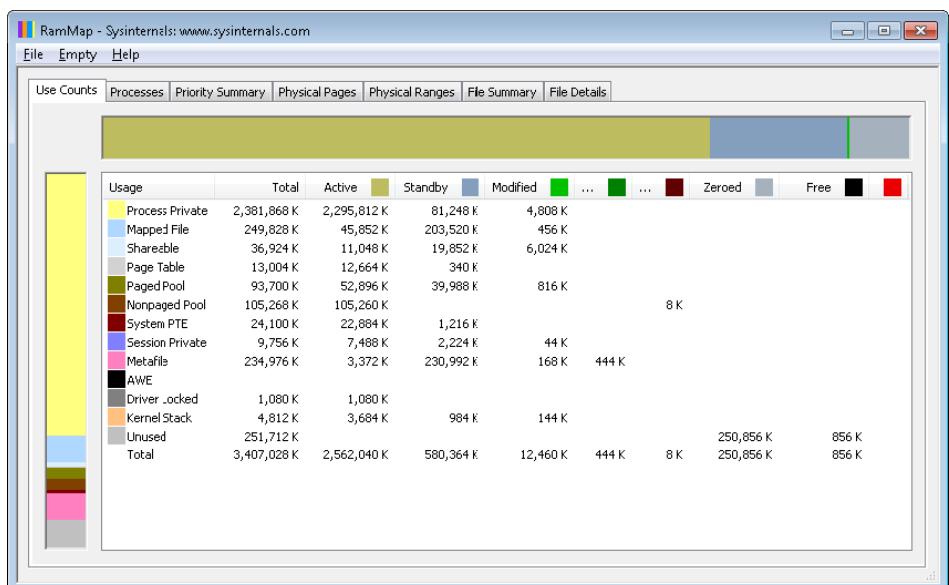
Process ID: 1000

Leaking private bytes with touch (MB) ...

Leaked 2017 MB of private memory (2017 MB total leaked). Lasterror: 8
 Not enough storage is available to process this command.

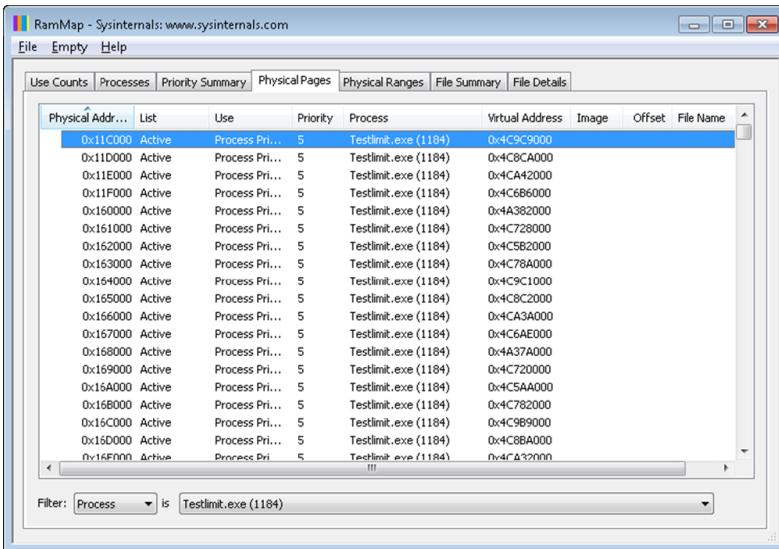
Теперь программа TestLimit создала 2017 областей выделения по 1 Мбайт каждая.

Для обновления экрана в программе RAMMap нужно воспользоваться командой File ▶ Refresh (Файл ▶ Обновить), поскольку самостоятельно программа RAMMap этого не делает (из-за больших накладных расходов на выполнение этой операции).

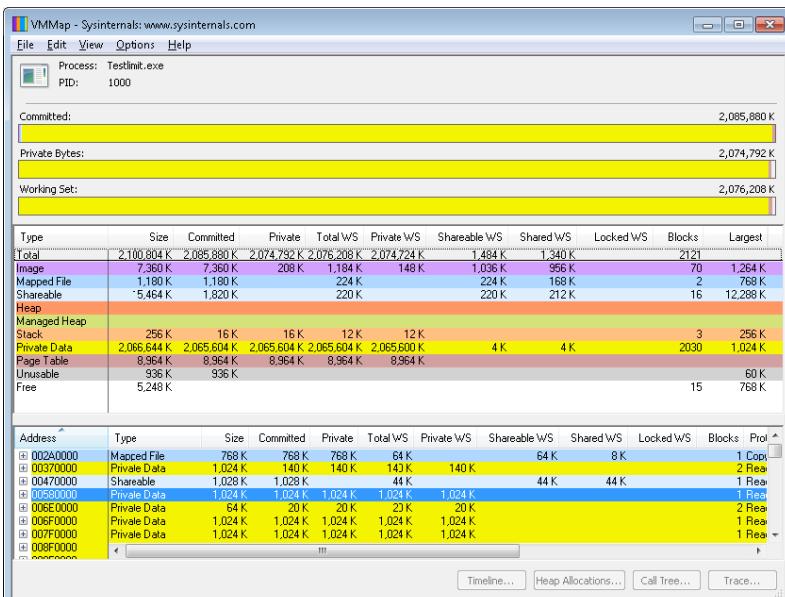


Как видите, свыше 2 Гбайт теперь относятся к активным страницам (столбец Active) и находятся в строке закрытых страниц процесса (строка Process Private). Это результат выделения памяти и доступа к ней со стороны процесса TestLimit. Также обратите внимание на то, что списки ожидающих (Standby), обнуленных (Zeroed) и свободных (Free) страниц теперь стали намного меньше. Большая часть памяти, выделенной программе TestLimit, взята из страниц, фигурировавших в этих списках.

Далее с помощью RAMMap нужно оценить выделение физических страниц процесса. Перейдите на вкладку Physical Pages (Физические страницы) и установите фильтр, находящийся в нижней части окна в столбец Process (Процесс), присвоив ему значение Testlimit.exe. В следующем окне показаны все физические страницы, являющиеся частью рабочего набора процесса.



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 обнаружим следующее:



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 0x580000.

Теперь вернемся к информации о физической памяти, выводимой на экран программой RAMMap. Перестройте столбцы так, чтобы хорошо был виден столбец Virtual Address (Виртуальный адрес). Щелкните на нем для сортировки таблицы по этому значению, и вы сможете найти нужный виртуальный адрес.

Physical Address	List	Use	Priority	Virtual Address	Image	Offset	File Name
0xA0619000	Active	Process Private	5	0x38E000			
0xA0376000	Active	Process Private	5	0x38F000			
0xFF120000	Active	Process Private	5	0x390000			
0xA06D1000	Active	Process Private	5	0x391000			
0x9FDE3000	Active	Process Private	5	0x392000			
0x97D78000	Active	Process Private	5	0x580000			
0xA05C1000	Active	Process Private	5	0x581000			
0x9FDEA000	Active	Process Private	5	0x582000			
0x9FDCCB000	Active	Process Private	5	0x583000			
0xA066C000	Active	Process Private	5	0x584000			
0x9FC3D000	Active	Process Private	5	0x585000			
0xA02E6000	Active	Process Private	5	0x586000			
0xA0AA7000	Active	Process Private	5	0x587000			
0x97D90000	Active	Process Private	5	0x588000			
0xA0549000	Active	Process Private	5	0x589000			
0x9FE32000	Active	Process Private	5	0x58A000			
0x9FE83000	Active	Process Private	5	0x58B000			
0xA0754000	Active	Process Private	5	0x58C000			
0x9FC35000	Active	Process Private	5	0x58D000			

Filter: Process is Testlimit.exe (1000)

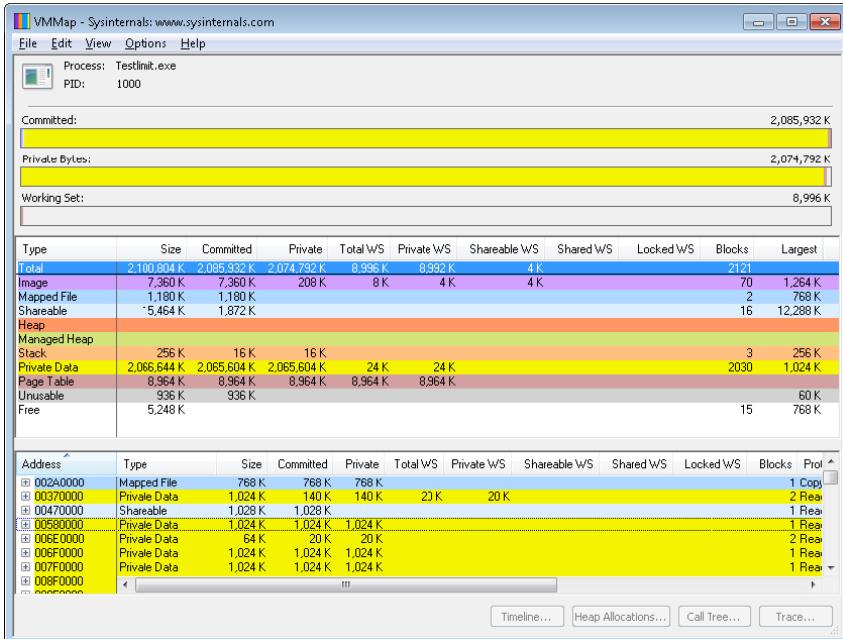
Здесь показано, что виртуальная страница, начинающаяся с адреса 0x01340000, в данный момент отображена на физический адрес 0x97D78000.

Использование в программе TestLimit ключа -d приводит к записи в первые байты каждой выделенной области собственного имени программы. Это можно продемонстрировать с помощью команды !dc локального отладчика ядра (dc — расшифровывается как display characters, то есть вывести символы, используя физический адрес):

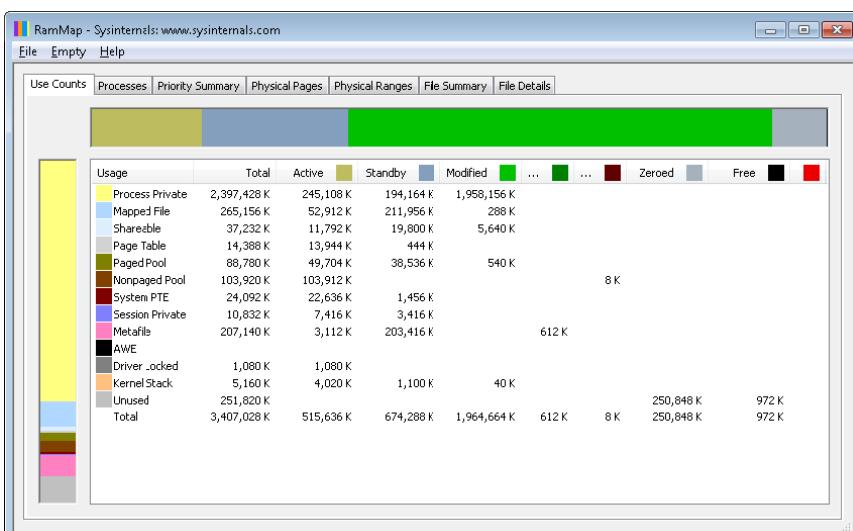
```
1kd> !dc 0x97d78000
#97d78000 74736554 696d694c 00000074 00000000 TestLimit.....
#97d78010 00000000 00000000 00000000 00000000 .....
#97d78020 00000000 00000000 00000000 00000000 .....
...
```

На заключительном этапе эксперимента мы продемонстрируем, что эти данные остаются неизменными (во всяком случае, на некоторое время) после того, как рабочий набор процесса сокращается и страница перемещается сначала в список измененных, а затем — ожидающих страниц.

Выбрав в программе VMMap процесс TestLimit, выберите команду View ▶ Empty Working Set (Вид ▶ Опустошить рабочий набор) для сокращения рабочего набора процесса до минимума. Теперь в окне VMMap должна выводиться следующая информация.



Обратите внимание, что линейка Working Set (Рабочий набор) практически пуста. В средней части для процесса показано, что общий размер рабочего набора равен всего лишь 9 Мбайт, причем почти все его пространство занято таблицами страниц с небольшим общим страничным пространством в 32 Кбайт, принадлежащим файлам образа и закрытым данным. Теперь вернитесь в RAMMap. На вкладке Use Counts (Итоги использования) видно, что количество активных страниц существенно сократилось наряду с большим количеством страниц в списке измененных и существенным количеством страниц в списке ожидающих.



Данные на вкладке Processes (Процессы) программы RAMMap подтверждают, что большинство этих страниц появилось в данных списках благодаря процессу TestLimit.

Process	Session	PID	Private	Standby	Modified	Page Table	Total
cmd.exe	1	2692	460 K	12 K	0 K	112 K	584 K
script-fu.exe	1	3788	0 K	6,344 K	0 K	252 K	6,596 K
vmmmap.exe	1	3508	4,444 K	0 K	0 K	236 K	4,680 K
RtHDVCpl.exe	1	2484	376 K	2,340 K	0 K	204 K	2,920 K
Testlimit.exe	1	1000	28 K	108,540 K	1,957,192 K	4,100 K	2,069,860 K
nusb3mon.exe	1	2504	464 K	736 K	0 K	148 K	1,348 K
SearchIndexer.	0	2972	4,936 K	3,224 K	0 K	320 K	8,480 K
svchost.exe	0	1436	824 K	2,952 K	0 K	228 K	4,004 K
crss.exe	0	452	324 K	600 K	0 K	108 K	1,032 K
taskhost.exe	1	1284	972 K	684 K	0 K	196 K	1,852 K
smss.exe	-1	280	0 K	76 K	0 K	48 K	124 K
MOM.exe	1	2536	444 K	3,660 K	0 K	348 K	4,452 K
crss.exe	1	520	5,312 K	264 K	0 K	132 K	5,708 K
wininit.exe	0	508	212 K	160 K	4 K	124 K	500 K
services.exe	0	568	1,836 K	828 K	0 K	136 K	2,800 K
svchost.exe	0	684	1,056 K	500 K	0 K	156 K	1,712 K
lsass.exe	0	584	1,308 K	1,084 K	0 K	140 K	2,532 K
lsm.exe	0	592	440 K	156 K	0 K	96 K	692 K
svchost.exe	0	760	1,428 K	792 K	8 K	132 K	2,360 K
atiesrxx.exe	0	800	40 K	304 K	0 K	112 K	456 K
winlogon.exe	1	868	388 K	304 K	0 K	148 K	840 K
cunction.exe	0	916	3,580 K	2,284 K	8 K	292 K	6,164 K

Оставаясь в RAMMap, перейдите на вкладку Physical Pages (Физические страницы). Отсортируйте таблицу по столбцу Physical Address (Физический адрес) и найдите ранее исследуемую страницу (в данном случае физический адрес равен 0xc09fa000). RAMMap почти наверняка покажет, что она находится в списке ожидающих или измененных страниц.

Physical Address	List	Use	Priority	Virtual Address	Image	Offset	Process	File Name
0x97D6F000	Modified	Process Private	5	0xE0AAC00			Testlimit.exe (1000)	
0x97D70000	Modified	Process Private	5	0x387000			Testlimit.exe (1000)	
0x97D75000	Modified	Process Private	5	0x49D24000			Testlimit.exe (1000)	
0x97D76000	Modified	Process Private	5	0x5DE7D000			Testlimit.exe (1000)	
0x97D78000	Modified	Process Private	5	0:580000			Testlimit.exe (1000)	
0x97D7D000	Modified	Process Private	5	0x49D25000			Testlimit.exe (1000)	
0x97D80000	Modified	Process Private	5	0x590000			Testlimit.exe (1000)	
0x97D85000	Modified	Process Private	5	0x49D26000			Testlimit.exe (1000)	
0x97D88000	Modified	Process Private	5	0x598000			Testlimit.exe (1000)	
0x97D8B000	Modified	Process Private	5	0x5EEF8000			Testlimit.exe (1000)	
0x97D90000	Modified	Process Private	5	0x588000			Testlimit.exe (1000)	
0x97D93000	Modified	Process Private	5	0x5ED6000			Testlimit.exe (1000)	
0x97D96000	Modified	Process Private	5	0x4402C000			Testlimit.exe (1000)	
0x97D99000	Modified	Process Private	5	0x49D27000			Testlimit.exe (1000)	
0x97D9E000	Modified	Process Private	5	0x5DE75000			Testlimit.exe (1000)	
0x97DA0000	Modified	Process Private	5	0x1061000			Testlimit.exe (1000)	
0x97DA1000	Modified	Process Private	5	0x5EB4000			Testlimit.exe (1000)	
0x97DA2000	Modified	Process Private	5	0x49D28000			Testlimit.exe (1000)	
0x97DA4000	Modified	Process Private	5	0x5F880000			Testlimit.exe (1000)	

Filter: Process Is Testlimit.exe (1000)

Обратите внимание, что страница по-прежнему связана с процессом TestLimit и его виртуальным адресом.

И наконец, мы можем опять воспользоваться отладчиком ядра для проверки страницы, которая не была переписана:

```
1kd> !dc 0x97d78000
#97d78000 74736554 696d694c 00000074 00000000 TestLimit.....
#97d78010 00000000 00000000 00000000 00000000 .....
#97d78020 00000000 00000000 00000000 00000000 .....
...
```

Локальный отладчик ядра можно также использовать для вывода номера страничного блока, или PFN-записи для страницы. (База данных PFN-номеров рассматривается в этой главе ранее.)

```
1kd> !pfn 97d78
PFN 00097D78 at address 84E9B920
flink 000A0604 blink / share count 000A05C1 pteaddress C0002C00
reference count 0000 Cached color 0 Priority 5
restore pte 00000080 containing page 097D60 Modified M
Modified
```

Обратите внимание, что страница по-прежнему связана с процессом TestLimit и с его виртуальным адресом.

Приоритеты страниц

У каждой физической страницы, присутствующей в системе, имеется значение приоритета страницы, присваиваемое ей диспетчером памяти. Приоритет страницы представляет собой число в диапазоне от 0 до 7. Его главное назначение состоит в определении 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 приоритета. Когда диспетчер памяти собирается взять страницу из списка ожидающих, то, как показано на рис. 10.40, 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 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 изучения PFN-записей, что показано в одном из следующих экспериментов данной главы).

На рис. 10.41 показана вкладка Threads (Потоки) программы Process Explorer, на которой представлена информация об основном потоке программы Winlogon. Хотя приоритет самого потока довольно высок, приоритет памяти составляет стандартное значение 5.



Рис. 10.40. Списки ожидающих страниц, распределенные по приоритетам

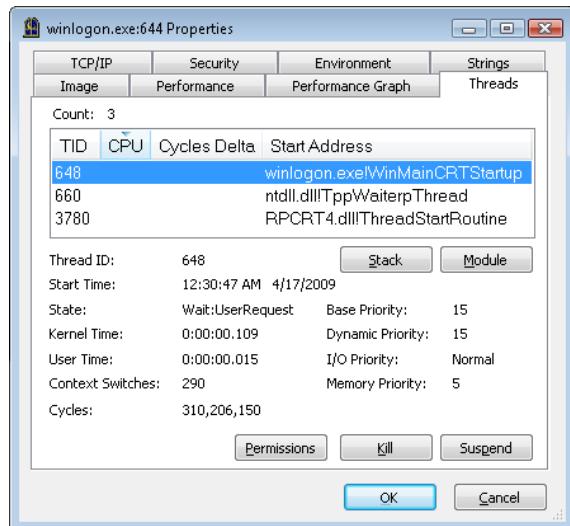


Рис. 10.41. Вкладка Threads (Потоки) программы Process Explorer

Реальное влияние приоритетов памяти проявляется, только когда относительные приоритеты страниц поднимаются на высокий уровень, в чем и заключается роль супервыборки, рассматриваемой в конце данной главы.

ЭКСПЕРИМЕНТ: ПРОСМОТР СПИСКОВ ОЖИДАЮЩИХ СТРАНИЦ, РАССТАВЛЕННЫХ ПО ПРИОРИТЕТАМ

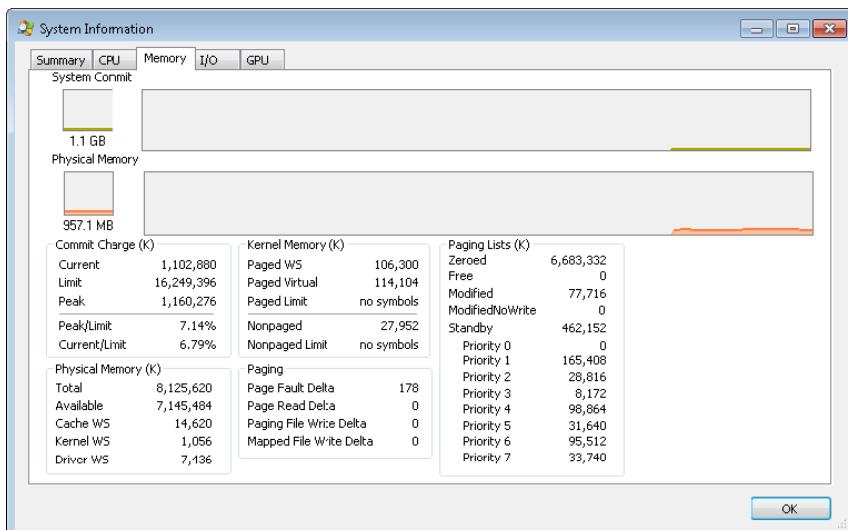
Для вывода дампа с размерами каждого списка ожидающих страниц можно воспользоваться созданной в Winsider Seminars & Solutions программой MemInfo с ключом `-c`. MemInfo покажет также количество повторно использованных страниц (*repurposed*) для каждого списка ожидающих страниц. Это то количество страниц, которые пришлось повторно использовать для удовлетворения потребностей в выделении памяти. Вот данные, полученные при запуске команды:

```
C:\Windows\system32>meminfo -c
MemInfo v2.10 - Show PFN database information
Copyright (C) 2007-2009 Alex Ionescu
www.alex-ionescu.com
```

Initializing PFN Database... Done

```
Priority Standby Repurposed
0 - Idle 0 ( 0 KB) 0 ( 0 KB)
1 - Very Low 41352 ( 165408 KB) 0 ( 0 KB)
2 - Low 7201 ( 28804 KB) 0 ( 0 KB)
3 - Background 2043 ( 8172 KB) 0 ( 0 KB)
4 - Background 24715 ( 98860 KB) 0 ( 0 KB)
5 - Normal 7895 ( 31580 KB) 0 ( 0 KB)
6 - Superfetch 23877 ( 95508 KB) 0 ( 0 KB)
7 - Superfetch 8435 ( 33740 KB) 0 ( 0 KB)
TOTAL 115518 ( 462072 KB) 0 ( 0 KB)
```

Для постоянного вывода состояния списков ожидающих страниц и счетчиков заново использованных страниц, который может оказаться полезным, чтобы отслеживать расходование памяти, а также для проведения следующего эксперимента при запуске MemInfo можно указать флаг `-i`. Кроме того, как показано в следующей копии экрана, для просмотра текущего состояния списков ожидающих страниц, распределенных по приоритетам, можно также воспользоваться панелью System Information (Системная информация) программы Process Explorer, выбрав в меню команду View ▶ System Information (Вид ▶ Системная информация).



На недавно запущенной системе x64, использованной в эксперименте (см. предыдущие данные, выведенные программой MemInfo), данных, кэшированных с приоритетом 0, нет вообще, данных с приоритетом 1 — 165 Мбайт, а данных с приоритетом 2 — 29 Мбайт. Возможно, на вашей системе также имеются данные с такими же приоритетами.

Далее показано, что произойдет, если для подтверждения памяти объемом 1 Гбайт и обращения к ней использовать программу TestLimit, разработанную в Sysinternals. Для этого применена следующая команда (с целью организации утечки и обращения к 20 областям памяти по 50 Мбайт каждая):

```
testlimit -d 50 -c 20
```

Выводимые программой MemInfo данные непосредственно перед запуском имели следующий вид:

```
Priority Standby Repurposed
0 - Idle 0 ( 0 KB) 2554 ( 10216 KB)
1 - Very Low 92915 ( 371660 KB) 141352 ( 565408 KB)
2 - Low 35783 ( 143132 KB) 0 ( 0 KB)
3 - Background 50666 ( 202664 KB) 0 ( 0 KB)
4 - Background 15236 ( 60944 KB) 0 ( 0 KB)
5 - Normal 34197 ( 136788 KB) 0 ( 0 KB)
6 - Superfetch 2912 ( 11648 KB) 0 ( 0 KB)
7 - Superfetch 5876 ( 23504 KB) 0 ( 0 KB)
TOTAL 237585 ( 950340 KB) 143906 ( 575624 KB)
```

А вот как выглядят выводимые данные после выделения памяти, но при еще существующем процессе TestLimit:

```
Priority Standby Repurposed
0 - Idle 0 ( 0 KB) 2554 ( 10216 KB)
1 - Very Low 5 ( 20 KB) 234351 ( 937404 KB)
2 - Low 0 ( 0 KB) 35830 ( 143320 KB)
3 - Background 9586 ( 38344 KB) 41654 ( 166616 KB)
4 - Background 15371 ( 61484 KB) 0 ( 0 KB)
5 - Normal 34208 ( 136832 KB) 0 ( 0 KB)
6 - Superfetch 2914 ( 11656 KB) 0 ( 0 KB)
7 - Superfetch 5881 ( 23524 KB) 0 ( 0 KB)
TOTAL 67965 ( 271860 KB) 314389 ( 1257556 KB)
```

Обратите внимание, что 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 понадобились для того, чтобы

избежать взаимных блокировок, возникающих, если запись страниц отображаемого файла становится причиной ошибки отсутствия страницы, которая, в свою очередь, требует свободной страницы при отсутствии доступных свободных страниц (требуя, таким образом, 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 17, и после инициализации они ждут для запуска своих операций отдельных объектов. Поток, записывающий отображаемые страницы, ждет наступления события `MmMappedPageWriterEvent`. Сигнал может быть подан в следующих случаях:

- ❑ В ходе операции со списками страниц (`MiInsertPageInLockedList` или `MiInsertPageInList`). Эти процедуры выдают сигнал в виде события, если количество страниц, предназначенных для файловой системы и занесенных в список измененных страниц, превысило 800 единиц, а количество доступных страниц упало ниже 1024, либо если количество доступных страниц стало меньше 256 единиц.
- ❑ 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 единиц.
- ❑ После запроса на сброс всех измененных страниц (`MmFlushAllPages`).
- ❑ После запроса 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.

Подсистема записи отображаемых страниц также ждет массива событий `MiMappedPageListHeadEvent`, связанного с 16 списками отображаемых страниц. При каждом внесении изменений в отображаемую страницу она вставляется в один из 16 списков отображаемых страниц по номеру корзины (`MiCurrentMappedPageBucket`). Этот номер корзины 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`, которое увеличивается на единицу при каждом запуске диспетчера рабочих наборов). Поводом для применения этих дополнительных событий служит стремление сократить потерю данных в случае краха системы или отказа электропитания путем периодической записи измененных отображаемых страниц, даже если список измененных страниц не достиг своего порога в 800 страниц.

Подсистема записи измененных страниц ждет сигнала на работу от отдельного объекта шлюза (`MmModifiedPageWriterGate`), от которого может поступить сигнал при развитии следующих сценариев:

- ❑ Поступление запроса на сброс всех страниц.
- ❑ Падение количества доступных страниц (`MmAvailablePages`) ниже уровня 128 единиц.
- ❑ Падение общего количества страниц, заполненных нулевыми байтами, и свободных страниц ниже уровня 20 000 единиц, 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 был усечен и не может вместить дополнительные страницы, если количество доступных страниц стало менее 15 000.
- ❑ В ходе операции со списком страниц (`MiInsertPageInLockedList` или `MiInsertPageInList`). Эти процедуры сигнализируют шлюзу, если количество страниц, предназначенных для страничных файлов в списке измененных страниц, стало больше 800 единиц, а количество доступных страниц упало ниже 1024 единиц или если количество доступных страниц стало меньше 256 единиц.

Кроме того, подсистема записи измененных страниц ждет наступления события `MiRescanPageFilesEvent` и внутреннего события в заголовке страничного файла (`MmPagingFileHeader`), которые 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 РТЕ, 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 числа 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.

Структура 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. 10.42.

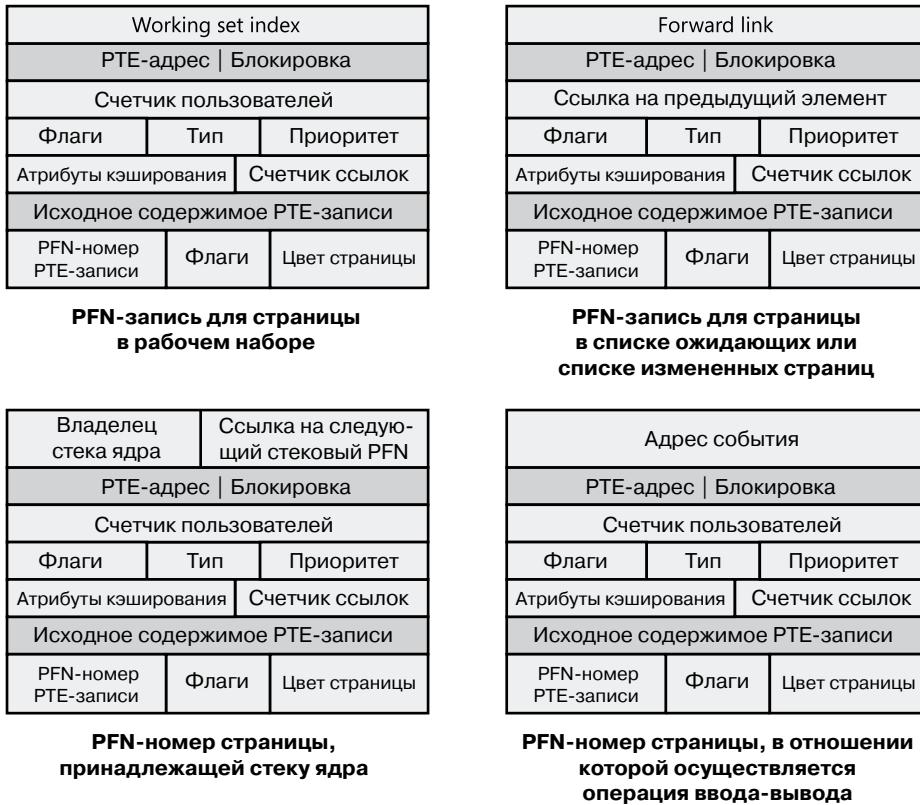


Рис. 10.42. Состояния записей базы данных PFN-номеров
(отдельные схемы представлены в абстрактном виде)

Для некоторых типов PFN-записей некоторые 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 выровнены по 4-байтовой границе (8-байтовой на 64-разрядных системах), два младших разряда используются в качестве блокировочного механизма для последовательного доступа к PFN-записи.
- ❑ **Счетчик ссылок** (reference count). Количество ссылок на данную страницу. Показание счетчика ссылок увеличивается на единицу, когда страница впервые добавляется к рабочему набору и (или) когда страница заблокирована в памяти для выполнения операции ввода-вывода (например, драйвером устройства). Показание счетчика

ссылок уменьшается на единицу, когда счетчик пользователей становится равным нулю или когда страница деблокируется в памяти. Когда счетчик пользователей становится равным нулю, страница больше не принадлежит рабочему набору. Затем, если счетчик ссылок также имеет нулевое значение, 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, или 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. 10.17.
- 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, используемое для кодировки дополнительной информации, касающейся PTE-записи (табл. 10.18).

Таблица 10.17. Флаги в записях базы данных PFN-номеров

Флаг	Значение
Выполняется запись (Write in progress)	Выполняется операция записи страницы. Первый DWORD-параметр содержит адрес объекта события, который подаст сигнал, когда операция ввода-вывода будет завершена
Измененное состояние (Modified state)	Страница была изменена. (Если страница была изменена, ее содержимое должно быть сохранено на диске перед удалением этой страницы из памяти.)
Выполняется чтение (Read in progress)	В отношении данной страницы выполняется страничная операция. Первый DWORD-параметр содержит адрес объекта события, который подаст сигнал, когда операция ввода-вывода будет завершена

Флаг	Значение
Получена из постоянной памяти (Rom)	Страница поступила из встроенного компьютерного программного обеспечения или из другого места памяти, предназначенного только для чтения, например из регистра устройства
Ошибка при проведении операции над страницей (In-page error)	В ходе страничной операции в отношении данной страницы произошла ошибка ввода-вывода. (В этом случае в первом поле PFN-записи содержится код ошибки.)
Стек ядра (Kernel stack)	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
Запрошено удаление (Removal requested)	Страница подлежит удалению (из-за ECC-кода или очистки либо из-за удаления блока памяти из работающей машины)
Ошибка четности (Parity error)	В физической странице содержится ошибка четности или ошибка контроля коррекции ошибок

Таблица 10.18. Вторичные флаги в записях базы данных PFN-номеров

Флаг	Значение
PFN-образ проверен (PFN image verified)	Код проверенной сигнатуры для данной PFN-записи (содержащийся в каталоге криптографической сигнатуры для образа, поддержанного этой PFN-записью)
AWE-выделение (AWE allocation)	Эта PFN-запись поддерживает AWE-выделение памяти
Прототипная PTE-запись (Prototype PTE)	Показывает, что PTE-запись, на которую ссылается данная PFN-запись, является прототипной. (Например, страница предназначена для совместного использования.)

Все остальные поля характерны для того или иного типа PFN-записей. Например, первая PFN-запись на рис. 10.42 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 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 подсказкой

того, что правильность 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 хэша списка рабочего набора, ссылающейся по виртуальному адресу в его дереве хэша рабочего набора. Тем самым обеспечивается уменьшение размера дерева хэша рабочего набора и ускоряется поиск именно таких прямых записей.

Вторая PFN-запись на рис. 10.42 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 устраниении ошибок отсутствия страницы. Если страница принадлежит одному из списков, счетчик пользователей по определению равен нулю (поскольку страница не используется ни одним рабочим набором), и поэтому 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 в отношении страницы выполняется операция ввода-вывода, например когда страница записывается на диск), то сначала страница удаляется из списка.

Третья PFN-запись на рис. 10.42 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. 10.42 предназначена для страницы, в отношении 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 страницей операцией, это поле содержит код статуса ошибки, отражающий ошибку ввода-вывода. Такой тип PFN-записи используется для разрешения конфликтных ошибок отсутствия страницы.

В дополнение к базе данных PFN-номеров общее состояние физической памяти описывается системными переменными, перечисленными в табл. 10.19.

Таблица 10.19. Системные переменные, описывающие физическую память

Переменная	Описание
MmNumberOfPhysicalPages	Общее количество физических страниц, доступных в системе
MmAvailablePages	Общее количество доступных страниц в системе (сумма страниц в списках обнуленных, свободных и ожидающих страниц)
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 качестве аргумента нужно снабдить эту команду номером PFN-записи. (Например, команда !pfn 1 покажет первую запись, !pfn 2 — вторую и т. д.) В следующем примере показана PTE-запись для виртуального адреса 0x50000, за которой следует PFN-запись, содержащая каталог страниц, затем следует фактическая страница:

```
lkd> !pte 50000
VA 00050000
PDE at 00000000C0600000 PTE at 00000000C0000280
contains 000000002C9F7867 contains 800000002D6C1867
pfn 2c9f7 ---DA--UWEV pfn 2d6c1 ---DA--UW-V

lkd> !pfn 2c9f7
PFN 0002C9F7 at address 834E1704
flink 00000026 blink /
share count 00000091 pteaddress C0600000
reference count 0001 Cached color 0 Priority 5
restore pte 00000080 containing page 02BAA5 Active M
Modified

lkd> !pfn 2d6c1
PFN 0002D6C1 at address 834F7D1C
flink 00000791 blink /
share count 00000001 pteaddress C0000280
reference count 0001 Cached color 0 Priority 5
restore pte 00000080 containing page 02C9F7 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 режим отладки. Вот как выглядят выводимые программой MemInfo данные для тех же самых двух PFN-записей:

```
C:\>meminfo -p 2c9f7
PFN: 2c9f7
PFN List: Active and Valid
PFN Type: Page Table
PFN Priority: 5
Page Directory: 0x866168C8
Physical Address: 0x2C9F7000

C:\>meminfo -p 2d6c1
PFN: 2d6c1
PFN List: Active and Valid
PFN Type: Process Private
PFN Priority: 5
EPROCESS: 0x866168C8 [windbg.exe]
Physical Address: 0x2D6C1000
```

MemInfo корректно 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 50000.

Лимиты физической памяти

Теперь, когда вы узнали, как Windows следит за физической памятью, поговорим о том, насколько большой объем этой памяти Windows фактически может поддерживать. Поскольку большинство систем в процессе своей работы обращаются к коду и данным, чей объем больше, чем может поместиться в физической памяти, эта память, по сути, является окном, которое открывает доступ к используемым коду и данным. Объем памяти может тем самым влиять на производительность, поскольку 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 сконфигурированного размера любых страничных файлов. Физическая память также может косвенно ограничивать максимальное количество процессов.

Поддержка физической памяти со стороны Windows диктуется аппаратными ограничениями, лицензированием, структурами данных операционной системы и совместимостью драйверов. Текущие поддерживаемые объемы физической памяти в различных редакциях Windows наряду с ограничивающими факторами перечислены в табл. 10.20.

Таблица 10.20. Поддержка физической памяти

Версия	32-разрядный лимит, Гбайт	64-разрядный лимит, Гбайт	Ограничивающие факторы
Максимальная (Ultimate), Корпоративная (Enterprise) и Профессиональная (Professional)	4	192	Лицензирование на 64-разрядной системе; лицензирование, аппаратная поддержка и совместимость драйверов на 32-разрядной системе
Домашняя расширенная (Home Premium)	4	16	Лицензирование на 64-разрядной системе; лицензирование, аппаратная поддержка и совместимость драйверов на 32-разрядной системе
Домашняя базовая (Home Basic)	4	8	Лицензирование на 64-разрядной системе; лицензирование, аппаратная поддержка и совместимость драйверов на 32-разрядной системе

Версия	32-разрядный лимит, Гбайт	64-разрядный лимит, Гбайт	Ограничивающие факторы
Начальная (Starter)	2	2	Лицензирование
Server Datacenter, Enterprise и Server for Itanium	Неприменимо	2	Тестирование и доступность системы
Server Foundation	Неприменимо	8	Лицензирование
Server Standard и Web Server	Неприменимо	32	Лицензирование
Server HPC Edition	Неприменимо	128	Лицензирование

Максимальный лимит физической памяти в 2 Тбайт не обуславливается каким-либо реализационным или аппаратным ограничением, а является причиной того, что Microsoft будет поддерживать только те варианты настройки, которые сможет протестировать. На момент написания данной книги наибольший протестированный и поддерживаемый объем памяти составлял 2 Тбайт.

Лимиты памяти клиентских версий Windows

64-разрядные клиентские версии Windows отличаются тем, что поддерживают различные объемы памяти, начиная с нижней границы в 2 Гбайт для начальной редакции (Starter Edition) и увеличиваясь до 192 Гбайт для универсальной (Ultimate), корпоративной (Enterprise) и профессиональной (Professional) редакций. 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 система рухнет, повиснет или станет незагружаемой, поскольку некоторые драйверы устройств (обычно это касается видео- и аудиоустройств, характерных для клиентских, но не серверных машин) не были запрограммированы на ожидаемые физические адреса, превышающие 4 Гбайт. В результате драйверы усекают такие адреса, из-за чего имеет место разрушение памяти и побочные разрушительные эффекты. Серверные системы обычно обладают более универсальными устройствами с более простыми и стабильными драйверами, поэтому, как правило, для них эти проблемы не характерны. Проблемные клиентские драйверные системы привели к решению для клиентских версий, в котором игнорируется физическая память, выходящая 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 несовершенных драйверах.

Фактические лимиты памяти на 32-разрядных клиентских системах

Хотя значение 4 Гбайт — это лицензированный лимит для 32-разрядных клиентских версий, реальный лимит фактически ниже и зависит от системного набора микропроцессоров 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, чтобы сохранить совместимость 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 ранее рассмотренных 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. 10.43, 500 Мбайт из 4 Гбайт оперативной памяти системы выйдет за адресную границу в 4 Гбайт.



Рис. 10.43. Схема физической памяти в системе с оперативной памятью размером 4 Гбайт

В результате, если имеется система с памятью размером 3 Гбайт и на ней запущена 32-разрядная клиентская версия Windows, то выгоды от доступа ко всей оперативной памяти можно и не получить. Какой объем установленной оперативной памяти «видит» операционная система, можно посмотреть в диалоговом окне **System Properties** (Свойства системы), но чтобы узнать, какой объем памяти фактически

доступен Windows, нужно открыть вкладку Performance (Быстродействие) диспетчера задач или использовать утилиты Msinfo32 и Winver. Как показала утилита Msinfo32, на одном конкретном ноутбуке с оперативной памятью в 4 Гбайт и загруженной 32-разрядной Windows объем доступной физической памяти составил 3,5 Гбайт:

- установленная оперативная память (RAM) – 4,00 Гбайт;
- полный объем физической памяти – 3,50 Гбайт.

План физической памяти можно посмотреть с помощью программы MemInfo, разработанной в Winsider Seminars & Solutions. На рис. 10.44 показан результат запуска утилиты MemInfo на 32-разрядной системе с ключом **-r**, предназначенным для вывода диапазонов физической памяти.

```
C:\>MemInfo.exe -r
MemInfo v1.11 - Show PFN database information
Copyright (C) 2007-2008 Alex Ionescu
www.alex-ionescu.com

Physical Memory Range: 00001000 to 0009F000 <158 pages, 632 KB>
Physical Memory Range: 00100000 to DFE6D000 <916845 pages, 3667380 KB>
MmHighestPhysicalPage: 917101
```

Рис. 10.44. Диапазоны памяти на 32-разрядной системе Windows

Обратите внимание на пропуск в диапазонах адресов памяти от страницы 9F0000 до страницы 100000, и еще один пропуск от DFE6D000 до FFFFFFFF (4 Гбайт). В то же время, когда загружена 64-разрядная версия Windows, все 4 Гбайт оказываются доступными (рис. 10.45), и Windows использует оставшиеся 500 Мбайт оперативной памяти, превышающие границу в 4 Гбайт.

```
MemInfo v1.11 - Show PFN database information
Copyright (C) 2007-2008 Alex Ionescu
www.alex-ionescu.com

Physical Memory Range: 00000000001000 to 00000000009F000 <158 pages, 632 KB>
Physical Memory Range: 00000000010000 to 00000000DFE6D000 <916845 pages, 3667380 KB>
Physical Memory Range: 0000000100002000 to 0000000120000000 <131070 pages, 524280 KB>
MmHighestPhysicalPage: 1179648
```

Рис. 10.45. Диапазоны памяти на платформе x64

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). Запустите файл оснастки Devmgmt.msc, выберите в меню команду **View ▶ Resources By Connection** (Вид ▶ Ресурсы по подключению), а затем раскройте узел **Memory** (Память). На ноутбуке, использованном для вывода информации, основным потребителем отображаемой памяти устройств вполне ожидаемо является видеокарта, на которую расходуется 256 Мбайт в диапазоне E0000000-EFFFFFFF (рис. 10.46).

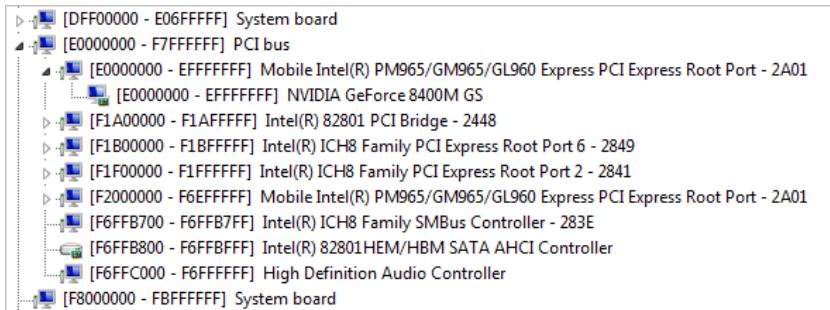


Рис. 10.46. Зарезервированные оборудованием диапазоны памяти на 32-разрядной системе Windows

Большая часть остальной памяти приходится на разные другие устройства, PCI-шина резервирует дополнительные диапазоны для устройств в рамках консервативной оценки встроенного программного обеспечения в ходе начальной загрузки.

На мощных игровых системах с большими видеокартами потребление адресов памяти ниже границы в 4 Гбайт может быть весьма существенным. Например, на тестируемой машине с оперативной памятью в 8 Гбайт и двумя видеокартами по 1 Гбайт 32-разрядной версии Windows была доступна память размером только 2,2 Гбайт. Как показано на рис. 10.47, в выводимых программой MemInfo данных на 64-разрядной версии Windows имеется большая дыра в адресах с 8FEF0000 по FFFFFFFF.

```
C:\>meminfo64.exe -r
MemInfo v1.11 - Show PFN database information
Copyright (C) 2007-2008 Alex Ionescu
www.alex-ionescu.com

Physical Memory Range: 0000000000001000 to 000000000009B000 <154 pages, 616 KB>
Physical Memory Range: 0000000001000000 to 000000008FEF0000 <589296 pages, 2357184 KB>
Physical Memory Range: 0000000100000000 to 0000000270000000 <1507328 pages, 6029312 KB>
MmHighestPhysicalPage: 2555904
```

Рис. 10.47. Диапазоны памяти на 64-разрядной системе Windows

Диспетчер памяти показывает, что 512 Мбайт из более чем 2 Гбайт пропуска предназначено для видеокарт (по 256 Мбайт для каждой), а драйвер PCI-шины зарезервировал еще больше памяти либо для динамического отображения, либо в связи с требованиями по выравниванию границ, либо, наверное, по причине того, что устройства заявили более обширные области, чем те, в которых они нуждаются. И наконец, даже системы с небольшим объемом памяти в 2 Гбайт, такие как 32-разрядная версия Windows, могут не получить в свое распоряжение всю готовую к использованию память из-за наборов микропроцессоров, агрессивно резервирующих диапазоны памяти для устройств.

Рабочие наборы

Итак, зная, как Windows отслеживает физическую память и какой объем этой памяти она может поддерживать, давайте выясним, 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, содержащие страницы, на которые ссылаются программные потоки отдельно взятого процесса.
- Системные рабочие наборы, содержащие резидентный поднабор системного кода со страничной организацией (например, Ntoskrnl.exe и драйверы), выгружаемый пул и системный кэш.
- Рабочий набор каждого сеанса, содержащий 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.

Перед подробным изучением рабочих наборов каждого типа давайте рассмотрим общую политику, чтобы понять, какие страницы попадают в физическую память и как долго они в этой памяти остаются. А уже после этого исследуем рабочие наборы различных типов.

Подкачки по требованию

Для загрузки страниц в память диспетчер памяти Windows использует алгоритм подкачки по требованию с кластеризацией. Когда поток получает ошибку отсутствия страницы, диспетчер памяти загружает в память ненайденную ранее страницу плюс небольшое количество страниц до и после нее. В этой стратегии предпринимается попытка минимизировать объем страничного ввода-вывода потока. Поскольку программы, особенно большие, в любой момент времени выполняются, как правило, в небольших областях своего адресного пространства, загрузка кластеров виртуальных страниц сокращает количество чтений с диска. Для устранения ошибок отсутствия страницы при ссылке на страницы данных в образах размер кластера составляет три страницы. Для всех остальных ошибок отсутствия страниц размер кластера составляет семь страниц.

Однако политика подкачки по требованию может привести к тому, что процессу придется сталкиваться с множеством ошибок отсутствия 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 рассмотрен в данной главе чуть позже.

Компонент логической предвыборки

В ходе обычной 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. Эти перескоки существенно замедляют каждый доступ, причем анализ показывает, что затраты времени на поиск страниц на диске являются доминирующим фактором в замедлении начальной загрузки и запуска приложения. Путем единой предвыборки пакетов страниц, более практического порядка обращения к диску без чрезмерных возвратов, чего вполне можно добиться, общее время запуска системы и приложений сокращается. 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 при чтении кода и данных в начале последующих начальных загрузок или запусков приложений. Когда компонент предвыборки активен, диспетчер памяти информирует код компонента предвыборки в ядре об ошибках отсутствия 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 Explorer), а в случае неудачи — через 60 секунд после инициализации служб Windows или через 120 секунд, в зависимости от того, что наступит 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, на которые были ссылки. Имея собранные результаты отслеживания, код компонента предвыборки ядра ожидает запросов от компонента супервыборки (%SystemRoot%\System32\Sysmain.dll), запущенной 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 сигнализирует о событии \KernelObjects\PrefetchTraces-Ready, чтобы информировать службу супервыборки, что теперь она может запросить отслеженные данные.

ПРИМЕЧАНИЕ

Предвыборку в ходе начальной загрузки или запуска приложений можно включать и выключать, модифицируя DWORD-параметр реестра в разделе HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\PrefetchParameters\EnablePrefetcher. Установка этого параметра в 0 выключает всю предвыборку, установка в 1 включает предвыборку только для приложений, установка в 2 включает предвыборку только для начальной загрузки, а установка в 3 включает предвыборку как для начальной загрузки, так и для запуска приложений.

Служба супервыборки (являющаяся хозяйствкой логической предвыборки, хотя последняя является совершенно отдельным компонентом, не имеющим отношения к фактическому 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. 10.48). Именем файла служит имя приложения, к которому применяются отслеженные данные, далее следует тире и шестнадцатеричное представление хэша пути к файлу. Этот файл имеет расширение .pf. В качестве примера можно привести файл `NOTEPAD.EXE-AF43252301.PF`.

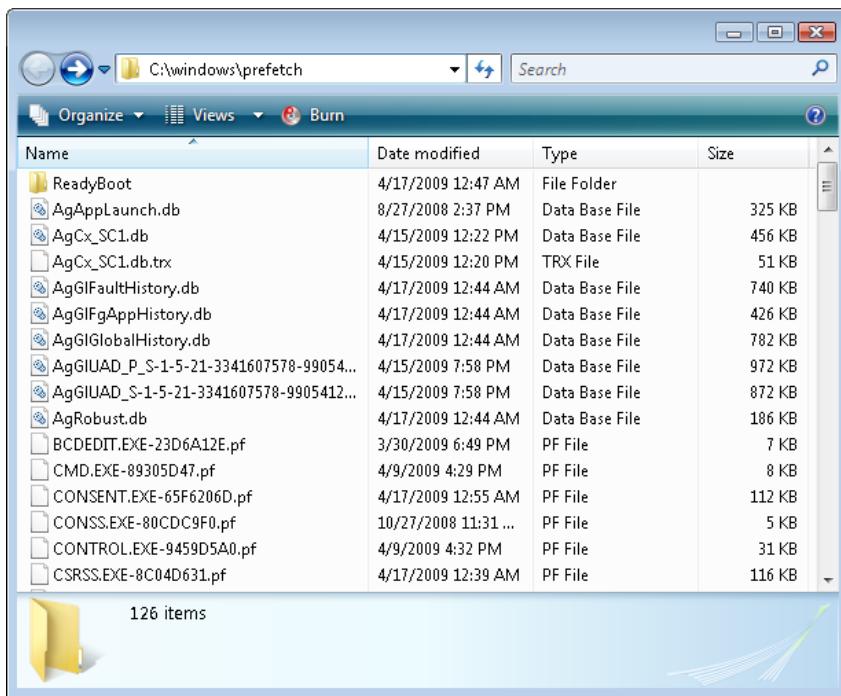


Рис. 10.48. Папка Prefetch

Из правила составления имени файла есть два исключения. Первое из них относится к образам, содержащим другие компоненты, включая образ консоли управления Microsoft (`%SystemRoot%\System32\Mmc.exe`), образ для запуска хост-процесса службы 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 хэш командную строку. Таким образом, запуск этих приложений с различными компонентами в командной строке позволит получать различные варианты трасс.

Еще одним исключением из правила составления имени файла является файл, в котором хранится трасса начальной загрузки. Этот файл всегда называется **NTOSBOOT-B00DFAAD.PF**. (Если читать его как слово, то «boodfaad» звучит как английские слова «boot fast», то есть «загрузить быстро».) Только после того, как компонент предвыборки завершит отслеживание начальной загрузки (время отслеживания было определено ранее), он приступит к сбору информации об ошибках конкретных приложений.

ЭКСПЕРИМЕНТ: ИЗУЧЕНИЕ СОДЕРЖИМОГО ФАЙЛА ПРЕДВЫБОРКИ

Содержимое файла предвыборки представляет собой перечень файлов и каталогов, к которым были обращения в ходе начальной загрузки или при запуске приложений. Для просмотра этой записи можно воспользоваться утилитой **Strings**, созданной в **Sysinternals**. Следующая команда выводит список всех файлов и каталогов, к которым были обращения в ходе последней начальной загрузки:

```
C:\Windows\Prefetch>Strings -n 5 ntosboot-b00dfaad.pf
```

```
Strings v2.4
Copyright (C) 1999-2007 Mark Russinovich
Sysinternals - www.sysinternals.com
```

```
4NTOSBOOT
\DEVICE\HARDDISKVOLUME1\$MFT
\DEVICE\HARDDISKVOLUME1\WINDOWS\SYSTEM32\DRIVERS\TUNNEL.SYS
\DEVICE\HARDDISKVOLUME1\WINDOWS\SYSTEM32\DRIVERS\TUNMP.SYS
\DEVICE\HARDDISKVOLUME1\WINDOWS\SYSTEM32\DRIVERS\I8042PRT.SYS
\DEVICE\HARDDISKVOLUME1\WINDOWS\SYSTEM32\DRIVERS\KBDCLASS.SYS
\DEVICE\HARDDISKVOLUME1\WINDOWS\SYSTEM32\DRIVERS\VMMOUSE.SYS
\DEVICE\HARDDISKVOLUME1\WINDOWS\SYSTEM32\DRIVERS\MOUCLASS.SYS
\DEVICE\HARDDISKVOLUME1\WINDOWS\SYSTEM32\DRIVERS\PARPORT.SYS
...
```

При начальной загрузке 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 каталогов, на которые есть ссылки, 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 на клиентской редакции Windows (в редакциях Windows Server предвыборка изначально отключена) захватить трассу запуска приложения, используя программу Process

Monitor, 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 трасса запуска приложения Блокнот (Notepad) с включающим фильтром (Include), настроенным на предвыборку, чтобы программа Process Monitor показывала только обращения к каталогу %SystemRoot%\Prefetch.

The screenshot shows the Process Monitor interface with a list of events. The columns are: Time of Day, Process Name, Operation, Path, Result, and Detail. The data is as follows:

Time of Day	Process Name	Operation	Path	Result	Detail
11:05:26 8:19:55.32 PM	notepad.exe	CreateFile	C:\Windows\Prefetch\NOTEPAD.EXE-E81B961A.pf	SUCCESS	Desired Access: Generic...
11:05:26 8:20:06.31 PM	notepad.exe	QueryStandardInformationFile	C:\Windows\Prefetch\NOTEPAD.EXE-E81B961A.pf	SUCCESS	AllocationSize: 16,384, E...
11:05:26 8:21:20.28 PM	notepad.exe	ReadFile	C:\Windows\Prefetch\NOTEPAD.EXE-E81B961A.pf	SUCCESS	Offset: 0, Length: 12,962...
11:05:26 8:21:73.56 PM	notepad.exe	CloseFile	C:\Windows\Prefetch\NOTEPAD.EXE-E81B961A.pf	SUCCESS	
11:05:36 8:52:57.58 PM	svchost.exe	CreateFile	C:\Windows\Prefetch\NOTEPAD.EXE-E81B961A.pf	SUCCESS	Desired Access: Generic...
11:05:36 8:52:70.32 PM	svchost.exe	QueryStandardInformationFile	C:\Windows\Prefetch\NOTEPAD.EXE-E81B961A.pf	SUCCESS	AllocationSize: 16,384, E...
11:05:36 8:52:75.15 PM	svchost.exe	QueryStandardInformationFile	C:\Windows\Prefetch\NOTEPAD.EXE-E81B961A.pf	SUCCESS	AllocationSize: 16,384, E...
11:05:36 8:53:08.15 PM	svchost.exe	CloseFile	C:\Windows\Prefetch\NOTEPAD.EXE-E81B961A.pf	SUCCESS	
11:05:36 8:53:57.34 PM	svchost.exe	CreateFile	C:\Windows\Prefetch\NOTEPAD.EXE-E81B961A.pf	SUCCESS	Desired Access: Generic...
11:05:36 8:55:55.77 PM	svchost.exe	WriteFile	C:\Windows\Prefetch\NOTEPAD.EXE-E81B961A.pf	SUCCESS	Offset: 0, Length: 13,166...
11:05:36 8:56:15.33 PM	svchost.exe	CloseFile	C:\Windows\Prefetch\NOTEPAD.EXE-E81B961A.pf	SUCCESS	

Showing 11 of 30,186 events (0.036%) Backed by page file

В строках с 1 по 4 показано, что файл предвыборки для приложения Notepad был считан в контексте процесса Notepad в ходе запуска. В строках с 5 по 11, имеющих метку времени на 10 секунд позже, чем у первых трех 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. 10.49). В этот список также включаются отслеженные службой супервыборки файлы, к которым происходят частые обращения.

Затем служба супервыборки запускает системную программу дефрагментации диска с ключом командной строки, предписывающим ей 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 будут 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. (Дополнительная информация о дефрагментации дана в главе 12.)

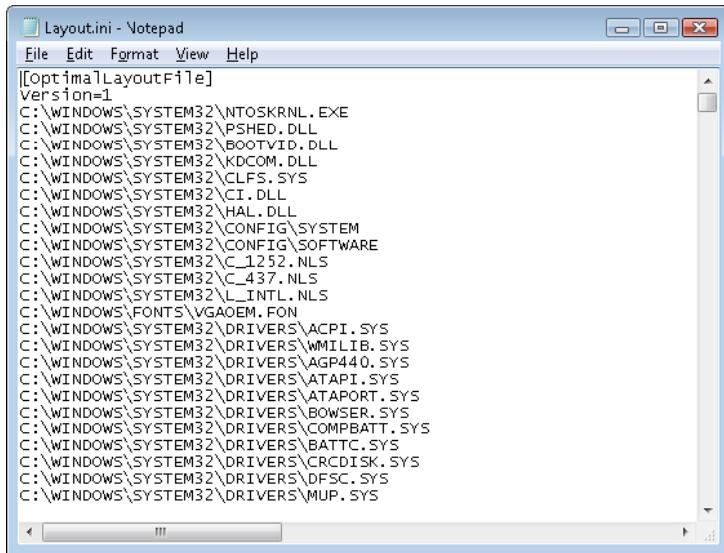


Рис. 10.49. Файл схемы дефрагментации предвыборки

Политика размещения

Когда поток сталкивается с ошибкой отсутствия 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, выбирая страничные блоки, 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.

Политики замещения могут далее характеризоваться либо как глобальные, либо как локальные. Глобальная политика замещения позволяет разрешать ошибки отсутствия страниц за счет любого страничного блока независимо от того, принадлежит он или не принадлежит другому процессу. 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 требует ограничить поиск наиболее старой из страниц того процесса, при выполнении которого произошла ошибка отсутствия страницы. Глобальные политики замещения делают процесс зависимым от поведения других процессов — приложение с ненадлежащим поведением может подорвать работу всей операционной системы, вызывая 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 привилегия 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 подкачуку при изобилии памяти (и наоборот, диспетчер памяти сократит рабочий набор процесса ниже минимума, если процесс не использует подкачуку 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 `M1MaximumWorkingSet` ядра, которая задает жесткий верхний предел на основе максимумов рабочих наборов, перечисленных в табл. 10.21.

Таблица 10.21. Верхний предел для максимумов рабочих наборов

Версия Windows	Максимум рабочих наборов
x86	2047,9 Мбайт
x86-версии Windows, загруженные с параметром <code>increaseuserserv</code>	2047,9 Мбайт плюс увеличение пользовательского виртуального пространства (Мбайт)
IA64	7152 Гбайт
x64	8192 Гбайт

Когда возникает ошибка отсутствия страницы, оцениваются лимиты рабочего набора процесса и объем имеющейся в системе свободной памяти. Если условия позволяют, диспетчер памяти разрешает процессу увеличить 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 отсутствия 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 Windows-функции `SetProcessWorkingSetSizeEx` также можно инициировать усечение рабочего набора собственного процесса, например после инициализации процесса.)

Диспетчер рабочих наборов исследует доступную память и принимает решение, какой из имеющихся рабочих наборов должен быть усечен. Если есть достаточно 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 РТЕ-записи, чтобы увидеть, осуществлялся ли доступ к странице. Если бит посещения сброшен, страница считается *возрастной* (aged), то есть 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 РТЕ-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

Исследовать размеры рабочих наборов процессов можно с помощью монитора производительности, наблюдая за значениями счетчиков производительности, показанных в следующей таблице.

Счетчик	Описание
Process: Working Set (Процесс: Рабочий набор)	Текущий размер рабочих наборов выбранных процессов в байтах
Process: Working Set Peak (Процесс: Рабочий набор (пик))	Пиковый размер рабочих наборов выбранных процессов в байтах
Process: Page Faults/sec (Процесс: Ошибок страницы/с)	Количество ошибок отсутствия страниц в секунду для процесса

Узнать размер рабочего набора процесса можно также с помощью нескольких других утилит просмотра процессов (таких, как диспетчер задач и Process Explorer).

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. Вы увидите, что полученная 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:

```
C:\temp>testlimit -d 1 -c 800
```

```
Testlimit v5.2 - test Windows limits
Copyright (C) 2012 Mark Russinovich
Sysinternals - www.sysinternals.com
```

```
Process ID: 700
```

продолжение ↗

```
Leaking private bytes 1 MB at a time...
Leaked 800 MB of private memory (800 MB total leaked). Lasterror: 0
The operation completed successfully.
```

Теперь запустите программу Process Explorer. Выберите в меню команду View ▶ Select Columns (Вид ▶ Выбрать столбцы), перейдите на вкладку Process Memory (Память процесса) и включите счетчики Private Bytes (Закрытые байты), Virtual Size (Виртуальный размер), Working Set Size (Размер рабочего набора), WS Shareable Bytes (Совместно используемая память рабочего набора) и WS Private Bytes (Закрытые байты рабочего набора). Затем, как показано на рисунке, найдите три экземпляра TestLimit.

Process	PID	CPU	Private Bytes	Virtual Size	Working Set	WS Private	WS Shareable
svchost.exe	3792		29,240 K	124,332 K	14,352 K	7,784 K	6,568 K
System	4	0.31	56 K	7,040 K	1,888 K	48 K	1,840 K
System Idle Pr...	0	97.19	0 K	0 K	24 K	0 K	0 K
taskhost.exe	2084		2,384 K	41,772 K	5,324 K	1,388 K	3,936 K
taskmgr.exe	3048	0.20	2,344 K	69,324 K	8,396 K	2,080 K	6,316 K
Testlimit.exe	1544		2,868 K	844,620 K	1,932 K	436 K	1,496 K
Testlimit.exe	2828		822,068 K	844,620 K	1,928 K	436 K	1,492 K
Testlimit.exe	700		822,064 K	844,620 K	822,772 K	821,232 K	1,540 K

CPU Usage: 2.81% | Commit Charge: 34.82% | Processes: 49 | Physical Usage: 44.00%

Новый процесс TestLimit (показан третьим по счету) имеет PID-идентификатор 700. 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 (822 064 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 частично показан выведенный на экран список рабочего набора процесса WinDbg.

```
1kd> !wsle 7

Working Set @ c0802000
FirstFree 209c FirstDynamic 6
LastEntry 242e NextSlot 6 LastInitialized 24b9
NonDirect 0 HashTable 0 HashTableSize 0
```

```
Reading the WSLE data.....
```

```
Virtual Address Age Locked ReferenceCount
```

```
c0600203 0 1 1
c0601203 0 1 1
c0602203 0 1 1
c0603203 0 1 1
c0604213 0 1 1
c0802203 0 1 1
2865201 0 0 1
1a6d201 0 0 1
3f4201 0 0 1
707ed101 0 0 1
2d27201 0 0 1
2d28201 0 0 1
772f5101 0 0 1
2d2a201 0 0 1
2d2b201 0 0 1
2d2c201 0 0 1
779c3101 0 0 1
c0002201 0 0 1
7794f101 0 0 1
7ffd1109 0 0 1
7ffd2109 0 0 1
7ffc0009 0 0 1
7ffb0009 0 0 1
77940101 0 0 1
77944101 0 0 1
112109 0 0 1
320109 0 0 1
322109 0 0 1
77949101 0 0 1
110109 0 0 1
77930101 0 0 1
111109 0 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 (те, чьи адреса укладываются в диапазон 0x7nnnnnnn), третий относятся к страницам с кодом самого файла Windbg.exe.

Диспетчер настройки баланса и поток 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 отсутствия страниц или список свободных страниц слишком мал, диспетчер памяти активирует диспетчер настройки баланса, чтобы тот вызвал диспетчера рабочих наборов и приступил к усечению этих наборов. При избытке памяти диспетчера 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 в связи с истечением тайм-аута своего односекундного таймера, он предпринимает следующие пять шагов:

1. Ставит в очередь DPC-вызовов, связанный с односекундным таймером. В качестве DPC-процедуры выступает процедура `KiScanReadyQueues`, которая ищет потоки, чей приоритет может быть повышен, чтобы они не вызывали перезагруженность центрального процессора. (См. раздел «Повышения приоритета, связанные с перезагруженностью центрального процессора (CPU starvation)» в главе 5 части I.)
2. При каждой четвертой активации диспетчера настройки баланса в связи с истечением тайм-аута односекундного таймера он сигнализирует об активации другого системного потока (`KiSwapperThread`), который называется потоком подкачки (процедура `KeSwapProcessOrStack`).
3. Диспетчер настройки баланса проверяет ассоциативные списки и настраивает в случае необходимости их глубину (для ускорения доступа, а также для снижения нагрузки на пул и уменьшения его фрагментации).
4. Диспетчер настройки баланса настраивает разрешения на передачу очередных пакетов запросов ввода-вывода для оптимизации использования ассоциативных списков, имеющихся у каждого процессора и применяемых при пополнении IRP-пакетов. Это позволяет улучшить масштабируемость при большой загруженности тех или иных процессоров операциями ввода-вывода.
5. Диспетчер настройки баланса вызывает диспетчера рабочих наборов диспетчера памяти. (У диспетчера рабочих наборов имеется собственный внутренний счетчик, регулирующий момент усечения рабочего набора и определяющий, насколько агрессивно это нужно делать.)

Поток подкачки также активируется имеющимся в ядре кодом планирования, если у потока, который нужно запустить, выгружен стек ядра или если выгружен весь процесс. Поток подкачки ищет потоки, находящиеся в режиме ожидания в течение 15 секунд (или трех секунд на системе, имеющей менее 12 Мбайт оперативной памяти). Если такой поток будет найден, его стек ядра переводится в переходное состояние (страницы перемещаются в список измененных или ожидающих), чтобы вернуть его физическую память, работая по принципу, если поток так долго ждал, то может подождать и еще. Когда из памяти удаляется стек ядра последнего потока, имеющегося в процессе, процесс помечается как полностью выгруженный. Поэтому, к примеру, у процессов, простояивающих длительный период времени (таких, как Winlogon после вашего входа в систему), может быть нулевой размер рабочего набора.

Системные рабочие наборы

По аналогии с тем, как рабочие наборы процессов используются для управления выгружаемыми порциями их адресных пространств, управление выгружаемым кодом и данными в системном адресном пространстве проводится 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 РТЕ-записей (`MmSystemPtesWs`) содержит выгружаемый код и данные загруженных драйверов и образа ядра, а также страницы из разделов, отображаемых на системное пространство.

Размеры этих рабочих наборов, как и размеры составляющих их компонентов, можно узнать с помощью счетчиков производительности или системных переменных (табл. 10.22). Следует иметь в виду, что счетчики производительности показывают значения в байтах, а системные переменные — в страницах.

Таблица 10.22. Счетчики производительности, относящиеся к системному рабочему набору

Счетчик производительности, байтов	Системная переменная, страниц	Описание
Memory: Cache Bytes (Память: Байт кэш-памяти), а также Memory: System Cache Resident Bytes (Память: Резидентных байт системного кэша)	<code>MmSystemCacheWs.WorkingSetSize</code>	Физическая память, потраченная на файл системного кэша
Memory: Cache Bytes Peak (Память: Байт кэш-памяти (пик))	<code>MmSystemCacheWs.Peak</code>	Пиковый размер системного рабочего набора
Memory: System Driver Resident Bytes (Память: Резидентных байт системных драйверов)	<code>MmSystemDriverPage</code>	Физическая память, потраченная на выгружаемый код драйверов устройств
Memory: Pool Paged Resident Bytes (Память: Байт в резидентном страницном пуле)	<code>MmPagedPoolWs.WorkingSetSize</code>	Физическая память, потраченная на выгружаемый пул

Страницную активность в рабочем наборе системного кэша можно также оценить путем изучения показаний счетчика `Memory: Cache Faults/sec` (Память: Ошибок кэш-памяти/с), в котором регистрируются ошибки отсутствия страниц, возникающие в рабочем наборе системного кэша (как серьезные, так и менее серьезные). Значение этого счетчика содержится в системной переменной `MmSystemCacheWs.PageFaultCount`.

События уведомлений в памяти

Windows обеспечивает уведомление процессов пользовательского режима и драйверов режима ядра о том, что объемы физической памяти, выгружаемого и невыгружаемого пуль, 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 предоставлен любой функции ожидания. Когда памяти слишком мало (или слишком много), ожидание завершается, и 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 устанавливает в каталоге `\KernelObjects`, поскольку уведомление, выполненное диспетчером памяти, сигнализирует об одном из определенных им глобальных именованных объектов событий (табл. 10.23).

Таблица 10.23. 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

Имя события	Описание
MemoryErrors	Обнаружена плохая страница (необнуленная страница, подлежащая обнулению)

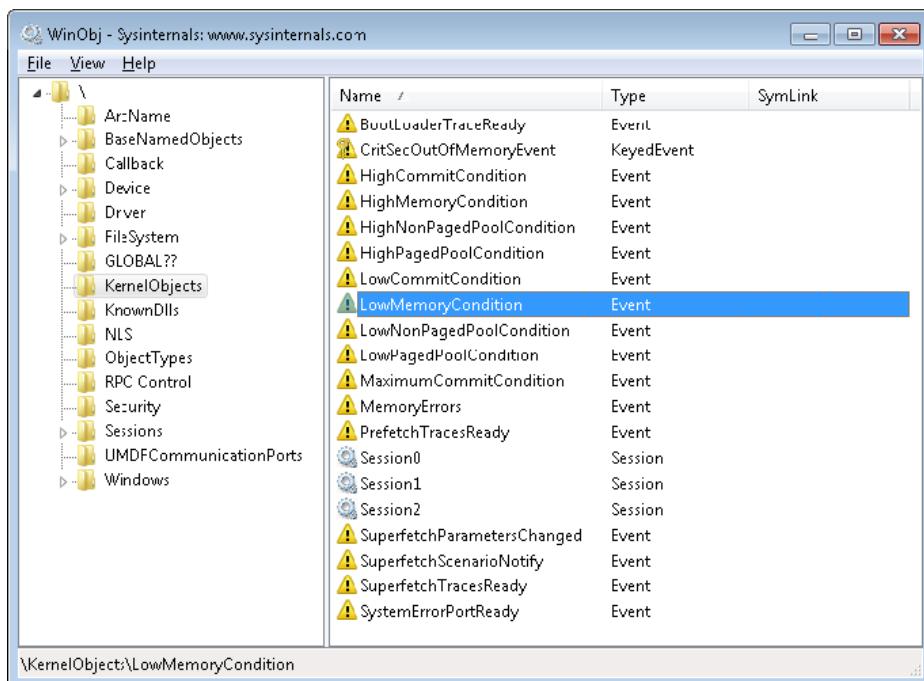
Когда обнаруживаются данные условия в состоянии памяти, сигнализируется о наступлении соответствующего события, в результате ожидающие программные потоки активируются.

ПРИМЕЧАНИЕ

Пороговые значения можно переопределить путем добавления в раздел HKLM\SYSTEM\CurrentControlSet\Session Manager\Memory Management DWORD-параметров реестра, LowMemoryThreshold or HighMemoryThreshold, в которых указывается количество мегабайтов для использования в качестве малых и больших показателей потребления памяти. Система может быть настроена так, чтобы при обнаружении плохой страницы аварийно завершать свою работу, а не сигнализировать о событии ошибки памяти. Для этого нужно в том же самом разделе реестра настроить DWORD-параметр PageValidationAction.

ЭКСПЕРИМЕНТ: ПРОСМОТР УВЕДОМЛЕНИЙ О СОСТОЯНИИ РЕСУРСОВ ПАМЯТИ

Чтобы увидеть события, уведомляющие о состоянии ресурсов памяти, нужно запустить программу WinObj, созданную в Sysinternals, и щелкнуть 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) ссылок на объекты.

Чтобы убедиться, что любые процессы в системе запрашивают уведомления о ресурсах памяти, нужно поискать в таблице дескрипторов ссылки вида «LowMemoryCondition» или «HighMemoryCondition». Это можно сделать, выбрав в Process Explorer команду Find (Поиск) и указав характеристику Handle (Дескриптор). Можно также воспользоваться программой WinDbg. (Описание таблицы дескрипторов можно найти в разделе «Диспетчер объектов» главы 3 части I.)

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, а прогноз их будущих требований к памяти не делался. Во-вторых, данные, использующиеся 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 запрашивать свои данные и код не из кэша, а с диска. В клиентских версиях Windows благодаря технологии супервыборки сделан большой шаг в направлении преодоления этих ограничений.

Компоненты

Технологию супервыборки реализуют в системе несколько компонентов, которые совместно осуществляют упреждающее управление памятью и минимизируют вли-

яние 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 трасс** (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 ходе 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). Основывая свою работу на информации, предоставляемой агентами супервыборки, а также на текущем состоянии системы (например, на состоянии списков страниц, составленных по приоритетам), перенастройщик баланса является специализированным агентом, который находится в службе пользовательского режима супервыборки, 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, и он является единственным агентом, который фактически предпринимает 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, находящиеся внутри диспетчера памяти и позволяющие запрашивать подробную информацию о состоянии каждой страницы в базе данных PFN-номеров, о текущем подсчете страниц для каждого списка страниц и приоритетного списка, а также о многом другом. На рис. 10.50 представлена диаграмма работы нескольких компонентов супервыборки. Компоненты супервыборки применяют также ввод-вывод с учетом приоритетов (дополнительные сведения о приоритетах ввода-вывода даны в главе 8), чтобы минимизировать влияние на пользователя.

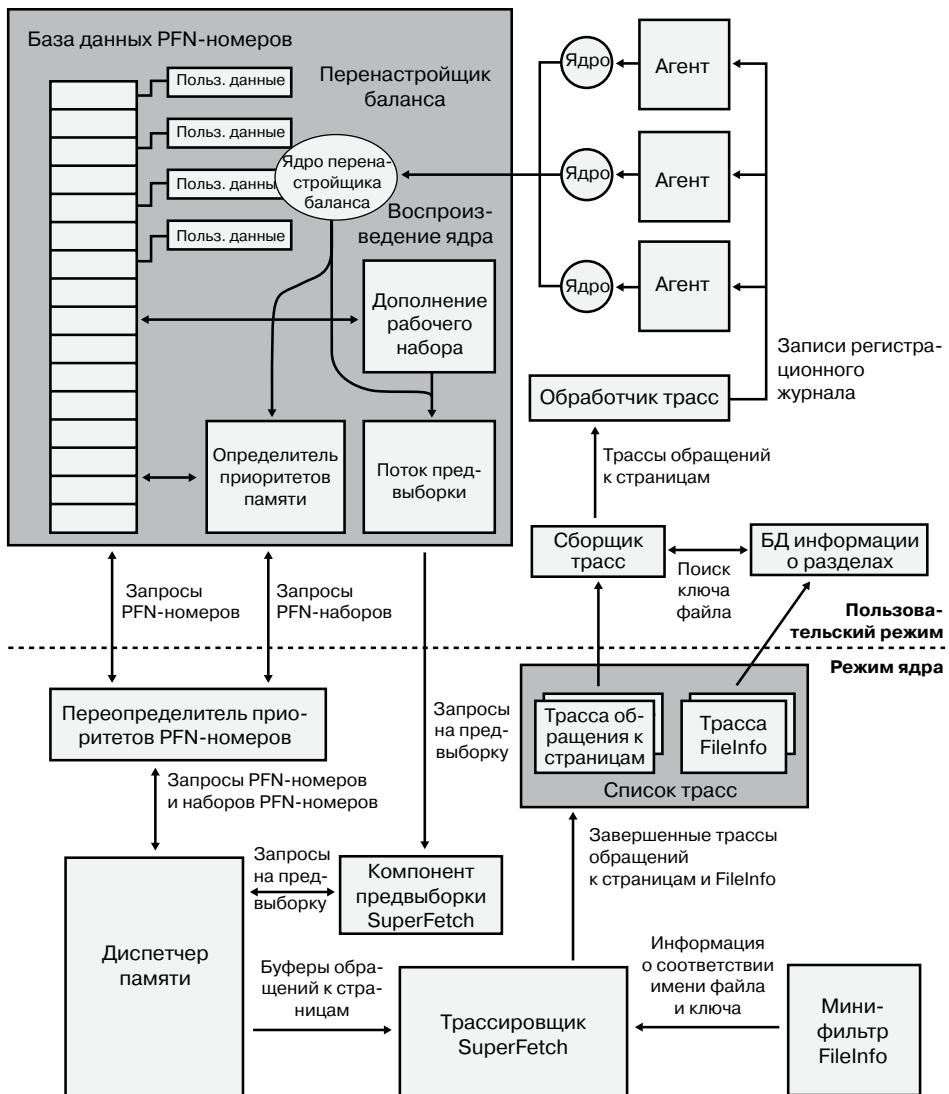


Рис. 10.50. Диаграмма супервыборки

Трассировка и протоколирование

Большинство своих решений механизм супервыборки принимает на основе той информации, которая была сведена в единый комплекс, 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 посещения (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. Для отсеивания этой информации механизм супервыборки задействует существующие функциональные возможности фильтрации и в дополнение к ним драйвер FileInfo. (Дополнительные сведения о фильтрующих драйверах есть в главе 8.) Этот драйвер находится в стеке 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 12), что дает ему возможность детально контролировать обращения к файлам. Основная работа драйвера FileInfo заключается в привязке 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 явной 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 быть активирован, чтобы заполнить страницу по-иному. Эти решения принимаются путем анализа такой информации, как выделение памяти внутри рабочих наборов, содержимое списков обнуленных, измененных и ожидающих страниц, количество ошибок отсутствия страниц, состояние битов посещения в РТЕ, результаты постраничного отслеживания использования страниц, текущий уровень потребления виртуальных адресов, размер рабочего набора.

Заданной трассой может быть либо трасса обращений к страницам, когда трассировщик отслеживает (с помощью бита посещения), к каким страницам было обращение

со стороны процесса (как к файловой странице, так и к закрытой памяти), либо трасса регистрации имен, когда отслеживаются обновления отображений имен файлов на ключи файлов (что позволяет механизму супервыборки отобразить страницу, связанную с файловым объектом) применительно к реальному файлу на диске.

Хотя в трассе супервыборки отслеживаются только обращения к страницам, служба супервыборки обрабатывает эту трассу 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 такие же 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. Этими сценариями являются ожидание и гибернация, а также быстрое переключение между пользователями. Каждый из этих сценариев имеет различные цели, но все они направлены на достижение главной цели: минимизация или полной ликвидации серьезных ошибок отсутствия страниц.

- ❑ При гибернации целью является разумный выбор 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.
- ❑ При ожидании целью является полное устранение серьезных ошибок отсутствия страниц после возобновления работы машины. Поскольку обычная система способна возобновить работу менее чем за 2 секунды, а на раскручивание жесткого диска после длительногоостояния может уйти 5 секунд, одна-единственная серьезная ошибка отсутствия страницы вызывает задержку в цикле возобновления работы. Для решения проблемы механизм супервыборки наделяет высокими приоритетами те страницы, которые понадобятся при выходе из режима ожидания.
- ❑ При быстром переключении между пользователями целью является сохранение точной расстановки приоритетов и осмысление состояния памяти для каждого пользователя, чтобы переключение на другого пользователя приводило к немедлен-

ному переходу в рабочее состояние нужного пользовательского сеанса и не вносило больших задержек в плане доступа к соответствующим страницам.

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`, с помощью которого контролируются компоненты ядра и генерируются запросы, например на запуск, на завершение выполнение сценария, на привязку к сценарию одной или нескольких трасс.

Каждый сценарий описывается файлом плана, который как минимум содержит список страниц, связанных со сценарием. Значения приоритетов страниц назначаются также в соответствии с определенными правилами, которые рассмотрены 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, но его истинная сила проявляется в супервыборке при присваивании приоритетов и перебалансировке, не требуя ручного ввода данных в приложениях или жесткого кодирования информации о степени важности процессов.

Механизм супервыборки присваивает страницы приоритет 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 учитываются, например, такие данные, как место, откуда эта страница поступила (из какого списка), как наличие других структур 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 самыми низкими приоритетами из списка ожидающих страниц, как показано в эксперименте «Просмотр списков ожидающих страниц, расставленных по приоритетам» (с. 361). 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 можно было дольше удержать определенное количество особо важных страниц от повторного использования. Все остальные приоритеты расставляются на основе оценки каждой страницы.

Поскольку механизм супервыборки «изучает» пользовательскую систему, он может начинать вообще без данных истории и постепенно 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 захвату нужных данных, а затем превратить их в заранее выстроенные трассы. Перед поставкой Windows команда разработчиков механизма супервыборки провела трассировку наиболее распространенных приемов и эталонов применения программ и данных, с которыми, вероятно, сталкиваются все пользователи, например 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). И 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. Фактически, 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 вредит системе. 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, например 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. Он также работает в связке со схемами расстановки приоритетов диспетчера ввода-вывода, чтобы операции ввода-вывода выполнялись с очень низким приоритетом и не создавали помех пользователю. 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 требует 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, к которым редко обращаются, хотя возможно, что в определенные периоды времени к ним могут обращаться часто. При обнаружении подобной ситуации перенастройщик баланса запускается еще раз, чтобы опустить 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 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, то есть 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, продолжить уклоняться от использования диска помогут такие технологии, как ReadyBoost и ReadyDrive.

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 кон-

кретному 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 и полезной программой, то вполне возможно, что ее приоритет был бы по крайней мере не ниже 5).

Механизм супервыборки реагирует на две конкретные схемы ввода-вывода: последовательный доступ к файлам (с перебором всех данных файла) и последовательный доступ к каталогам (с перебором всех файлов каталога). 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). Эти страницы, которые можно назвать устойчивыми, получают, по сути, новый приоритет, равный 2.

Поскольку данный компонент супервыборки реагирует на ситуацию, а не занимается прогнозированием, 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 поискать во всех исполняемых файлах строку, которая является частью каталога. Если найдется множество таких файлов, 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 от него модели поведения.

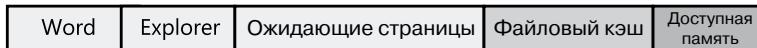
Размышляя о повышении устойчивости, важно помнить, что оптимизация супервыборки в основном заключается в том, что компонент супервыборки постоянно отслеживает характерные схемы работы и обновляет свое понимание системы, благодаря чему ему удается избежать извлечения бесполезных данных. Хотя изменения 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, самое худшее, что может произойти, заключается в том, что 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.

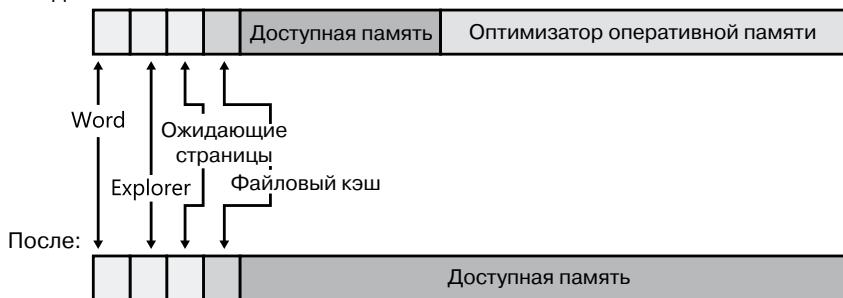
ПРОГРАММЫ ОПТИМИЗАЦИИ ОПЕРАТИВНОЙ ПАМЯТИ

Хотя супервыборка обеспечивает весьма ценную и реальную оптимизацию использования памяти для различных сценариев, на поддержку которых она нацелена, распространением так называемых «Оптимизаторов оперативной памяти», призванных существенно повысить объем доступной памяти в пользовательских системах, занимаются многие сторонние производители программного обеспечения. В пользовательском интерфейсе подобных оптимизаторов обычно присутствует график с надписью «Доступная память», и горизонтальная линия, как правило, показывает тот объем памяти, который оптимизатор попытается освободить при запуске. После запуска задания на оптимизацию имеющийся в утилите счетчик доступной памяти зачастую идет вверх, иногда показывая весьма впечатляющие темпы, подразумевая, что программа реально освобождает память для приложений. Оптимизаторы памяти работают путем выделения больших объемов виртуальной памяти и ее последующего освобождения. Следующая иллюстрация демонстрирует эффект от применения оптимизатора оперативной памяти.

До:



В ходе:



В панели «До:» показаны рабочие наборы процессов и системы, страницы в списках ожидания и свободная память перед оптимизацией. В панели «В ходе:» показано, что оптимизатор оперативной памяти обеспечил высокий спрос на память, что выразилось в возникновении в короткий период времени множества ошибок отсутствия страниц. В ответ на это диспетчер памяти увеличил объем рабочего набора оптимизатора. Эта экспансия рабочего набора оперативной памяти происходит за счет свободной памяти, за которой следуют ожидающие страницы, а когда доступной памяти становится мало,

экспансия идет за счет рабочих наборов других процессов. В панели «После:» показано, что после освобождения памяти оптимизатором диспетчер памяти переместил все страницы, которые были выделены оптимизатору, в список свободных страниц (эти страницы в итоге обнуляются потоком обнуления страниц и перемещаются в список обнуленных страниц), внося тем самым вклад в увеличение объема свободной памяти.

Хотя обретение большего объема свободной памяти может показаться весьма ценным результатом, подобный способ ее получения нельзя признать таковым. По мере того как оптимизаторы оперативной памяти заставляют показатель счетчика свободной памяти расти, из памяти выбрасываются код и данные других процессов. Если у вас, к примеру, был запущен редактор Microsoft Word, то текст открытых документов и программный код, которые были частью рабочего набора Word до оптимизации (и поэтому присутствовали в физической памяти), придется заново считывать с диска, как только вы продолжите редактировать свой документ. Кроме того, из-за очистки списков ожидающих страниц теряются ценные кэшированные данные, включая и основную часть кеша супервыборки. Особенно серьезное падение производительности ожидает серверы, где усечение системного рабочего набора приводит к удалению кэшированных файловых данных из физической памяти, что при следующих к ним обращениях вызовет ошибки отсутствия нужных страниц с серьезными последствиями.

Служба ReadyBoost

Хотя сегодня оперативная память стала намного доступнее и относительно дешевле, чем десять лет назад, она все еще дороже такого вторичного хранилища 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 поиске данных, поэтому хранение данных супервыборки на диске — такая же неудачная затея, как выгрузка страницы с последующей серьезной ошибкой отсутствия страницы во внутренней памяти. (У твердотельных дисков некоторые недостатки сглажены, но они дороже и по-прежнему 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. (Практически карты CompactFlash и Secure Digital почти всегда сопрягаются через USB-адаптер, поэтому в системе все они фигурируют как UFD-диски.) Они дешевле оперативной памяти и более емкие, кроме того, ввиду отсутствия подвижных частей, время поиска у них намного меньше, чем у жестких дисков.

Особенно затратным по времени является произвольный дисковый ввод-вывод, потому что время наведения 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 13 миллисекунд — это целая вечность для современных процессоров с тактовой частотой 3 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, находящегося между памятью и дисками.

Служба ReadyBoost 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, как показано на рис. 10.51. Стока «Emd» представляет собой сокращение от External Memory Device (внешнее запоминающее устройство) — рабочее название ReadyBoost.

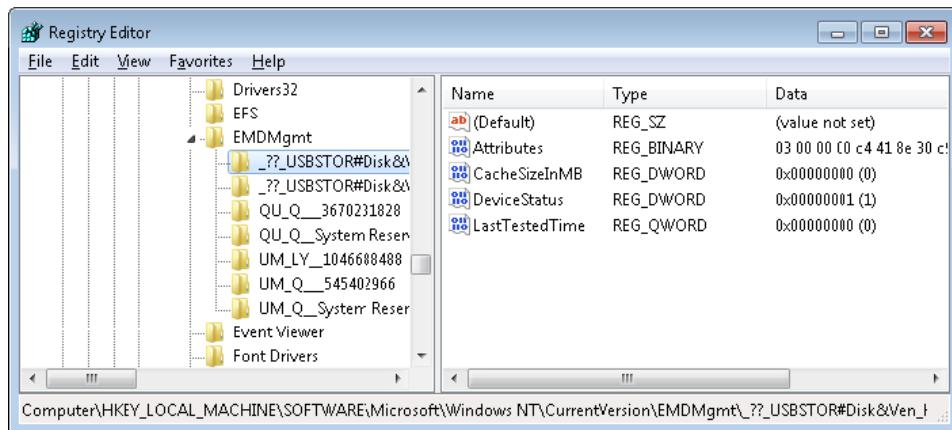


Рис. 10.51. Результаты тестирования устройства в реестре

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 отдать 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. Но есть исключения, например данные, которые долго 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 ключ сеанса произвольно генерируется во время каждой начальной загрузки, чтобы гарантировать закрытость данных в кэше после удаления устройства из системы.

Когда служба ReadyBoost замечает произвольные считывания, которые могут быть удовлетворены из кэша, она обслуживает их оттуда, но поскольку у жесткого диска лучшие показатели последовательного считывания, 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 предлагает дополнительные преимущества и позволяет решить проблему.

Технология ReadyDrive

ReadyDrive – это инструмент Windows, использующий 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, а лимит кэша, установленный в Windows, составляет 2 Тбайт.

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 включенной, 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 некоторыми 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.

Унифицированное кэширование

Для лучшего понимания концепций функционирования технологий супервыборки, ReadyBoost и ReadyDrive, мы рассматривали их независимо друг от друга. Однако

функции выделения внешней памяти и отслеживания содержимого, выполняемые в рамках этих технологий, реализованы в операционной системе в виде унифицированного кода в комплексе друг с другом. Единый механизм кэширования зачастую называют *диспетчером хранилищ* (store manager), хотя на самом деле диспетчер хранилищ является лишь одним из компонентов.

Унифицированное кэширование было разработано с целью задействовать возможности аппаратуры хранения данных различных типов, которая может присутствовать в системе. Например, компонент супервыборки вместо системной оперативной памяти может использовать либо флэш-память гибридного жесткого диска (при ее доступности), либо UFD-диск (при его доступности). Поскольку флэш-память гибридного жесткого диска, скорее всего, нужно приберечь для циклов завершения работы системы и ее начальной загрузки, ее предпочтительнее использовать для кэширования данных, помогающих оптимизировать начальную загрузку, а для других данных лучше выбрать оперативную память системы. (В дополнение к оптимизации начальной загрузки, NVRAM-память на гибридном жестком диске, если таковая имеется, в целом предпочтительнее для размещения кэша, чем UFD-диск. UFD-диск может быть в любое время вынут, то есть он может исчезнуть, поэтому кэш на UFD-диске должен всегда использоваться для сквозной записи на реальный жесткий диск. NVRAM-памяти на гибридном жестком диске можно разрешить работать в режиме обратной записи, поскольку она не может исчезнуть без исчезновения самого жесткого диска.)

Полностью архитектура унифицированного механизма кэширования показана на рис. 10.52.

Основной компонент кэширования называется «хранилищем» (store). В каждом хранилище реализуются функции для добавления данных в обслуживающее запоминающее устройство (которое может быть в оперативной памяти системы или в NVRAM-памяти) для считывания из него данных или для удаления данных.

Все данные в хранилище обрабатываются с помощью *страниц хранения* (store pages), часто называемых просто *страницами* (pages). Размер страницы хранения равен размеру страницы физической и виртуальной памяти системы (4 или 8 Кбайт на платформах Itanium), независимо от «размера блока» (иногда называемого «размером сектора»), представляемого основным устройством хранения данных. Это позволяет страницам хранения эффективно отображаться и перемещаться между хранилищами, оперативной памятью системы и страничными файлами (которые всегда группируются в блоки одинакового размера). Современные жесткие диски, продвигающиеся в направлении «расширенного формата» и экспортирующие размер блока в 4 Кбайт, хорошо вписываются в этот подход. Страницы хранения в пределах блока идентифицируются с помощью «ключей хранения», интерпретация которых зависит от конкретного хранилища.

При записи в хранилище само хранилище отвечает за буферизацию данных, поэтому при вводе-выводе на реальном устройстве хранения данных используются большие буферы. Это повышает производительность, поскольку NVRAM-устройства, так же как и физические жесткие диски, с небольшими произвольными записями работают плохо. Перед записью на устройство хранения хранилище может также выполнить сжатие и кодирование данных.

Управление всеми хранилищами и их содержимым осуществляют компонент, который называется *диспетчером хранилищ* (store manager). Он реализован в виде

компоненты службы супервыборки в файле Sysmain.dll, набора исполняющих служб (`SmXxx`, например `SmPageRead`) внутри файла Ntoskrnl.exe и фильтрующих драйверов в стеке дискового хранилища, Storemgr.sys. Логически он работает на уровне, находящемся чуть выше всех хранилищ. С хранилищами связан только диспетчер хранилищ; все остальные компоненты взаимодействуют с диспетчером хранилищ. Запросы к диспетчеру хранилищ очень похожи на запросы из диспетчера хранилищ к хранилищу, к ним относятся запросы на хранение данных, на извлечение данных или на удаление данных из хранилища. Однако запросы к диспетчеру хранилищ на хранение данных содержат параметр, указывающий на то, в какие хранилища они должны записываться.

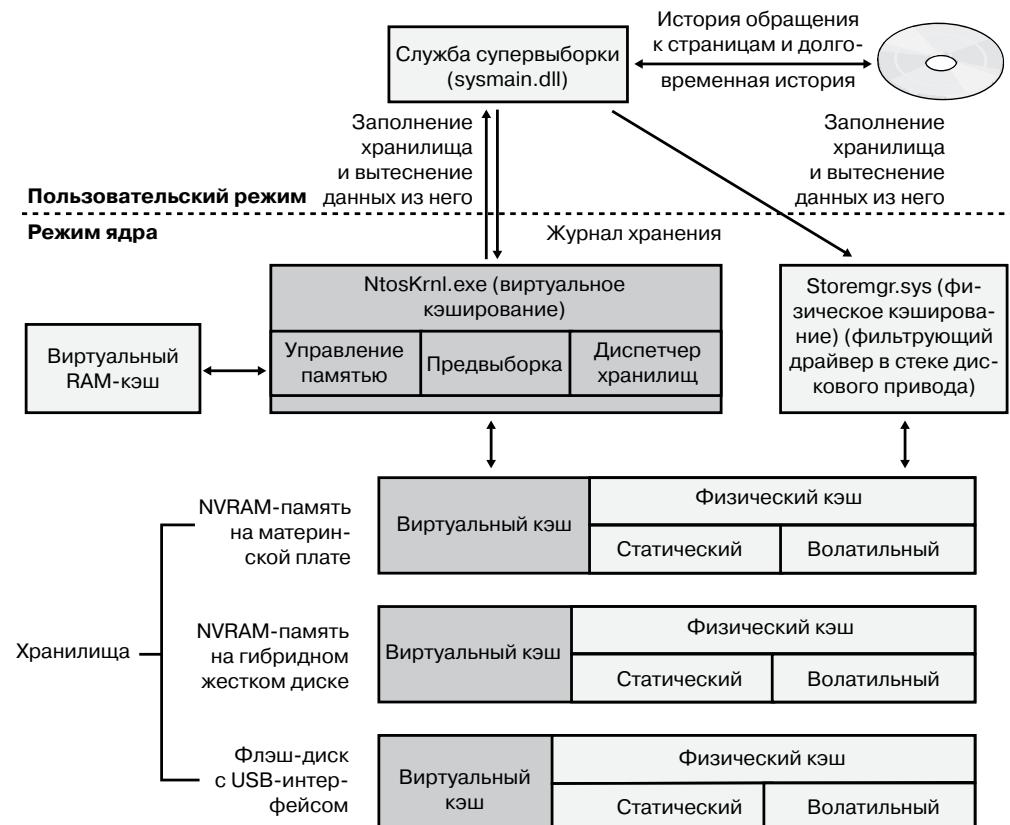


Рис. 10.52. Архитектура унифицированного механизма кэширования

Диспетчер хранилищ отслеживает, в каких хранилищах располагается каждая кэшируемая страница. Если кэшируемая страница находится в одном или нескольких хранилищах, запросы на извлечения этой страницы направляются диспетчером хранилищ к тому или иному хранилищу в соответствии с тем, какое из хранилищ работает быстрее или какое из них меньше загружено.

Диспетчер хранилищ оценивает хранилища следующим образом. Во-первых, хранилище может находиться в оперативной памяти системы или в какой-нибудь из разновидностей энергонезависимой памяти (либо на UFD-диске, либо в NVRAM-памяти гибридного жесткого диска). Во-вторых, хранилища на основе NVRAM-памяти делятся на «виртуальные» и «физические» части, в то время как хранилище в оперативной памяти системы работает только как виртуальное хранилище.

Виртуальные хранилища содержат только информацию, поддерживаемую страничными файлами, включая закрытую память процессов и разделы, поддерживаемые страничными файлами. Физические кэши содержат страницы с диска за тем исключением, что в физических кэшах никогда не содержатся страницы из страничных файлов. В то же время хранилище в оперативной памяти системы может содержать страницы из страничных файлов.

Физически кэши делятся на «статические» и «волатильные» (или «динамические») области. Содержимое статической области полностью определяется службой диспетчера хранилищ пользовательского режима. Для заполнения статической области диспетчера хранилищ прибегает к журналам историй обращений к данным. В то же время волатильная или динамическая область каждого хранилища заполняется самостоятельно на основе запросов на чтение и запись, которые проходят через стек дискового хранилища, что во многом напоминает работу автоматического кэша на оперативной памяти с обычным жестким диском. Хранилища, реализующие динамическую область, отвечают за отправку отчетов диспетчеру хранилищ о каждом таком автоматически кэшированном (и сброшенном) содержимом.

Здесь мы лишь кратко познакомились с организацией и работой унифицированного механизма кэширования. На момент написания данной книги в мониторе производительности не было соответствующих счетчиков, как и иных средств операционной системы, позволяющих оценить работу этого механизма, за исключением счетчиков, относящихся к объекту «Кэш», который является предшественником диспетчера хранилищ.

Отражение процессов

Довольно часто случается, что процесс демонстрирует некое проблемное поведение, но поскольку он работает в рамках какой-нибудь службы, его приостановка для создания полного дампа памяти или интерактивная отладка нежелательны. Продолжительность приостановки процесса для создания дампа может быть сведена к минимуму при получении мини-дампа, захватывающего регистры и стеки программных потоков наряду со страницами памяти, на которые ссылаются регистры. Однако у этого типа дампа 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 из клона может быть получен более объемный дамп, охватывающий всю доступную процессу память пользователяского 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`. Сначала она создает раздел общей памяти, заполняет его параметрами и отображает на текущий и целевой процессы. Затем создаются два объекта событий, которые дублируются в целевой процесс, чтобы текущий и целевой процессы могли синхронизировать свои операции. После этого в целевой процесс посредством вызова 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`. 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.

Программный поток, внедренный в целевой процесс, назначает дополнительный объект события, предназначенный для синхронизации с клонированным процессом, как только тот будет создан. Затем он вызывает функцию `RtlCloneUserProcess`, передавая ей параметры, полученные из отображения общей с исходным процессом памяти. 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 связанные блокировки. Это может пригодиться для отладки, поскольку копия дампа памяти со структурой данных останется в согласованном состоянии.

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`, предназначеннной для создания объекта процесса и ини-

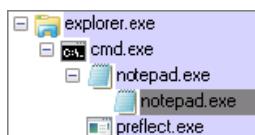
циализации программного потока, она вызывает функцию `MmInitializeProcessAddressSpace` с флагом, указывающим на то, что адресом должна быть копия (полученная копированием при записи) целевого процесса, а не исходное адресное пространство процесса. Для эффективного клонирования адресного пространства диспетчер памяти использует ту же поддержку, которую он предоставляет службам API-функции `fork` 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 `RtlDetectHeapsLeaks`, 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`.

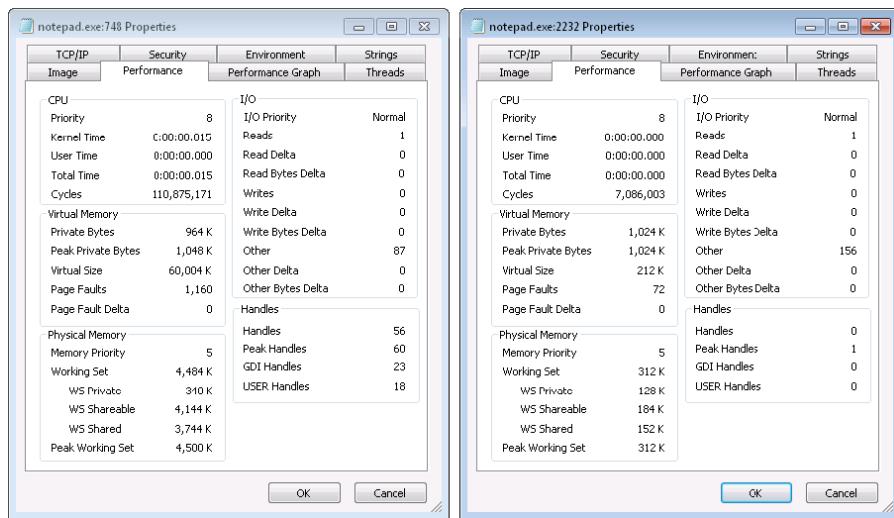
Когда RADAR и WDI используют механизм отражения процессов, они вызывают функцию `RtlCreateProcessReflection` с запросом к ней вернуть им дескриптор клонированного процесса и с запросом к клону приостановиться после инициализации. Затем они генерируют мини-дамп целевого процесса, что приостанавливает этот процесс на время генерации дампа, затем они 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 несогласования не оказывают влияния на поиск и устранение неисправностей. Такие же этапы проходит разработанная в Sysinternals утилита ProcDump при задании ключа `-r`, который заставляет ее создать дамп целевого процесса.

ЭКСПЕРИМЕНТ: ИСПОЛЬЗОВАНИЕ УТИЛИТЫ PREFLECT ДЛЯ НАБЛЮДЕНИЯ ЗА ПОВЕДЕНИЕМ СИСТЕМЫ ПРИ ОТРАЖЕНИИ ПРОЦЕССА

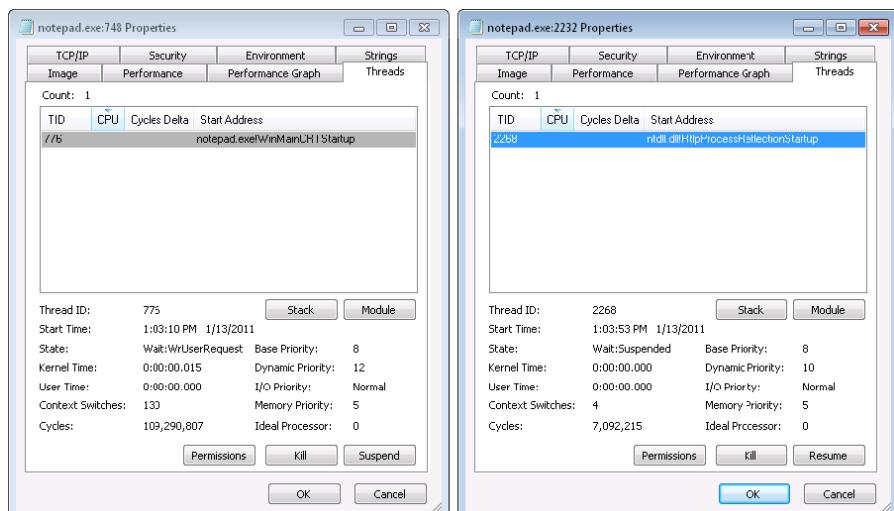
Понаблюдать за эффектами отражения процесса позволяет утилита Preflect. Сначала следует запустить программу Notepad.exe и получить идентификатор ее процесса в утилите управления процессами, такой как Process Explorer или диспетчер задач. Затем нужно открыть окно командной строки и выполнить команду `Preflect` с идентификатором процесса, указанным в качестве аргумента. Путем отражения процесса будет создан его клон. В Process Explorer можно будет наблюдать два экземпляра процесса Notepad: тот, который был запущен вами, и его клонированный дочерний экземпляр, выделенный серым фоном (серый фон свидетельствует о приостановке всех потоков процесса).



Откройте окно свойств процесса для каждого экземпляра, перейдите на вкладку Performance (Быстродействие) и поместите окна рядом для сравнения.



Эти два экземпляра легко отличить друг от друга, поскольку целевой процесс выполняется, в силу чего имеет существенно более высокие показатели счетчика рабочих циклов и более объемный рабочий набор, а у клона нет никаких ссылок на какие-либо объекты управления ядра или окна, о чем свидетельствует нулевое значение в счетчиках дескрипторов ядра, а также GDI- и USER-дескрипторов. Далее, если посмотреть на вкладку Threads (Потоки) и настроить Process Explorer на получение символов операционной системы, можно будет увидеть, что программный поток целевого процесса приступил к выполнению кода Notepad.exe, тогда как поток клона относится к потоку, который был внедрен в целевой процесс для выполнения функции RtlpProcessReflectionStartup.



Заключение

В данной главе рассказывается о том, как диспетчер памяти управляет виртуальной памятью в Windows. 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 процессам разрешается эффективно и безопасно работать с общей памятью. Описываются также и такие нетривиальные возможности диспетчера памяти, как включение отображаемых файлов и выделение разреженной памяти. Посредством Windows API подсистема среды Windows делает большинство возможностей диспетчера памяти доступными приложениям.

В следующей главе рассматривается диспетчер кэша — компонент, который тесно связан с диспетчером памяти.

Глава 11. Диспетчер кэша

Диспетчер кэша в Windows представляет собой набор функций и системных программных потоков режима ядра, совместно с диспетчером памяти обеспечивающих кэширование данных для всех драйверов файловой системы (как локальных, так и сетевых). В данной главе рассматривается порядок работы диспетчера кэша, в частности мы узнаем, как работают его ключевые внутренние структуры данных и функции, как он устанавливает размеры кэша в ходе инициализации системы, как диспетчер кэша взаимодействует с другими элементами операционной системы, как отслеживать его работу с помощью счетчиков производительности. Также описываются пять флагов Windows-функции `CreateFile`, влияющих на кэширование файлов.

ПРИМЕЧАНИЕ

Все внутренние функции диспетчера кэша рассматриваются в этой главе ровно в том объеме, который достаточен для понимания его работы. Документацию по программным интерфейсам диспетчера кэша можно найти в Windows Driver Kit (WDK). Дополнительная информация о WDK находится по адресу <http://www.microsoft.com/whdc/devtools/wdk/default.mspx>.

Основные возможности диспетчера кэша

Диспетчер кэша обладает следующими основными возможностями:

- ❑ Поддержка всех типов файловых систем (как локальных, так и сетевых), благодаря чему отпадает необходимость писать код реализации собственного диспетчера кэша для каждой файловой системы.
- ❑ Использование диспетчера памяти для контроля над тем, какие части каких файлов находятся в физической памяти (с целью нахождения компромисса между потребностями пользовательских процессов и операционной системы в физической памяти).
- ❑ Кэширование данных на основе виртуальных блоков (смещения в пределах файла), в отличие от многих других систем кэширования, работающих на основе логических блоков (смещения в пределах дискового тома), что позволяет проводить интеллектуальное упреждающее чтение и осуществлять высокоскоростной доступ к кэшу без привлечения драйверов файловой системы. (Этот метод кэширования, называемый быстрым вводом-выводом, мы рассмотрим чуть позже.)
- ❑ Поддержка «подсказок», передаваемых приложениями в ходе открытия файла (таких, как произвольный или последовательный доступ, создание временного файла и т. д.).

- Поддержка самовосстанавливающихся файловых систем (например, тех, которые ведут журналы транзакций) для восстановления данных после системного сбоя.

В данной главе речь в основном идет о том, как диспетчер кэша распоряжается этими своими возможностями, но сначала мы познакомимся с концепциями, лежащими в их основе.

Единый централизованный системный кэш

Некоторые операционные системы при кэшировании данных полагаются на каждую отдельно взятую файловую систему, и эта практика приводит либо к дублированию кода управления кэшем и памятью в операционной системе, либо к ограничениям в отношении кэшируемых данных. В отличие от подобных операционных систем, Windows поддерживает централизованное кэширование, когда кэшированием охватываются все внешние данные независимо от места их хранения (локальные жесткие диски, гибкие диски, сетевые файловые серверы или компакт-диски). При этом кэшироваться могут любые данные, будь то потоки пользовательских данных (части содержимого какого-нибудь файла и данные, получаемые при его чтении и записи) или *системные метаданные* (system metadata) файлов (такие, как заголовки каталогов и файлов). В этой главе вы узнаете, что метод обращения к кэшу, используемый в Windows, зависит от типа кэшируемых данных.

Диспетчер памяти

Одним из необычных аспектов работы диспетчера кэша является то, что он никогда не знает, каков фактический объем кэшированных данных, находящихся в физической памяти. Такое утверждение может звучать странно, поскольку целью кэша является хранение поднабора наиболее часто востребованных данных в физической памяти с целью ускорения ввода-вывода. Причиной того, что диспетчер кэша не знает, сколько данных находится в физической памяти, является способ его обращения к данным, который заключается в отображении представлений на файлы в системном виртуальном адресном пространстве с использованием стандартных *объектов разделов* (section objects); по терминологии Windows API они называются *объектами отображения файлов* (file mapping objects). (Объекты разделов являются основными примитивами диспетчера памяти и подробно рассматриваются в главе 10.) По мере обращения к этим отображаемым представлениям диспетчер памяти подкачивает блоки, отсутствующие в физической памяти. А когда возникает потребность в памяти, диспетчер памяти прекращает отображение этих страниц в кэше и, если данные изменились, выгружает их обратно в файлы.

Благодаря кэшированию на основе виртуального адресного пространства с использованием отображаемых файлов диспетчеру кэша удается отказаться от создания пакетов запросов на ввод-вывод (I/O request packets, IRP) при чтении или записи, чтобы получить доступ к данным для кэшируемых файлов. Вместо этого он при отображении части кэшируемого файла просто копирует данные в виртуальное адресное пространство или из него, а для сброса данных в память или из памяти при необходимости полагается на диспетчера памяти. Этот подход позволяет диспетчеру памяти идти

на глобальные компромиссы в отношении того, сколько памяти нужно предоставлять системному кэшу, а сколько — пользовательским процессам. (Диспетчер кэша также способен инициировать ввод-вывод, например отложенную запись, которая рассматривается в данной главе чуть позже, но для записи страниц он вызывает диспетчер памяти.) Кроме того, как показано в следующем разделе, этот подход позволяет процессам, открывающим кэшированные файлы, видеть те же данные, что и процессы, отображающие эти же файлы в своем пользовательском адресном пространстве.

Согласованность кэша

Одной из важных функций диспетчера кэша является предоставление любому процессу, обращающемуся к кэшированным данным, самой последней версии этих данных. Проблема может возникнуть в том случае, когда один процесс открывает файл (и, следовательно, этот файл кэшируется), в то время как другой процесс отображает файл непосредственно на свое адресное пространство (с помощью Windows-функции `MapViewOfFile`). Эта потенциальная проблема не проявляется под управлением Windows, потому что и диспетчер кэша, и пользовательские приложения, отображающие файлы на свое адресное пространство, задействуют одни и те же службы отображения файлов диспетчера памяти. Поскольку диспетчер памяти гарантирует, что у него есть только одно представление каждого уникального отображаемого файла (независимо от количества объектов разделов или отображенных представлений), он отображает все представления файла (даже если они перекрываются) на единый набор страниц в физической памяти, как показано на рис. 11.1. (Дополнительные сведения о том, как диспетчер памяти работает с отображаемыми файлами, можно найти в главе 10.)

К примеру, если процесс 1 имеет представление (представление 1) файла, отображаемого на его пользовательское адресное пространство, а процесс 2 обращается к тому же самому представлению через системный кэш, процесс 2 увидит любые изменения, которые вносит процесс 1, сразу после их внесения, а не после того, как они будут сброшены на диск. Диспетчер памяти не сбрасывает на диск *все* страницы, отображенные в пользовательском адресном пространстве, ограничиваясь только теми, в которые, по его сведениям, велась запись (потому что у них установлен бит изменения). Следовательно, любой процесс в Windows, обращающийся к файлу, всегда видит самую свежую версию этого файла, даже если у некоторых процессов этот файл открыт через систему ввода-вывода, а у других он отображен на их адресное пространство с помощью Windows-функций отображения файлов.

ПРИМЕЧАНИЕ

В данном случае под согласованностью данных кэша понимается согласованность данных, отображенных в пользовательском адресном пространстве и кэшируемых при вводе-выводе, а не согласованность некэшируемого и кэшируемого аппаратного доступа и ввода-вывода, когда данные практически гарантированно не согласованы. Кроме того, согласованность кэшадается несколько сложнее для сетевых систем переадресации, чем для локальных файловых систем, потому что сетевые системы переадресации должны поддерживать дополнительные операции сброса и очистки, чтобы обеспечивать согласованность кэша при обращении к сетевым данным. (Об уступающей блокировке, являющейся механизмом согласованности распределенного кэша в Windows, рассказывается в главе 12.)

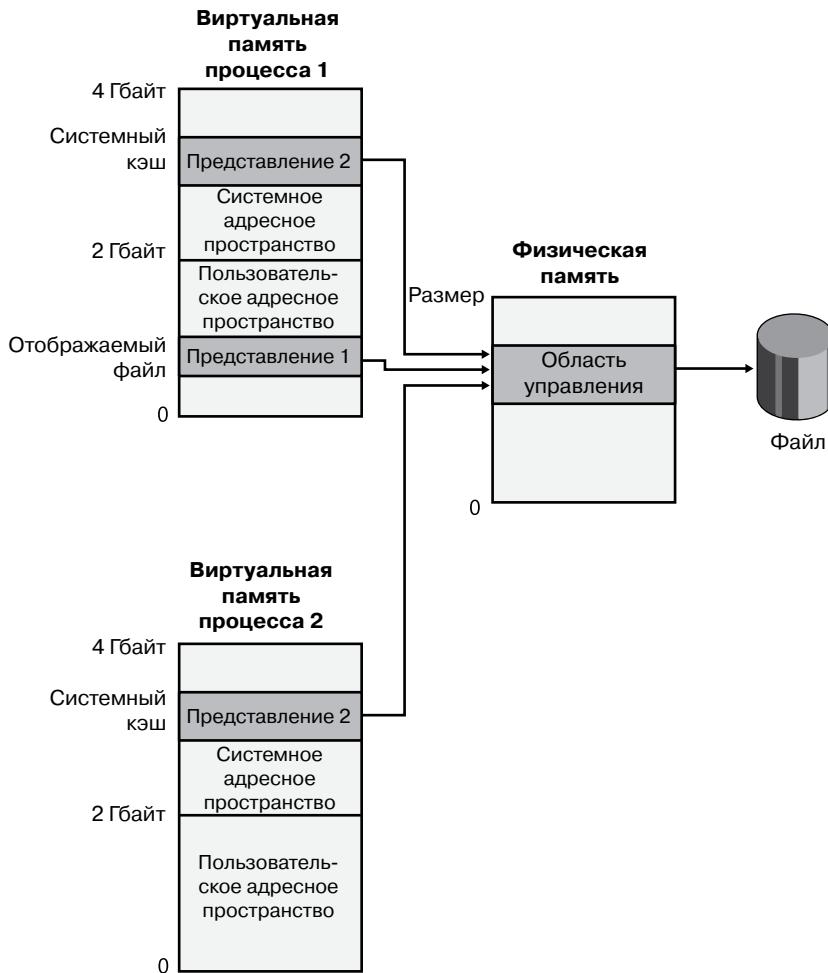


Рис. 11.1. Схема обеспечения согласованности данных кэша

Кэширование виртуальных блоков

Диспетчер кэша в Windows использует поход, известный как *кэширование виртуальных блоков* (virtual block caching), отслеживая, какие части каких файлов находятся в кэше. Диспетчер кэша может отслеживать эти части файлов путем отображения представлений файлов размером 256 Кбайт на пространства виртуальных адресов системы с помощью специальных системных процедур кэша, находящихся в диспетчере памяти. Такой подход имеет следующие основные преимущества:

- Он открывает возможность проводить интеллектуальное упреждающее чтение. Поскольку кэш отслеживает, какие части каких файлов находятся в кэше, он может прогнозировать следующий шаг вызывающей программы.

- Он позволяет системе ввода-вывода обойтись без обращения к файловой системе при запросе данных, которые уже находятся в кэше (быстрый ввод-вывод). Поскольку диспетчер кэша знает, какие части каких файлов находятся в кэше, он может возвратить адрес кэшированных данных для удовлетворения запроса на ввод-вывод без обращения к файловой системе.

Детали, касающиеся интеллектуального упреждающего чтения и быстрого ввода-вывода, рассматриваются чуть позже.

Кэширование на основе потоков данных

Диспетчер кэша рассчитан также на *кэширование потоков данных* (stream caching), которое отличается от кэширования файлов. Поток данных является последовательностью байтов внутри файла. Некоторые файловые системы, например NTFS, позволяют файлу содержать более одного потока данных; диспетчер кэша подстраивается под такие файловые системы, выполняя независимое кэширование каждого такого потока. NTFS может воспользоваться этой возможностью, организуя в потоки данных свою главную таблицу файлов (см. главу 12) и кэшируя эти потоки. Хотя о диспетчере кэша говорят, что он кэширует файлы, на самом деле он кэширует потоки данных (у всех файлов имеется, по крайней мере, один поток данных), идентифицируемые как по имени файла, так и по имени потока, если в файле имеется более одного потока данных.

ПРИМЕЧАНИЕ

Диспетчер кэша не знает ни имен файлов, ни имен потоков данных, а использует указатели на эти объекты.

Поддержка самовосстанавливающихся файловых систем

Самовосстанавливающиеся файловые системы, такие как NTFS, разрабатывались в расчете на восстановление структуры дискового тома после системного сбоя. Такая возможность означает, что операции ввода-вывода, проводившиеся в момент системного сбоя, при перезапуске системы должны быть либо полностью завершены, либо полностью отменены, не оставив на диске никаких следов. Не до конца проведенные операции ввода-вывода могут повредить дисковый том и даже сделать весь том недоступным. Чтобы этого не допустить, самовосстанавливающиеся файловые системы ведут файл журнала, в который перед записью изменений в том записывают каждое обновление, которое они намереваются внести в структуру файловой системы (в метаданные файловой системы). Если в системе происходит сбой, прерывающий внесение изменений в том, самовосстанавливающаяся файловая система использует информацию, сохраненную в журнале, чтобы снова попытаться внести в том изменения.

ПРИМЕЧАНИЕ

Понятие «метаданные» применяется только к изменениям в структуре файловой системы: созданию, переименованию и удалению файлов и каталогов.

Чтобы гарантировать успешное восстановление тома, каждая запись в файл журнала, документирующая обновление тома, должна быть полностью записана на диск перед тем, как само это обновление будет применено к тому. Поскольку записи на диск кэшируются, диспетчер кэша и файловая система должны скоординировать обновление метаданных, обеспечивброс инфомации в файл журнала перед тем, как обновлять метаданные. В целом происходит последовательное выполнение следующих действий:

1. Файловая система делает в файле журнала запись, документирующую намерение обновить метаданные.
2. Файловая система вызывает диспетчер кэша для сброса записи файла журнала на диск.
3. Файловая система записывает обновления тома в кэш, то есть обновляет свои кэшированные метаданные.
4. Диспетчер кэша сбрасывает измененные метаданные на диск, обновляя тем самым структуру тома. (Фактически, перед сбросом на диск записи в файле журнала группируются, то же самое делается и при изменении тома.)

Когда файловая система записывает данные в кэш, она может предоставить *логический порядковый номер* (Logical Sequence Number, LSN), по которому идентифицируется запись в ее файле журнала, соответствующая обновлению кэша. Диспетчер кэша их отслеживает, записывая самые меньшие и самые большие LSN-номера (представляющие самые старые и самые новые записи в файле журнала), связанные с каждой страницей в кэше. Кроме того, потоки данных, защищенные записями в журнале транзакций, помечаются файловой системой NTFS как незаписываемые, чтобы подсистема записи отображаемых файлов не смогла ненароком записать эти страницы обратно на диск до внесения записи в соответствующий журнал. (Когда подсистема записи отображаемого файла видит помеченную таким образом страницу, она перемещает эту страницу в специальный список, который затем в подходящее время используется диспетчером кэша для сброса страницы на диск, примерно так же, как это делается при отложенной записи.)

При подготовке к сбросу на диск группы измененных страниц (с установленным битом изменения) диспетчер кэша определяет наивысший LSN-номер, связанный со сбрасываемыми страницами, и сообщает этот номер файловой системе. Далее файловая система опять вызывает диспетчер кэша, заставляя его сбросить на диск данные файла журнала вплоть до точки, соответствующей полученному ранее LSN-номеру. После того как диспетчер кэша сбросит на диск эту часть файла журнала, он сбрасывает на диск соответствующие обновления структуры тома, гарантируя тем самым, что он записывает то, что собирается сделать, прежде чем сделать это на самом деле. Этот механизм взаимодействия между файловой системой и диспетчером кэша обеспечивает возможность восстановления дискового тома после сбоя системы.

Управления виртуальной памятью кэша

Поскольку системный диспетчер кэша в Windows кэширует данные на виртуальной основе, он использует области виртуального адресного пространства (а не физическую

память) и управляет ими в структурах, которые называются *блоками управления виртуальными адресами* (Virtual Address Control Blocks, VACB). VACB-блоки определяют области адресного пространства в слотах по 256 Кбайт, называемые *представлениями* (views). Когда диспетчер кэша проводит инициализацию своих данных в ходе начальной загрузки, он выделяет исходный массив VACB-блоков для описания кэшируемой памяти. По мере возрастания потребностей в кэшировании и, соответственно, в большем объеме памяти, диспетчер кэша выделяет больше массивов VACB-блоков. Он также может сокращать виртуальное адресное пространство при изменении потребностей системы в памяти.

При первой же файловой операции ввода-вывода (связанной с чтением или записью) диспетчер кэша отображает 256-килобайтное представление выровненной 256-килобайтной области файла, который содержит запрошенные данные, на свободный слот адресного пространства системного кэша. Например, если 10 байт, начинающиеся со смещения 300 000 байт, были считаны в файл, отображаемое представление начнется со смещения 262 144 (со второй выровненной по 256-килобайтной границе области файла) и будет простираться на величину 256 Кбайт.

Диспетчер кэша отображает представления файлов в слотах адресного пространства кэша на круговой основе, отображая первое запрошенное представление в первом 256-килобайтном слоте, второе представление — во втором 256-килобайтном слоте и т. д., как показано на рис. 11.2. В этом примере первым отображается файл *B*, вторым — файл *A*, третьим — файл *C*. Следовательно, отображенный участок файла *B* занимает в кэше первый слот. Следует заметить, что отображается только первая 256-килобайтная часть файла *B*, поскольку обращение было только к части файла, что же касается файла *C*, то хотя его объем и составляет всего 100 Кбайт (меньше, чем размер одного представления в системном кэше), он требует выделения ему в кэше собственного 256-килобайтного слота.

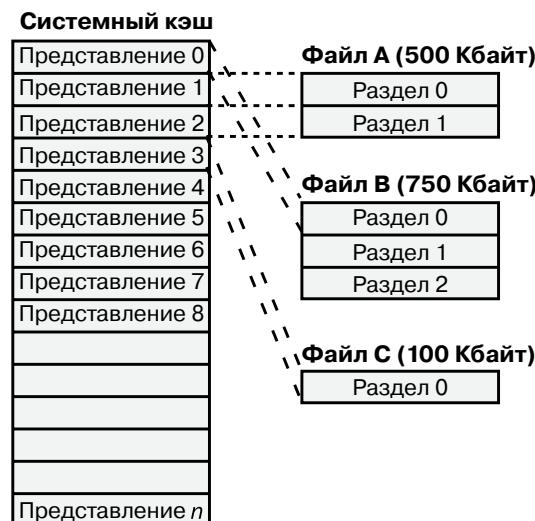


Рис. 11.2. Файлы разного размера, отображаемые в системном кэше

Диспетчер кэша гарантирует, что представление будет отображаться до тех пор, пока оно активно (хотя представления могут оставаться отображенными и после того, как перестают быть неактивными). Однако представление помечается как активное только в ходе чтения из файла или записи в файл. Если процесс не открывает файл с установкой флага `FILE_FLAG_RANDOM_ACCESS` при вызове функции `CreateFile`, диспетчер кэша прекращает отображение неактивных представлений файла по мере отображения новых представлений, если он обнаруживает, что доступ к файлу осуществляется последовательно. Страницы, соответствующие представлениям, отображение которых прекращено, отправляются в список ожидающих или в список измененных (если они были изменены) страниц, а поскольку диспетчер памяти экспортирует для диспетчера кэша специальный интерфейс, диспетчер кэша может направлять страницы в конец или в начало этих списков. Страницы, соответствующие представлениям файлов, открытых с флагом `FILE_FLAG_SEQUENTIAL_SCAN`, перемещаются в начало списков, а все остальные страницы — в конец. Эта схема стимулирует повторное использование страниц, принадлежащих файлам последовательного чтения, и реально предотвращает влияние на физическую память операций копирования больших файлов, допуская подобное влияние лишь для небольшой части физической памяти. Этот флаг также оказывается на прекращении отображения: если он предоставляется функции, диспетчер кэша будет прекращать отображение представлений более агрессивно.

Если диспетчуру кэша нужно отобразить представление файла, а в кэше нет свободных слотов, он прекращает отображение того неактивного представления, которое было отображено раньше других, и использует освободившийся слот. Если ни одно из представлений недоступно, возвращается ошибка ввода-вывода, свидетельствующая о том, что для выполнения операции недостаточно доступных системных ресурсов. Однако учитывая, что представления помечаются как активные только в ходе операции чтения или записи, подобное развитие событий маловероятно, поскольку для этого нужно одновременно обратиться к тысячам файлов.

Размер кэша

В следующих разделах рассматривается порядок вычисления в Windows виртуального и физического размеров системного кэша. Как и в большинстве других вычислений, связанных с управлением памятью, размер системного кэша зависит от множества факторов.

Виртуальный размер кэша

В 32-разрядных системах Windows виртуальный размер системного кэша ограничен только лишь объемом виртуального адресного пространства ядра и параметром `SystemCacheLimit` реестра, который должен быть оптимальным. (Дополнительные сведения об ограничении размера виртуального адресного пространства ядра можно найти в главе 10.) Это означает, что размер кэша не может превышать размера системного адресного пространства (2 Гбайт), но обычно он существенно меньше, поскольку системное адресное пространство используется совместно с другими ресурсами,

к которым относятся записи системных таблиц страниц (PTE-записи), невыгружаемый и выгружаемый пулы, таблицы страниц. В 64-разрядных системах Windows максимальный размер виртуального кэша составляет 1024 Гбайт (1 Тбайт).

Размер рабочего набора кэша

Как уже упоминалось, одно из основных отличий диспетчера кэша в Windows от аналогичных диспетчеров в других операционных системах заключается в делегировании обязанностей по управлению физической памятью глобальному диспетчеру памяти. По этой причине существующий код, предназначенный для управления увеличением и уменьшением рабочего набора, а также для управления списками измененных и ожидающих страниц, служит также и для управления размером системного кэша, обеспечивая динамическую сбалансированность потребностей в физической памяти между процессами и операционной системой.

У системного кэша нет собственного рабочего набора, он совместно использует единый системный набор, включающий данные кэша, выгружаемый пул, страничный Ntoskrnl-код и код драйверов. Согласно объяснениям в разделе «Системные рабочие наборы» главы 10, этот единый рабочий набор внутри системы называется *рабочим набором системного кэша* (system cache working set), хотя системный кэш является лишь одним из его компонентов. Поэтому, а также в соответствии с назначением данной книги мы будем называть этот рабочий набор просто *системным рабочим набором* (system working set). Как отмечено в главе 10, при установке для параметра LargeSystemCache реестра значения 1 диспетчер памяти отдает предпочтение системному рабочему набору, ставя его потребности выше потребностей процессов, запущенных в системе.

ЭКСПЕРИМЕНТ: ПРОСМОТР РАБОЧЕГО НАБОРА КЭША

Команда !filecache отладчика в этом эксперименте выводит дамп с информацией о физической памяти, используемой кэшем, о текущем и пиковых размерах рабочего набора, о количестве достоверных страниц, связанных с представлениями, и об именах файлов, отображаемых на представления там, где это приемлемо. (Метаданные кэша для драйверов файловой системы, такие как структуры каталогов и битовые массивы томов, выводятся с помощью безымянных файловых потоков данных.)

```
1kd> !filecache
***** Dump file cache*****
Reading and sorting 999 VACBs ...
ReadVirtual: 85b77038 not properly sign extended
ReadVirtual: 85ba7010 not properly sign extended
Processing 998 active VACBs ...
File Cache Information
Current size 30528 kb
Peak size 65752 kb
461 Control Areas
Skipping view @ 91980000 - no VACB, but PTE is a prototype!
Loading file cache database (100% of 523264 PTEs)
SkippedPageTableReads = 882
File cache has 7668 valid pages
Usage Summary (in Kb):
Control Valid Standby/Dirty Shared Locked FsContext Name
```

```
85fa5be0 0 4 0 0 add0dbf8 $Directory
85f971b8 0 8 0 0 ad9bc918 $Directory
87c489f0 4 4 0 0 93b390f8 $Directory
87c4a9c0 4 0 0 0 93b38c30 $Directory
87c451a8 0 4 0 0 93b35780 $Directory
86a83710 4512 45432 0 0 86a90168 $Mft
85f96770 0 8 0 0 ad9c00f8 No Name for File
85e90998 0 512 0 0 abb83510 No Name for File
88062008 4 0 0 0 9e6c40f8 $Directory
87c291e8 44 164 0 0 93b400f8 $Directory
87c27e10 0 16 0 0 93b4bd08 $Directory
87b4bc88 236 84 0 0 93b28d08 $Directory
86ce23a8 12 0 0 0 a2051528 $Directory
87c2bb20 4 0 0 0 93b3b850 $Directory
87d51480 0 4 0 0 824f9830 $Directory
87c8c900 0 4 0 0 825b06d0 utmpx
87c2aa30 44 216 0 0 93b3fc70 $Directory
86ecc168 12 4088 0 0 9c3c5c50 Microsoft-Windows-
GroupPolicy%40operational.evtx
```

Физический размер кэша

Хотя в системный рабочий набор входит тот объем физической памяти, который отображен на представления в виртуальном адресном пространстве кэша, он совсем не обязательно равен всему объему данных файла, кэшированного в физической памяти. Различие между двумя значениями обусловливается тем, что дополнительные данные файла могут находиться в списках ожидающих или измененных страниц, обслуживаемых диспетчером памяти.

Как показано в главе 10, в ходе усечения рабочего набора или замещения страниц диспетчер памяти может переместить измененные страницы из рабочего набора либо в список ожидающих, либо в список измененных страниц, в зависимости от того, содержат ли страницы данные, которые перед повторным использованием страницы нужно записать в страничный или какой-нибудь другой файл. Если бы диспетчер памяти не вел такие списки, то при каждом обращении процесса к ранее удаленным из его рабочего набора данным диспетчеру памяти пришлось бы сталкиваться с серьезными ошибками отсутствия страниц и загружать данные с диска. Если же данные, к которым идет обращение, присутствуют в любом из этих списков, диспетчер памяти просто сталкивается с несерьезными ошибками и возвращает страницу в рабочий набор процесса. Таким образом, списки служат в качестве кэшированных в памяти данных, которые хранятся в страничном файле, исполняемых образах или файлах данных. Стало быть, общий объем кэшированных файловых данных включает в себя не только системный рабочий набор, но и объединенные объемы списков ожидающих и измененных страниц.

Рассмотрим пример, показывающий, что диспетчер кэша может кэшировать в физической памяти намного больше данных, чем их смогло бы уместиться в системном рабочем наборе. Возьмем систему,ирующую в качестве выделенного файлового сервера. Клиентское приложение обращается к файловым данным по сети, а некая серверная программа, например драйвер файлового сервера (%SystemRoot%\System32\Drivers\Srv2.sys, см. главу 12), использует интерфейсы диспетчера кэша для чтения

и записи файловых данных от имени клиента. Если клиент считывает несколько тысяч файлов по 1 Мбайт каждый, диспетчеру кэша придется начать повторно использовать представления, как только он выйдет за пределы отображаемого пространства (и не сможет расширить отображаемую VACB-область). Для каждого считываемого после этого файла диспетчер кэша прекращает отображение представлений и отображает их для новых файлов. Когда диспетчер кэша прекращает отображение представлений, диспетчер памяти не избавляется от тех файловых данных в рабочем наборе кэша, которые соответствовали представлению, а перемещает их в список ожидающих страниц. При отсутствии какого-либо другого спроса на физическую память под данные, фигурирующие в списке ожидающих страниц, может быть занята почти вся физическая память, оставшаяся за пределами системного рабочего набора. Иными словами, как показано на рис. 11.3, фактически вся физическая память сервера будет использована для кэширования файловых данных.



Рис. 11.3. Пример, в котором основной объем физической памяти выделен под файловый кэш

Поскольку в общий объем кэшированных файловых данных входят системный рабочий набор, а также списки измененных и ожидающих страниц, причем их объемы управляются диспетчером памяти, то по смыслу этот диспетчер и есть настоящий диспетчер кэша. Диспетчер кэша просто предоставляет удобные интерфейсы для обращения к файловым данным через диспетчер памяти. Благодаря своим политикам упреждающего чтения и кэширования при записи (отложенной записи) он также играет важную роль, оказывая влияние на то, какие именно данные диспетчер памяти оставляет в физической памяти, а также управляя представлениями в пространстве системных виртуальных адресов.

Пытаясь поточнее отразить общий объем файловых данных, кэшированных в системе, диспетчер задач выводит значение поля *Cache* (Кэшировано), которое отражает объединенный объем системного рабочего набора вместе со списками ожидающих и измененных страниц. Что же касается такого средства, как *Process Explorer*, то он распределяет эти значения по полям *Cache WS* (Рабочий набор системного кэша), *Standby* (Ожидающие страницы) и *Modified* (Измененные страницы). В нижней левой части окна *Process Explorer* на рис. 11.4 системная информация располагается в поле *Cache WS* (Рабочий набор системного кэша) раздела *Physical Memory* (Физическая память), а в разделе *Paging Lists* (Страницочные списки), расположенным ближе к середине,

представлены объемы списков ожидающий и измененных страниц. Следует учесть, что значение в поле Cache (Кэшировано) диспетчера задач включает также значения, которые в Process Explorer фигурируют в полях Paged WS (Выгружаемый рабочий набор), Kernel WS (Рабочий набор ядра) и Driver WS (Рабочий набор драйверов). Когда выбирались названия этих полей, абсолютное большинство системного рабочего набора состояло из рабочего набора системного кэша. Сейчас ситуация изменилась, но в диспетчере памяти этот анахронизм остался.

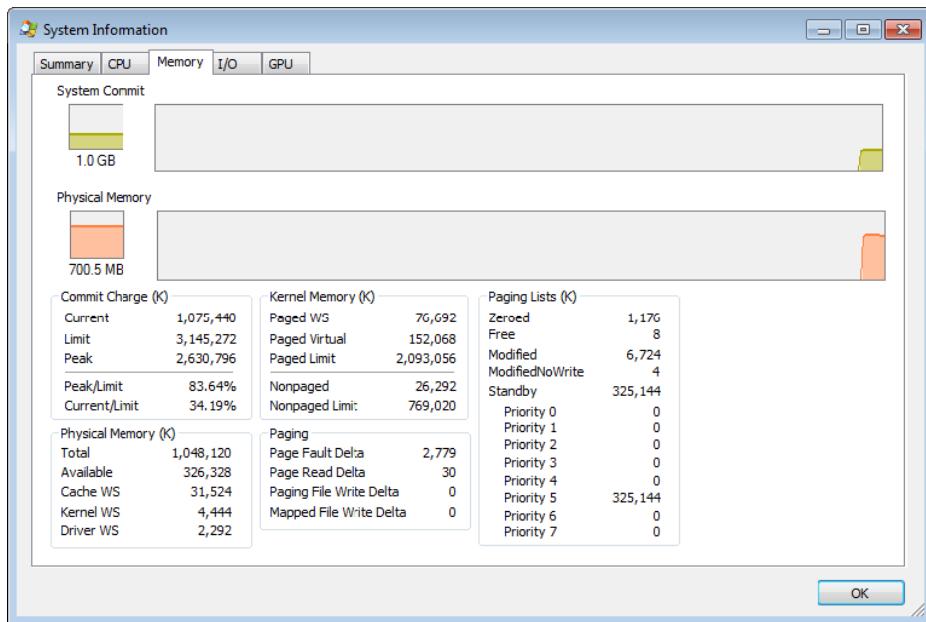


Рис. 11.4. Диалоговое окно System Information программы Process Explorer

Структуры данных кэша

Для отслеживания кэшируемых файлов диспетчер кэша использует следующие структуры данных:

- ❑ VACB-блок, описывающий каждый 256-килобайтный слот системного кэша.
- ❑ Закрытая карта кэша для каждого отдельно открытого кэшируемого файла, в которой содержится информация, используемая для управления упреждающим чтением (см. далее).
- ❑ Единая общая (совместно используемая) карта кэша для каждого кэшируемого файла, указывающая на слоты в системном кэше, в которых содержатся отображаемые представления файла.

Эти структуры и связи между ними рассматриваются в следующих разделах.

Общесистемные структуры данных кэша

Как уже упоминалось, диспетчер кэша отслеживает состояние представлений в системном кэше с помощью массива структур данных, который называется массивом *блоков управления виртуальными адресами* (Virtual Address Control Block, VACB) и хранится в невыгружаемом пуле. Размер каждого VACB-блока в 32-разрядной системе составляет 32 байта, а VACB-массива – 128 Кбайт, стало быть, в каждом VACB-массиве содержится 4096 VACB-блоков. В 64-разрядной системе VACB-блок содержит 64 байта, то есть получается, что в каждом VACB-массиве содержится 2048 VACB-блоков. Диспетчер кэша распределяет исходный VACB-массив в ходе инициализации системы и связывает его в общесистемный список VACB-массивов, который называется `CcVacbArrays`. Как показано на рис. 11.5, каждый VACB-блок соответствует одному 256-килобайтному представлению в системном кэше. Структура VACB-блока показана на рис. 11.6.

Кроме того, каждый VACB-массив состоит из VACB-блоков двух типов: *низкоприоритетных отображенных VACB-блоков* (low priority mapping VACBs) и *высокоприоритетных отображенных VACB-блоков* (high priority mapping VACBs). Для каждого VACB-массива система выделяет 64 VACB-блока, которые изначально имеют высокий приоритет. Высокоприоритетные VACB-блоки отличаются тем, что память для их представлений выделяется из системного адресного пространства заранее. Если к моменту отображения каких-либо данных у диспетчера памяти нет представлений для их передачи диспетчеру кэша, а запрос на отображение имеет метку высокоприоритетного, диспетчер кэша использует одно из заранее выделенных представлений в высокоприоритетном VACB-блоке. Такие высокоприоритетные VACB-блоки применяются им, к примеру, для важных метаданных файловой системы, а также для удаления данных из кэша. Однако когда высокоприоритетные VACB-блоки заканчиваются, любые операции, требующие VACB-представления, перестают выполняться из-за нехватки ресурсов. Обычно приоритет отображения по умолчанию устанавливается низким, но если при фиксации (см. далее) кэшированных данных используется флаг `PIN_HIGH_PRIORITY`, файловая система может при необходимости запросить вместо него высокоприоритетный VACB-блок.

Как показано на рис. 11.6, в первом поле VACB-блока содержится виртуальный адрес данных в системном кэше. Во втором поле находится указатель на общую (совместно используемую) карту кэша, которая определяет, какой именно файл кэшируется. В третьем поле определяется смещение внутри файла, с которого начинается представление (это смещение всегда базируется на 256-килобайтной гранулярности). С учетом этой гранулярности младшие 16 бит файлового смещения всегда нулевые, следовательно, эти биты можно повторно использовать для хранения количества ссылок на представление, то есть количества активных обращений к представлению при чтении или записи данных. Четвертое поле связывает VACB-блок со списком VACB-блоков, дольше всех не используемых (Least Recently Used, LRU) на момент освобождения VACB-блока диспетчером кэша; диспетчер сначала проверяет этот список, а затем выделяет память для новых VACB-блоков. И наконец, пятое поле связывает этот VACB-блок с заголовком VACB-массива, в котором он хранится.

В ходе операции ввода-вывода файла показатель счетчика обращений VACB-блока этого файла увеличивается на единицу, а затем, когда операция ввода-вывода завершается, он уменьшается на единицу. Если показатель обращений отличается от нуля,

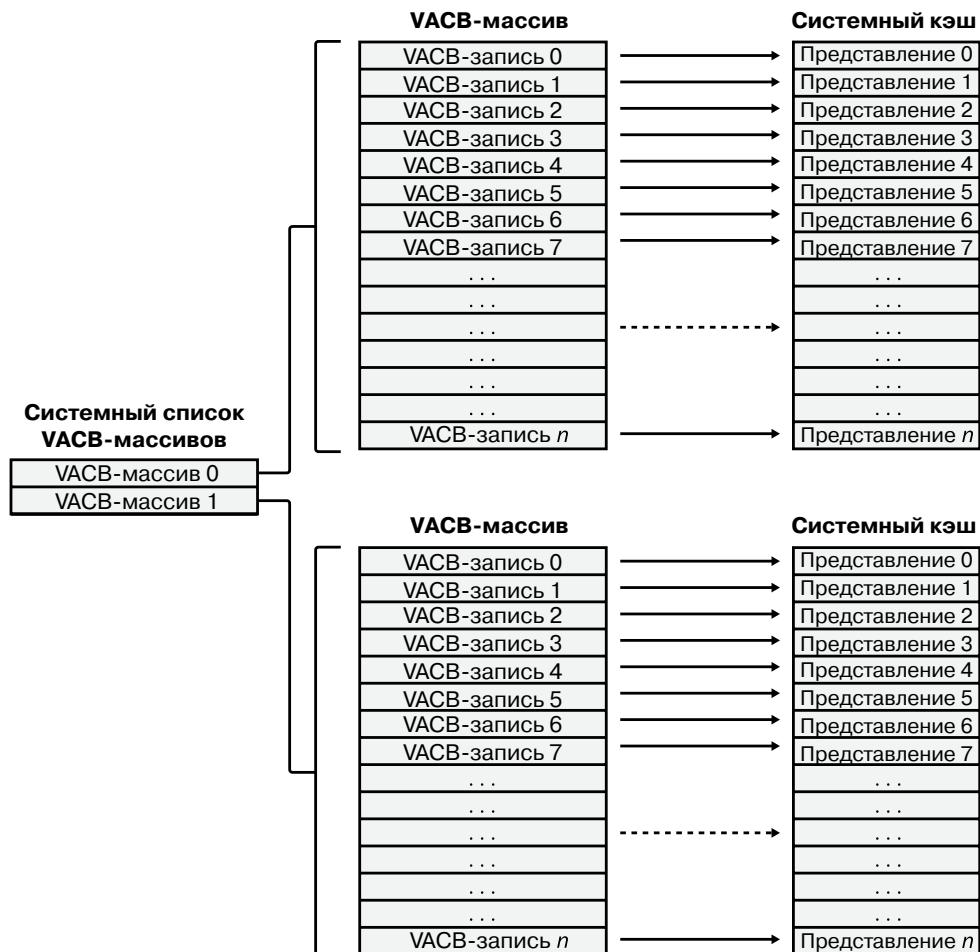


Рис. 11.5. Системный VACB-массив

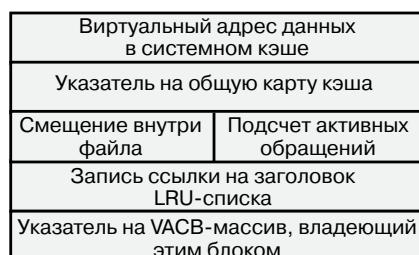


Рис. 11.6. Структура VACB-блока

VACB-блок *активен*. При доступе к метаданным файловой системы подсчет активных обращений дает представление о том, сколько драйверов файловой системы обладают страницами в данном представлении, заблокированными в памяти.

ЭКСПЕРИМЕНТ: ПРОСМОТР VACB-БЛОКОВ И ИХ СТАТИСТИКИ

В диспетчере кэша имеется механизм отслеживания различных значений, которые могут пригодиться разработчикам и специалистам программной поддержки при отладке после получения аварийных дампов. Имена всех этих отладочных переменных начинаются с префикса CcDbg, что упрощает просмотр всего списка с помощью команды x:

```
1kd> x nt!*ccdbg*
8194ba84 nt!CcDbgNumberOfCcUnmapInactiveViews = <no type information>
8197c740 nt!CcDbgNumberOfFailedMappingsDueToVacbSpace =
<no type information>
8197c730 nt!CcDbgNumberOfFailedBitmapAllocations =
<no type information>
8197c73c nt!CcDbgNumberOfFailedHighPriorityMappingsDueToMmResources =
<no type information>
...
...
```

На некоторых системах из-за разницы в реализации 32- и 64-разрядных версий имена переменных могут быть другими. Однако точные имена для данного эксперимента не важны, главное внимание в нем уделяется рассматриваемой методологии. Используя эти переменные и ваши знания структуры данных заголовка VACB-массива, вы можете воспользоваться отладчиком ядра для вывода списка заголовков всех VACB-массивов. Переменная CcVacbArrays представляет собой массив указателей на заголовки VACB-массивов, с помощью которых можно получить значения, позволяющие извлечь дамп содержимого переменных _VACB_ARRAY_HEADER. Сначала получим наивысший индекс массива:

```
1kd> dd nt!CcVacbArraysHighestUsedIndex 1 1
8194ba7c 00000000
```

А теперь можно получить нужные значения по каждому индексу, вплоть до максимально-го. На данной системе (что является нормой) наивысший индекс равен 0, что означает наличие только одного заголовка, получаемого по этому индексу:

```
1kd> ?? (*((nt!_VACB_ARRAY_HEADER***))@((nt!CcVacbArrays)))[0]
struct _VACB_ARRAY_HEADER * 0x8315b000
+0x000 VacbArrayIndex : 0
+0x004 MappingCount : 0x5ab
+0x008 HighestMappedIndex : 0x9a9
+0x00c Reserved : 0
```

Если бы индексов было больше, можно было бы менять индекс массива в конце команды на больший вплоть до достижения самого большого индекса. Из выведенных данных следует, что в системе имеется только один VACB-массив из активных VACB-блоков в количестве 1451 (0x5ab).

И наконец, в переменной CcNumberOfFreeVacbs хранится количество VACB-блоков в списке свободных VACB-блоков. В выводе дампа этой переменной в системе, используемой в эксперименте, результат равен 2645 (0xa55):

```
1kd> dd nt!CcNumberOfFreeVacbs 1 1  
8197c768 00000a55
```

Как и ожидалось, сумма свободных (0x5ab, или 1451 в десятичном исчислении) и активных (0xa55, или 2645 в десятичном исчислении) VACB-блоков в 32-разрядной системе с одним VACB-массивом равна 4096, что соответствует количеству VACB-блоков в одном VACB-массиве. Если в системе закончатся свободные VACB-блоки, диспетчер кэша попытается выделить память для нового VACB-массива. В силу неустойчивого характера данного эксперимента, между этими двумя шагами в вашей системе могут создаваться и (или) освобождаться дополнительные VACB-блоки (и в дампах будут фигурировать активные, а затем освобожденные VACB-блоки). Это может привести к тому, что общее количество свободных и активных VACB-блоков не будет в точности равно 4096. При таком результате попробуйте быстро пару раз повторить эксперимент, хотя устойчивых результатов вы все равно не получите, особенно в условиях высокой активности файловой системы.

Структуры данных кэша, относящиеся к каждому файлу

У каждого открытого дескриптора файла имеется соответствующий файловый объект. (Файловые объекты подробно рассматриваются в главе 8.) Если файл кэшируется, файловый объект указывает на *закрытую карту кэша* (private cache map), в которой содержатся данные о местоположении двух последних операций чтения, чтобы диспетчер кэша мог выполнять интеллектуальное упреждающее чтение (см. далее раздел «Интеллектуальное упреждающее чтение»). Кроме того, все закрытые карты кэша для открытых экземпляров файла связаны друг с другом.

Каждый кэшированный файл (в противоположность файловому объекту) имеет *общую карту кэша* (shared cache map), которая описывает состояние кэшируемого файла, включая его размер и длину достоверных данных. (Назначение поля длины достоверных данных рассматривается в разделе «Кэширование с обратной записью и отложенная запись».) Общая карта кэша указывает также на объект раздела (обслуживаемый диспетчером памяти и описывающий отображение файла на виртуальную память), на список закрытых карт кэша, связанных с этим файлом, и на любые VACB-блоки, которые описывают текущие отображенные представления файла в системном кэше. (Более подробно указатели на объекты разделов рассматриваются в главе 10.) Взаимосвязь этих структур данных кэша, относящихся к каждому файлу, иллюстрирует рис. 11.7.

При запросе на считывание данных из конкретного файла диспетчер кэша должен ответить на два вопроса:

1. Имеется ли этот файл в кэше?
2. Если он в кэше, то какой VACB-блок, если таковой имеется, ссылается на запрошенное место?

Иными словами, диспетчер кэша должен определить, отображается ли представление файла на нужный адрес в системном кэше. Если VACB-блока, содержащего нужное файловое смещение, нет, значит, запрошенные данные на системный кэш не отображены.

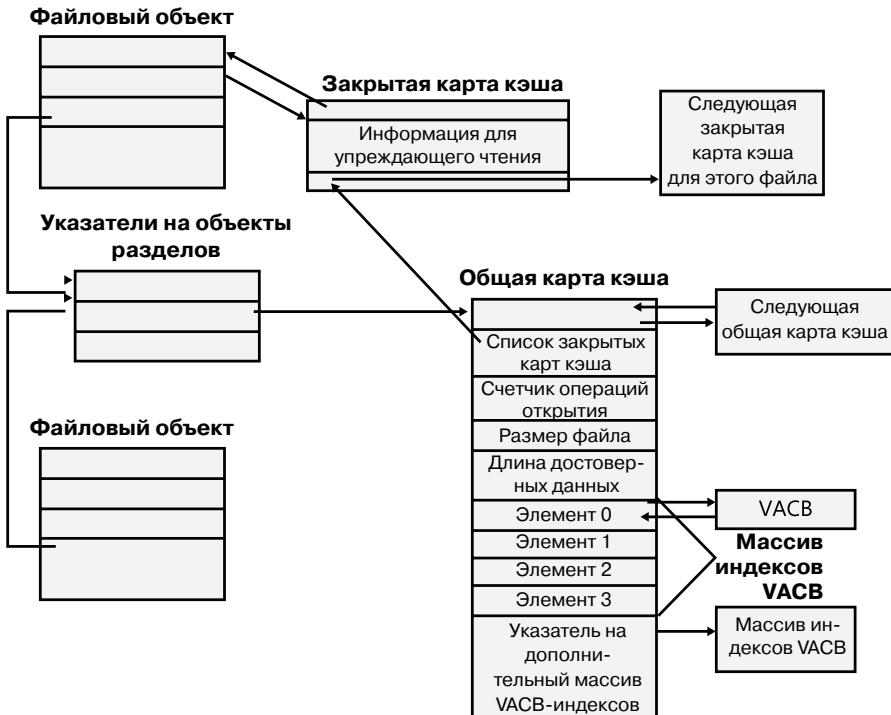


Рис. 11.7. Структуры данных кэша, относящиеся к каждому файлу

Для того чтобы отслеживать, какие представления заданного файла отображаются на системный кэш, диспетчер кэша содержит массив указателей на VACB-блоки, известный как *массив VACB-индексов* (VACB index array). Первый элемент массива VACB-индексов ссылается на первые 256 Кбайт файла, второй элемент — на вторые 256 Кбайт и т. д. На рис. 11.8 показаны четыре разных раздела трех разных файлов, которые в данный момент отображаются на системный кэш.

При обращении процесса к конкретному файлу в заданном месте диспетчер кэша проверяет соответствующий элемент массива VACB-индексов файла, чтобы узнать, отображены ли запрошенные данные на кэш. Если элемент имеет ненулевое значение (а следовательно, содержит указатель на VACB-блок), значит, область файла, на которую дается ссылка, находится в кэше. В свою очередь, VACB-блок указывает на место в системном кэше, где отображено представление файла. Если элемент имеет нулевое значение, диспетчер кэша должен найти свободный слот в системном кэше (а стало быть, и свободный VACB-блок) для отображения запрошенногопредставления.

Из соображений оптимизации размера в общей карте кэша содержится массив VACB-индексов размером в четыре элемента. Поскольку в каждом VACB-блоке описывается 256 Кбайт, элементы в этом небольшом массиве индексов с фиксированным размером могут указывать на элементы VACB-массива, которые вместе описывают файл

длиной до 1 Мбайт. Если файл больше 1 Мбайт, из невыгружаемого пула выделяется память для отдельного массива VACB-индексов на базе размера файла, деленного на 256 Кбайт и в случае остатка округляемого в большую сторону. После этого общая карта кэша начинает указывать на эту отдельную структуру.

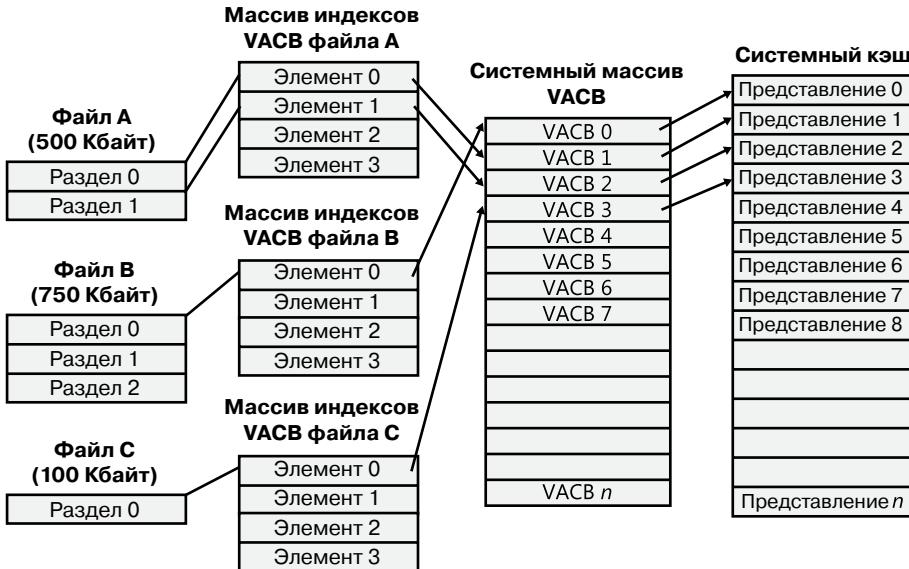


Рис. 11.8. Массивы VACB-индексов

Если размер файла превышает 32 Мбайт, для проведения дальнейшей оптимизации массив VACB-индексов, выделенный из невыгружаемого пула, становится разреженным многоуровневым массивом индексов, в котором каждый массив индексов содержит 128 элементов. Количество уровней, требуемых для файла, можно рассчитать по следующей формуле:

$$(количество битов, требуемых для представления размера файла - 18) / 7$$

Результат вычисления нужно округлить до следующего целого числа. Значение 18 в этой формуле берется на основании того факта, что VACB-блок представляет 256 Кбайт, а 256 Кбайт — это 2^{18} . Значение 7 берется на основании того факта, что каждый уровень в массиве имеет 128 элементов, а 2^7 равно 128. Таким образом, для файла, имеющего максимальный размер, который может быть описан с помощью 63 бит (максимальный размер, поддерживаемый диспетчером кэша), требуется всего семь уровней. Массив является разреженным, поскольку единственными выделяемыми диспетчером кэша ветвями являются те ветви, для которых имеются активные представления, фигурирующие в массиве индексов самого низкого уровня. Пример многоуровневого VACB-массива для разреженного файла, размера которого вполне достаточно, чтобы потребовались три уровня, показан на рис. 11.9.

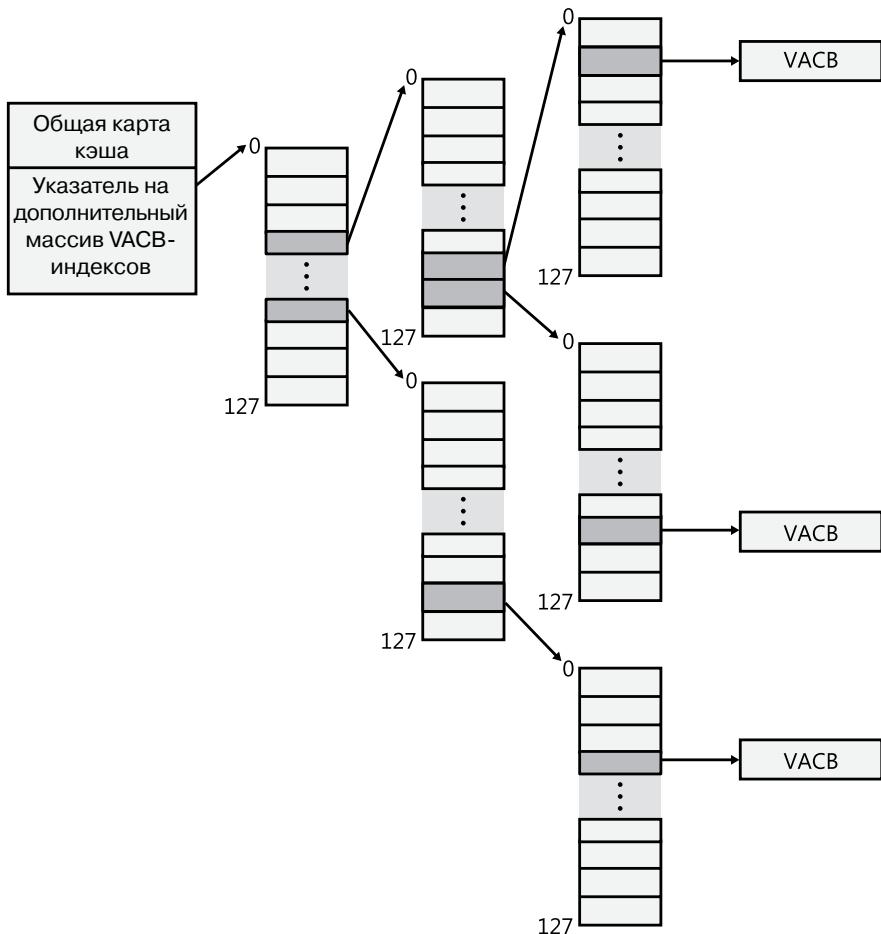


Рис. 11.9. Многоуровневые VACB-массивы

Такая схема нужна для эффективной обработки разреженных файлов, у которых при очень большом размере достоверными данными занята лишь малая часть, поскольку для управления текущими отображенными представлениями файла выделяется массив строго необходимого размера. Например, для разреженного файла размером 32 Кбайт, в котором в виртуальном адресном пространстве кэша отображено лишь 256 Кбайт, требуется VACB-массив с тремя распределенными индексными массивами, поскольку отображение имеет только одна ветвь массива, а файл размером 32 Гбайт (2^{35} байт) требует трехуровневого массива. Если бы диспетчер кэша не использовал для этого файла оптимизированный многоуровневый массив VACB-индексов, ему пришлось бы выделять массив VACB-индексов из 128 000 элементов, что эквивалентно 1000 массивов VACB-индексов.

ЭКСПЕРИМЕНТ: ПРОСМОТР ОБЩИХ И ЗАКРЫТЫХ КАРТ КЭША

Для просмотра определений структур данных общих и закрытых карт кэша, а также для исследования структур на действующей системе можно воспользоваться командой `dt` отладчика ядра. Сначала нужно выполнить команду `!filecache` и найти в выводимом VACB-блоке запись с именем требуемого файла. В данном примере этим файлом является журнал событий системы:

```
8742a008 120 160 0 0 System.evtx
```

Первый адрес указывает на структуру данных области управления, которую диспетчер памяти использует для отслеживания диапазона адресов. (Детали см. в главе 10.) Область управления хранит указатель на файловый объект, соответствующий представлению в кэше. Этот файловый объект определяет экземпляр открытого файла. Введите следующую команду, используя адрес области управления той записи, которая была определена для просмотра структуры области управления:

```
1kd> !ca 8742a008
ControlArea @ 87cd7248
Segment 824157e0 Flink 00000000 Blink 00000000
Section Ref 1 Pfn Ref 1117 Mapped Views 3
User Ref 0 WaitForDel 0 Flush Count 0
File Object 87bcab60 ModWriteCount 0 System Views 3
WritableRefs 0
Flags (c080) File WasPurged Accessed
```

```
\Windows\System32\winevt\Logs\System.evtx
```

```
...
```

Затем с помощью следующей команды изучите файловый объект, на который ссылается область управления:

```
1kd> dt nt!_FILE_OBJECT 87bcab60
+0x000 Type : 0n5
+0x002 Size : 0n128
+0x004 DeviceObject : 0x86a4c4d0 _DEVICE_OBJECT
+0x008 Vpb : 0x86a0c270 _VPB
+0x00c FsContext : 0x93b2a8e0 Void
+0x010 FsContext2 : 0x93b2aa38 Void
+0x014 SectionObjectPointer : 0x87c1b6f0 _SECTION_OBJECT_POINTERS
+0x018 PrivateCacheMap : 0x87cd59e8 Void
+0x01c FinalStatus : 0n0
+0x020 RelatedFileObject : (null)
+0x024 LockOperation : 0 ''
...
```

Закрытая карта кэша находится по смещению 0x18:

```
1kd> dt nt!_PRIVATE_CACHE_MAP 0x87cd59e8
+0x000 NodeTypeCode : 0n766
+0x000 Flags : _PRIVATE_CACHE_MAP_FLAGS
+0x000 UlongFlags : 0x1402fe
+0x004 ReadAheadMask : 0xffff
+0x008 FileObject : 0x87bcab60 _FILE_OBJECT
+0x010 FileOffset1 : _LARGE_INTEGER 0x1000
```

продолжение ↗

```
+0x018 BeyondLastByte1 : _LARGE_INTEGER 0x1080
+0x020 FileOffset2 : _LARGE_INTEGER 0x1000
+0x028 BeyondLastByte2 : _LARGE_INTEGER 0x1080
...
```

И наконец, в поле SectionObjectPointer можно определить местонахождение общей карты кэша, а затем просмотреть ее содержимое:

```
1kd> dt nt!_SECTION_OBJECT_POINTERS 0x87c1b6f0
+0x000 DataSectionObject : 0x87cd7248
+0x004 SharedCacheMap : 0x87cd58f8
+0x008 ImageSectionObject : (null)

1kd> dt nt!_SHARED_CACHE_MAP 0x87cd58f8
+0x000 NodeTypeCode : 767
+0x002 NodeByteSize : 0n352
+0x004 OpenCount : 1
+0x008 FileSize : _LARGE_INTEGER 0x1211000
+0x010 BcbList : _LIST_ENTRY [ 0x87cd5908 - 0x87cd5908 ]
+0x018 SectionSize : _LARGE_INTEGER 0x1300000
+0x020 ValidDataLength : _LARGE_INTEGER 0x1116200
+0x028 ValidDataGoal : _LARGE_INTEGER 0x1116200
+0x030 InitialVacbs : [4] (null)
+0x040 Vacbs : 0x87dc3a20 -> 0x85ba9df0 _VACB
+0x044 FileObjectFastRef : _EX_FAST_REF
+0x048 VacbLock : _EX_PUSH_LOCK
...
```

В качестве альтернативы для нахождения и автоматического вывода большей части этой информации можно воспользоваться командой !fileobj. Например, используя эту команду для ранее упомянутого файлового объекта, можно получить следующий результат:

```
1kd> !fileobj 87bcab60
\Windows\System32\winevt\Logs\System.evtx

Device Object: 0x86a4c4d0 \Driver\volmgr
Vpb: 0x86a0c270
Event signalled
Access: Read Write SharedRead
Flags: 0xc3042
Synchronous IO
Cache Supported
Modified
Size Changed
Handle Created
Fast IO Read

FsContext: 0x93b2a8e0 FsContext2: 0x93b2aa38
Private Cache Map: 0x87cd59e8
CurrentByteOffset: 1116180
Cache Data:
Section Object Pointers: 87c1b6f0
Shared Cache Map: 87cd58f8 File Offset: 1116180 in VACB number 44
Vacb: 85ba9df0
Your data is at: 82756180
```

Интерфейсы файловых систем

При первом же обращении к данным файла с целью чтения или записи драйвер файловой системы должен определить, отображаются ли в системном кэше нужные части файла. Если они не отображаются, драйвер файловой системы должен вызвать функцию `CcInitializeCacheMap` для задания рассмотренных в предыдущем разделе структур данных, относящихся к отдельно взятому файлу.

Как только файл настраивается на кэшированный доступ, драйвер файловой системы вызывает для доступа к данным файла одну из нескольких функций. Для доступа к кэшированным данным существуют три основных метода, каждый из которых предназначен для применения в той или иной ситуации:

- В методе копирования пользовательские данные копируются между буферами кэша в системном пространстве и буфером процесса в пользовательском пространстве.
- В методе отображения и фиксации для непосредственного чтения данных из буферов кэша и записи данных в эти буферы используются виртуальные адреса.
- В методе доступа к физической памяти для непосредственного чтения данных из буферов кэша и их записи в эти буферы используются физические адреса.

Чтобы избежать бесконечного цикла при обработке диспетчером памяти ошибок отсутствия страниц, драйверы файловой системы должны предоставлять две версии операции чтения из файла: с кэшированием и без кэширования. Когда диспетчер памяти разрешает ошибку отсутствия страницы, вызывая файловую систему для извлечения данных из файла (разумеется, через драйвер устройства), он должен указать, что операция чтения выполняется без кэширования, установив в IRP-пакете соответствующий флаг.

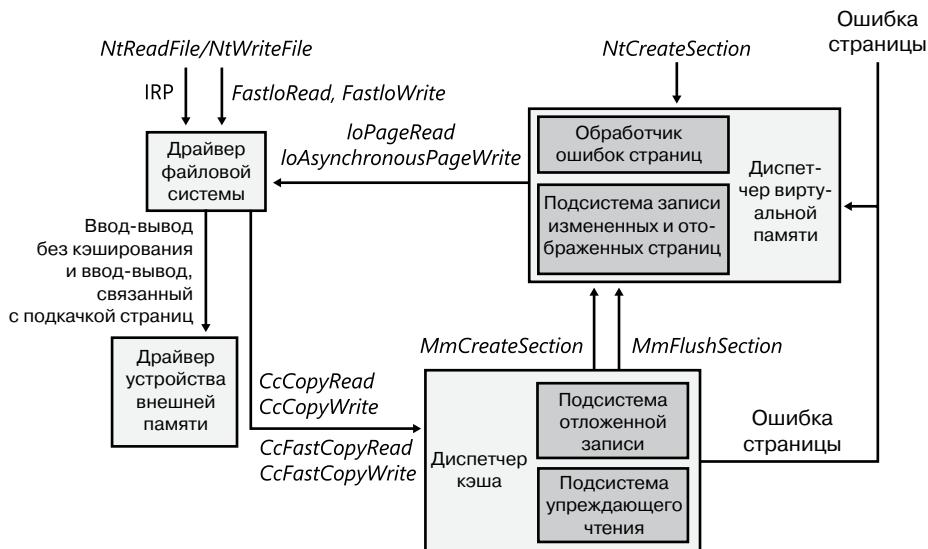


Рис. 11.10. Взаимодействие файловой системы с диспетчерами кэша и памяти

На рис. 11.10 показана схема обычного взаимодействия диспетчера кэша, диспетчера памяти и драйверов файловой системы в ответ на пользовательские операции чтения или записи, связанные с файловым вводом-выводом. Диспетчер кэша активируется файловой системой через интерфейсы копирования (функции `CcCopyRead` и `CcCopyWrite`). Например, для обработки операции чтения, инициированной функцией `CcFastCopyRead` или `CcCopyRead`, диспетчер кэша создает представление в кэше для отображения считываемой части файла и считывает файловые данные в пользовательский буфер, копируя их из представления. По мере обращения к каждой предварительно недостоверной странице в представлении операция копирования генерирует ошибку отсутствия страницы, а в ответ на это диспетчер памяти инициирует ввод-вывод без использования кэша в драйвере файловой системы для извлечения данных, соответствующих части файла, отображененной на страницу, в отношении которой возникла ошибка.

Эти механизмы доступа, их назначение и порядок использования рассматриваются в следующих трех разделах.

Копирование в кэш и из кэша

Поскольку системный кэш находится в системном пространстве, он отображается на адресное пространство каждого процесса. Однако как и все страницы системного пространства, страницы кэша недоступны из пользовательского режима, потому что это стало бы потенциальной брешью в системе безопасности. (Например, у процесса могут отсутствовать права на чтение файла, чьи данные в настоящее время частично содержатся в системном кэше.) Таким образом, файловые операции чтения и записи кэшированных файлов со стороны пользовательского приложения должны обслуживаться процедурами режима ядра, которые копируют данные между буферами кэша в системном пространстве и буферами приложения, находящимися в адресном пространстве процесса.

Кэширование через интерфейсы отображения и фиксации

Точно так же, как пользовательские приложения считывают данные из файлов на диске и записывают в них данные, драйверы файловой системы нуждаются в чтении и записи данных, которые описывают сами эти файлы (метаданных, то есть данных о структуре тома). Однако поскольку драйверы файловой системы запускаются в режиме ядра, они могут при правильном информировании диспетчера кэша менять данные непосредственно в системном кэше. Чтобы допустить такую оптимизацию, диспетчер кэша предоставляет функции, которые позволяют драйверам файловой системы определять местонахождение метаданных файловой системы в виртуальной памяти, разрешая таким образом вносить непосредственные изменения, минуя промежуточные буферы.

Если драйверу файловой системы нужно прочитать метаданные файловой системы в кэше, он вызывает интерфейс отображения диспетчера кэша для получения виртуального адреса желаемых данных. Диспетчер кэша получает все запрашиваемые страницы, чтобы поместить их в память, а затем возвращает управление драйверу файловой системы. После этого драйвер файловой системы может обращаться к данным напрямую.

Если драйверу файловой системы нужно изменить страницы в кэше, он вызывает службы фиксации диспетчера кэша, которые сохраняют страницы активными в виртуальной памяти, не давая от них освободиться. Фактически, страницы не блокируются в памяти (как при блокировании драйвером устройства страниц для передачи данных путем прямого доступа к памяти). Чаще всего драйвер файловой системы дает своему потоку метаданных метку «не для записи», а заставляет подсистему записи отображенных страниц диспетчера памяти (см. главу 10) не записывать страницы на диск до тех пор, пока ей это не будет предписано явным образом. Когда драйвер файловой системы отменяет фиксацию страниц (освобождает страницы), диспетчер кэша освобождает свои ресурсы, чтобы осуществить отложенный сброс любых изменений на диск и освободить представление кэша, занимаемое метаданными.

Интерфейсы отображения и фиксации решают одну непростую проблему, возникающую при реализации файловой системы: управление буферами. Без непосредственного воздействия на кэшированные метаданные файловой системе приходится прогнозировать, какое максимальное количество буферов ей понадобится при обновлении структуры томов. Давая файловой системе возможность непосредственного доступа и обновления метаданных в кэше, диспетчер кэша позволяет отказаться от буферов, просто обновляя структуру тома в виртуальной памяти, предоставляемой диспетчером памяти. Единственным ограничением, с которым сталкивается файловая система, является объем доступной памяти.

Кэширование через интерфейсы прямого доступа к памяти

В дополнение к интерфейсам отображения и фиксации, используемым для непосредственного доступа к метаданным в кэше, диспетчер кэша предоставляет третий интерфейс для доступа к кэшированным данным — *прямой доступ к памяти* (Direct Memory Access, DMA). DMA-функции служат для чтения кэшированных страниц или для записи в эти страницы без промежуточных буферов, которые, к примеру, применяются при передаче данных по сети сетевой файловой системой.

DMA-интерфейс возвращает файловой системе физические адреса кэшированных пользовательских данных (а не виртуальные адреса, возвращаемые интерфейсами отображения и фиксации), которые затем могут использоваться для прямой передачи данных из физической памяти на сетевое устройство. Хотя небольшие объемы данных (от 1 до 2 Кбайт) можно передавать через обычные интерфейсы копирования, основанные на применении буферов, при более масштабных передачах данных DMA-интерфейс способен существенно повысить производительность сетевого сервера при обработке файловых запросов, поступивших от удаленных систем. Для описания этих ссылок на физическую память используется *список дескрипторов памяти* (Memory Descriptor List, MDL), о котором рассказывается главе 10.

Быстрый ввод-вывод

По возможности операции чтения и записи кэшированных файлов обрабатываются высокоскоростным механизмом *быстрого ввода-вывода* (fast I/O). Как показано в гла-

ве 8, при быстром вводе-выводе чтение и запись кэшированных файлов происходит без генерирования IRP-пакета. В этом случае диспетчер ввода-вывода вызывает процедуру быстрого ввода-вывода, имеющуюся в драйвере файловой системы, и проверяет, можно ли выполнить ввод-вывод непосредственно из диспетчера кэша, не генерируя IRP-пакет.

Поскольку архитектурно диспетчер кэша находится на вершине подсистемы виртуальной памяти, драйверы файловой системы могут использовать диспетчера кэша для доступа к данным файла, просто копируя данные в страницы, отображенные на нужный файл, или копируя данные из этих страниц, не тратя ресурсы на генерирование IRP-пакета.

Быстрый ввод-вывод возможен не всегда. Например, самое первое чтение из файла или запись в файл требует настройки файла на кэширование (отображения файла в кэше и настройки структур данных кэша, как описано в разделе «Структуры данных кэша»). Кроме того, если вызывающая процедура указала на необходимость чтения или записи в асинхронном режиме, быстрый ввод-вывод не используется, поскольку эта вызывающая процедура может быть приостановлена на время страничных операций ввода-вывода, требуемых для проведения буферизированного копирования в системный кэш или из него, тем самым не предоставляя реальной возможности на выполнение запрошенной асинхронной операции ввода-вывода. Но даже при выполнении синхронного ввода-вывода драйвер файловой системы может решить, что обрабатывать операцию ввода-вывода с помощью механизма быстрого ввода-вывода невозможно, к примеру, если интересующий нас файл имеет заблокированный диапазон байтов (в результате вызова функций `LockFile` и `UnlockFile`). Поскольку диспетчер кэша не знает, какие именно части файла заблокированы, драйвер файловой системы должен проверять действительность чтения или записи, для чего требуется генерировать IRP-пакет. Дерево решений для быстрого ввода-вывода показано на рис. 11.11.

При обслуживании чтения или записи с использованием быстрого ввода-вывода предпринимаются следующие действия:

1. Программный поток выполняет операцию чтения или записи.
2. Если файл кэширован, а ввод-вывод проводится в синхронном режиме, запрос передается точке входа в процедуру быстрого ввода-вывода в стеке драйвера файловой системы. Если файл не кэширован, драйвер файловой системы настраивает файл на кэширование, чтобы в следующий раз при запросе чтения или записи можно было воспользоваться быстрым вводом-выводом.
3. Если процедура быстрого ввода-вывода, принадлежащая драйверу файловой системы, определяет, что быстрый ввод-вывод возможен, она вызывает процедуру чтения или записи диспетчера кэша для непосредственного доступа к находящимся в кэше данным файла. (Если быстрый ввод-вывод невозможен, драйвер файловой системы возвращает управление системе ввода-вывода, которая затем генерирует IRP-пакет для ввода-вывода и в итоге вызывает обычную процедуру чтения файловой системы.)
4. Диспетчер кэша преобразует предоставленное смещение в файле в виртуальный адрес в кэше.
5. При операциях чтения диспетчер кэша копирует данные из кэша в буфер процесса, пославшего запрос, а при операциях записи — из буфера в кэш.



Рис. 11.11. Дерево решений для быстрого ввода-вывода

6. Осуществляется одно из следующих действий:

- Для чтения файла, при открытии которого не был указан флаг `FILE_FLAG_RANDOM_ACCESS`, обновляется информация упреждающего чтения в закрытой карте кэша того процесса, который вызвал драйвер. Для тех файлов, которые были открыты без указания флага `FO_RANDOM_ACCESS`, упреждающее чтение может также быть поставлено в очередь.
- В случае записи для любой измененной страницы в кэше устанавливается бит изменения, чтобы подсистема отложенной записи знала, что ее нужно сбросить на диск.
- Для файлов, требующих сквозной записи, все измененные данные тут же собираются на диск.

Упреждающее чтение и отложенная запись

В этом разделе рассказывается о том, как диспетчер кэша реализует чтение и запись данных файла в интересах системных драйверов. Следует учесть, что драйвер кэша

привлекается к вводу-выводу файлов, только когда файл открыт без флага FILE_FLAG_NO_BUFFERING с последующим чтением или записью с помощью Windows-функций ввода-вывода (например, `ReadFile` и `WriteFile`). Отображаемые файлы, а также файлы с установленным при их открытии флагом FILE_FLAG_NO_BUFFERING через диспетчер кэша не проходят.

ПРИМЕЧАНИЕ

Когда для открытия файла приложение использует флаг FILE_FLAG_NO_BUFFERING, его файловый ввод-вывод должен начинаться с выровненных устройствами смещений и с размерами, кратными размеру выравнивания; входной и выходной буфера этого приложения должны также располагаться по выровненным под устройства виртуальным адресам. Для файловых систем это значение обычно соответствуют размеру сектора (как правило, 512 байт для NTFS и 2048 байт для CDFS). Одним из преимуществ диспетчера кэша, если не считать фактическое повышение производительности кэширования, является тот факт, что он осуществляет промежуточную буферизацию, обеспечивающую произвольно выровненный и имеющий произвольные размеры ввод-вывод.

Интеллектуальное упреждающее чтение

Интеллектуальное упреждающее чтение (intelligent read-ahead) диспетчер кэша выполняет по принципу пространственной локальности, выстраивая предположение, что данные, которые вызывающий процесс, скорее всего, будет читать в следующий раз, основаны на данных, которые он читает сейчас. Поскольку в системном кэше используются виртуальные адреса, идущие для конкретного файла последовательно, сохранение такой же последовательности адресов в физической памяти не играет никакой роли. Упреждающее чтение файла для кэширования логических блоков — задача более сложная, требующая тесного взаимодействия между драйверами файловой системы и кэшем блоков, потому что такая система кэширования базируется на относительных позициях тех данных на диске, к которым идет обращение, и конечно же, файлы не обязательно должны храниться на диске строго последовательно. Активность упреждающего чтения можно отследить с помощью счетчика производительности `Cache: Read Aheads/sec` (Кэш: Упреждающий чтений/с) или переменной `CcReadAheadIosSystem`.

Считывание следующего блока файла, доступ к которому ведется последовательно, дает очевидные преимущества в плане повышения производительности, с тем лишь недостатком, что оно становится причиной дополнительного позиционирования головки жесткого диска. Чтобы распространить преимущества упреждающего чтения на доступ к разрозненным данным (как вперед, так и назад по файлу), диспетчер кэша хранит историю последних двух запросов на чтение в закрытой карте кэша для дескриптора файла, к которому было обращение, и такой метод известен как *асинхронное упреждающее чтение с использованием истории* (asynchronous read-ahead with history). Если в предположительно произвольных чтениях вызывающего приложения можно усмотреть какую-то схему, диспетчер кэша проводит ее экстраполяцию. Например, если вызывающее приложение считывает страницу номер 4000, а затем страницу номер 3000, диспетчер кэша предполагает, что в следующий раз этому приложению понадобится страница номер 2000, и осуществляет ее упреждающее чтение.

ПРИМЕЧАНИЕ

Хотя для построения предполагаемой последовательности вызывающее приложение должно выполнить как минимум три операции чтения, в закрытой карте кэша запоминаются только две из них.

Чтобы сделать упреждающее чтение еще эффективнее, Win32-функция `CreateFile` предоставляет флаг `FILE_FLAG_SEQUENTIAL_SCAN`, задающий последовательный доступ к файлу в прямом направлении. Если этот флаг установлен, диспетчер кэша не хранит историю чтений вызывающего приложения с целью прогнозирования, а выполняет вместо этого последовательное упреждающее чтение. Однако по мере того как файл считывается в рабочий набор кэша, диспетчер кэша прекращает отображение неактивных представлений файла, и если они не подверглись изменениям, предписывает диспетчеру памяти поместить страницы, принадлежащие таким представлениям, в верхнюю часть списка ожидающих страниц, чтобы их можно было быстро использовать снова. Также при упреждающем чтении считывается двукратный объем данных (например, не 1, а 2 Мбайт). По мере того как вызывающее приложение продолжает чтение, диспетчер кэша заранее считывает дополнительные блоки данных, всегда опережая читающее приложение на один блок (равный по размеру текущему считываемому блоку).

Упреждающее чтение диспетчера кэша носит асинхронный характер, поскольку оно выполняется в отдельном программном потоке, а не в потоке вызывающего приложения, и происходит параллельно с выполнением вызывающего приложения. При вызове для извлечения кэшированных данных диспетчер кэша с целью выполнения запроса сначала обращается к запрошенней виртуальной странице, а затем ставит в очередь дополнительный запрос на ввод-вывод в системный рабочий поток, чтобы извлечь дополнительные данные. Затем рабочий поток выполняется в фоновом режиме, считывая дополнительные данные, упреждая следующий запрос на чтение со стороны вызывающего приложения. Предварительно считанные страницы сбрасываются в память, чтобы к моменту запроса данных они уже были в памяти, а приложение продолжает выполняться.

Для приложений, не имеющих предсказуемых схем чтения, при вызове функции `CreateFile` может быть установлен флаг `FILE_FLAG_RANDOM_ACCESS`. Этот флаг предписывает диспетчеру кэша не делать попыток предугадать следующее чтение со стороны приложения, таким образом, режим упреждающего чтения отключается. Флаг также не дает диспетчеру кэша проводить агрессивную политику прекращения отображения представлений файла при обращениях к этому файлу, чтобы таким образом свести для файла к минимуму действия по отображению и прекращению отображения в случае нового обращения приложения к частям этого файла.

Кэширование с обратной записью и отложенная запись

Диспетчер кэша реализует кэширование с обратной записью, используя отложенную запись. Это означает, что данные, записываемые в файлы, сначала сохраняются в памяти на страницах кэша, а позже записываются на диск. Таким образом, операции записи могут накапливаться на короткое время, а затем разом сбрасываться на диск, сокращая общее количество дисковых операций ввода-вывода.

Для сброса кэшированных страниц диспетчер кэша должен явным образом вызвать диспетчера памяти, потому что в противном случае диспетчер памяти запишет содержимое памяти на диск, только когда спрос на физическую память превысит предложение (и при наличии соответствующих измененных данных). Однако кэшированные файловые данные являются отражением неизменяемых дисковых данных. Если процесс изменяет кэшированные данные, то пользователь ожидает своевременного отражения изменений на диске.

Кроме того, диспетчер кэша имеет возможность запретить выполнение диспетчёром памяти программного потока записи отображенных данных. Поскольку список измененных страниц (см. главу 10) не отсортирован по адресам логических блоков (Logical Block Address, LBA), попытки диспетчера кэша собрать страницы в кластер для более масштабных последовательных дисковых операций ввода-вывода не всегда бывают успешными и фактически приводят к многократным позиционированиям головки жесткого диска. Для борьбы с этим эффектом диспетчер кэша имеет возможность агрессивным образом запрещать выполнение программного потока записи отображенных данных, выполняя запись в порядке виртуального байтового смещения (Virtual Byte Offset, VBO), который намного ближе к порядку следования адресов логических блоков на диске. Поскольку теперь владельцем этих записей является диспетчер кэша, он также может применить собственные алгоритмы планирования и ограничения, чтобы отдать предпочтение упреждающему чтению над обратной записью и меньше влиять на систему.

Одним из важных решений является выбор частоты сброса кэша. Если кэш сбрасывается слишком часто, быстродействие системы снизится из-за ненужных операций ввода-вывода. Если же кэш будет сбрасываться слишком редко, появится риск утраты измененных файловых данных в случае краха системы (что особенно раздражает тех пользователей, которые пытались сохранить изменения) и нехватки физической памяти (из-за того что она была занята слишком большим количеством измененных страниц).

Для достижения разумного баланса один раз в секунду в системном рабочем программном потоке диспетчер кэша выполняет функцию *отложенной записи* (*lazy writer*), которая ставит в очередь для записи на диск одну восьмую часть измененных (с установленным битом изменения) страниц из системного кэша. Если частота появления измененных страниц требует увеличить объем, который по оценке подсистемы отложенной записи может быть в данной ситуации ею записан, она записывает дополнительное количество измененных страниц, необходимое по ее расчетам для соответствия частоте их появления. Реально операции ввода-вывода выполняются системными рабочими программными потоками из общесистемного пула критических рабочих потоков. Подсистеме отложенной записи также известно, когда принадлежащая диспетчеру памяти подсистема записи отображаемых страниц приступает к выполнению сброса. В таких случаях она откладывает свои механизмы обратной записи для соответствующего потока данных, чтобы избежать ситуации, когда две подсистемы сброса ведут запись в один и тот же файл.

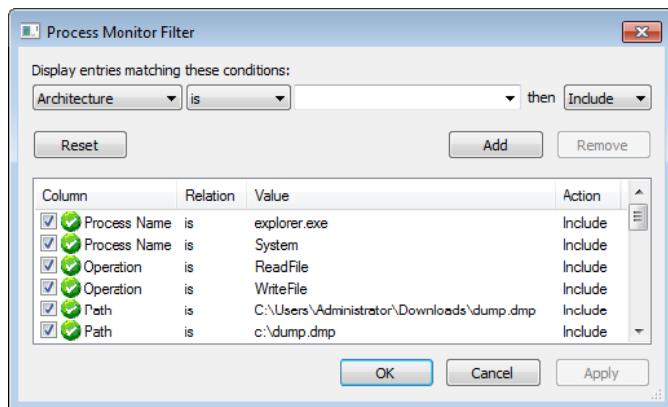
ПРИМЕЧАНИЕ

Диспетчер кэша предоставляет драйверам файловой системы средства, позволяющие отслеживать, когда и сколько данных записывается в файл. После того как подсистема отложенной записи сбросит измененные страницы на диск, диспетчер кэша уведомляет об этом файловую систему, предписывая ей провести обновление ее представления о длине достоверных данных файла. (Диспетчер кэша и файловая система отслеживают длину имеющихся в памяти достоверных данных файла независимо друг от друга.)

ЭКСПЕРИМЕНТ: НАБЛЮДЕНИЕ ЗА РАБОТОЙ ДИСПЕТЧЕРА КЭША

В данном эксперименте для просмотра исходной активности файловой системы, включая осуществляемые диспетчером кэша операции упреждающего чтения и обратной записи при копировании Windows Explorer большого файла (в данном примере — образа компакт-диска) из одного локального каталога в другой, используется программа Process Monitor.

Сначала настроим фильтр программы Process Monitor для включения исходных и целевых путей к файлам, процессов Explorer.exe и System, а также операций ReadFile и WriteFile. В данном примере файл C:\Users\Administrator\Downloads\dump.dmp копируется в C:\dump.dmp, поэтому настройка фильтра выглядит следующим образом.

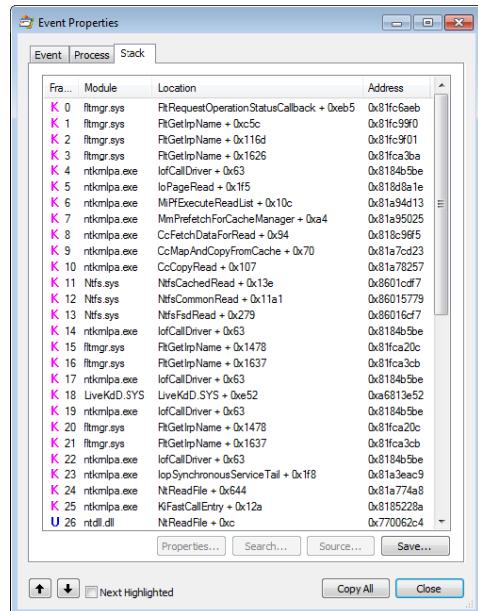


Трасса, созданная программой Process Monitor после копирования файла, должна выглядеть примерно так.

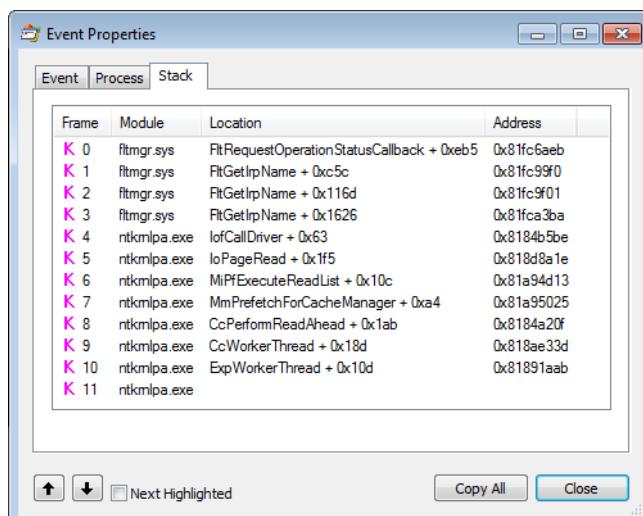
Process Name	Operation	Path	Result	Detail
Explorer.EXE	ReadFile	C:\Users\Admin... SUCCESS	Offset: 0, Length: 1,048,576, Priority: Normal	
Explorer.EXE	ReadFile	C:\Users\Admin... SUCCESS	Offset: 0, Length: 1,048,576, I/O Flags: Non-cached, Paging I/O, Priority: Normal	
Explorer.EXE	WriteFile	C:\dump.dmp	SUCCESS	Offset: 0, Length: 65,536, Priority: Normal
Explorer.EXE	WriteFile	C:\dump.dmp	SUCCESS	Offset: 65,536, Length: 65,536
Explorer.EXE	WriteFile	C:\dump.dmp	SUCCESS	Offset: 131,072, Length: 65,536
System	ReadFile	C:\Users\Admin... SUCCESS	Offset: 1,048,576, Length: 1,048,576, I/O Flags: Non-cached, Paging I/O, Priority: Normal	
Explorer.EXE	WriteFile	C:\dump.dmp	SUCCESS	Offset: 196,608, Length: 65,536
System	ReadFile	C:\Users\Admin... SUCCESS	Offset: 2,097,152, Length: 1,048,576, I/O Flags: Non-cached, Paging I/O, Priority: Normal	
Explorer.EXE	WriteFile	C:\dump.dmp	SUCCESS	Offset: 262,144, Length: 65,536
Explorer.EXE	WriteFile	C:\dump.dmp	SUCCESS	Offset: 327,680, Length: 65,536
Explorer.EXE	WriteFile	C:\dump.dmp	SUCCESS	Offset: 393,216, Length: 65,536
Explorer.EXE	WriteFile	C:\dump.dmp	SUCCESS	Offset: 458,752, Length: 65,536
Explorer.EXE	WriteFile	C:\dump.dmp	SUCCESS	Offset: 524,288, Length: 65,536
Explorer.EXE	WriteFile	C:\dump.dmp	SUCCESS	Offset: 589,824, Length: 65,536
Explorer.EXE	WriteFile	C:\dump.dmp	SUCCESS	Offset: 655,360, Length: 65,536
Explorer.EXE	WriteFile	C:\dump.dmp	SUCCESS	Offset: 720,896, Length: 65,536
Explorer.EXE	WriteFile	C:\dump.dmp	SUCCESS	Offset: 786,432, Length: 65,536
Explorer.EXE	WriteFile	C:\dump.dmp	SUCCESS	Offset: 851,968, Length: 65,536
Explorer.EXE	WriteFile	C:\dump.dmp	SUCCESS	Offset: 917,504, Length: 65,536
Explorer.EXE	WriteFile	C:\dump.dmp	SUCCESS	Offset: 983,040, Length: 65,536
Explorer.EXE	ReadFile	C:\Users\Admin... SUCCESS	Offset: 1,048,576, Length: 1,048,576	
Explorer.EXE	WriteFile	C:\dump.dmp	SUCCESS	Offset: 1,048,576, Length: 65,536
Explorer.EXE	WriteFile	C:\dump.dmp	SUCCESS	Offset: 1,114,112, Length: 65,536
Explorer.EXE	WriteFile	C:\dump.dmp	SUCCESS	Offset: 1,179,648, Length: 65,536
Explorer.EXE	WriteFile	C:\dump.dmp	SUCCESS	Offset: 1,245,184, Length: 65,536
Explorer.EXE	WriteFile	C:\dump.dmp	SUCCESS	Offset: 1,310,720, Length: 65,536
System	ReadFile	C:\Users\Admin... SUCCESS	Offset: 3,145,728, Length: 1,048,576, I/O Flags: Non-cached, Paging I/O, Priority: Normal	
Explorer.EXE	WriteFile	C:\dump.dmp	SUCCESS	Offset: 3,736,256, Length: 65,536
System	ReadFile	C:\Users\Admin... SUCCESS	Offset: 4,194,304, Length: 1,048,576, I/O Flags: Non-cached, Paging I/O, Priority: Normal	
Explorer.EXE	WriteFile	C:\dump.dmp	SUCCESS	Offset: 4,841,732, Length: 65,536

Первые несколько записей иллюстрируют начальную работу по вводу-выводу, выполняемую механизмом копирования, а также первые операции диспетчера кэша. Здесь можно проследить следующие моменты:

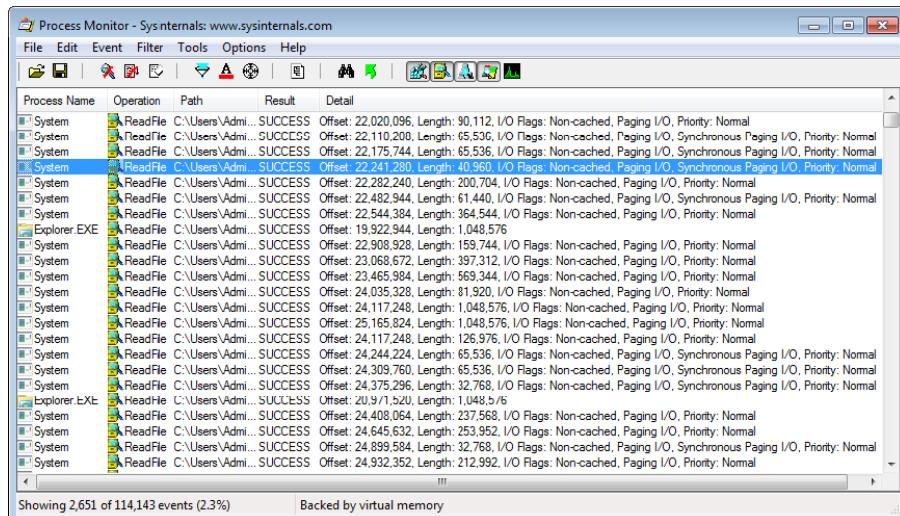
- Исходное 1-мегабайтное кэшированное чтение из Explorer в первую записи. Объем чтения зависит от внутреннего матричного вычисления на основе размера файла и может варьироваться от 128 Кбайт до 1 Мбайт. Поскольку был выбран довольно большой файл, механизм копирования предпочел размер 1 Мбайт.
- За 1-мегабайтным чтением последовало еще одно некэшируемое 1-мегабайтное чтение. Некэшируемое чтение обычно свидетельствует о действиях, связанных с ошибками отсутствия страниц или об обращении диспетчера кэша. Более пристальное изучение трассы стека для этих событий, на которую можно посмотреть, дважды щелкнув кнопкой мыши на записи и перейдя на вкладку Stack (Стек), показывает, что, конечно же, процедура CcCopyRead диспетчера кэша, вызванная процедурой чтения NTFS-драйвера, заставила диспетчер памяти сбросить исходные данные в физическую память.



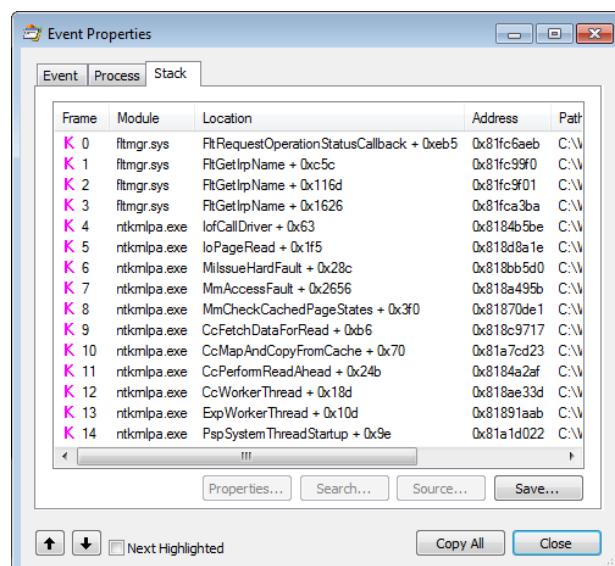
- После этой 1-мегабайтной операции ввода-вывода, связанной с ошибкой отсутствия страницы, механизм упреждающего чтения диспетчера кэша приступил к чтению файла, что подразумевает выполняемое процессом System последующее 1-мегабайтное чтение с 1-мегабайтным смещением. Исходя из размеров файла и операций ввода-вывода процесса Explorer, диспетчер кэша выбрал в качестве оптимального размера упреждающего чтения размер 1 Мбайт. Показанная следующей трассой стека для одной из операций упреждающего чтения подтверждает, что один из рабочих программных потоков диспетчера кэша выполняет упреждающее чтение.



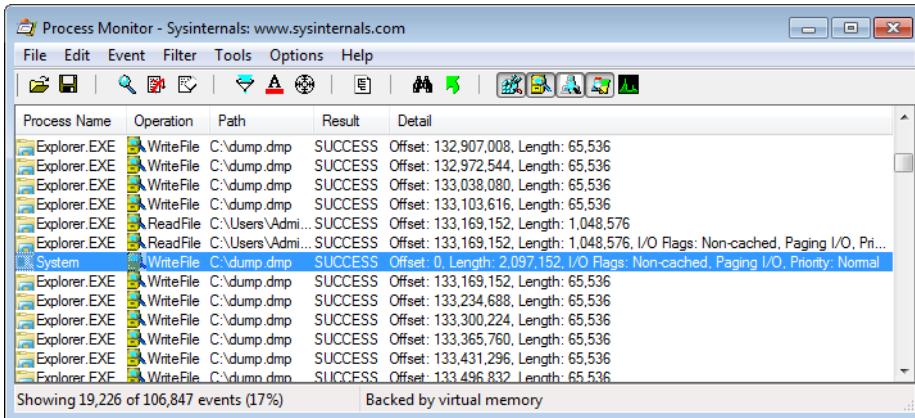
После этого операции 1-мегабайтного чтения, выполняемые Explorer, уже не следуют за ошибками отсутствия страниц, поскольку программный поток упреждающего чтения опережает Explorer, заранее извлекая файловые данные в ходе своих 1-мегабайтных некэшированных операций чтения. Однако время от времени поток упреждающего чтения не может заранее получить достаточный объем данных, в результате случаются групповые ошибки отсутствия страниц, которые помечаются как Synchronous Paging I/O (синхронный страницочный ввод-вывод).



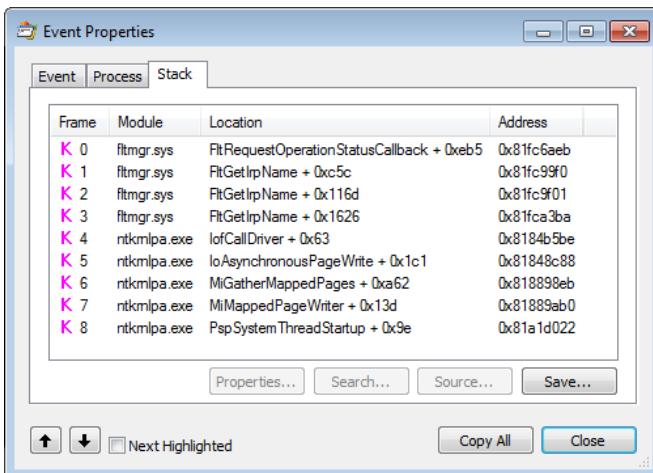
Если взглянуть на стек для таких записей, можно увидеть, что вместо процедуры `MmPrefetchForCacheManager` вызываются процедуры `MmAccessFault`/`MlIssueHardFault`.



Начиная чтение, Explorer приступает также к записи в целевой файл. Запись ведется в формате последовательных кэшированных 64-килобайтных записей. После считывания примерно 132 Мбайт из процесса System выполняется первая операция WriteFile, показанная на следующем рисунке.

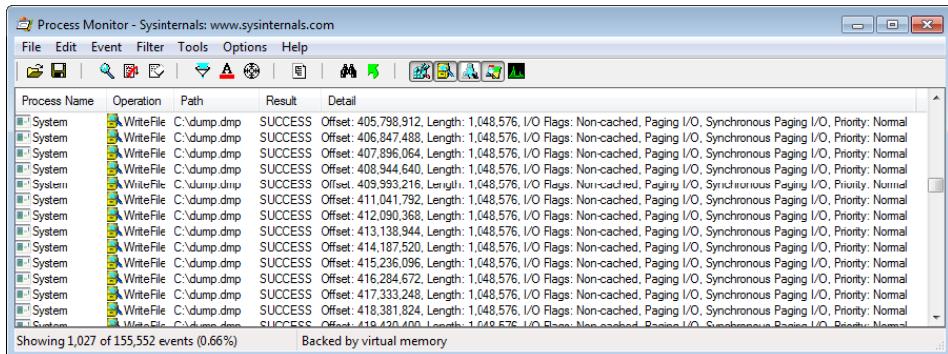


Представленная на следующем рисунке трасса стека операции записи свидетельствует о том, что фактически за запись отвечает программный поток записи отображаемых страниц диспетчера памяти.

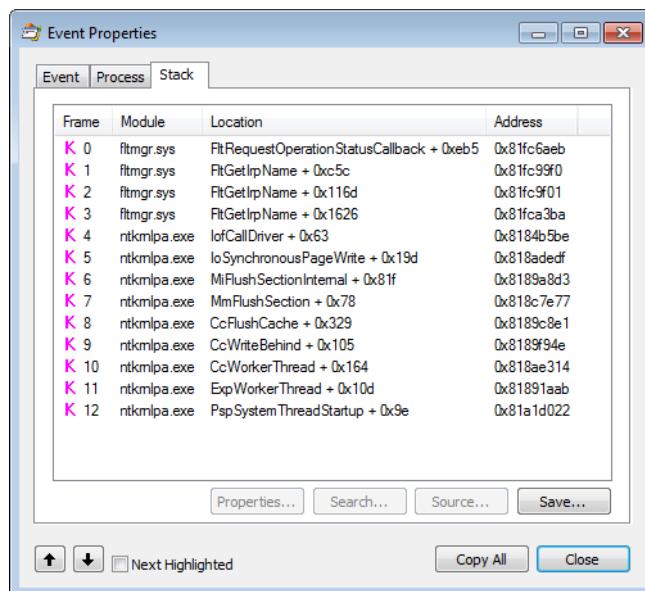


Это происходит из-за того, что для первых двух мегабайтов данных диспетчер кэша не выполняет упреждающего чтения, поэтому подсистема записи отображаемых страниц диспетчера памяти начинает сбрасывать на диск измененные данные целевого файла. (Дополнительные сведения о подсистеме записи отображаемых страниц есть в главе 10.)

Чтобы получить точное представление об операциях диспетчера кэша, удалите процесс Explorer из фильтра утилиты Process Monitor, оставив видимыми, как показано на следующем рисунке, только операции процесса System.



При просмотре этого окна намного проще видеть 1-мегабайтные операции обратной записи, выполняемые диспетчером кэша (на клиентских версиях Windows максимальные размеры записей составляют 1 Мбайт, на серверных — 32 Мбайт; этот эксперимент проводился на клиентской системе). Показанная на следующем рисунке трасса стека для одной из операций обратной записи подтверждает, что обратную запись выполняет рабочий программный поток диспетчера кэша.



В качестве дополнения к эксперименту попытайтесь воспроизвести его не с обычным, а с удаленным копированием (с одной системы Windows на другую), выбирая для копирования файлы различных размеров. Вы заметите несколько иные действия со стороны механизма копирования и диспетчера кэша, как на получающей, так и на отправляющей стороне.

Отключение режима отложенной записи для файла

При создании временного файла с применением флага `FILE_ATTRIBUTE_TEMPORARY` при вызове Windows-функции `CreateFile` подсистема отложенной записи не станет записывать измененные страницы на диск, пока не возникнет серьезная нехватка физической памяти или пока файл не будет сброшен на диск в принудительном порядке. Эта особенность подсистемы отложенной записи повышает производительность системы, поскольку не происходит немедленной записи на диск данных, которые в конечном счете могут оказаться ненужными. Приложения обычно удаляют временные файлы вскоре после закрытия.

Принудительное включение в кэш режима сквозной записи на диск

Поскольку некоторые приложения не терпят ни малейшей задержки между записью в файл и появлением обновлений на диске, диспетчер кэша поддерживает кэширование со сквозной записью для отдельно взятых файловых объектов, при этом изменения записываются на диск по мере их внесения. Для включения режима кэширования со сквозной записью при вызове функции `CreateFile` нужно установить флаг `FILE_FLAG_WRITE_THROUGH`. В качестве альтернативы в тот момент, когда данные должны быть записаны на диск, программный поток может явно сбросить на диск открытый файл, воспользовавшись Windows-функцией `FlushFileBuffers`.

Сброс отображаемых файлов

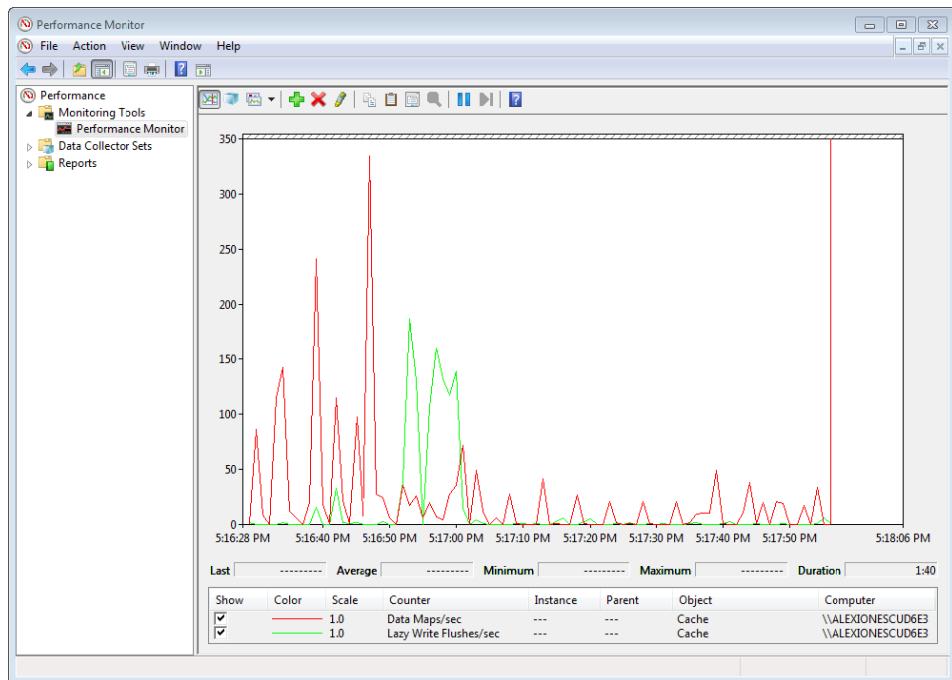
Если подсистема отложенной записи должна записать на диск данные из представления, которое также отображается на адресное пространство другого процесса, ситуация немного осложняется, потому что диспетчер кэша знает только о тех страницах, в которые он внес изменения. (Страницы, измененные другим процессом, известны только этому процессу, поскольку бит изменения в записях таблицы страниц, касающихся измененных страниц, сохраняется в закрытых таблицах страниц процесса.) Для решения этой проблемы диспетчер памяти сразу же оповещает диспетчер кэша, как только пользователь отображает какой-нибудь файл. Когда такой файл сбрасывается в кэш (например, в результате вызова Windows-функции `FlushFileBuffers`), диспетчер кэша записывает измененные страницы в кэш, а затем проверяет, не отображается ли файл также и на другой процесс. Когда диспетчер кэша обнаруживает этот факт, он сбрасывает все представление раздела, чтобы записать на внешний носитель те страницы, которые могли быть изменены вторым процессом. Если пользователь прекращает отображение представления того файла, который также был открыт в кэше, для измененных страниц устанавливается бит изменения, чтобы потом, когда программный поток отложенной записи будет сбрасывать представление, эти измененные страницы были записаны на диск. Эта процедура выполняется при развитии событий в следующем порядке:

1. Пользователь прекращает отображение представления.
2. Процесс сбрасывает на диск файловые буферы.

Если такая последовательность не выстраивается, то предугадать, какие именно страницы будут записаны на диск, невозможно.

ЭКСПЕРИМЕНТ: НАБЛЮДЕНИЕ ЗА ОПЕРАЦИЯМИ СБРОСА КЭША

Увидеть, как диспетчер кэша отображает представления в системный кэш и сбрасывает страницы на диск, можно с помощью системного монитора, добавив к нему счетчики Data Maps/sec (Кэш: Отображений данных/с) и Lazy Write Flushes/sec (Сбросов ленивой записи/с). Затем нужно выполнить копирование большого файла из одного места в другое. Самая высокая линия на следующем экране показывает значения счетчика Data Maps/sec (Кэш: Отображений данных/с), другая линия — значения счетчика Lazy Write Flushes/sec (Сбросов ленивой записи/с). В ходе копирования файла линия, показывающая значения счетчика Lazy Write Flushes/sec (Сбросов ленивой записи/с), резко идет вверх.



Ограничение записи

Файловая система и диспетчер кэша должны определить, скажется ли на производительности системы запрос на кэшированную запись, а затем спланировать все отложенные записи. Сначала файловая система через функцию `CcCanIWrite` узнает у диспетчера кэша, можно ли без ущерба для производительности прямо сейчас записать определенное количество байтов, и при необходимости блокирует эту запись. Чтобы получился асинхронный ввод-вывод, файловая система задает обратный вызов диспетчера кэша для автоматической записи байтов, когда запись опять будет разрешена путем вызова функции `CcDeferWrite`. В противном случае просто выставляется блокировка и ожидается ответ процедуры `CcCanIWrite` для продолжения работы. Как только будет получено уведомление о предстоящей операции записи, диспетчер кэша

определяет количество измененных страниц, имеющихся в кэше, и объем доступной физической памяти. Если свободными остаются всего несколько страниц физической памяти, диспетчер кэша тут же блокирует программный поток файловой системы, запросивший запись данных в кэш. Подсистема отложенной записи диспетчера кэша сбрасывает часть измененных страниц на диск, после чего разрешает заблокированному потоку файловой системы продолжить работу. Такое *ограничение записи* (write throttling) не дает резко упасть производительности системы из-за нехватки памяти, когда файловая система или сетевой сервер выполняют объемные операции записи.

ПРИМЕЧАНИЕ

Эффект ограничения записи дифференцируется по используемым томам, то есть если пользователь копирует большой файл, скажем, на твердотельный диск RAID-0 SSD, и при этом переносит документ на флэш-диск с USB-интерфейсом, то запись на флэш-диск не приведет к ограничению записи в ходе переноса данных на SSD-диск.

Порог измененных страниц (dirty page threshold) представляет собой допустимое системным кэшем количество страниц перед началом ограничения кэшированных записей. Это значение вычисляется при инициализации системы и зависит от типа системы (клиент или сервер). Вычисляются также два других значения: *верхний* (top) и *нижний* (bottom) пороги измененных страниц. В зависимости от потребления памяти и скорости обработки измененных страниц подсистема отложенной записи вызывает внутреннюю функцию `CcAdjustThrottle`, которая на серверных системах выполняет динамическую настройку текущего порога на основе вычисления верхнего и нижнего значений. Эта настройка делается с целью сохранения кэша для чтения в случае высокой загруженности из-за записи, что неизбежно приведет к переполнению кэша, а также для ограничения записи. В табл. 11.1 перечислены алгоритмы, используемые для вычисления пороговых значений измененных страниц.

Таблица 11.1. Алгоритмы, используемые для вычисления порогов измененных страниц

Тип системы	Порог измененных страниц	Верхнее число измененных страниц	Нижний порог измененных страниц
Клиент	Число физических страниц / 8	Число физических страниц / 8	Число физических страниц / 8
Сервер	Число физических страниц / 2	Число физических страниц / 2	Число физических страниц / 8

Ограничение записи может также применяться в системах переадресации при передаче данных по медленным линиям связи. Предположим, к примеру, что локальный процесс записывает большой объем данных на удаленную файловую систему по линии с пропускной способностью 9600 бод. Данные не запишутся на удаленный диск, пока подсистема отложенной записи диспетчера кэша сбрасывает данные кэша на диск. Если у системы переадресации накопится множество измененных страниц, которые должны сбрасываться на диск за один прием, адресат может перед завершением передачи данных получить задержку. С помощью функции `CcSetDirtyPageThreshold`

диспетчер кэша позволяет сетевым системам переадресации устанавливать лимит на число измененных кэшированных страниц, которое они в состоянии обработать (для каждого потока данных), предотвращая тем самым неблагоприятное развитие событий. Ограничивая число измененных страниц, система переадресации гарантирует, что операции сброса данных кэша не повлекут за собой задержку.

ЭКСПЕРИМЕНТ: ПРОСМОТР ПАРАМЕТРОВ ОГРАНИЧЕНИЯ ЗАПИСИ

Команда `!defwrites` отладчика ядра выводит дамп переменных ядра, используемых диспетчером кэша, включая число измененных страниц в файловом кэше (`CcTotalDirtyPages`). Это значение позволяет принять решение о необходимости ограничения записи:

```
1kd>
!defwrites
*** Cache Write Throttle Analysis ***

CcTotalDirtyPages: 39 ( 156 Kb)
CcDirtyPageThreshold: 32753 ( 131012 Kb)
MmAvailablePages: 81569 ( 326276 Kb)
MmThrottleTop: 450 ( 1800 Kb)
MmThrottleBottom: 80 ( 320 Kb)
MmModifiedPageListHead.Total: 4337 ( 17348 Kb)
Write throttles not engaged
```

По этим данным видно, что число измененных страниц далеко от того, при котором инициируется ограничение записи (`CcDirtyPageThreshold`), поэтому система в нем не заинтересована.

Системные программные потоки

Как уже упоминалось, диспетчер кэша выполняет операции ввода-вывода, связанные с отложенной записью и упреждающим чтением, передавая запросы общему пулу критически важных системных рабочих потоков. Но он накладывает на применение этих программных потоков определенные ограничения, разрешая использовать на один поток меньше их общего количества в системах с небольшим и средним объемами памяти (и на два потока меньше их общего количества в системах с большим объемом памяти).

В своих внутренних структурах диспетчер кэша организует свои рабочие запросы в виде четырех списков (хотя все они обслуживаются одним и тем же набором рабочих потоков исполнительной системы):

- Экспресс-очередь** (*express queue*) служит для операций упреждающего чтения.
- Обычная очередь** (*regular queue*) предназначена для операций сканирования, связанных с отложенной записью (в поисках измененных данных, подлежащих сбросу на диск), для обратных записей и для отложенных операций закрытия файлов.
- Очередь быстрого освобождения** (*fast teardown queue*) используется, когда диспетчер памяти ждет освобождения раздела данных, которым владеет диспетчер кэша, чтобы

файл можно было открыть с помощью образа (вместо этого раздела данных), заставив функцию `CcWriteBehind` сбросить весь файл и освободить общую карту кэша.

- ❑ *Последиковая очередь* (post tick queue) применяется диспетчером кэша для внутренней регистрации уведомлений после каждого «тика» программного потока подсистемы отложенной записи, иными словами, после завершения каждого прохода.

Для отслеживания рабочих элементов, подлежащих выполнению рабочими потоками, диспетчер кэша для каждого процессора создает собственный внутренний ассоциативный список фиксированного размера, содержащий структуры рабочих элементов для поставленных в очередь рабочих потоков. (Ассоциативные списки изучаются в главе 10.) Число элементов в очереди рабочих потоков зависит от размеров системы: 32 – для систем с небольшим объемом памяти, 64 – для систем со средним объемом памяти, 128 – для клиентских систем с большим объемом памяти и 256 – для серверных систем с большим объемом памяти. Для повышения межпроцессорной производительности диспетчер кэша выделяет также *глобальный ассоциативный список* (global look-aside list), размер которого определяется аналогично.

Заключение

Диспетчер кэша предоставляет высокоскоростной интеллектуальный механизм для сокращения объема дискового ввода-вывода и повышения общей пропускной способности системы. Благодаря кэшированию на основе виртуальных блоков диспетчер кэша может выполнять интеллектуальное упреждающее чтение. А благодаря использованию при доступе к файловым данным примитивов отображения файлов диспетчера памяти, диспетчер кэша способен предложить специальный механизм быстрого ввода-вывода, экономящий время центрального процессора при выполнении операций чтения и записи. Кроме того, диспетчер кэша оставляет все вопросы, связанные с управлением физической памятью, в ведении единого глобального диспетчера памяти Windows, сокращая тем самым дублирование кода и повышая эффективность системы.

Глава 12. Файловые системы

В этой главе представлен обзор форматов поддерживаемых в Windows файловых систем. Кроме того, описываются типы драйверов файловых систем и принципы их работы, в том числе способы взаимодействия с другими компонентами операционной системы, например с диспетчерами памяти и кэша. Также вы научитесь пользоваться программой Process Monitor, созданной в Windows Sysinternals (с продукцией этой компании можно ознакомиться на странице <http://technet.microsoft.com/ru-ru/sysinternals/default.aspx>) и предназначеннной для анализа проблем, связанных с доступом к файловой системе.

Основной материал главы начинается с описания файловой системы с типовым протоколированием (Common Log File System, CLFS) – транзакционной виртуальной файловой системы с поддержкой протоколирования, реализованной на базе родного для Windows формата NTFS. Затем рассматриваются сама файловая система NTFS и ее нетривиальные функциональные возможности, в частности сжатие данных, способность к восстановлению, дисковые квоты, символические ссылки, транзакции (которые пользуются службами, предоставляемыми CLFS) и шифрование.

Для понимания материала данной главы необходимо знакомство с терминологией, введенной в главе 9. Вы должны понимать, что такое «тот» и «раздел». Также вам должны быть знакомы следующие термины:

- *Секторы* (sectors) – аппаратно адресуемые блоки носителя. На жестких дисках обычно используются секторы размером 512 байт, но постепенно происходит переход к 4096-байтным секторам (см. главу 9). Таким образом, если операционная система при размере сектора 512 байт собирается модифицировать 632-й байт диска, она записывает 512-байтовый блок во второй сектор диска.
- *Форматы файловых систем* (file system formats) определяют принципы хранения данных на носителях и влияют на характеристики файловой системы. К примеру, в файловой системе, формат которой не допускает ассоциирование прав доступа с файлами и папками, обеспечение безопасности невозможно. Формат файловой системы также может налагать ограничения на размеры файлов и емкость поддерживаемых устройств внешней памяти. Наконец, некоторые форматы файловых систем эффективно реализуют поддержку либо больших, либо малых файлов и дисков. В качестве примера файловых систем, предлагающих разный набор возможностей и сценариев применения, можно привести NTFS и exFAT.
- *Кластеры* (clusters) представляют собой адресуемые блоки, используемые многими файловыми системами. Размер кластера всегда кратен размеру сектора, как показано на рис. 12.1. Файловые системы применяют кластеры для более эффективного управления дисковым пространством: кластеры, размер которых превышает размер сектора, позволяют разбить диск на блоки меньшей длины, которые проще

в управлении, чем секторы. К потенциальным недостаткам кластеров большого размера можно отнести неэффективное использование дискового пространства или внутреннюю фрагментацию, которая возникает из-за того, что размеры файлов редко бывают кратными размеру кластера.

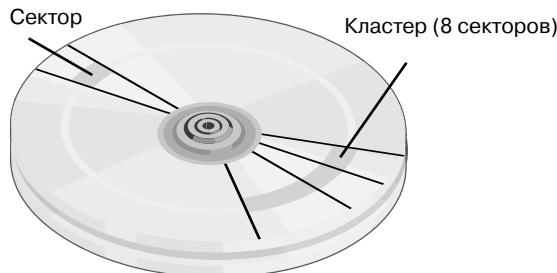


Рис. 12.1. Секторы и кластер на диске

- *Метаданные* (metadata) — хранящиеся на томе данные, необходимые для управления файловой системой. Приложениям они, как правило, недоступны. В метаданные входит, к примеру, информация, определяющая расположение файлов и папок на томе.

Форматы файловых систем в Windows

В Windows поддерживаются следующие форматы файловых систем:

- CDFS;
- UDF;
- FAT12, FAT16 и FAT32;
- exFAT;
- NTFS.

Далее вы увидите, что каждая из этих систем оптимальна для определенной среды.

CDFS

CDFS (CD-ROM File System — файловая система для CD-ROM) представляет собой предназначенный только для чтения драйвер файловой системы (%SystemRoot%\System32\Drivers\Cdfs.sys), поддерживающий расширенный набор дисковых форматов ISO-9660 и Joliet. Если формат ISO-9660 относительно прост и ограничивается именами длиной в 32 символа, содержащими только символы верхнего регистра в кодировке ASCII, то формат Joliet более гибок и поддерживает имена в формате Unicode произвольной длины. При наличии на диске структуры для обоих форматов

(для обеспечения максимальной совместимости), CDFS использует формат Joliet. CDFS имеет два ограничения:

- максимальная длина файлов не должна превышать 4 Гбайт;
- максимальное число папок не может превышать 65 535.

Формат CDFS считается устаревшим, так как в качестве стандарта для оптических носителей был принят формат UDF (Universal Disk Format).

UDF

Файловая система UDF в Windows является UDF-совместимой реализацией OSTA (Optical Storage Technology Association – ассоциация по технике и технологии оптических запоминающих устройств). UDF относится к подмножеству формата ISO-13346 с расширениями для поддержки таких форматов, как CD-R и DVD-R/RW. В 1995 году ассоциация OSTA определила UDF как формат для магнитооптических носителей, главным образом DVD-ROM, предназначенный для замены формата ISO-9660. Формат UDF включен в спецификацию DVD и гибкостью превосходит CDFS. Он обладает следующими преимуществами:

- длина имен папок и файлов в кодировке ASCII может доходить до 254 символов, а в кодировке Unicode – до 127;
- файлы могут быть разреженными (о том, что это такое, мы поговорим позже);
- размеры файлов задаются 64-разрядными значениями;
- поддерживаются списки контроля доступа (Access Control Lists, ACL);
- поддерживаются альтернативные потоки данных.

UDF-драйвер поддерживает версии UDF до 2.60. Формат UDF разрабатывался с учетом особенностей перезаписываемых носителей. UDF-драйвер в Windows (%SystemRoot%\System32\Drivers\Udfs.sys) поддерживает запись и чтение на дисках Blu-ray, DVD-RAM, CD-R/RW и DVD+R/RW в версии UDF 2.50, в то время как в версии UDF 2.60 поддерживаются носители только для чтения. Кроме того, в Windows отсутствует реализация ряда возможностей UDF, например именованных потоков данных и списков контроля доступа.

FAT12, FAT16 и FAT32

В Windows файловая система FAT поддерживается в основном для совместимости с другими операционными системами при многовариантной загрузке и в качестве формата для устройств флэш-памяти или карт памяти. Драйвер этой файловой системы находится в файле %SystemRoot%\System32\Drivers\Fastfat.sys.

Число в названии каждого формата FAT означает разрядность, применяемую для идентификации кластеров на диске. 12-разрядный идентификатор кластеров в FAT12 ограничивает размер дискового раздела значением в 2^{12} (4096) кластеров. В Windows допустимы размеры кластеров от 512 байт до 8 Кбайт, что ограничивает размер тома FAT12 значением в 32 Мбайт.

ПРИМЕЧАНИЕ

Все файловые системы FAT резервируют первые два и последние шестнадцать кластеров на томе, поэтому, например, в случае FAT12 число доступных для использования кластеров чуть меньше, чем 4096.

Файловая система FAT16, с 16-разрядным идентификатором кластеров, допускает наличие 2^{16} (65 536) кластеров. В Windows размер кластера FAT16 варьируется от 512 байт (размер сектора) до 64 Кбайт (на дисках с размером сектора 512 байт), что ограничивает размер тома FAT16 значением в 4 Гбайт. На дисках с размером сектора 4096 байт допустимы кластеры размером 256 Кбайт. Windows выбирает размер кластера в зависимости от размера тома. Возможные варианты перечислены в табл. 12.1. Когда при помощи команды `format` или оснастки *Disk Management* под FAT форматируется том размером менее 16 Мбайт, Windows использует FAT12 вместо FAT16.

Таблица 12.1. Размер кластеров FAT16 в Windows

Размер тома	Размер кластера по умолчанию
<8 Мбайт	Не поддерживается
8–32 Мбайт	512 байт
32–64 Мбайт	1 Кбайт
64–128 Мбайт	2 Кбайт
128–256 Мбайт	4 Кбайт
256–512 Мбайт	8 Кбайт
512–1024 Мбайт	16 Кбайт
1–2 Гбайт	32 Кбайт
2–4 Гбайт	64 Кбайт
>16 Гбайт	Не поддерживается

Том в FAT делится на несколько областей, как показано на рис. 12.2. Таблица размещения файлов (File Allocation Table, FAT), от которой собственно пошло название этой файловой системы, имеет по одной записи для каждого кластера на томе. Так как наличие этой таблицы необходимо для успешной интерпретации содержимого тома, формат FAT поддерживает две ее копии. В результате, если драйвер файловой системы или программа проверки целостности диска (например, Chkdsk) не может получить доступ к одной копии FAT (например, из-за битого сектора на диске), в ход идет вторая копия.

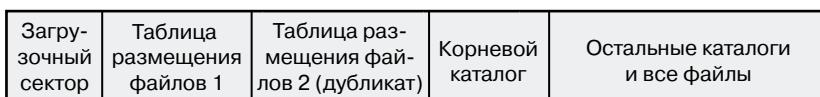


Рис. 12.2. Структура формата FAT

Записи в таблице определяют цепочки размещения файлов и папок (как показано на рис. 12.3), в которых отдельные звенья представляют собой указатели на следующий кластер с данными файла. Запись, относящаяся к папке, в которой находится файл, хранит начальный кластер файла. Последняя запись в цепочке содержит зарезервированное значение 0xFFFF для FAT16 и 0xFFF для FAT12. Записи в FAT, соответствующие свободным кластерам, имеют нулевые значения. На рис. 12.3 видно, что файлу FILE1 назначены кластеры 2, 3 и 4; файл FILE2 фрагментирован и использует кластеры 5, 6 и 8; а файл FILE3 занимает только кластер 7. Чтение файла с FAT-тома может потребовать чтения больших фрагментов таблицы размещения файлов для обхода всех цепочек размещения.

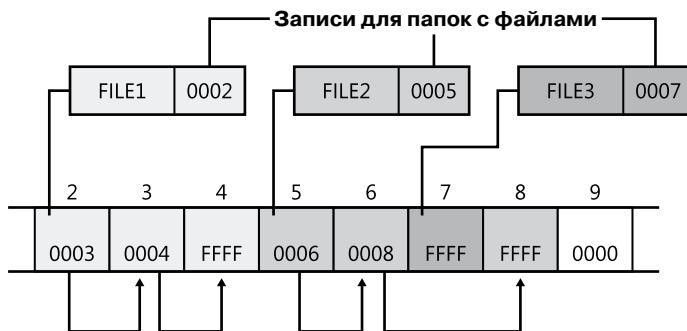


Рис. 12.3. Примеры цепочек размещения файлов в FAT

В начале тома FAT12 или FAT16 заранее выделяется место для корневого каталога, достаточное для хранения 256 записей, что ограничивает количество файлов и папок в корневом каталоге. (В FAT32 подобное ограничение отсутствует.) Размер каждой записи в папке составляет 32 байта. Запись содержит имя файла, его размер, начальный кластер и временную метку (время последнего доступа к файлу, время создания и т. п.). Если имя файла состоит из Unicode-символов или не соответствует принятым в MS-DOS правилам именования по формуле «8.3», оно считается длинным и для его хранения выделяются дополнительные записи, которые предшествуют основной записи. На рис. 12.4 показан пример записи для файла с именем «The quick brown fox». Система создала представление этого имени в формате «8.3» THEQUI~1.FOX (в записи вы не увидите точки, так как предполагается, что точка появляется после восьмого символа) и воспользовалась двумя дополнительными записями для хранения длинного имени в формате Unicode. На рисунке каждая строка состоит из 16 байт.

В FAT32 используются 32-разрядные идентификаторы кластеров, но старшие 4 бита резервируются, поэтому в действительности идентификаторов всего 28. Так как размер кластера в FAT32 может доходить до 64 Кбайт, теоретически эта файловая система может функционировать с 16-терабайтными томами. Хотя Windows работает с существующими томами FAT32 больших размеров (которые созданы в других операционных системах), в настоящее время она ограничивает новые тома FAT32

размером в 32 Гбайт. Большее количество кластеров, поддерживаемое в FAT32, позволяет этой файловой системе управлять дисками более эффективно, чем FAT16; она может манипулировать томами размером до 128 Гбайт с 512-байтными кластерами. В табл. 12.2 перечислены размеры кластеров, используемые по умолчанию на томах в FAT32.



Рис. 12.4. Запись для папки в FAT

Таблица 12.2. Размер кластеров для томов в FAT32

Размер раздела	Размер кластера по умолчанию
<32 Мбайт	Не поддерживается
32–64 Мбайт	512 байт
64–128 Мбайт	1 Кбайт
128–256 Мбайт	2 Кбайт
256 Мбайт – 8 Гбайт	4 Кбайт
8–16 Гбайт	8 Кбайт
16–32 Гбайт	16 Кбайт
>32 Гбайт	Не поддерживается

Кроме большего максимального количества кластеров, FAT32 имеет перед FAT12 и FAT16 такое преимущество, как отсутствие предопределенного места хранения корневого каталога. А это значит, что на размер данного каталога не накладывается ограничений. Кроме того, для надежности FAT32 хранит вторую копию загрузочного сектора. Общим ограничением для FAT32 и FAT16 является максимальный размер файла. Он не может превышать 4 Гбайт, так как размер файла в папке описывается 32-разрядным числом.

exFAT

Разработанная Microsoft расширенная таблица размещения файлов (Extended File Allocation Table, exFAT), также называемая FAT64, представляет собой усовершенствованную версию традиционных файловых систем FAT и предназначена главным образом для флэш-накопителей. Ее основным предназначением является обеспечение ряда расширенных функциональных возможностей, предлагаемых NTFS, но без структур метаданных для служебной информации и журналов метаданных, алгоритмы записи которых не подходят для большинства флэш-накопителей. (Описание флэш-накопителей было дано в главе 9.) В табл. 12.3 перечислены размеры кластеров в exFAT, используемые по умолчанию.

Таблица 12.3. Размеры кластеров для томов в exFAT

Размер тома	Размер кластера по умолчанию
<7 Мбайт	Не поддерживается
7–256 Мбайт	4 Кбайт
256 Мбайт – 32 Гбайт	32 Кбайт
32 Гбайт – 256 Тбайт	128 Кбайт
>256 Тбайт	Не поддерживается

Как следует из названия файловой системы FAT64, размер файла увеличился до 2^{64} , то есть в ней допустимы файлы размером до 16 эксабайт. Это изменение соответствует увеличению максимального размера кластера, который в настоящее время реализуется как 32-мегабайтный и может включать в себя до 2255 секторов. В exFAT также появилась бит-карта, отслеживающая свободные кластеры, что благотворно сказалось на процедурах выделения места и удаления. Наконец, exFAT позволяет поместить в одну папку более 1000 файлов. Результатом всего этого явились лучшая масштабируемость и поддержка дисков больших размеров.

Кроме того, в exFAT реализован ряд возможностей, ранее доступных только в NTFS, например поддержка списков контроля доступа (ACL) и транзакций — в этом случае exFAT называют FAT с поддержкой безопасных транзакций (Transaction-Safe FAT, TFAT). Эти функциональные возможности реализованы в exFAT операционной системы Windows Embedded CE, но не реализованы в версии exFAT для Windows.

ПРИМЕЧАНИЕ

Технология ReadyBoost, описанная в главе 10, работает с флэш-накопителями, отформатированными для exFAT, что позволяет поддерживать файлы кэша, размер которых превышает 4 Гбайт.

NTFS

Как уже упоминалось в начале этой главы, файловая система NTFS относится к родным для Windows форматам. В ней используются 64-разрядные номера кластеров, что позволяет адресовать тома размером до 16 экзабайт. Однако Windows ограничивает

размеры NTFS-томов значениями, при которых возможна адресация 32-разрядными кластерами, что чуть меньше, чем 256 Тбайт (с использованием кластеров по 64 Кбайт). В табл. 12.4 перечислены размеры кластеров на томах в NTFS, используемые по умолчанию. (Их можно переопределить при форматировании NTFS-тома.) Количество файлов на томе в NTFS может доходить до $2^{32}-1$. Формат NTFS допускает файлы размером 16 эксабайт, но пределы реализации ограничивают максимальный размер файла величиной в 16 Тбайт.

Таблица 12.4. Размеры кластеров на томах в NTFS

Размер тома	Размер кластера по умолчанию
<7 Мбайт	Не поддерживается
7 Мбайт – 16 Тбайт	4 Кбайт
16–32 Тбайт	8 Кбайт
32–64 Тбайт	16 Кбайт
64–128 Тбайт	32 Кбайт
128–256 Тбайт	64 Кбайт

NTFS поддерживает ряд дополнительных функциональных возможностей, например защиту файлов и папок, альтернативные потоки данных, дисковые квоты, разрешенные файлы, сжатие файлов, символические (мягкие) и жесткие ссылки, семантику транзакций, точки соединения и шифрование. Одной из важнейших характеристик этой файловой системы является *восстановляемость* (recoverability). При неожиданной остановке работы системы целостность метаданных FAT-тома может быть утрачена, что приведет к повреждению структуры папок и значительного объема данных. Журнал изменений метаданных, который ведет NTFS, реализован с помощью транзакций, поэтому целостность файловой системы может быть восстановлена без потери информации о структуре файлов и папок. (Хотя, если пользователь не применяет систему TxF, о которой пойдет речь чуть позже, данные файлов могут быть потеряны.) Кроме того, NTFS-драйвер в Windows реализует технологию *самовосстановления* (self-healing), которая позволяет исправить большую часть мелких повреждений файловой системы на структурах диска без перезагрузки Windows.

Далее в этой главе мы более подробно рассмотрим структуры данных в NTFS и нетривиальные возможности этой файловой системы.

Архитектура драйверов файловой системы

Драйверы файловой системы (File System Drivers, FSD) управляют форматами файловой системы. Хотя драйверы файловой системы запускаются в режиме ядра, по сравнению со стандартными драйверами режима ядра они имеют ряд особенностей. Важнейшей из этих особенностей является необходимость регистрироваться у диспетчера ввода-вывода и более тесное взаимодействие с диспетчером памяти. Для достижения более высокой производительности драйверы файловой системы обычно опираются на службы диспетчера кэша. То есть они пользуются куда большим набором

экспортируемых функций из файла Ntoskrnl.exe, чем стандартные драйверы. При этом как и для стандартных драйверов ядра, для разработки драйверов файловой системы требуется Windows Driver Kit (WDK). (Подробно набор WDK рассматривался в главе 1 части I, кроме того, о нем можно прочитать на странице <http://www.microsoft.com/whdc/devtools/wdk>.)

В Windows есть два типа драйверов файловой системы:

- ❑ локальные FSD-драйверы управляют томами, подключенными непосредственно к компьютеру;
- ❑ сетевые FSD-драйверы позволяют обращаться к дисковым томам, подключенными к удаленным компьютерам.

Локальные FSD-драйверы

К локальным FSD-драйверам относятся Ntfs.sys, Fastfat.sys, Exfat.sys, Udfs.sys и Cdfs.sys, а также драйвер Raw (встроенный в Ntoskrnl.exe). Упрощенная схема взаимодействия локальных FSD-драйверов с диспетчером ввода-вывода и драйверами устройств внешней памяти представлена на рис. 12.5. Как описывалось в разделе «Монтирование томов» главы 9, локальный FSD-драйвер должен регистрироваться у диспетчера ввода-вывода. После этого диспетчера может вызывать его для распознавания томов при первом обращении к тому со стороны системы или одного из приложений. Процесс распознавания включает анализ загрузочного сектора тома и часто в качестве проверки целостности — анализ метаданных файловой системы. Если том не распознается ни одной из зарегистрированных файловых систем, ему назначается FSD-драйвер Raw, после чего появляется диалоговое окно, в котором пользователя спрашивают, хочет ли он отформатировать том. В случае отрицательного ответа доступ к тому обеспечивает драйвер Raw, но исключительно на уровне секторов — другими словами, возможны чтение и запись только секторов целиком.

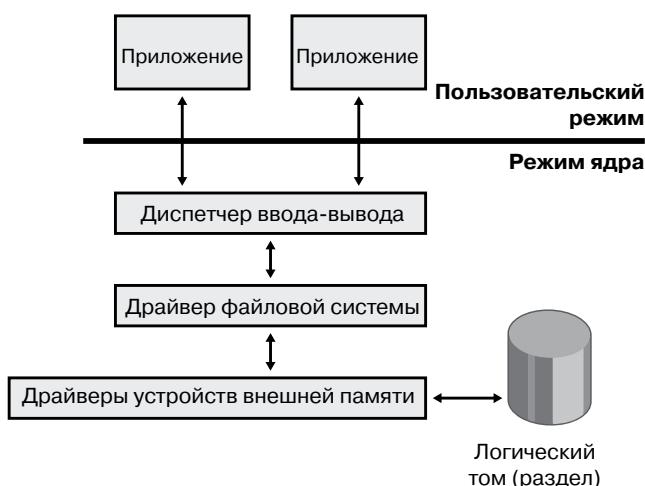


Рис. 12.5. Локальный драйвер файловой системы

Целью распознавания файловой системы является предоставление операционной системе дополнительного варианта в виде отличной от Raw, легитимной, но нераспознанной файловой системы. Для этого в первом секторе тома находится фиксированный тип структуры данных (**FILE_SYSTEM_RECOGNITION_STRUCTURE**). Именно эта структура данных распознается операционной системой, которая затем уведомляет пользователя о наличии на томе легитимной, но нераспознанной файловой системы. После этого на томе все равно будет загружена файловая система Raw, но предложения отформатировать том уже не поступит. Пользовательское приложение или драйвер режима ядра могут попросить копию **FILE_SYSTEM_RECOGNITION_STRUCTURE**, воспользовавшись новым управляющим кодом ввода-вывода файловой системы **FSCTL_QUERY_FILE_SYSTEM_RECOGNITION**.

Первый сектор любого поддерживаемого Windows формата файловой системы зарезервирован в качестве загрузочного сектора тома. Там содержится достаточное количество данных, чтобы локальный FSD-драйвер смог идентифицировать том, на котором расположен сектор, как содержащий управляемый драйвером формат и локализовать все метаданные, необходимые для определения места хранения метаданных на томе.

После распознавания тома локальный FSD-драйвер создает объект устройства, представляющий смонтированную файловую систему. Диспетчер ввода-вывода связывает объект устройства для тома, созданный драйвером устройства внешней памяти, с объектом устройства, созданным FSD-драйвером через блок параметров тома (VPB). Это приводит к тому, что диспетчер ввода-вывода направляет запросы на ввод и вывод, адресованные объекту тома, объекту FSD-драйвера. (Подробно VPB рассматривается в главе 9.)

Для повышения производительности локальные FSD-драйверы обычно пользуются диспетчером кэша для кэширования данных файловой системы, в том числе метаданных. (Этот процесс рассматривается в главе 11.) Также эти драйверы объединяются с диспетчером памяти, что позволяет корректно реализовать отображение файлов. К примеру, при любой попытке приложения обрезать файл, FSD-драйвер должен запрашивать диспетчер памяти, чтобы убедиться, что за точкой отсечения файл не отображается ни одним процессом. (Дополнительные сведения о диспетчере памяти можно найти в главе 10.) Windows не разрешает удалять отображеные приложением данные ни путем удаления файла, ни путем отсечения.

Локальные FSD-драйверы поддерживают также операции размонтирования файловой системы, позволяющие операционной системе отсоединить FSD-драйвер от объекта тома. Размонтирувание происходит всякий раз при обращении приложения напрямую к содержимому тома или при смене ассоциированного с томом носителя. При первом после размонтирования обращении приложения к носителю диспетчер ввода-вывода повторно инициирует для этого носителя операцию монтирования тома.

Удаленные FSD-драйверы

Все удаленные FSD-драйверы состоят из двух компонентов: клиента и сервера. Удаленный FSD-драйвер на стороне клиента позволяет приложениям обращаться к удаленным файлам и папкам. Клиентский FSD-драйвер принимает запросы на ввод и вывод от приложений и преобразует их в команды протокола сетевой файловой

системы (такие, как SMB), посылаемые через сеть компоненту на серверной стороне, то есть удаленному FSD-драйверу. Серверный FSD-драйвер принимает поступающие по сетевому соединению команды и выполняет их. При этом он выдает запросы на ввод и вывод локальному FSD-драйверу, управляющему томом, на котором расположен нужный файл или папка.

В Windows входит клиентский удаленный FSD-драйвер, который называется LANMan Redirector (его принято называть просто редиректором), и серверный удаленный FSD-драйвер под названием LANMan Server (%SystemRoot%\System32\Drivers\Srv2.sys). Взаимодействие между сервером и клиентом при доступе к файлам на стороне сервера через редиректор и серверный FSD-драйвер иллюстрирует рис. 12.6. Дополнительную информацию о редиректорах и RDBSS можно найти в главе 7 части I.

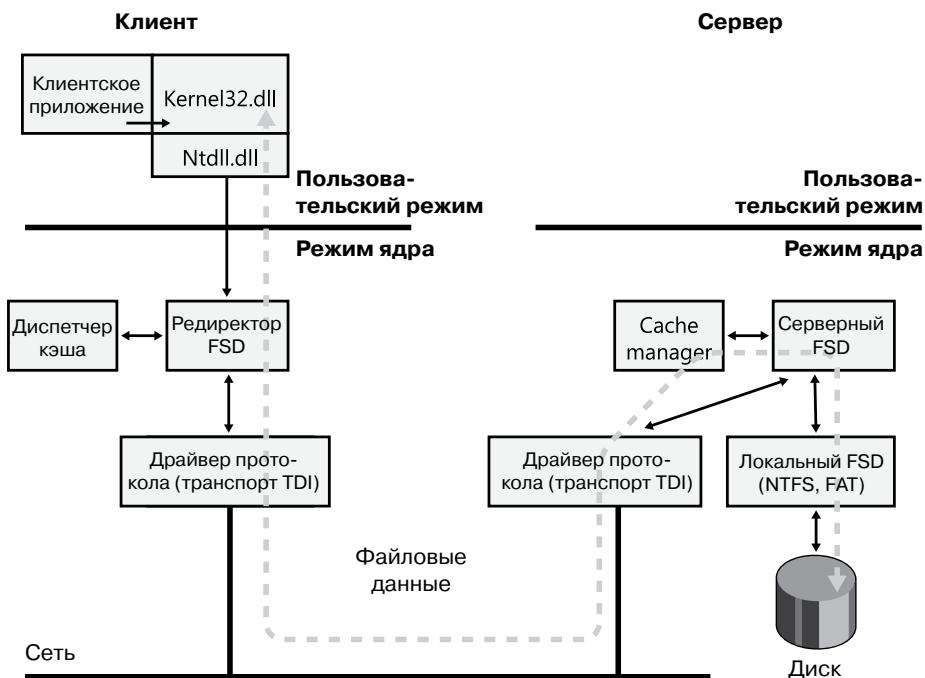


Рис. 12.6. Обмен файлами по протоколу CIFS

Для форматирования сообщений, которыми обмениваются редиректор и сервер, Windows использует протокол Common Internet File System (CIFS). Это версия протокола Server Message Block (SMB) от Microsoft. (Сведения о SMB можно найти на странице [http://msdn.microsoft.com/en-us/library/windows/desktop/aa365233\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/aa365233(v=vs.85).aspx).)

Как и локальные FSD-драйверы, удаленные FSD-драйверы на стороне клиента обычно пользуются службами диспетчера кэша для локального кэширования относящихся к удаленным файлам и папкам данных, поэтому они должны поддерживать распределенный механизм блокировки как на клиентской, так и на серверной стороне. Работающий по протоколу SMB удаленный FSD-драйвер на стороне клиента реализует протокол поддержки когерентности распределенного кэша, называемый протоколом

уступающей блокировки (opportunistic locking, или oplock). Этот протокол гарантирует, что любое приложение при обращении к файлу получит те же данные, что и приложения на других компьютерах сети. Файловые системы сторонних производителей могут использовать протокол уступающей блокировки или реализовывать собственные протоколы. Удаленные FSD-драйверы на стороне клиента участвуют в поддержке ко-герентности клиентских кэшей, но не кэшируют данные локальных FSD-драйверов, так как те кэшируют свои данные самостоятельно.

Блокировка

Крайне важно, чтобы при одновременном доступе к ресурсу всегда включался механизм сериализации, определяющий, кто будет осуществлять запись, и гарантирующий, что запись в каждый конкретный момент времени осуществляется только одним приложением. Без этого механизма возможно повреждение данных. Всеми файловыми серверами, реализующими протокол SMB, используются такие механизмы, как уступающая блокировка или ее аренда. Выбор в данном случае зависит от характеристик сервера и клиента, причем предпочтение отдается второму варианту.

Уступающая блокировка. Функциональность уступающей блокировки (oplock) реализована в библиотеке времени выполнения для файловой системы (функции `FsRtlXxx`) и может использоваться любым драйвером файловой системы. Клиенту удаленного файлового сервера уступающая блокировка помогает динамически определить, какая стратегия кэширования на стороне клиента минимизирует поток данных через сеть. Уступающая блокировка для файла совместного доступа запрашивается драйвером файловой системы или редиректором от лица приложения, пытающегося открыть файл. Предоставление уступающей блокировки позволяет клиенту кэшировать файл вместо передачи по сети каждой операции чтения и записи файловому серверу. К примеру, клиент может открыть файл для монопольного доступа, что даст ему возможность кэшировать результаты чтения и записи в файл и после закрытия файла скопировать на файловый сервер результат обновлений. Если же сервер не предоставит клиенту уступающей блокировки, на него потребуется отправить все результаты чтения и записи.

После предоставления уступающей блокировки клиент может приступить к кэшированию файла. При этом вид доступного кэширования определяется типом предоставленной блокировки. Блокировка не обязана удерживаться до момента, пока клиент не закончит работу с файлом. Она снимается, если серверу требуется выполнить операцию, несовместимую с предоставленными блокировками. Поэтому клиент должен уметь быстро реагировать на изменение условий и динамически менять стратегию кэширования. До появления SMB 2.1 существовало четыре типа уступающей блокировки:

- ❑ **Уровень 1, монопольный доступ.** Эта блокировка позволяет клиенту открывать файл для монопольного доступа и выполнять буферизацию опережающего считывания, а также кэширование операций чтения и записи.
- ❑ **Уровень 2, совместный доступ.** Эта блокировка позволяет одновременно осуществлять чтение из файла, но не запись в него. Клиент может выполнять буферизацию опережающего чтения и кэширование считываемых из файла данных и атрибутов. Запись в файл приводит к снятию блокировки.
- ❑ **Пакетная блокировка, монопольный доступ.** В данном случае название произошло от имени блокировки, используемой при обработке пакетных файлов (`.bat`), которые

открываются и закрываются для обработки каждой строчки. Клиент может держать файл на сервере открытым, даже если приложение его закрыло (возможно, временно). Эта блокировка поддерживает кэширование чтения, записи и дескриптора.

- ❑ **Фильтр, монопольный доступ.** Эта блокировка обеспечивает приложения и файловую систему фильтрами с механизмом, позволяющим отказаться от блокировок при попытке доступа к используемому файлу со стороны других клиентов. В отличие от блокировки уровня 2, файл невозможно открыть, предоставив права на удаление, а второй клиент не получает уведомления о нарушении режима совместного использования файла. Блокировка поддерживает кэширование чтения и записи.

Проще говоря, если несколько клиентских систем кэшировали один и тот же файл, к которому сервер предоставил общий доступ, то до тех пор, пока каждое обращающееся к файлу (со стороны любого клиента или сервера) приложение выполняет только операцию чтения, ему достаточно локального кэша системы. Это значительно снижает нагрузку на сеть, так как исчезает необходимость пересылать каждой системе содержимое файла с сервера. Разумеется, клиентские системы и сервер должны обмениваться данными о блокировке, но для этого высокой пропускной способности не требуется. Однако если хотя бы один из клиентов открывает файл для чтения и записи (или для монопольной записи), ни один из остальных клиентов не сможет воспользоваться своим локальным кэшем, и все результаты операций ввода и вывода должны немедленно пересылаться на сервер, *даже если запись в файл вообще не осуществляется*. (Режимы блокировки зависят от способа открытия файла, а не от отдельных запросов на ввод-вывод.)

Работу уступающей блокировки иллюстрирует рис. 12.7. Первому клиенту, открывающему файл, сервер автоматически предоставляет блокировку уровня 1. Редиректор на стороне клиента помещает данные файла при чтении и записи в кэш файловой системы локальной машины. Второй клиент, открывая тот же файл, также запрашивает блокировку уровня 1. Но сервер в данном случае должен принять меры для согласования данных, предоставляемых обоим клиентам. Если первый клиент произвел запись в файл, как показано на рисунке, сервер отзывает блокировку и никому ее больше не предоставляет. После снятия блокировки первый клиент сбрасывает все кэшированные данные файла обратно на сервер.

Если бы первый клиент не произвел запись в файл, его уступающая блокировка была бы понижена до уровня 2, то есть до блокировки того же типа, который предоставляется второму клиенту. После этого оба клиента получили бы возможность кэшировать операции чтения, но после записи в файл сервер отзывал бы их блокировки, и все последующие операции стали бы некэшируемыми. Один раз снятая блокировка повторно тому же экземпляру открытого файла не предоставляется. Но если клиент закроет и повторно откроет файл, сервер заново решит, блокировку какого уровня следует предоставить. Выбор зависит от того, открыт ли файл другими клиентами и производил ли хоть один из них запись в этот файл.

Аренда уступающей блокировки. Предшествующий SMB 2.1 протокол SMB предполагал безошибочное соединение между клиентом и сервером и не допускал отключений, вызванных временным сбоем в сети, перезагрузкой сервера или переключением на резервный кластер. При отключении от сети клиент терял доступ к ресурсам на удаленных серверах, и все последующие операции ввода-вывода на этих ресурсах не имели шанса на завершение. Аналогичным образом сервер освобождал все ресурсы,

связанные с прерванным пользователем сеансом. В результате происходил сбой в работе приложений и возникал ненужный поток данных через сеть.

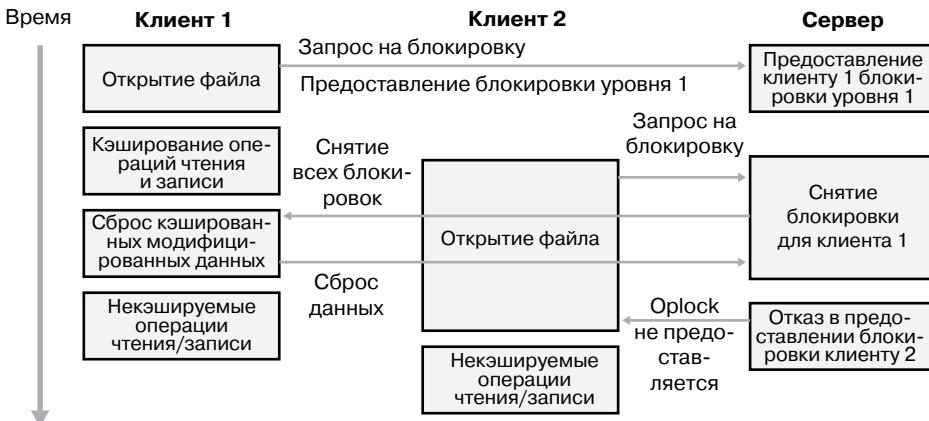


Рис. 12.7. Пример функционирования уступающей блокировки

Вместе с протоколом SMB 2.1 был введен новый тип механизма кэширования клиента, который называется *ареной* (*lease*) и напоминает уступающую блокировку. Аренда имеет аналогичное уступающей блокировке предназначение, но обеспечивает большую гибкость и значительно повышает производительность.

- **Чтение (R), общий доступ.** Допустимы одновременные операции чтения из файла, но запись запрещена. Этот вариант аренды позволяет клиенту выполнять буферизацию опережающего чтения и кэширование чтения.
- **Чтение-дескриптор (RH), общий доступ.** Аналог уступающей блокировки уровня 2, в котором клиенту предоставлена возможность оставлять файл на сервере открытым даже после того, как средство обеспечения клиентского доступа этот файл закрыло. (Диспетчер кэша будет медленно сбрасывать незаписанные данные и удалять неизменявшиеся страницы кэша исходя из доступности памяти.) Это превосходит возможности уступающей блокировки уровня 2, так как между открытиями и закрытиями дескриптора файла отказ от аренды не происходит. (В данном случае мы имеем семантику, аналогичную пакетной уступающей блокировке.) Этот тип аренды особенно полезен для неоднократно открывавшихся и закрывающихся файлов, так как кэш не становится недействительным при закрытии файла и не заполняется повторно при его открытии, что значительно повышает производительность приложений, связанных с интенсивным вводом-выводом данных.
- **Чтение-запись (RW), монопольный доступ.** Этот тип аренды дает клиенту возможность открывать файлы для монопольного доступа. Блокировка позволяет клиенту выполнять буферизацию опережающего чтения и кэширование чтения и записи.
- **Чтение-запись-дескриптор (RWH), монопольный доступ.** Эта блокировка позволяет клиенту открывать файлы для монопольного доступа. Такой тип аренды поддерживает кэширование чтения, записи и дескрипторов.

Еще одним преимуществом аренды по сравнению с уступающей блокировкой является возможность кэширования файла даже при открытых со стороны клиента множественных дескрипторах файла. (Такое поведение встречается у многих приложений.) Данная возможность осуществляется через ключ аренды (реализованный с помощью GUID), который создается клиентом и связан с блоком управления файлом (File Control Block, FCB) кэшированного файла.

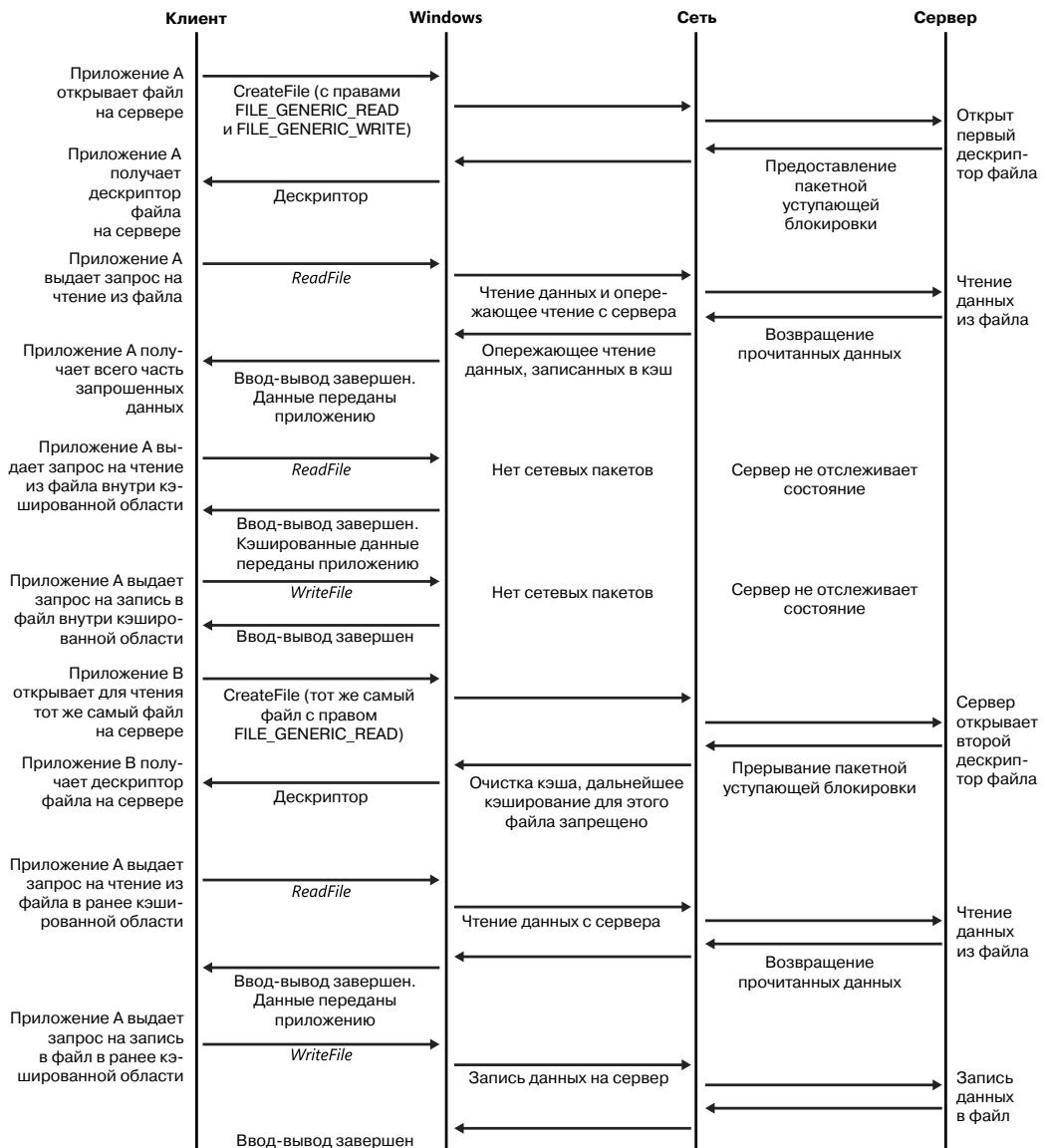


Рис. 12.8. Уступающая блокировка с набором дескрипторов от одного клиента

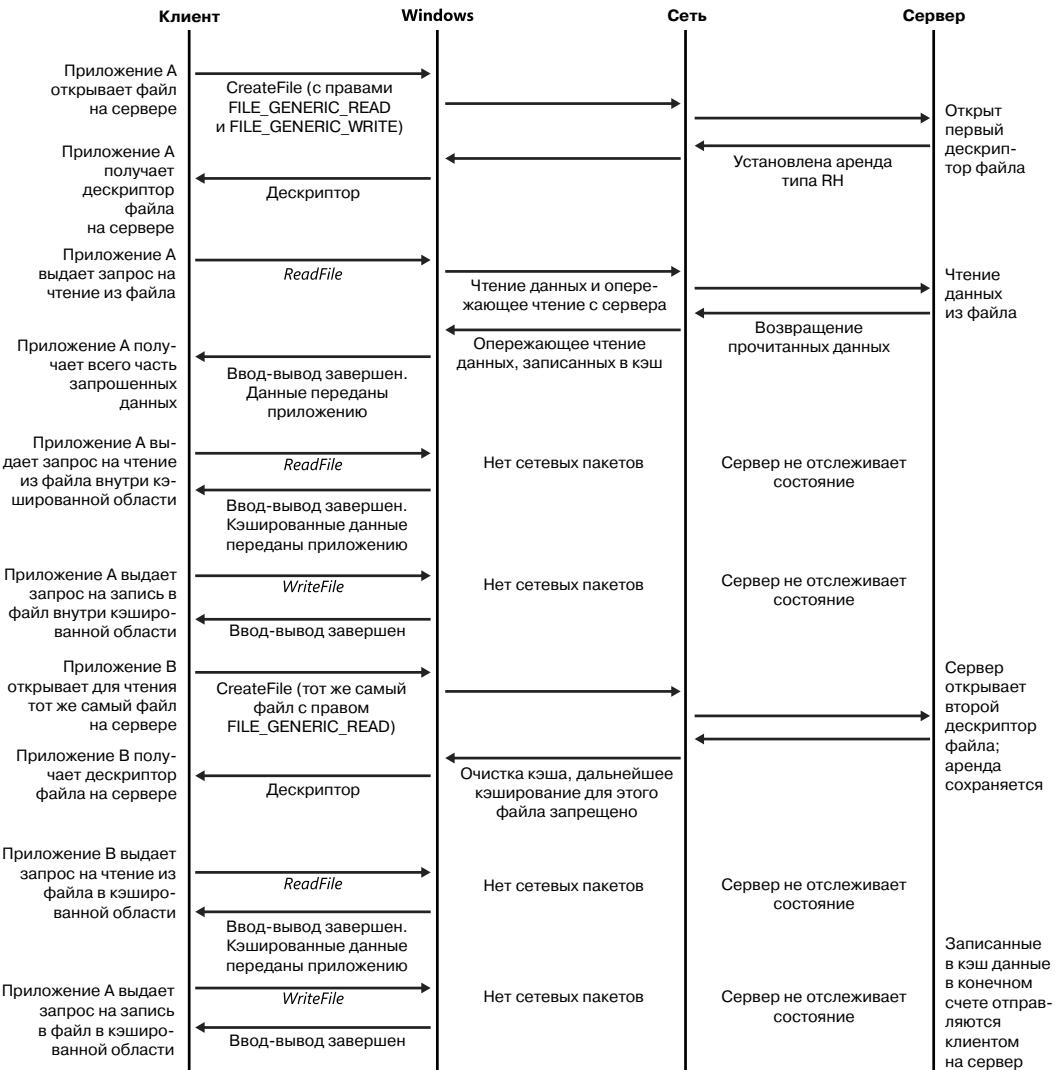


Рис. 12.9. Аренда с набором дескрипторов от одного клиента

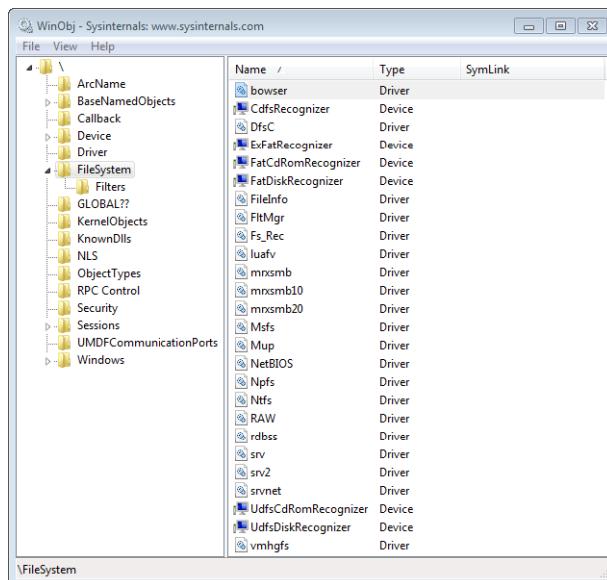
Именно ключ позволяет всем дескрипторам одного файла получить один и тот же вариант аренды, который обеспечивается кэшированием не дескриптора, а файла. До появления аренды уступающая блокировка снималась при любом открытии нового дескриптора файла, даже если операцию выполнял тот же самый клиент. Поведение уступающей блокировки показано на рис. 12.8, а рис. 12.9 демонстрирует поведение аренды.

До появления протокола SMB 2.1 уступающие блокировки можно было только предоставлять и снимать, аренда же допускает еще и возможность преобразования.

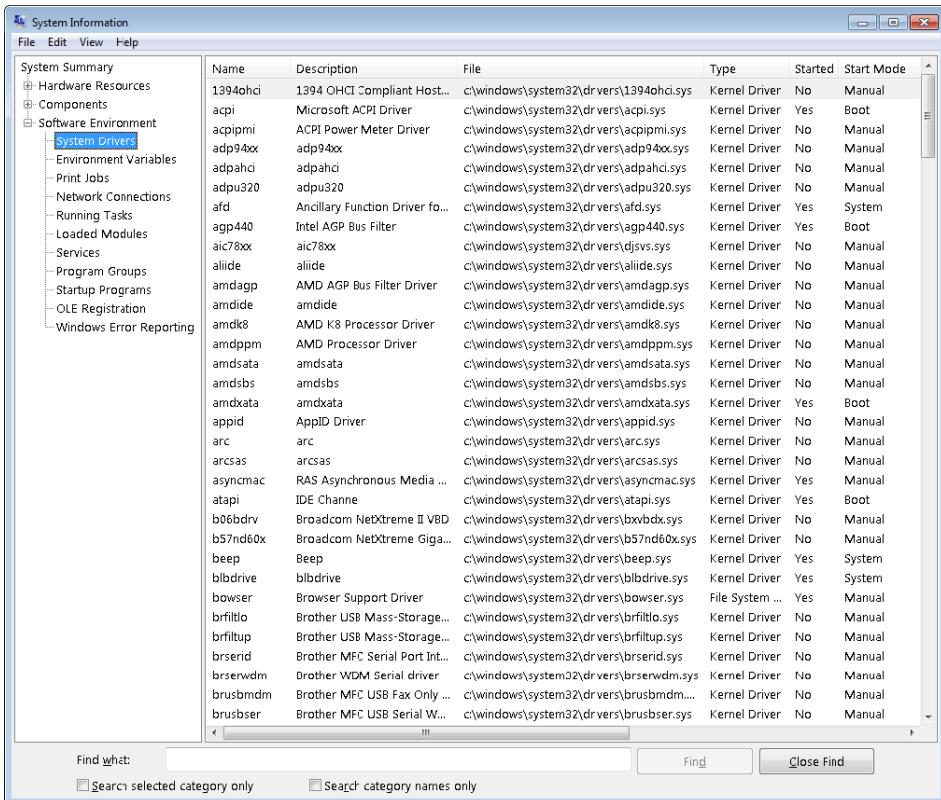
К примеру, аренду типа R можно преобразовать в аренду типа RW, что значительно снижает нагрузку на сеть, так как исчезает необходимость объявлять недействительным и повторно формировать кэш конкретного файла, как это происходит при снятии уступающей блокировки (уровня 2), за которой следует запрос и предоставление блокировки уровня 1.

ЭКСПЕРИМЕНТ: ПРОСМОТР СПИСКА ЗАРЕГИСТРИРОВАННЫХ ФАЙЛОВЫХ СИСТЕМ

Загружая в память драйвер устройства, диспетчер ввода-вывода обычно присваивает имя объекту устройства, созданному для представления данного драйвера. Этот объект помещается в папку \Driver диспетчера объектов. Объекты устройств для любого драйвера, загружаемого диспетчером ввода-вывода, атрибут которых имеет значение SERVICE_FILE_SYSTEM_DRIVER (2), помещаются диспетчером в папку \FileSystem. Соответственно, такой инструмент, как WinObj (производства Sysinternals), позволяет увидеть зарегистрированные файловые системы, как показано на следующем снимке экрана (обратите внимание, что некоторые драйверы файловых систем помещают в папку \FileSystem еще и объекты устройств).



Другим средством просмотра зарегистрированных файловых систем является программа System Information (Сведения о системе). Введите в диалоговое окно Run (Запуск) меню Start (Пуск) команду Msinfo32 и выберите вариант System Drivers (Системные драйверы) в узле Software Environment (Программная среда). Отсортируйте список драйверов щелчком на заголовке столбца Type (Тип), и драйверы с атрибутом SERVICE_FILE_SYSTEM_DRIVER окажутся в одной группе.



The screenshot shows the Windows System Information window with the 'System Drivers' category selected. The table lists various drivers with their names, descriptions, file paths, types, start times, and start modes.

Name	Description	File	Type	Started	Start Mode
1394ohci	1394 OHCI Compliant Host...	c:\windows\system32\drivers\1394ohci.sys	Kernel Driver	No	Manual
acpi	Microsoft ACPI Driver	c:\windows\system32\drivers\acpi.sys	Kernel Driver	Yes	Boot
acpipmi	ACPI Power Meter Driver	c:\windows\system32\drivers\acpipmi.sys	Kernel Driver	No	Manual
adp94xx	adp94xx	c:\windows\system32\drivers\adp94xx.sys	Kernel Driver	No	Manual
adphaci	adphaci	c:\windows\system32\drivers\adphaci.sys	Kernel Driver	No	Manual
adpu320	adpu320	c:\windows\system32\drivers\adpu320.sys	Kernel Driver	No	Manual
afd	Ancillary Function Driver fo...	c:\windows\system32\drivers\afd.sys	Kernel Driver	Yes	System
agp440	Intel AGP Bus Filter	c:\windows\system32\drivers\agp440.sys	Kernel Driver	Yes	Boot
aic78xx	aic78xx	c:\windows\system32\drivers\djvs.sys	Kernel Driver	No	Manual
alilide	alilide	c:\windows\system32\drivers\alilide.sys	Kernel Driver	No	Manual
amdgap	AMD AGP Bus Filter Driver	c:\windows\system32\drivers\amdgap.sys	Kernel Driver	No	Manual
amdiide	amdiide	c:\windows\system32\drivers\amdiide.sys	Kernel Driver	No	Manual
amdk8	AMD K8 Processor Driver	c:\windows\system32\drivers\amdk8.sys	Kernel Driver	No	Manual
amdppm	AMD Processor Driver	c:\windows\system32\drivers\amdppm.sys	Kernel Driver	No	Manual
amdsata	amdsata	c:\windows\system32\drivers\amdsata.sys	Kernel Driver	No	Manual
amdsbs	amdsbs	c:\windows\system32\drivers\amdsbs.sys	Kernel Driver	No	Manual
amdxata	amdxata	c:\windows\system32\drivers\amdxata.sys	Kernel Driver	Yes	Boot
appid	AppID Driver	c:\windows\system32\drivers\appid.sys	Kernel Driver	No	Manual
arc	arc	c:\windows\system32\drivers\arc.sys	Kernel Driver	No	Manual
arcsas	arcsas	c:\windows\system32\drivers\arcsas.sys	Kernel Driver	No	Manual
asyncmac	RAS Asynchronous Media ...	c:\windows\system32\drivers\asyncmac.sys	Kernel Driver	Yes	Manual
atapi	IDE Channe...	c:\windows\system32\drivers\atapi.sys	Kernel Driver	Yes	Boot
b06bdrv	Broadcom NetXtreme II VBD	c:\windows\system32\drivers\b06bdrv.sys	Kernel Driver	No	Manual
b57nd60x	Broadcom NetXtreme Giga...	c:\windows\system32\drivers\b57nd60x.sys	Kernel Driver	No	Manual
beep	Beep	c:\windows\system32\drivers\beep.sys	Kernel Driver	Yes	System
blbdrive	blbdrive	c:\windows\system32\drivers\blbdrive.sys	Kernel Driver	Yes	System
bowser	Browser Support Driver	c:\windows\system32\drivers\bowser.sys	File System ...	Yes	Manual
brfilto	Brother USB Mass-Storage...	c:\windows\system32\drivers\brfilto.sys	Kernel Driver	No	Manual
brfilup	Brother USB Mass-Storage...	c:\windows\system32\drivers\brfilup.sys	Kernel Driver	No	Manual
brserid	Brother MFC Serial Port Int...	c:\windows\system32\drivers\brserid.sys	Kernel Driver	No	Manual
brserwdm	Brother WDM Serial driver	c:\windows\system32\drivers\brserwdm.sys	Kernel Driver	No	Manual
brusbrndm	Brother MFC USB Fax Only ...	c:\windows\system32\drivers\brusbrndm...	Kernel Driver	No	Manual
brusbsr	Brother MFC USB Serial W...	c:\windows\system32\drivers\brusbsr.sys	Kernel Driver	No	Manual

Следует заметить, что простая регистрация драйвера как драйвера файловой системы не означает, что он становится локальным или удаленным FSD-драйвером. К примеру, Npfs (Named Pipe File System) является драйвером сетевого прикладного программного интерфейса, который поддерживает именованные каналы, но реализует закрытое пространство имен, и поэтому в определенной степени аналогичен драйверу файловой системы. Эксперимент, в котором исследуется пространство имен Npfs, был приведен в главе 7 части I.

Работа файловой системы

Система и приложения могут обращаться к файлам двумя способами: напрямую, через функции ввода-вывода (такие, как `ReadFile` и `WriteFile`), и косвенно, путем чтения или записи части своего адресного пространства, представляющей собой раздел отображенного файла. (Об отображеных на память файлах рассказывается в главе 10.) Представленная на рис. 12.10 упрощенная схема иллюстрирует компоненты, участвующие в работе файловой системы, и способы их взаимодействия. Как видите, существует несколько способов вызова FSD-драйвера:

- из пользовательского или системного программного потока, выполняющего явный ввод или вывод;
- из подсистем записи модифицированных и отображенных страниц, принадлежащих диспетчеру памяти;
- неявно из подсистемы отложенной записи, принадлежащей диспетчеру кэша;
- неявно из потока данных опережающего чтения, принадлежащего диспетчеру кэша;
- из обработчика ошибок страниц, который принадлежит диспетчеру памяти.

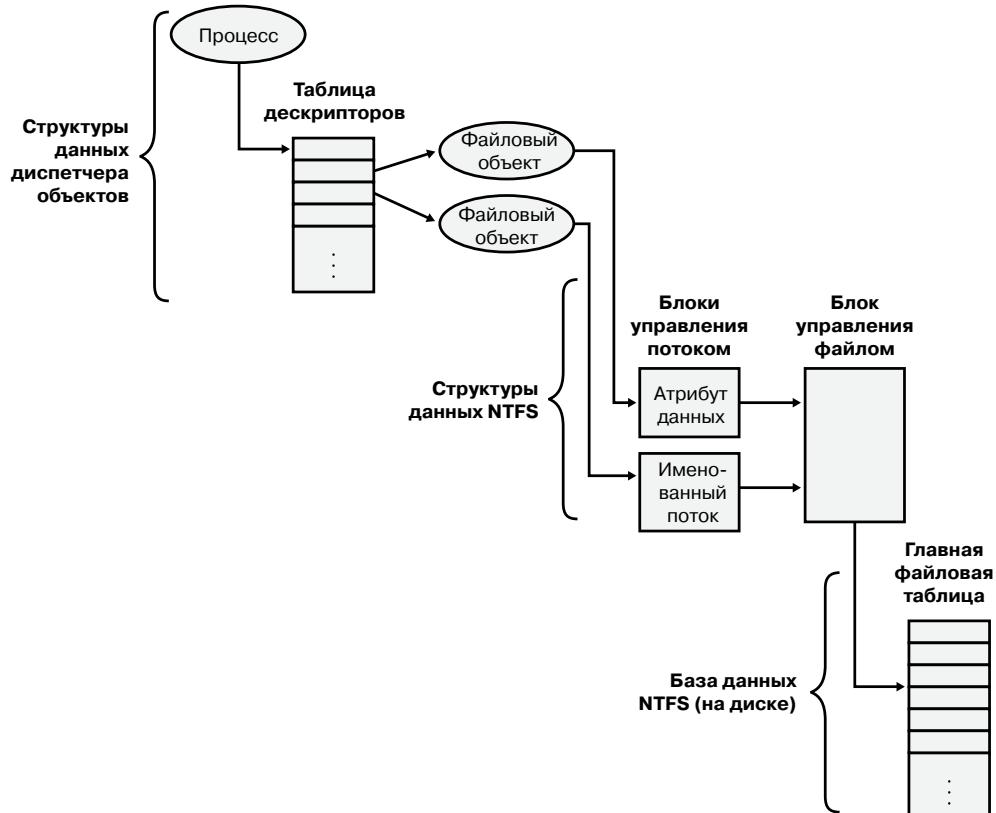


Рис. 12.10. Компоненты, участвующие в операциях ввода-вывода файловой системы

Далее мы выясним, при каких обстоятельствах возникает каждый из перечисленных сценариев и каким образом FSD реагирует в каждом из случаев. Вы увидите, в какой степени FSD-драйвер зависит от диспетчеров памяти и кэша.

Явный ввод-вывод

Наиболее очевидным способом доступа приложений к файлам является вызов Windows-функций ввода-вывода, таких как `CreateFile`, `ReadFile` и `WriteFile`. Открытие

файла происходит при помощи функции `CreateFile`, после чего приложение осуществляет чтение, запись или удаление, передавая возвращаемый функцией `CreateFile` дескриптор файла другим Windows-функциям. Реализованная в динамической библиотеке Kernel32.dll функция `CreateFile` вызывает встроенную функцию `NtCreateFile`, формируя для пути, который передает ей приложение, полное имя относительно корневого каталога (обрабатывая в имени пути символы «.» и «..») и добавляя к этому имени префикс \?? (например, \??\C:\Daryl\Todo.txt).

Для открытия файла системная служба `NtCreateFile` использует функцию `ObOpenObjectByName`, которая выполняет анализ имени начиная с корневого каталога диспетчера объектов и первого компонента имени пути (??). Подробно разрешение имен диспетчером объектов описывалось в главе 3 части I, а здесь мы поясним, как происходит поиск букв диска для томов.

Первым делом диспетчер объектов преобразует префикс \?? в папку пространства имен для текущего сеанса. На эту папку ссылается поле `DosDevicesDirectory`, принадлежащее структуре карты устройства в ссылках на объект процесса (который получен от первого процесса в сеансе авторизации путем применения поля ссылок в маркере сеанса). В такой папке обычно хранятся только имена томов для общих сетевых ресурсов и букв дисков, отображенных программой Subst.exe, поэтому диспетчер объектов, не найдя имени (в данном примере – С) в папке сеанса, переходит к поиску в папке, на которую ссылается поле `GlobalDosDevicesDirectory` ассоциированной с папкой сеанса карты устройств. Поле `GlobalDosDevicesDirectory` всегда указывает на папку \Global??, в которой Windows хранит буквы дисков для локальных томов. (Подробно эта тема рассматривалась в разделе «Пространство имен сеанса» главы 3 части I.)

Символическая ссылка для присвоенной тому буквы диска указывает на объект устройства для тома в папке \Device, поэтому диспетчер объектов, распознав этот объект, передает остаток строки с именем функции `IoParseDevice`, зарегистрированной диспетчером ввода-вывода для объектов устройств. (Символическая ссылка в томах на динамических дисках указывает на промежуточную символическую ссылку, которая, в свою очередь, указывает на объект устройства для тома.) Рисунок 12.11 демонстрирует доступ к объектам устройств для томов через пространство имен диспетчера объектов. В данном случае символическая ссылка \GLOBAL??\C: указывает на объект устройства \Device\HarddiskVolume1.

После блокировки контекста безопасности вызывающего потока и получения данных о безопасности из его маркера функция `IoParseDevice` генерирует пакет запроса на ввод-вывод (IRP) типа `IRP_MJ_CREATE`, создает файловый объект, в котором сохраняется имя открытого файла, и следует в VPB объекта устройства для тома, чтобы найти объект устройства для смонтированной на нем файловой системы. Затем с помощью функции `IoCallDriver` она передает IRP драйверу файловой системы, которому принадлежит объект устройства для данной файловой системы.

Когда FSD-драйвер получает IRP-пакет типа `IRP_MJ_CREATE`, он ищет указанный файл, выполняет проверку безопасности, и если файл существует, а пользователь имеет право на доступ к нему запрошенным способом, возвращает код успешного завершения. Диспетчер объектов создает дескриптор для файлового объекта в таблице дескрипторов процесса, и этот дескриптор передается назад по цепочке вызовов, в конечном счете достигая приложения в виде параметра, возвращаемого функцией `CreateFile`.

Если же файловой системе создать файл не удается, диспетчер ввода-вывода удаляет созданный для файла файловый объект.

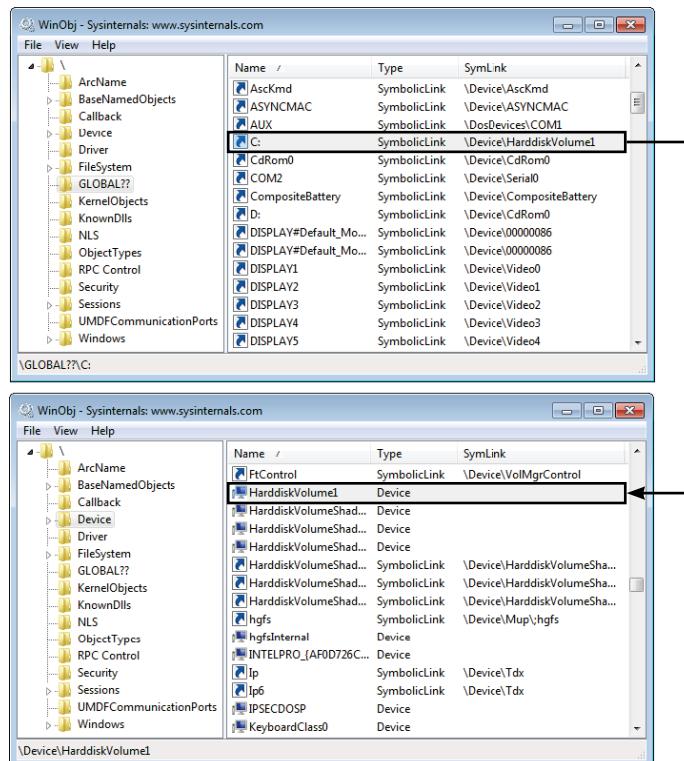


Рис. 12.11. Разрешение букв дисков

Мы не будем детально рассматривать, каким образом FSD-драйвер находит открытый на томе файл, но упомянем, что операция вызова функции `ReadFile` происходит во многих вариантах взаимодействия FSD с диспетчером кэша и драйвером запоминающего устройства. Обе функции, `ReadFile` и `CreateFile`, относятся к системным вызовам, которые отображаются на функции диспетчера ввода-вывода, но системной службе `NtReadFile` в этом случае не приходится заниматься поиском имени — она вызывает диспетчер объектов для преобразования переданного из функции `ReadFile` дескриптора в указатель на файловый объект. Если дескриптор открытого файла указывает на наличие у вызывающего потока прав на чтение файла, служба `NtReadFile` создает IRP-пакет типа `IRP_MJ_READ` и отправляет его драйверу файловой системы тома, на котором находится файл. Затем она получает объект устройства для FSD-драйвера, хранящийся в файловом объекте, и вызывает функцию `IoCallDriver`, а диспетчер ввода-вывода находит FSD-драйвер в объекте устройства и передает ему IRP-пакет.

Если считываемый файл допускает кэширование (то есть при открытии файла в функцию `CreateFile` не передается флаг `FILE_FLAG_NO_BUFFERING`), FSD-драйвер

проверяет, инициировано ли кэширование файлового объекта на данный момент. Если поле `PrivateCacheMap` в файловом объекте указывает на структуру закрытой карты кэша (она описывается в главе 11), значит, кэширование уже происходит. Если же оно пока не было инициировано, поле `PrivateCacheMap` будет иметь значение `null`. Кэширование файлового объекта инициируется FSD при первой операции чтения или записи для объекта. Для этого FSD вызывает функцию `CcInitializeCacheMap` диспетчера кэша, которая создает закрытую карту кэша, а если на тот же самый файл ссылается еще один файловый объект с неиницированным кэшированием, создается еще и общая карта кэша, а также объект раздела.

Убедившись, что кэширование файла включено, FSD копирует данные запрошенного файла из виртуальной памяти диспетчера кэша в буфер, который программный поток передал функции `ReadFile`. Файловая система выполняет копирование файла внутри блока `try/except`, что позволяет перехватывать все ошибки, которые могут появиться, если приложение неверно укажет буфер. Копирование файловая система осуществляет с помощью функции `CcCopyRead` диспетчера кэша, которая принимает в качестве параметров файловый объект, смещение внутри файла и длину данных.

В процессе выполнения функции `CcCopyRead` диспетчер кэша загружает указатель на общую карту кэша, хранящуюся в файловом объекте. В главе 11 рассказывается, что эта карта хранит указатели на блоки управления виртуальными адресами (Virtual Address Control Block, VACB), и один VACB-элемент соответствует блоку файла размером 256 Кбайт. Если VACB-указатель для считываемой части файла имеет значение `null`, функция `CcCopyRead` создает VACB, резервируя в виртуальном адресном пространстве диспетчера кэша представление размером 256 Кбайт, на которое отображает указанную часть файла (используя функцию `MmMapViewInSystemCache`). Затем функция `CcCopyRead` просто копирует данные файла из отображенного представления в переданный ей буфер (тот самый буфер, который изначально был передан функции `ReadFile`). Если файловые данные отсутствуют в физической памяти, операция копирования генерирует ошибки страниц, обслуживаемые функцией `MmAccessFault`.

При возникновении ошибки страницы функция `MmAccessFault` изучает ставший причиной ошибки виртуальный адрес и локализует дескриптор виртуального адреса (Virtual Address Descriptor, VAD) в VAD-дереве вызвавшего ошибку процесса. (Подробно VAD-деревья рассматриваются в главе 10.) В данном сценарии VAD описывает представление считываемого файла, проецируемое диспетчером кэша, поэтому для обработки ошибки страницы с действительным виртуальным адресом функция `MmAccessFault` вызывает функцию `MiDispatchFault`, которая сначала находит область управления (на нее указывает VAD) и через эту область обнаруживает файловый объект, соответствующий открытому файлу. (Если файл открывался несколько раз, возможно наличие списка файловых объектов, связанных указателями в закрытых картах кэша.)

Локализовав файловый объект, функция `MiDispatchFault` вызывает функцию `IoPageRead` диспетчера ввода-вывода, чтобы создать IRP-пакет (типа `IRP_MJ_READ`) и отправить его FSD-драйверу, владеющему объектом устройства, на который указывает файловый объект. Соответственно, происходит повторный вход в файловую систему для чтения запрошенных через функцию `CcCopyRead` данных, но на этот раз IRP-пакет снабжен флагами, указывающими на некэшируемый ввод-вывод,

связанный с подкачкой. Эти флаги информируют FSD-драйвер, что он должен извлечь данные непосредственно с диска. Драйвер так и поступает, определяя, какие кластеры диска содержат запрошенные данные (конкретный механизм зависит от того, в какой файловой системе это происходит), и передавая IRP-пакеты диспетчеру томов, владеющему объектом устройства для тома, на котором находится файл. Поле блока параметров тома (VPB) в объекте устройства для FSD указывает на объект устройства для тома.

Диспетчер памяти ждет, пока FSD закончит чтение IRP-пакета и вернет управление диспетчеру кэша, продолжающему прерванную ошибкой страницы операцию копирования. После завершения работы функции `CcCopyRead` FSD возвращает управление вызвавшему службу `NtReadFile` программному потоку. На этот момент запрошенные данные из файла уже скопированы — с помощью диспетчера кэша и диспетчера памяти — в буфер потока.

Функция `WriteFile` работает аналогичным образом, за исключением того, что системная служба `NtWriteFile` генерирует IRP-пакеты типа `IRP_MJ_WRITE`, а FSD вызывает функцию `CcCopyWrite`, а не `CcCopyRead`. Эта функция, как и `CcCopyRead`, проверяет, отображены ли на кэш части записываемого файла, и копирует в кэш содержимое переданного в функцию `WriteFile` буфера.

В случае, когда данные файла уже кэшированы (в рабочем наборе системы), существует несколько вариантов вышеописанного сценария. Если данные уже хранятся в кэше, функция `CcCopyRead` не вызывает ошибок страницы. Кроме того, в определенных обстоятельствах службы `NtReadFile` и `NtWriteFile`, вместо того чтобы немедленно создать IRP-пакет и послать его FSD-драйверу, вызывают FSD в точку входа для быстрого ввода-вывода. Это происходит, к примеру, когдачитываются первые 4 Гбайт файла, файл не имеет блокировок, а считываемая или записываемая часть файла не выходит за пределы его текущей длины.

Для большинства FSD-драйверов точки быстрого ввода-вывода вызывают функции `CcFastCopyRead` и `CcFastCopyWrite` диспетчера кэша. Эти варианты стандартных процедур копирования требуют, чтобы перед копированием файловые данные были отображены на кэш файловой системы. Если это условие не выполняется, функции `CcFastCopyRead` и `CcFastCopyWrite` сообщают о невозможности быстрого ввода-вывода. В этом случае службы `NtReadFile` и `NtWriteFile` возвращаются к созданию IRP. (Более подробно быстрый ввод-вывод описывается в одноименном разделе главы 11.)

Подсистема записи модифицированных и отображенных страниц

Для сброса модифицированных страниц периодически (и при нехватке доступной памяти) пробуждаются программные потоки принадлежащих диспетчеру памяти подсистем записи модифицированных и отображенных страниц. Они вызывают функцию `IoAsynchronousPageWrite` для создания IRP-пакетов типа `IRP_MJ_WRITE` и записи страниц либо в файл подкачки, либо в модифицированный после отображения файл. Подобно IRP-пакетам, которые создает функция `MidDispatchFault`, эти IRP-пакеты помечаются флагами, указывающими на некэшируемый и связанный с подкачкой ввод-вывод. Соответственно, для записи содержимого памяти на диск FSD-драйвер обходит кэш файловой системы и передает IRP-пакеты непосредственно драйверу устройств внешней памяти.

Подсистема отложенной записи

Программный поток принадлежащей диспетчеру кэша подсистемы отложенной записи тоже участвует в записи модифицированных страниц, поскольку периодически сбрасывает на диск измененные представления разделов файлов, отображенных на кэш. Операция сброса, которую диспетчер кэша выполняет, вызывая функцию `MmFlushSection`, заставляет диспетчер памяти записать на диск все модифицированные страницы в сбрасываемой части раздела. Подобно подсистемам записи модифицированных и отображенных страниц, функция `MmFlushSection` посыпает данные FSD-драйверу через функцию `IoSynchronousPageWrite`.

Программный поток опережающего чтения

Реализация ссылок на код и данные в кэше строится по двум принципам: временной и пространственной локальности. Принцип временной локальности состоит в том, что если произошло обращение по некоторому адресу, то следующее обращение с большой вероятностью произойдет в ближайшее время. Принцип пространственной локальности гласит, что если произошло обращение по некоторому адресу, то с высокой степенью вероятности в ближайшее время произойдет обращение к соседним адресам. Поэтому кэш обычно хорошо ускоряет доступ к областям памяти, к которым недавно происходило обращение, но не может ускорить доступ к области памяти, обращений к которой пока не было (он не способен к прогнозированию). Попытка заполнить кэш данными, которые, скорее всего, скоро будут использоваться, реализуется в диспетчере кэша при помощи механизмов опережающего чтения и супервыборки.

Диспетчер кэша содержит программный поток, отвечающий за чтение данных из файлов до того, как их явным образом запросит приложение, драйвер или системный программный поток. Чтобы определить объем подлежащих чтению данных, программный поток опережающего чтения использует хронологию операций чтения, хранящуюся в закрытой карте кэша файлового объекта. Выполняя опережающее чтение, программный поток просто отображает на кэш ту часть файла, которую он хочет считать (создавая при необходимости VACB-блоки), и обращается к отображенными данным. Если при попытках обращения возникают ошибки страниц, активируется обработчик страниц, подгружающий нужные страницы в системный рабочий набор.

К сожалению, опережающее чтение работает только для открытых файлов. Поэтому в Windows была добавлена служба супервыборки (superfetch), позволяющая заранее добавлять в кэш еще даже не открывавшиеся файлы. В частности, диспетчер памяти посыпает службе супервыборки (%SystemRoot%\System32\Sysmain.dll) сведения об использовании страницы, а мини-фильтр файловой системы предоставляет данные о разрешении имен файлов. Служба супервыборки пытается определить схему использования файлов, например платежная ведомость просматривается каждую пятницу в 12:00, а Outlook запускается каждое утро в 8:00. На основе этой информации создается база данных, постоянно обновляемая по ходу работы пользователя, и к ней добавляются таймеры. В момент наиболее вероятного использования программы таймер срабатывает и пробуждает службу супервыборки, которая заставляет диспетчер памяти прочитать файл, дав ему низкий приоритет (задействуя низкоприоритетный дисковый ввод-вывод). В этом случае при открытии файла пользователю не приходится

ждать, пока данные считаются с диска, так как они уже находятся в памяти. Если же файл не открывается, занимаемая им память возвращается системе.

Обработчик ошибок страниц

Об использовании обработчика ошибок страниц уже рассказывалось в контексте явного файлового ввода-вывода и опережающего чтения диспетчера кэша. Но он также активируется всякий раз, когда приложение обращается к виртуальной памяти, являющейся представлением отображенного файла, и встречает страницы, которые представляют фрагменты отсутствующего в памяти файла. Обработчик `MmAccessFault` диспетчера памяти предпринимает те же действия, что и в ситуации, когда диспетчер кэша генерирует ошибки страниц в результате вызова функций `CcCopyRead` или `CcCopyWrite`, отправляя через функцию `ToPageRead` IRP-пакеты файловой системе, в которой хранится нужный файл.

Фильтрующие драйверы файловой системы

Фильтрующий драйвер, занимающий в иерархии более высокий уровень, чем драйвер файловой системы, называется *фильтрующим драйвером файловой системы* (file system filter driver). (Фильтрующие драйверы подробно рассматриваются в главе 8.) Его способность видеть все запросы к файловой системе и при необходимости модифицировать или выполнять их, делает возможным появление таких приложений, как службы репликации удаленных файлов, шифрования файлов, эффективного резервного копирования и лицензирования. В любой коммерческий антивирусный сканер, проверяющий файлы «на лету», входит фильтрующий драйвер файловой системы, перехватывающий IRP-пакеты, которые передают команды `IRP_MJ_CREATE`, выдаваемые при каждом открытии файла приложением. Прежде чем передать такой пакет драйверу файловой системы, которому адресована данная команда, сканер проверяет файл на наличие вирусов. «Чистый» файл передается дальше по цепочке, если же он окажется зараженным, сканер обратится к связанному с ним служебному процессу для удаления или лечения файла. Если вылечить файл невозможно, фильтрующий драйвер отклоняет IRP-пакет (обычно с ошибкой запрещения доступа), чтобы вирус не смог активироваться.

Программа Process Monitor

Неоднократно встречавшаяся вам в этой книге служебная программа Process Monitor (Procmon) от Sysinternals, следящая за работой операционной системы, является примером пассивного фильтрующего драйвера, который не влияет на IRP-пакеты, курсирующие между приложениями и драйверами файловой системы. В Windows включен диспетчер фильтров (`%SystemRoot%\System32\Fltmgmgr.sys`) как часть модели порт/мини-порт для фильтрующих драйверов файловой системы. Этот диспетчер значительно упрощает разработку фильтрующих драйверов, взаимодействуя в Windows с фильтрующим драйвером мини-порта подсистемы ввода-вывода и предоставляя службы для запросов имен файлов, присоединения томов и взаимодействия с другими фильтрами. Наблюдение за файловой системой в служебной программе Process Monitor реализовано как драйвер мини-фильтра.

Программа Process Monitor извлекает фильтрующий драйвер устройства из образа исполняемой программы (хранящегося как ресурс внутри файла `Procmon.exe`) при

первом ее запуске после загрузки системы, устанавливает драйвер в память, стирая после этого образ файла с диска. Через графический интерфейс программы можно заставить драйвер следить за файловой системой на локальных томах, которым были назначены буквы, в общих сетевых ресурсах, именованных каналах и в почтовых каталогах. Получив такую команду, драйвер регистрирует обратный вызов фильтра через диспетчер фильтра, связанный с объектом устройства для смонтированной на томе файловой системы. После присоединения диспетчера ввода-вывода направляет предназначенные исходному объекту устройства IRP-пакеты драйверу, владельцу присоединенным устройством. В данном случае это диспетчера фильтра, который отправляет событие зарегистрированным драйверам мини-фильтра, то есть приложению Process Monitor.

Перехватив IRP-пакет, драйвер Process Monitor записывает такую информацию, как имя целевого файла и прочие связанные с командой параметры (например, значения длины чтения и записи, а также величины смещений) в невыгружаемый буфер ядра. Каждые 500 миллисекунд GUI программы Process Monitor отправляет пакет объекту устройства для своего интерфейса, который запрашивает копию буфера с данными о последних действиях и затем выводит их в окне. Применение программы Process Monitor описывается далее в разделе «Решение проблем файловой системы».

ЭКСПЕРИМЕНТ: ПРОСМОТР ФИЛЬТРУЮЩЕГО ДРАЙВЕРА ПРОГРАММЫ PROCESS MONITOR

Чтобы посмотреть, какой из фильтрующих драйверов файловой системы загружен, через интерпретатор командной строки в режиме администратора запустите диспетчер фильтров (%SystemRoot%\System32\Fltmc.exe). Откройте приложение Process Monitor (ProcMon.exe) и снова запустите Fltmc. Вы увидите, что загружен фильтрующий драйвер программы Process Monitor (PROCMON20) и в столбце Instances ему соответствует не-нулевое значение. Теперь закройте Process Monitor и снова запустите Fltmc. На этот раз вы увидите, что фильтрующий драйвер приложения Process Monitor все еще загружен, но число его экземпляров стало равным нулю.

```
Administrator: Admin Command Prompt
Microsoft Windows [Version 6.1.7601]
Copyright <c> 2009 Microsoft Corporation. All rights reserved.

C:\Windows\system32>fltmc

Filter Name           Num Instances   Altitude   Frame
luafv                1             135000     0
FileInfo              5             45000      0

C:\Windows\system32>procmon

C:\Windows\system32>fltmc

Filter Name           Num Instances   Altitude   Frame
PROCMON20            0             385200     0
luafv                1             135000     0
FileInfo              5             45000      0

C:\Windows\system32>
```

Решение проблем файловой системы

В главе 4 части I описывалось, как система и приложения сохраняют данные в реестре. Проблемы, связанные с реестром, например неверная конфигурация механизмов безопасности или недостаток параметров и разделов, становятся причиной различных сбоев в работе системы и приложений. Кроме того, система и приложения используют для хранения данных файлы и обращаются к образам исполняемых файлов и DLL. Поэтому второй распространенной причиной сбоев в работе системы и приложений является неверная конфигурация механизмов безопасности в NTFS и отсутствие некоторых файлов и папок. Ведь система и приложения зачастую функционируют исходя из возможности беспрепятственного доступа к нужным файлам и при ее отсутствии начинают вести себя непредсказуемо.

Программа Process Monitor отражает все операции с файлами по мере их выполнения, что превращает ее в идеальный инструмент для анализа сбоев в работе операционной системы и приложений из-за проблем файловой системы. Для первого запуска этой программы учетная запись должна обладать привилегиями загрузки драйвера и отладки. После загрузки драйвер остается резидентным в памяти, поэтому для последующих запусков Process Monitor достаточно привилегии отладки.

Базовый и расширенный режимы программы Process Monitor

После запуска программа Process Monitor стартует в базовом режиме, в котором выводятся наиболее полезные для решения возникающих с файловой системой проблем операции. Часть происходящих в файловой системе операций в этом режиме не показывается, в том числе:

- обращение к NTFS-файлам метаданных;
- ввод-вывод в файл подкачки;
- ввод-вывод, генерируемый процессом System;
- ввод-вывод, генерируемый процессом Process Monitor.

В базовом режиме Process Monitor также сообщает об операциях файлового ввода-вывода, используя описательные имена вместо типов IRP-пакетов, которыми они на самом деле представлены. К примеру, операции IRP_MJ_WRITE и FASTIO_WRITE выводятся как операция WriteFile, а операции IRP_MJ_CREATE — как Open при открытии существующих файлов и как Create при создании новых файлов.

ЭКСПЕРИМЕНТ: НАБЛЮДЕНИЕ ЗА АКТИВНОСТЬЮ ПРОСТАИВАЮЩЕЙ ФАЙЛОВОЙ СИСТЕМЫ

Драйверы файловых систем в Windows поддерживают уведомления об изменениях в файлах (file change notification), что позволяет приложениям узнавать об изменениях в файловой системе без ее постоянного опроса. За уведомления отвечают функции ReadDirectoryChangesW, FindFirstChangeNotification и FindNextChangeNotification. Поэтому при запуске программы Process Monitor в простаивающей файловой системе вы не увидите повторяющихся обращений к файлам и папкам, так как данная активность не влияет негативным образом на общую производительность системы.

Запустите приложение Process Monitor и через несколько секунд проверьте в журнале вывода, происходит ли периодический опрос. Обнаружив соответствующую строку, щелкните на ней правой кнопкой мыши и выберите в появившемся меню команду Properties (Свойства). Для просмотра данных о процессе, который выполняет опрос, перейдите на вкладку Process (Процесс) открывшегося диалогового окна.

Устранение неисправностей с помощью Process Monitor

Приложение Process Monitor предлагает два основных способа анализа проблем, связанных с файловой системой и реестром. Следует найти в трассировочной информации Process Monitor сведения о последней операции, выполненной приложением перед тем, как произошел сбой, или сравнить след от «упавшего» приложения со следом работающей системы. Подробно эти способы рассматривались в части I (см. раздел «Технологии поиска и устранения неисправностей с помощью Process Monitor» в главе 4).

После трассировки записи в столбце Result могут иметь значения NAME NOT FOUND, NO SUCH FILE, PATH NOT FOUND, SHARING VIOLATION и ACCESS DENIED. Первые три указывают на попытку приложения открыть несуществующий файл или папку. Чаще всего это не является указанием на серьезную проблему. К примеру, при попытке запустить какую-либо программу через диалоговое окно Run (Запуск) без указания полного пути к ней приложение Проводник будет искать в папках, перечисленных в переменной окружения PATH, пока не найдет нужный образ или не закончит просмотр всех указанных папок. Каждая попытка найти образ в папке, в которой он отсутствует, будет отражаться на результатах вывода программы Process Monitor примерно такой строкой:

```
25314 7:44:27.4180943 PM Explorer.EXE 1640 CreateFile  
C:\Program Files\Microsoft Windows Performance Toolkit\test.exe NAME NOT FOUND Desired  
Access: Read Attributes, Disposition: Open, Options: Open Reparse Point, Attributes:  
n/a, ShareMode: Read, Write, Delete, AllocationSize: n/a
```

Ошибки, связанные с отклонением попыток доступа, являются частой причиной сбоев приложений при работе с файловой системой и возникают при отсутствии у приложения соответствующего разрешения на открытие файла или папки. Некоторые приложения не проверяют коды ошибок и не занимаются их устранением, что приводит к сбою или аварийному завершению работы. Существуют также приложения, выводящие неверные сообщения об ошибках, что мешает обнаружению источника проблемы.

Уязвимости, связанные с переполнением буфера, представляют серьезную угрозу безопасности, но сообщение BUFFER OVERFLOW является всего лишь средством драйвера файловой системы сообщить приложению о нехватке в выделенном буфере места для хранения данных. Разработчики приложений используют это поведение, чтобы определить корректный размер буфера, так как драйвер файловой системы заодно сообщает, сколько места ему требуется для сохранения данных. За операциями с кодом результата BUFFER OVERFLOW обычно следуют те же операции с успешным завершением.

Приложение Process Monitor широко используется в Microsoft и других компаниях для решения сложных вопросов или проблем, которые практически невозможно диагностировать.

Файловая система с типовым протоколированием

Транзакционная семантика баз данных или файловых систем с протоколированием часто требует слежения за хранящимися в файлах или записях данными и метаданными. Как правило, эти изменения сохраняются в структурах данных, называемых *записями журнала* (*log records*), с помощью операции, называемой *протоколированием* (*logging*). Наличие таких записей позволяет отменять, повторять и проверять изменения даже после перезагрузки системы.

В Windows данная служба реализована через файловую систему с типовым протоколированием (CLFS). Это сделано для поддержки встроенных в Windows транзакционных инструментов, таких как транзакционная файловая система NTFS (TxF) и транзакционный реестр (TxR), а также для предоставления сторонним разработчикам возможности пользоваться преимуществами подобных технологий. В CLFS существуют прикладные программные интерфейсы пользовательского режима и режима ядра, предназначенные для создания, чтения и записи CLFS-файлов журналов. Эти интерфейсы весьма гибкие и расширяемые, что позволяет вызывающей стороне определять хранящиеся в журналах детали реализации и структуру записей. Файловая система CLFS используется самыми разными приложениями, например базами данных; она применяется для хранения и пересылки очередей сообщений и агентов репликации; полезна для таких операций, как протоколирование событий, фиксация данных о соответствии или даже поддержка истории отмены и повтора операций в редакторе. Прикладные программные интерфейсы файловой системы CLFS обеспечивают согласованное представление журнала и доступ к нему компонентам как в режиме пользователя, так и в режиме ядра.

Хотя CLFS и называется файловой системой, на самом деле она с помощью *потоков данных* (*streams*) и *контейнеров* (*containers*), о которых пойдет речь далее, предоставляет виртуальный уровень абстракции, расположенный поверх NTFS. То, что в CLFS представляется как один виртуальный файл журнала, на самом деле может быть одним физическим файлом журнала, файлом журнала, разделенным на набор физических файлов, или даже набором файлов журнала, каждый из которых разделен на несколько физических файлов. Далее мы поговорим о том, каким образом NTFS взаимодействует с CLFS для поддержки транзакций.

Маршализование

В CLFS встроена функциональность алгоритма восстановления и изоляции с применением семантики (Algorithm for Recovery and Isolation Exploiting Semantics, ARIES), обеспечивающая надежное восстановление и репликацию операций в соответствии с промышленным стандартом. Но файловая система CLFS не ограничивается поддержкой ARIES; она предназначена для самых разных вариантов протоколирования. Полная спецификация ARIES доступна по адресу www.sai.msu.su/~megera/postgres/gist/papers/concurrency/p94-mohan.pdf.

Основным назначением любого высокопроизводительного журнала транзакций является предоставление клиентам возможности точного воспроизведения истории.

В CLFS это осуществляется путем маршалирования записей в клиентском журнале в буферы памяти, принудительной отправки их в стационарное хранилище (тот диск) и чтения записей по запросу. При условии сохранности накопителя CLFS может прочитать записанные данные даже после системного сбоя.

Клиенты как в режиме пользователя, так и в режиме ядра преобразуют буферы данных в записи журнала, которые являются частью *области маршалирования* (marshalling area), поддерживаемой в адресном пространстве клиента. При создании этой области клиент обязан указать количество и размер буферов ввода-вывода журнала, которые он собирается поддерживать. В ходе маршалирования выделяется место под буферы ввода-вывода журнала, они присоединяются к внутренней очереди журнала и сбрасываются на диск. Клиенты могут переопределять используемую по умолчанию политику маршалируемого кода, реализуя добавление в очередь и сброс данных на диск через API-вызовы.

Механизм маршалирования в CLFS разрабатывался, в частности, для минимизации транзакций ядра, что достигается, помимо прочего, за счет резервирования места под журнал и обязательной поддержки таких сценариев, как отмена транзакций. Каждое обращение области маршалирования журнала к CLFS-драйверу (что означает обращение работающего в режиме пользователя клиента к ядру) сопровождается попыткой области маршалирования согласовать требуемый объем зарезервированного места, обычно превосходящего необходимое на данный момент. Благодаря этому, если в будущем клиенту потребуется больше места, область маршалирования сможет немедленно удовлетворить запрос, не переходя в режим ядра. Однако следует заметить, что если пользователь запросит больше, чем зарезервировано, попытка области маршалирования удовлетворить этот запрос может привести к дополнительным переходам в режим ядра.

Типы журналов

В CLFS поддерживаются журналы двух типов: *выделенные* (dedicated logs) и *мультиплексные* (multiplexed logs), называемые также *типовыми* (common logs). В первом случае мы имеем дело с единым потоком записей, которым пользуются все клиенты журнала. Мультиплексные журналы работают с несколькими потоками записей, каждый из которых имеет собственный клиент и собственные буферы памяти для записей журнала маршалирования. При этом записи из всех буферов объединяются в одну очередь и записываются в один журнал на стационарном накопителе. Это позволяет свести воедино операции ввода-вывода нескольких потоков данных. При создании или открытии журнала CLFS определяет его тип в зависимости от вида пути к журналу.

При запросе к клиенту, работающему с выделенным журналом — он называется *физическим клиентом* (physical client), — CLFS находит для соответствующего файла физический блок управления файлом (FCB) и обрабатывает запрос.

Если же запрос поступил к клиенту, работающему с мультиплексным журналом — он называется *виртуальным клиентом* (virtual client), — CLFS находит для объектов соответствующие виртуальный блок управления файлом (FCB) и блок контроля контекста (Context Control Block, CCB) с целью преобразования запроса в операцию на физическом FCB-объекте. После этого CLFS осуществляет операцию на этом объекте, как было описано ранее.

В любом случае, если запрос является кэшированным чтением, CLFS использует службы диспетчера кэша для доступа к кэшированным данным. (Дополнительную информацию о диспетчере кэша вы найдете в главе 11.) Как и в случае запросов от драйверов других файловых систем, диспетчер кэша отображает представление файла и ссылается на это представление, которое может заставить диспетчера памяти произвести некэшированное чтение из CLFS, а не из физического журнала. Для сброса данных и некэшированного чтения CLFS находит целевой объект-контейнер в метаданных журнала и отправляет IRP-пакет непосредственно файловой системе NTFS. На рис. 12.12 демонстрируются возможные маршруты, которыми CLFS пересыпает запросы из режима пользователя в режим ядра.

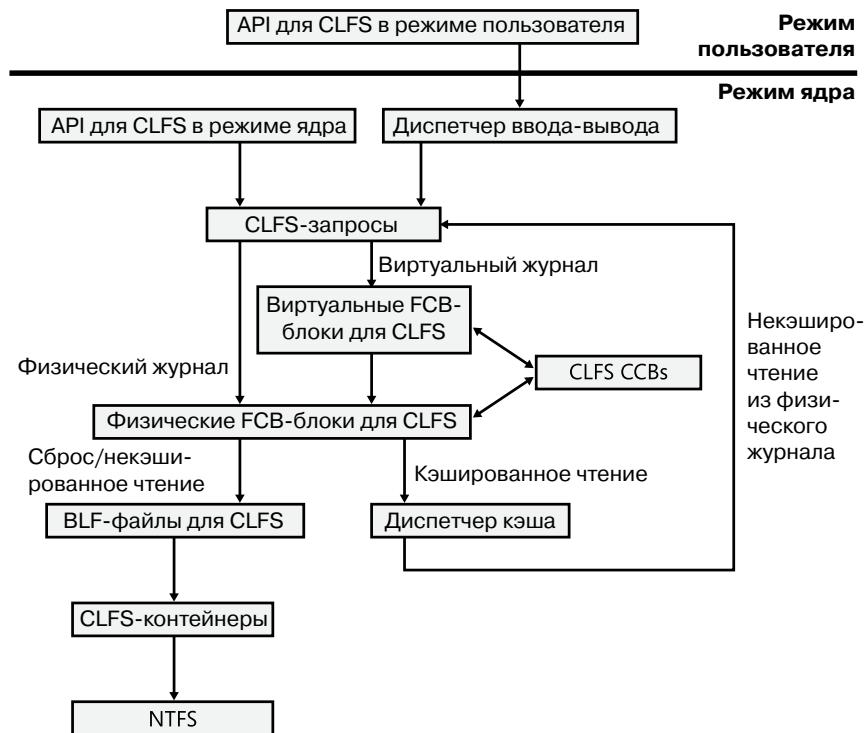


Рис. 12.12. Пути запросов в CLFS

Так как каждый поток данных в мультиплексном журнале выглядит для клиента как целый журнал, файловая система CLFS должна включить в физический журнал метаданные, показывающие, какому клиенту принадлежит каждый блок данных. Эти метаданные называются *страницей владельца* (owner page) и всегда представляют собой ровно одну страницу размером 4 Кбайт. Каждому фрагменту клиентских данных размером 512 Кбайт соответствует описывающая их страница владельца. Выделенным журналам не требуется следить за клиентом и отображением данных, поэтому они страниц владельца не имеют. На рис. 12.13 демонстрируется запись данных от двух

клиентов в мультиплексный журнал и иллюстрируется механизм формирования единой очереди, позволяющей сбросить данные на физический носитель за одну операцию ввода-вывода.

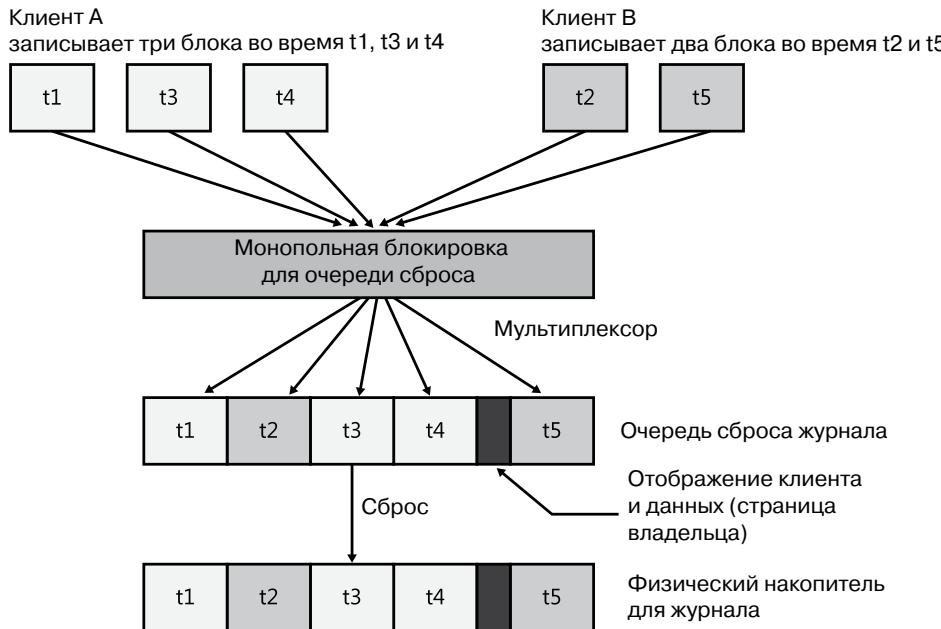


Рис. 12.13. Мультиплексирование в CLFS

Опустошение очереди сброса происходит при следующих условиях:

- объем данных в очереди превосходит заданное пороговое значение (по умолчанию 40 000 байт);
- в CLFS вызывается API сброса;
- произошла запись в область перезапуска и нужно сбросить данные журнала, находящиеся за пределами этой области (об области перезапуска рассказывается далее в разделе «Служба файла журнала»).

В процессе сброса CLFS сканирует очередь сброса, определяя количество находящихся в ней записей. После этого к соответствующим файлам журналов в NTFS посылаются IRP-пакеты, а CLFS ждет, пока дойдут все пакеты. Недошедшие пакеты отправляются повторно (причиной повтора может стать нехватка памяти, отсутствие квоты и т. п.).

Структура журнала

Файл журнала состоит из *базового файла журнала* (Base Log File, BLF), в котором содержатся метаданные, и набора контейнеров с реальными данными, количество которых может доходить до 1023. Базовый файл журнала изначально имеет размер

64 Кбайт и увеличивается по мере необходимости. В качестве метаданных хранятся такие сведения о журнале, как его начало, размер контейнера, путь к контейнеру, место, откуда выполняется операция перезапуска, состояние и имя журнала, используемые им клиенты. Для сохранения целостности в случае сбоя в процессе обновления журнала базовый файл хранит две копии метаданных и при обновлении переписывает более старую копию. В BLF хранится *счетчик дампов* (dump count), позволяющий определить возраст копий.

Контейнер представляет собой блок размещения для потока данных активного физического журнала. Все контейнеры одного журнала имеют одинаковый размер, кратный 512 Кбайт. Максимальный размер контейнера составляет 4 Гбайт. CLFS-клиент увеличивает или уменьшает поток данных журнала, добавляя в файл журнала контейнеры или удаляя их оттуда. CLFS реализует контейнеры как непрерывные файлы на томе, на котором находится BLF. Соотношение между базовым файлом журнала и хранящимися в контейнерах данными иллюстрирует рис. 12.14.

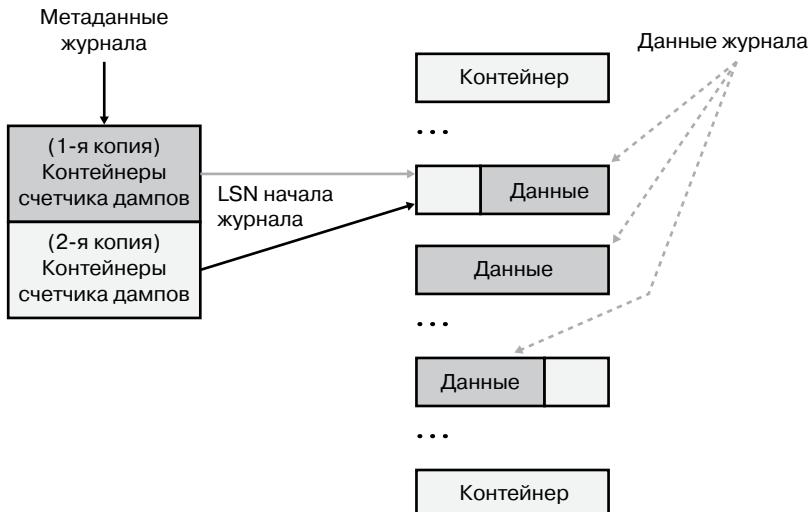


Рис. 12.14. Базовый файл журнала и контейнеры в CLFS

CLFS-драйвер помещает контейнеры в очередь, предлагая клиентам логическое представление единого непрерывного потока данных физического журнала. При этом *идентификаторы физических контейнеров* (physical container identifier) отображаются драйвером на *идентификаторы логических контейнеров* (logical container identifier). Когда хвост активного журнала выходит за границы последнего сектора контейнера, контейнеры начинают использоваться повторно. При этом контейнер перемещается из конца очереди контейнеров в ее начало, а его идентификатор обновляется.

Регистрационные номера транзакций в журнале

Когда клиент делает запись в поток данных, CLFS возвращает *регистрационный номер транзакции в журнале* (Log Sequence Number, LSN), по которому в дальнейшем можно

будет сослаться на запись. Номера, назначенные записям конкретного потока данных, формируют возрастающую последовательность. То есть LSN каждой записи больше, чем LSN предыдущей записи в том же потоке данных. Базовый файл журнала отслеживает два важных LSN-номера — LSN начала ведения журнала и LSN перезагрузки. Как уже упоминалось, они хранятся в метаданных BLF-файла.

LSN имеет ширину 64 бита и состоит из трех частей, как показано на рис. 12.15:

- 32-разрядный индекс контейнера идентифицирует контейнер, в котором находится запись журнала;
- 23-разрядное смещение блока задает смещение внутри контейнера;
- 9-разрядное смещение записи идентифицирует запись в блоке.

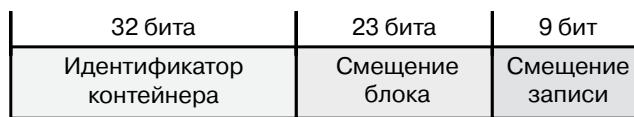


Рис. 12.15. Структура LSN-номера в CLFS

Блоки журнала

Так как в процессе записи в журнал не исключена неудача, называемая *прерванной записью* (torn write), CLFS при помощи *блоков журнала* (log blocks) отслеживает, полностью ли записи журнала перенесены на накопитель. Именно в блоках, соответствующих секторам размером 512 байт, CLFS сохраняет записи и именно с их помощью осуществляет чтение данных из журнала и запись в него. В каждый блок журнала входит сигнатура сектора размером 2 байта, расположенная в конце каждого сектора в блоке. Блок хранит регистрационный номер транзакции и флаги, а в самом конце — массив из последних подтвержденных сигнатур, как показано на рис. 12.16. Блок считается корректным только при корректности всех сигнатур в журнале блока и их совпадения с сигнатурами массива. Если же, к примеру, когда возникает системный сбой, блок записан только наполовину, сигнатурь не совпадут, и CLFS посчитает блок недействительным.



Рис. 12.16. Блоки журнала в CLFS

Страницы владельца

Как уже упоминалось, каждый блок данных размером 512 Кбайт, называемый *областью* (region), в мультиплексном журнале согласуется с виртуальным журналом через страницу владельца. Каждая область состоит из *страниц* (pages) размером 4 Кбайт, а каждая страница — из одного или нескольких секторов, содержащих блоки журнала. Как показано на рис. 12.17, страница владельца является последней страницей в области. Так как она сама по себе является блоком журнала, CLFS может найти на ней прерванные записи, используя массив сигнатур.



Рис. 12.17. Области и страницы владельца в CLFS

Страница владельца содержит два вида данных:

- ❑ Для каждого сектора в области есть виртуальный журнал, которому принадлежит сектор, а также серийный номер сектора (начиная с 0). Максимальное количество секторов в области равно 1024.
- ❑ Для каждого виртуального журнала есть минимальный и максимальный LSN-номера в данной области. Эти значения задают диапазон действительных виртуальных LSN-номеров для области.

Изучив LSN виртуального журнала на странице владельца, CLFS может определить, принадлежит ли запись с этим LSN-номером текущей области. Если запись области не принадлежит, файловая система CLFS сравнивает LSN ее виртуального журнала с заданным диапазоном и решает, в предыдущей или в следующей области продолжать поиск.

При вставке блока журнала в очередь сброса физического FCB-блока мультиплексного журнала может оказаться, что текущий блок перекрывает страницу владельца текущей области. В этом случае CLFS разбивает этот блок и вставляет страницу владельца после первой половины (как показано на рис. 12.17). Другими словами, страница владельца записывается на диск только после заполнения описываемой ею области. При повторном открытии файла мультиплексного журнала клиентом CLFS сканирует области и восстанавливает в памяти страницу владельца, описывая последнюю область, для которой не был записан блок журнала.

Следует заметить, что при повторном открытии файла журнала CLFS точно не знает, где находится LSN-номер конца журнала, и этот номер следует найти, чтобы избежать потери данных или работы с поврежденными данными. В случае выделенного журнала CLFS последовательно читает все блоки, пока не обнаружит недопустимый блок. Именно это место CLFS считает концом журнала. В случае же мультиплексного журнала CLFS читает последнюю страницу владельца (базовый файл журнала сохра-

няет копию LSN последней сброшенной страницы при последнем сбросе метаданных журнала) и проверяет ее корректность. После этого CLFS начинает одну за другой читать страницы владельцев следующих областей, пока не обнаружит некорректную. Затем CLFS выполняет обратное сканирование для обнаружения первой области, содержащей только действительные блоки данных. С точки зрения CLFS, конец журнала должен располагаться внутри следующей области. Она сканирует блок за блоком, пока не обнаружит недопустимый блок. Это место и обозначается как конец журнала.

Преобразование виртуальных LSN-номеров в физические

CLFS идентифицирует блоки внутри физического журнала при помощи физических LSN-номеров. Но в случае мультиплексных журналов CLFS составляет физический журнал из нескольких виртуальных, а для локализации блоков в последних пользуется виртуальными LSN-номерами. Следовательно, в клиенте для виртуального журнала адрес блока может указываться как через физический, так и через виртуальный LSN-номер.

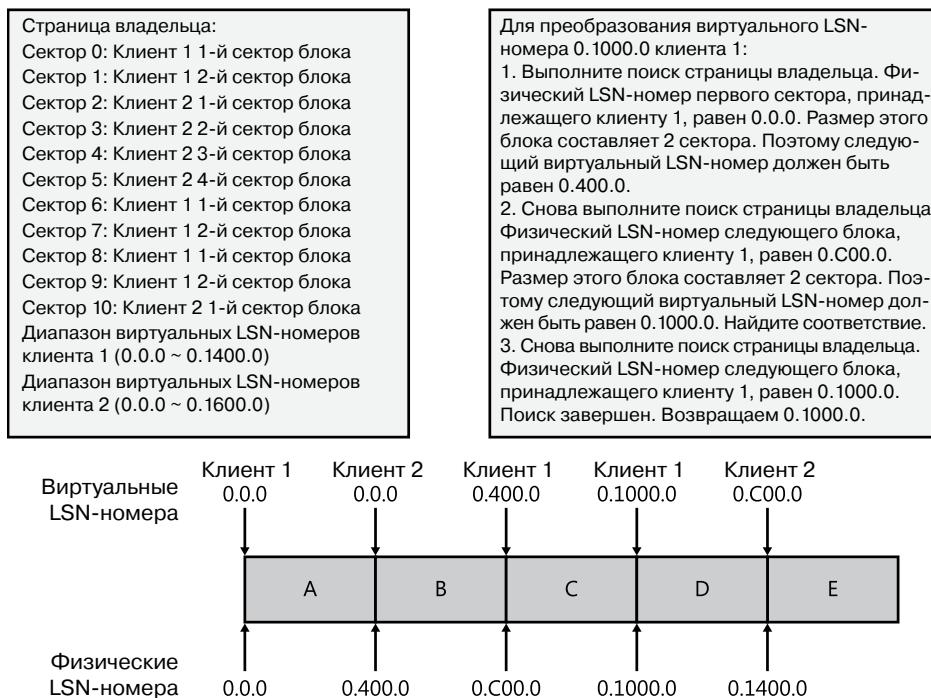


Рис. 12.18. Преобразование виртуальных LSN-номеров в физические в CLFS

Вот таким образом осуществляется преобразование виртуальных LSN-номеров журнала в физические:

- Читается страница владельца для области, обозначенной виртуальным LSN-номером журнала.

2. Проверяется область виртуального LSN-номера страницы владельца, чтобы понять, попадает LSN в эту область или нет. В большинстве случаев блок журнала оказывается внутри области.
3. Если виртуальный LSN-номер принадлежит области, то для поиска физического смещения блока для LSN-номера CLFS обращается к *сектору в клиентском отображении* на странице владельца. Зная виртуальный LSN-номер клиента и его размер, CLFS может вычислить виртуальный LSN-номер следующего блока журнала. По этому правилу CLFS однозначно вычисляет физические LSN-номера всех виртуальных блоков в области, как показано на рис. 12.18.
4. Если виртуальный LSN-номер не попадает в рассматриваемую область, CLFS осуществляет поиск в предыдущей или в следующей области, в зависимости от того, как соотносится этот LSN-номер с диапазоном виртуальных LSN-номеров текущей области.

Политики управления

Каждый журнал в CLFS определяется набором политик управления, конфигурация которых выбирается клиентом. Они перечислены в табл. 12.5.

Таблица 12.5. Политики управления в CLFS

Политика	Описание
ClfsMgmtPolicyMaximumSize	Максимальный размер журнала
ClfsMgmtPolicyMinimumSize	Минимальный размер журнала
ClfsMgmtPolicyNewContainerSize	Размер новых создаваемых контейнеров
ClfsMgmtPolicyGrowthRate	Количество контейнеров, которые будут добавляться в журнал при увеличении его размера. Этот параметр может быть указан как в процентах, так и в виде конкретного числа
ClfsMgmtPolicyLogTail	Размер свободного пространства, которое будет запрашиваться при получении клиентом уведомления о необходимости сдвинуть последний фрагмент журнала. Указывается как минимальный процент свободного места или как минимальное количество контейнеров
ClfsMgmtPolicyAutoShrink	Момент сжатия журнала в зависимости от процента оставшегося свободного места
ClfsMgmtPolicyAutoGrow	Определяет, должен ли увеличиваться размер журнала при менее чем двух свободных контейнерах
ClfsMgmtPolicyNewContainerPrefix	Префикс имени файла каждого контейнера и полный путь к папке размещения контейнеров

Цели разработки и особенности NTFS

В этом разделе мы рассмотрим требования, которые учитывались при разработке NTFS, после чего перейдем к рассмотрению нетривиальных функциональных возможностей этой файловой системы.

Требования к профессиональной файловой системе

Разработка NTFS с самого начала велась с учетом требований, предъявляемых к файловой системе корпоративного класса. Чтобы свести к минимуму потери данных при неожиданном сбое или прекращении работы, файловая система должна гарантировать целостность своих метаданных. Для предотвращения несанкционированного доступа к конфиденциальным данным в файловую систему следует встроить модель защиты. Наконец, она должна поддерживать безопасность пользовательских данных за счет программной избыточности, которая предлагается как недорогая альтернатива аппаратным решениям. В этом разделе показано, как все эти возможности реализованы в NTFS.

Восстанавливаемость

В соответствии с требованиями к надежности хранения данных и доступа к ним NTFS обеспечивает возможность восстановления, в основе которого лежит концепция *атомарной транзакции* (atomic transaction). Этим термином называют метод обработки изменений в базе данных, при котором сбои в работе системы не влияют на целостность базы. Суть метода заключается в том, что ряд операций, называемых транзакциями, проводится по принципу «всё или ничего». (Транзакцию можно определить как операцию ввода-вывода, меняющую данные файловой системы или структуру каталогов тома.) Отдельные изменения, из которых состоит транзакция, выполняются атомарно — то есть если транзакция началась, все изменения диска подлежат завершению. Если транзакция прерывается сбоем в работе системы, уже завершенные операции подлежат отмене, или *rollback* (roll back). После отката база данных возвращается в предшествующее транзакции согласованное состояние.

С помощью транзакций NTFS реализует возможность восстановления. Если какая-то программа начинает операцию ввода-вывода, меняющую структуру NTFS-тома, то есть модифицирует структуру папок, увеличивает длину файла, выделяет место под новый файл и т. п., файловая система обрабатывает эту операцию как атомарную транзакцию. Это гарантирует, что действия будут либо выполнены полностью, либо — если хотя бы одну операцию не удастся завершить из-за системного сбоя — отменены. О том, как это делается, мы поговорим в разделе «Поддержка восстановления в NTFS». Добавим, что для хранения критически важной для функционирования NTFS информации применяется избыточность, поэтому если на диске будет поврежден сектор, файловая система все равно сможет получить доступ к хранящимся на нем данным.

Безопасность

Безопасность в NTFS строится на модели Windows-объектов. Неавторизованные пользователи не имеют доступа к файлам и папкам. (Подробно вопросы безопасности освещаются в главе 6 части I.) Открытый файл реализуется в виде файлового объекта

с дескриптором безопасности, хранящимся на диске в скрытом метафайле **\$Secure**, в потоке данных, называемом **\$SDS** (Security Descriptor Stream — поток данных дескриптора безопасности). Прежде чем процесс сможет открыть дескриптор любого объекта, в том числе файлового, система безопасности Windows должна убедиться в наличии у этого процесса соответствующих полномочий. Дескриптор безопасности в сочетании с требованием авторизации пользователя при входе в систему гарантируют, что ни один процесс не получит доступа к файлу без разрешения системного администратора или владельца файла. (Сведения о дескрипторах безопасности можно найти в главе 6 части I, а файловые объекты рассматриваются в главе 8.)

Избыточность данных и отказоустойчивость

В дополнение к возможности восстановления данных файловой системы некоторые заказчики требуют, чтобы их собственные данные не подвергались риску при отключении питания или фатальном сбое диска. Механизм восстанавливаемости в NTFS действительно гарантирует, что файловая система на томе останется доступной, но пользовательских файлов это не касается. Защита приложений, которые не могут позволить себе утрату информации, осуществляется за счет избыточности данных.

Избыточность данных для пользовательских файлов реализуется в Windows через многоуровневую модель драйверов (см. главу 8), поддерживающую отказоустойчивые диски. Чтобы записать данные на диск, NTFS взаимодействует с диспетчером томов, который, в свою очередь, взаимодействует с драйвером жесткого диска. Диспетчер томов может *зеркаливать* (mirror), или дублировать, данные одного диска на другом, что позволяет при необходимости воспользоваться данными с избыточной копии. Поддержку такой возможности обычно называют *RAID уровня 1*. Диспетчеры томов также могут записывать данные в *чередующиеся области* (stripes) на три и более дисков, используя один диск для хранения информации о четности. Если данные на одном диске станут недоступными, драйвер может восстановить содержимое этого диска при помощи логической операции *XOR*. Такую поддержку называют *RAID уровня 5*. (Чередующиеся и зеркальные тома, а также тома RAID-5 рассматриваются в главе 9.)

Нетривиальные возможности NTFS

NTFS не только является восстанавливаемой, безопасной, надежной и эффективной файловой системой, она также предлагает несколько нетривиальных возможностей, позволяющих поддерживать широкий спектр приложений. Некоторые из этих возможностей доступны приложениям через API, другие являются внутренними:

- множественные потоки данных;
- имена на базе Unicode;
- универсальный механизм индексации;
- динамическое переназначение поврежденных кластеров;
- жесткие ссылки;
- символические (мягкие) ссылки и точки соединения;
- сжатие и разреженные файлы;

- протоколирование изменений;
- квоты томов, отдельные для каждого пользователя;
- отслеживание ссылок;
- шифрование;
- поддержка POSIX;
- дефрагментация;
- поддержка доступа только для чтения и динамическое разбиение на разделы.

Далее мы подробно рассмотрим каждую из этих возможностей.

Множественные потоки данных

В NTFS каждая ассоциированная с файлом единица информации, включая имя файла, сведения о его владельце, метки времени, содержимое и прочее, реализуется как атрибут файла (атрибут NTFS-объекта). Каждый атрибут состоит из одного *потока данных* (stream), то есть простой последовательности байтов. Такая обобщенная реализация позволяет легко добавлять к файлу новые атрибуты (и, соответственно, новые потоки данных). Так как данные файла являются всего лишь «еще одним атрибутом», и благодаря возможности присоединения атрибутов, в NTFS файлы (и папки, в которых они находятся) могут включать в себя несколько потоков данных.

Любой файл в NTFS по умолчанию содержит один безымянный поток данных. Приложения могут дополнительно создавать именованные потоки данных и обращаться к ним по именам. Чтобы избежать модификации API-интерфейсов ввода-вывода, использующих в качестве аргумента строку с именем файла, имена потоков данных указываются через двоеточие (:) после имен файлов. Двоеточие является специальным символом и поэтому служит разделителем между именем файла и именем потока данных, как показано в следующем примере:

```
myfile.dat:stream2
```

Каждый поток данных имеет свой выделенный размер (который определяет количество зарезервированного под поток дискового пространства), реальный размер (количество байтов, использованное вызывающей функцией) и длину действительных данных (указывает, какая часть потока была инициализирована). Кроме того, каждому потоку предоставляется отдельная файловая блокировка, применяемая для фиксации диапазонов байтов и предоставления параллельного доступа.

Множественные потоки данных в Windows используют, к примеру, служба Attachment Execution Service, которая вызывается в случае, когда такие приложения, как Internet Explorer или Outlook, применяют стандартный в Windows API-интерфейс для сохранения интернет-вложений. В зависимости от того, откуда загружается файл (например, из зоны My Computer, Intranet или Untrusted), Проводник может предостеречь пользователя, сообщив, что загрузка осуществляется из недоверенного источника, или даже полностью заблокировать доступ к файлу. К примеру, на рис. 12.19 показано диалоговое окно, которое появляется в результате работы программы Process Explorer после завершения загрузки с сайта Sysinternals.

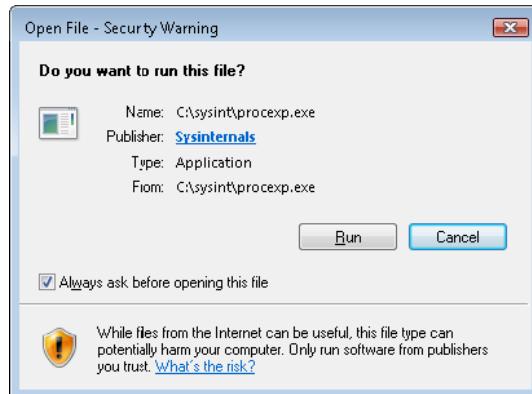


Рис. 12.19. Предупреждение системы безопасности для загруженных из Интернета файлов

ПРИМЕЧАНИЕ

При сбросе флажка Always ask before opening this file (Всегда спрашивать при открытии этого файла) отключает от файла поток данных идентификатора зоны.

Другие приложения также могут использовать множественные потоки данных. К примеру, службе резервного копирования дополнительный поток данных помогает сохранять временные метки, связанные с резервированием файлов. А служба архивирования реализует иерархическое хранилище, в котором файлы, созданные ранее определенной даты или не востребованные в течение заданного периода, перемещались бы на ленточные накопители. При этом служба после копирования файла «на ленту» обнуляет используемый по умолчанию поток данных и добавляет новый поток данных, задающий место хранения файла.

ЭКСПЕРИМЕНТ: ПРОСМОТР ПОТОКОВ ДАННЫХ

Большинство Windows-приложений не рассчитано на работу с дополнительными именованными потоками данных, но команды echo и more поддерживают эту функциональность. Соответственно, существует простой способ наблюдать за потоками данных в действии. Нам нужно создать именованный поток данных при помощи команды echo, а затем вывести его на экран командой more. В результате мы получим файл test с потоком stream:

```
C:\>echo hello > test:stream
C:\>more < test:stream
hello
C:\>
```

При выводе содержимого папки размер файла test не отражает хранящиеся в дополнительном потоке данные, так как NTFS возвращает при запросах к файлу размер только безымянного потока данных.

```
C:\>dir test
Volume in drive C is WINDOWS
Volume Serial Number is 3991-3040

Directory of C:\

08/01/00 02:37p 0 test
1 File(s) 0 bytes
112,558,080 bytes free
```

Служебная программа Streams от Sysinternals позволяет определить, какие файлы и папки в вашей системе содержат дополнительные потоки данных. Аналогичный результат дает переключатель /r в команде dir.

```
C:\>streams test

Streams v1.56 - Enumerate alternate NTFS data streams
Copyright (C) 1999-2007 Mark Russinovich
Sysinternals - www.sysinternals.com

C:\test:
:stream:$DATA 8
```

Имена на базе Unicode

Как и Windows в целом, NTFS поддерживает 16-разрядные Unicode-символы 1.0/UTF-16 для хранения имен файлов, папок и томов. (В последней на момент написания книги версии стандарта Unicode, появившейся в феврале 2012 года версии 6.1, символы выводятся последовательностями длиной до 4 байт. В режиме ядра эта версия не поддерживается.) Unicode обеспечивает уникальное представление любого символа основных мировых языков, что упрощает обмен информацией между странами. Это улучшенное по сравнению с традиционным представление международных символов. В нем применяется двухбайтовая схема кодирования, сохраняющая часть символов в виде 8 бит, а часть — в виде 16. Для установки доступных символов эта технология требует загрузки различных кодовых страниц. Так как Unicode обеспечивает уникальное представление каждого символа, вид символов не зависит от загруженных кодовых страниц. Длина имен файлов и папок, фигурирующая в маршрутах доступа, может достигать 255 символов. В именах допустимы Unicode-символы, пробелы и идущие подряд точки.

Универсальный механизм индексации

Архитектура NTFS позволяет индексировать атрибуты файлов на дисковом томе при помощи структуры Б-дерева. (Создание индексов по произвольным атрибутам пользователям не предлагается.) Данная структура дает файловой системе возможность эффективно искать файлы по указанным критериям, например все файлы, находящиеся в определенной папке. А, например, файловая система FAT индексирует, но не сортирует имена файлов, что замедляет поиск в больших папках.

Некоторые возможности NTFS опираются на преимущества универсальной индексации, в том числе консолидированные дескрипторы безопасности. При этом дескрипторы

безопасности файлов и папок тома хранятся в едином внутреннем потоке данных без дубликатов и индексируются с применением внутреннего идентификатора безопасности, который задает NTFS. Подробно данный вариант индексации рассматривается далее в разделе «NTFS-структура на диске».

Динамическое переназначение поврежденных кластеров

Обычно попытки приложений считывать данные из поврежденных секторов оканчиваются неудачей, а данные соответствующего кластера становятся недоступными. Но если отформатировать диск как отказоустойчивый NTFS-том, диспетчер томов динамически считает «хорошую» копию данных, хранившихся на поврежденном секторе, а NTFS получит уведомление о проблемах с этим сектором. После этого NTFS выделит новый кластер на замену поврежденному и скопирует в него данные. Поврежденный кластер попадет в особый список (хранящийся в скрытом файле мтаданных \$BadClus) и в дальнейшем использоваться не будет. Такое восстановление данных и динамическое переназначение поврежденных секторов особенно полезно для файловых серверов и отказоустойчивых систем, а также для приложений, в которых потеря данных недопустима. Если диспетчер томов не был загружен в момент повреждения сектора (к примеру, все произошло на ранних этапах загрузки), NTFS все равно заменит кластер и не допустит его повторного использования, хотя восстановить данные из поврежденного сектора уже не удастся.

Жесткие ссылки

Жесткая ссылка (hard links) позволяет обратиться к одному и тому же файлу по различным путям. (Для папок они не поддерживаются.) При создании жесткой ссылки вида C:\Documents\Spec.doc, указывающей на файл C:\Users\Administrator\Documents\Spec.doc, вы связываете с файлом два пути. И оба пути можно будет использовать для последующих обращений к данному файлу. Процессы создают жесткие ссылки, вызывая Windows-функцию CreateHardLink или POSIX-функцию ln.

В NTFS жесткие ссылки реализуются через счетчик ссылок на существующие данные, в котором при создании каждой следующей жесткой ссылки появляется дополнительная ссылка на данные по имени файла. Таким образом, при наличии нескольких жестких ссылок на файл можно удалить исходный путь к данным (в нашем примере C:\Users\Administrator\Documents\Spec.doc). Останется вторая жесткая ссылка (C:\Documents\Spec.doc), которая будет указывать на данные. Но жесткие ссылки являются локальными и привязаны к диску (их представляет номер записи файла), они существуют только в пределах одного тома и не могут указывать на другой том или другой компьютер.

ЭКСПЕРИМЕНТ: СОЗДАНИЕ ЖЕСТКОЙ ССЫЛКИ

Существует два средства создания жесткой ссылки: команда fsutil hardlink create или утилита mklink с параметром /H. В данном эксперименте мы рассмотрим второй вариант, так как данная утилита потребуется нам неоднократно, например для создания символической ссылки. Для начала создадим файл test.txt (добавьте в него какой-нибудь текст):

```
C:\>echo hello > test.txt
```

Теперь создадим жесткую ссылку hard.txt:

```
C:\>mklink hard.txt test.txt /H  
Hardlink created for hard.txt <<===>> test.txt
```

Если вывести содержимое папки, вы обнаружите, что два файла совершенно идентичны по всем параметрам. У них одна и та же дата создания, одни и те же разрешения и размеры файлов; отличаются только их имена:

```
C:\>dir *.txt  
Volume in drive C is OS  
Volume Serial Number is 38D4-EA71  
Directory of C:\  
05/12/2012 11:55 PM 8 hard.txt  
05/12/2012 11:55 PM 8 test.txt  
2 File(s) 16 bytes  
0 Dir(s) 10,646,011,904 bytes free
```

Символические (мягкие) ссылки и соединения

В дополнение к жестким ссылкам в NTFS поддерживается еще один тип псевдонимов для имен файлов, который называется *символическими* (symbolic), или *мягкими* (soft), *ссылками*. Такие ссылки представляют собой динамически интерпретируемые строки, которые могут быть относительными или абсолютными путями к месту на любом накопителе, в том числе на другом локальном томе или в общем доступе на другом компьютере. Это означает, что символические ссылки не увеличивают счетчик ссылок исходного файла и удаление этого файла приведет к потере данных. Наконец, в отличие от жестких ссылок, символические ссылки могут указывать не только на файлы, но и на папки, что дает им дополнительное преимущество.

К примеру, если путь C:\Drivers представляет собой символическую ссылку на папку, которая направляется по адресу %SystemRoot%\System32\Drivers, приложение, читающее файл C:\Drivers\Ntfs.sys, будет иметь дело с файлом %SystemRoot%\System\Drivers\Ntfs.sys. Символические ссылки являются распространенным средством подъема расположенных глубоко в дереве каталогов папок на более высокий уровень без нарушения структуры или содержимого исходного дерева. В приведенном примере папка Drivers поднялась до корневой папки на томе, уменьшив глубину расположения файла Ntfs.sys с третьего уровня на первый. Аналогичным способом работают символические ссылки на файлы. Их можно представить в виде ярлыков, просто они реализуются в файловой системе, а не являются управляемыми Проводником Ink-файлами. Символические ссылки, как и жесткие ссылки, создаются программой mklink (без параметра /H) или через API CreateSymbolicLink.

Так как некоторые устаревшие приложения при наличии символьических ссылок могут работать с нарушениями безопасности, особенно если приходится обращаться к файлам с другого компьютера, для создания символьской ссылки требуется привилегия SeCreateSymbolicLink, обычно имеющаяся только у администраторов. Файловая система также обладает параметром поведения SymLinkEvaluation, конфигурация которого задается следующей командой:

```
fsutil behavior set SymLinkEvaluation
```

По умолчанию политика оценки символических ссылок в Windows допускает только ссылки с локальной машины на локальную и с локальной машины на удаленную, но не наоборот:

```
C:\>fsutil behavior query SymLinkEvaluation  
Local to local symbolic links are enabled  
Local to remote symbolic links are enabled.  
Remote to local symbolic links are disabled.  
Remote to Remote symbolic links are disabled.
```

Символические ссылки реализуются с применением такого NTFS-механизма, как *точки повторной обработки* (reparse points). (Подробно они рассматриваются далее.) Такая точка представляет собой файл или папку, с которыми ассоциирован блок данных, называемых *данными повторной обработки* (reparse data). Это пользовательские сведения о файле или папке, например информация о состоянии или местоположении. Их можно считать из точки повторной обработки при помощи приложения, в котором были созданы данные, фильтрующего драйвера файловой системы или диспетчера ввода-вывода. Обнаружив при поиске файла или папки точку повторной обработки, NTFS возвращает код состояния STATUS_REPARSE, который является сигналом о необходимости анализа данных повторной обработки для подключенных к дисковому тому фильтрующих драйверов файловой системы и диспетчера ввода-вывода. Каждому типу точек повторной обработки соответствует уникальный тег, позволяющий компоненту, отвечающему за интерпретацию данных конкретной точки повторной обработки, распознать такую точку без проверки данных. После этого владелец тега (фильтрующий драйвер файловой системы или диспетчер ввода-вывода) может выбрать один из следующих вариантов действий:

- ❑ Изменить указанное в операции ввода-вывода полное имя файла, при анализе которого была обнаружена точка повторной обработки, и заново инициализировать вывод по измененному пути. Именно этот подход используют точки соединения (о них мы поговорим чуть позже), к примеру, для изменения направления поиска папки.
- ❑ Удалить из файла точку повторной обработки, каким-то образом отредактировать файл и инициализировать новую операцию файлового ввода-вывода.

Windows-функций для создания точек повторной обработки не существует. Вместо этого процессам нужно использовать управляющий код FSCTL_SET_REPARSE_POINT файловой системы в сочетании с функцией DeviceIoControl. Процесс может запросить содержимое точки повторной обработки с помощью управляющего кода FSCTL_GET_REPARSE_POINT. В атрибутах ассоциированного с этой точкой файла присутствует флаг FILE_ATTRIBUTE_REPARSE_POINT, что позволяет приложениям проверять наличие точек повторной обработки путем вызова функции GetFileAttributes.

Другим типом поддерживаемых NTFS точек повторной обработки является *соединение* (junction). Это унаследованное NTFS понятие, которое работает практически аналогично символическим ссылкам на папки, но исключительно в пределах одного тома. Единственным преимуществом соединений в данном случае является совместимость с предыдущими версиями Windows, которой нет у символических ссылок на папки.

ЭКСПЕРИМЕНТ: СОЗДАНИЕ СИМВОЛИЧЕСКОЙ ССЫЛКИ

Этот эксперимент продемонстрирует, чем различаются символьическая и жесткая ссылки даже при работе с файлами на одном томе. Создайте символьическую ссылку soft.txt, как показано в следующем фрагменте, нацелив ее на созданный в предыдущем эксперименте файл test.txt:

```
C:\>mklink soft.txt test.txt  
symbolic link created for soft.txt <<===> test.txt
```

Если вывести на экран содержимое папки, вы обнаружите, что для символьической ссылки отсутствует указание на размер файла, а идентифицируется она типом <SYMLINK>. Более того, в этой строке фигурирует время создания ссылки, а не файла, на который она указывает. Права доступа у символьической ссылке и целевого файла также могут различаться:

```
C:\>dir *.txt  
Volume in drive C is 0S  
Volume Serial Number is 38D4-EA71  
  
Directory of C:\  
  
05/12/2012 11:55 PM  8 hard.txt  
05/13/2012 12:28 AM <SYMLINK> soft.txt [test.txt]  
05/12/2012 11:55 PM  8 test.txt  
3 File(s) 16 bytes  
0 Dir(s) 10,636,480,512 bytes free
```

Наконец, при удалении исходного файла test.txt вы обнаружите, что обе ссылки — жесткая и символьическая — никуда не делись, но последняя больше не указывает на нужный файл, в то время как переход по жесткой ссылке откроет вам доступ к данным.

Сжатие и разреженные файлы

NTFS поддерживает сжатие, или компрессию, файловых данных. Так как процедуры компрессии и декомпрессии в NTFS прозрачны, приложения не нуждаются в модификации, чтобы ими воспользоваться. Сжатие применяется и к папкам, что влечет за собой сжатие всех файлов, которые после этого будут созданы в данной папке.

Приложения выполняют компрессию и декомпрессию файлов, передавая управляющий код FSCTL_SET_COMPRESSION функции DeviceIoControl. Запрос состояния сжатия файла или папки осуществляется при помощи управляющего кода FSCTL_GET_COMPRESSION. В атрибутах сжатых файлов и папок установлен флаг FILE_ATTRIBUTE_COMPRESSED, поэтому приложения могут определять состояние их сжатия при помощи функции GetFileAttributes.

Второй тип сжатия основан на так называемых *разреженных файлах* (sparse files). Если файл помечен как разреженный, NTFS не выделяет на томе место для тех фрагментов файла, которые определены приложением как пустые. При чтении приложением этих фрагментов NTFS возвращает буферы, заполненные нулевыми значениями. Этот тип сжатия полезен для клиент-серверных приложений, реализующих протоколирование с циклическими буферами, при котором сервер регистрирует информацию в файле, а клиенты асинхронно считывают ее. Так как считанная клиентом информация

в дальнейшем уже не требуется, нет нужды хранить ее в файле. Сделав такой файл разреженным, клиент сможет помечать считанные области как пустые, освобождая место на томе. Сервер может продолжить добавлять в файл информацию, не опасаясь, что файл в итоге займет все свободное пространство.

Как и в случае сжатых файлов, управление разреженными файлами осуществляется в NTFS совершенно прозрачно. Приложения указывают состояние разреженности файла, передавая функции `DeviceIoControl` управляющий код `FSCTL_SET_SPARSE`. Чтобы пометить диапазон файла как пустой, приложения пользуются кодом `FSCTL_SET_ZERO_DATA`. Выяснить у NTFS, какие части файла являются разреженными, приложения могут при помощи управляющего кода `FSCTL_QUERY_ALLOCATED_RANGES`. Разреженные файлы используются в NTFS, например в журнале изменений, о котором мы сейчас и поговорим.

Протоколирование изменений

Приложениям многих типов приходится следить за изменениями файлов и папок тома. К примеру, программа автоматического резервного копирования начинает с создания полной копии, а в дальнейшем сохраняет только измененные файлы. Очевидным способом слежения за изменениями в данном случае является сканирование тома с записью состояния файлов и папок и сравнением с результатами предыдущего сканирования. Но этот процесс может негативно повлиять на производительность системы, особенно на компьютерах с тысячами и десятками тысяч файлов.

Альтернативным подходом для приложения является регистрация с помощью функции `FindFirstChangeNotification` или `ReadDirectoryChangesW` с целью получения уведомлений об изменении содержимого папок. В качестве входного параметра приложение указывает имя папки, состояние которой требуется отследить, а функция сообщает обо всех изменениях в содержимом этой папки. Это более эффективный по сравнению со сканированием тома подход, но он требует непрерывной работы приложения. Кроме того, приложению все равно может потребоваться сканирование папок, так как функция `FindFirstChangeNotification` сообщает только о факте изменений, не указывая подробностей. Приложение может передать буфер, в который FSD записывает данные об изменениях, в функцию `ReadDirectoryChangesW`. Но в случае переполнения буфера приложение должно быть готово к тому, что придется снова сканировать папку.

В NTFS предусмотрен и третий вариант, лишенный недостатков первых двух: приложение может настроить журнал изменений с помощью управляющего кода `FSCTL_CREATE_USN_JOURNAL` (где `USN` — это номер последовательного обновления) функции `DeviceIoControl`. В этом случае NTFS будет записывать сведения об изменениях в файлах и папках во внутренний журнал, называемый *журналом изменений* (*change journal*). Обычно он достаточно велик, что дает приложению шанс обработать все изменения, ничего не упустив. Приложения пользуются управляющим кодом `FSCTL_QUERY_USN_JOURNAL` для чтения записей из журнала изменений и могут сделать так, чтобы работа функции `DeviceIoControl` не завершалась, пока в журнале остаются новые записи.

Квоты томов для пользователей

Системным администраторам часто требуется отслеживать или ограничивать дисковое пространство, занимаемое пользователями на томах общего доступа, для этого NTFS

поддерживает механизм управления дисковым пространством на основе квот. В этом случае каждому пользователю выделяются квоты, а NTFS следит за их потреблением и за моментом достижения порогового значения. Можно настроить NTFS таким образом, что событие, возникающее при превышении пользователем порогового значения, будет записываться в системный журнал. Аналогично, при попытке пользователя занять больше места, чем разрешает квота, NTFS регистрирует событие в системном журнале и завершает ввод-вывод для приложения, ставшего причиной нарушения квоты с кодом ошибки заполненности диска («disk full»).

Слежение за использованием тома в NTFS возможно благодаря тому, что файлы и папки помечены идентификаторами безопасности (Security ID, SID) пользователей, создавших эти объекты на томе. (Определение SID дано в главе 6 части I.) Сумма логических размеров принадлежащих пользователю файлов и папок сравнивается с квотой, определенной администратором. Соответственно, пользователь не может превысить квоту, создав пустой разреженный файл и заполнив его ненулевыми значениями. Поэтому, несмотря на возможность сжать файл размером 50 Кбайт до 10 Кбайт, при учете используется его исходный размер — 50 Кбайт.

По умолчанию режим отслеживания квот на томах отключен. Для его включения, указания пороговых значений для выдачи предупреждений и настройки поведения NTFS при достижении пороговых значений служит вкладка Quota (Квота) окна свойств тома, показанная на рис. 12.20. Окно Quota Entries (Записи квот), которое можно открыть с этой вкладки, позволяет администратору задавать различные лимиты и варианты поведения для каждого пользователя. Приложения, которым требуется управление на основе NTFS-квот, задействуют COM-интерфейсы квот, в том числе `IDiskQuotaControl`, `IDiskQuotaUser` и `IDiskQuotaEvents`.

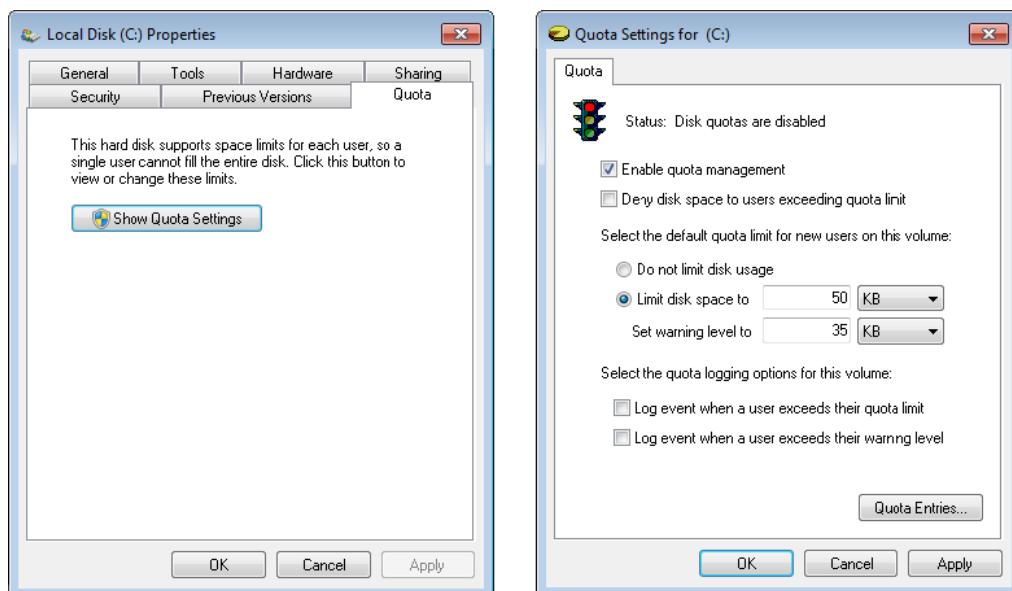


Рис. 12.20. Окна свойств тома и записи квот

Отслеживание связей

В пространстве имен Windows (например, на рабочем столе) можно создавать ярлыки файлов, находящихся в пространстве имен файловой системы. Такие ярлыки активно используются, например в меню Start (Пуск). Аналогичным образом технология связывания и внедрения объектов (Object Linking and Embedding, OLE) позволяет вставлять документы из одних приложений в документы, созданные другими. Этую технологию поддерживают программы пакета Microsoft Office, включая PowerPoint, Excel и Word.

Несмотря на простую возможность соединения файлов друг с другом и с пространством имен, предоставляемую OLE-связями, при перемещении источника ярлыка оболочки или OLE-связи (источником в данном случае называется файл или папка, на которую дается ссылка) возникают проблемы. Файловая система NTFS поддерживает служебные приложения, выполняющие *отслеживание изменившихся связей* (distributed link-tracking). Именно они поддерживают целостность ярлыков и OLE-ссылок при перемещении источника. В результате, если источник ссылки, расположенный на NTFS-тome, перемещается на другой NTFS-том в пределах того же домена, служба отслеживания изменившихся связей обновит ссылку в соответствии с выявленными изменениями.

Отслеживание связей в NTFS базируется на таком необязательном атрибуте файла, как *идентификатор объекта* (object ID). Приложения назначают его файлам при помощи управляющих кодов FSCTL_CREATE_OR_GET_OBJECT_ID (назначает идентификатор, если он еще не назначен) и FSCTL_SET_OBJECT_ID. Для запроса идентификаторов объектов применяются управляющие коды FSCTL_CREATE_OR_GET_OBJECT_ID и FSCTL_GET_OBJECT_ID. Код FSCTL_DELETE_OBJECT_ID позволяет приложениям удалить идентификаторы объектов из файлов.

Шифрование

На компьютерах корпоративных пользователей часто хранится конфиденциальная информация. Данные на серверах компании обычно надежно защищены системой безопасности сети и средствами контроля физического доступа, а вот портативный компьютер может быть потерян или украден, и тогда появляется риск, что данные попадут в чужие руки. В NTFS права доступа к файлам в таких случаях не могут защитить информацию, так как достаточно воспользоваться программой, умеющей читать NTFS-файлы вне среды Windows. Более того, права доступа к NTFS-файлам перестают играть какую-либо роль при доступе с правами администратора из другой копии Windows. В главе 6 части I мы говорили о том, что учетная запись администратора обладает привилегиями владения и резервного копирования, позволяющими получить доступ к любому объекту в обход его параметров защиты.

В NTFS входит средство шифрования файлов под названием EFS (Encrypting File System – шифрующая файловая система). С его помощью пользователи могут шифровать конфиденциальную информацию. Как и механизм сжатия файлов, EFS работает совершенно прозрачно для приложений. То есть при чтении данных приложением, работающим под управлением учетной записи пользователя, они автоматически шифруются и дешифруются при изменении их авторизованным приложением.

ПРИМЕЧАНИЕ

NTFS не допускает шифрования файлов, расположенных в корневом каталоге системного тома или в каталоге \Windows, так как многие из этих файлов требуются в процессе загрузки, когда система EFS еще не активна. Описанный в главе 9 инструмент BitLocker реализует намного более подходящую к подобным средам технологию, так как обеспечивает шифрование всего тома.

В EFS используются криптографические службы, предоставляемые Windows в пользовательском режиме, поэтому эта файловая система состоит из драйвера устройства режима ядра, тесно связанного с NTFS, и DLL-модулями пользовательского режима, которые взаимодействуют с подсистемой локальной аутентификации (Local Security Authority Subsystem, LSASS) и криптографическими DLL-библиотеками.

Доступ к зашифрованным файлам можно получить только с помощью закрытого ключа из криптографической EFS-пары (она состоит из закрытого и открытого ключей). При этом закрытые ключи защищены паролем учетной записи. Соответственно, без пароля учетной записи, авторизованной для просмотра данных, доступ к зашифрованным EFS-файлам на потерянных или украденных портативных компьютерах нельзя получить никакими средствами (кроме криптоаналитических атак путем перебора паролей).

В Windows приложения могут пользоваться API-функциями `EncryptFile` и `DecryptFile` для шифрования и дешифрования файлов, а также функцией `FileEncryptionStatus` для получения EFS-атрибутов файла или папки, позволяющих, к примеру, выяснить, зашифрован ли файл. В этом случае в атрибутах файла присутствует флаг `FILE_ATTRIBUTE_ENCRYPTED`, что позволяет приложениям узнать состояние шифрования файла при помощи функции `GetFileAttributes`.

Поддержка POSIX

Как отмечено в главе 2, одним из требований к Windows была полная поддержка стандарта POSIX 1003.1. Стандарт POSIX требует, чтобы файловая система поддерживала имена файлов и папок, чувствительные к регистру символов, цепочки разрешений (когда для доступа к файлу нужны права на доступ к каждой из папок на пути к нему), меток времени изменения файла (отличающихся от меток времени последней модификации файла в MS-DOS) и жестких ссылок. В NTFS вся эта функциональность реализована.

Дефрагментация

Несмотря на то что NTFS при выделении блоков для увеличения файла пытается сохранить его непрерывность, файлы на томе со временем могут стать фрагментированными, особенно при многократном увеличении или при недостатке свободного пространства. Файл является фрагментированным, если его данные занимают несмежные кластеры. К примеру, на рис. 12.21 представлен фрагментированный файл, состоящий из пяти фрагментов. Но как и большинство файловых систем (в том числе FAT в Windows), NTFS не предпринимает специальных мер по поддержанию непрерывности файлов (это делается с помощью встроенных средств дефрагментации), разве что резервирует область дискового пространства, в которой и хранится *главная таблица файлов*.

(Master File Table, MFT). (NTFS разрешает выделять место под файлы из MFT-зоны при недостатке свободного места на томе.) Выделение свободной области для MFT-зоны может помочь сохранению ее непрерывности, но фрагментация может появиться и там. (Подробно MFT рассматривается далее в одноименном разделе.)

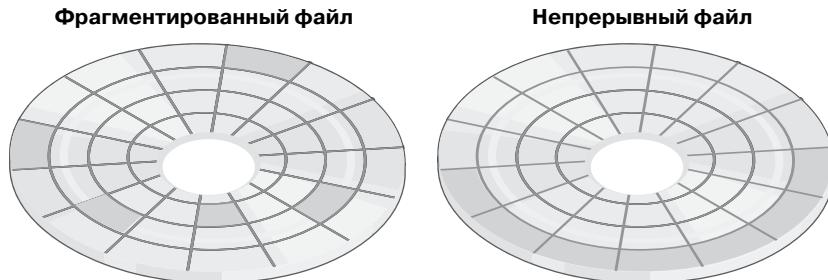


Рис. 12.21. Фрагментированный и непрерывный файлы

Чтобы упростить сторонним производителям разработку инструментов дефрагментации, в Windows включен API-интерфейс дефрагментации, которым данные инструменты могут пользоваться для перемещения файловых данных таким образом, чтобы файл в итоге занял смежные кластеры. Данный API-интерфейс состоит из управляющих кодов файловой системы, позволяющих приложениям получать карту свободных и занятых кластеров (`FSCTL_GET_VOLUME_BITMAP`), получать карту кластеров, занятых файлом (`FSCTL_GET_RETRIEVAL_POINTERS`), и перемещать файл (`FSCTL_MOVE_FILE`).

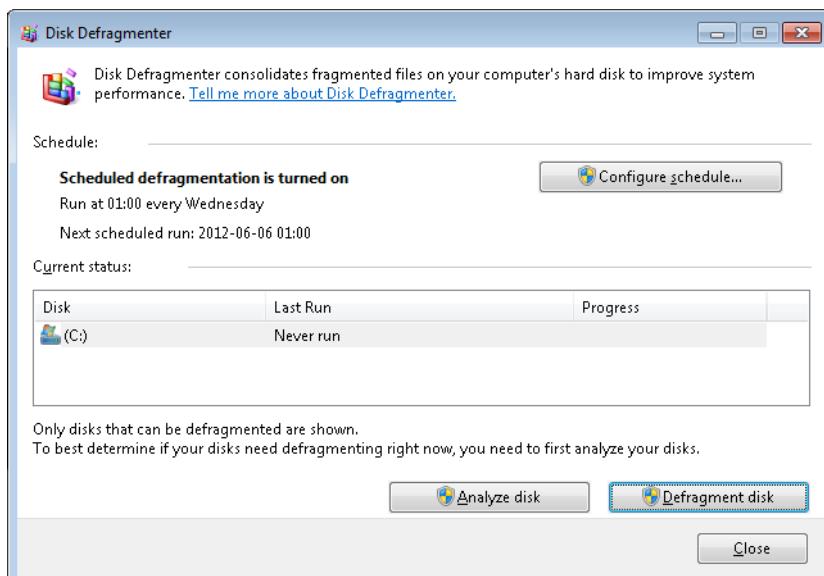


Рис. 12.22. Программа дефрагментации диска

В Windows имеется встроенная программа дефрагментации, доступная через служебную программу Disk Defragmenter (%SystemRoot%\System32\Defrag.exe), окно которой показано на рис. 12.22, а также интерфейс командной строки %SystemRoot%\System32\Defrag.exe, который можно запустить в диалоговом режиме, а можно по заданному расписанию. К сожалению, этот интерфейс не позволяет ни получить подробный отчет, ни управлять процессом дефрагментации, например исключая из него отдельные файлы и папки.

Реализация дефрагментации в NTFS обладает единственным ограничением — нельзя дефрагментировать страничные файлы и файлы NTFS-журналов.

Динамическое разбиение на разделы

NTFS-драйвер дает пользователям возможность динамически менять размеры любого раздела, в том числе системного, уменьшая или увеличивая его. Увеличить раздел, если на диске достаточно места, легко, воспользовавшись управляющим кодом FSCTL_EXPAND_VOLUME. Сложнее уменьшить раздел, так как при этом требуется переместить файловые данные из удаляемых областей в область, которые будут принадлежать разделу после уменьшения (аналогичный механизм применяется при дефрагментации). В этот процесс вовлекаются два компонента: *механизм уплотнения* (shrinking engine) и драйвер файловой системы.

Механизм уплотнения реализуется в режиме пользователя. Он взаимодействует с NTFS, чтобы определить максимальное количество байтов, которыми он сможет воспользоваться, то есть узнает, сколько данных можно перенести в область, которая останется после изменения размера. При этом применяется стандартный механизм дефрагментации, который не поддерживает перенос фрагментов используемых файлов и любых других файлов, помеченных с помощью управляющего кода FSCTL_MARK_HANDLE как не подлежащие перемещению (например, файл спящего режима). Невозможно переместить резервную копию главной таблицы файлов (\$MftMirr), NTFS-журнал транзакций метаданных (\$LogFile) и файл с меткой тома (\$Volume), что ограничивает минимальный размер уплотненного тома и становится причиной нерационального расходования места.

Код уплотнения в драйвере файловой системы отвечает за сохранение целостности тома в процессе уплотнения. Для этого он предоставляет интерфейс, в котором используются три описывающих текущую операцию запроса, отправляемые через управляющий код FSCTL_SHRINK_VOLUME:

- ❑ Запрос ShrinkPrepare должен предшествовать любой другой операции. Он получает желаемый размер нового тома в секторах и использует его таким образом, что файловая система перестает выделять место вне границ нового тома. Запрос ShrinkPrepare не проверяет, можно ли уплотнить том на указанную величину, но гарантирует, что величина уплотнения выражается корректным числом и что параллельно не будут выполняться никакие другие операции уплотнения. Обратите внимание, что после подготовительной операции файловый дескриптор тома связывается с запросом на уплотнение. При закрытии файлового дескриптора операция считается прерванной.
- ❑ Затем следует запрос ShrinkCommit. При этом файловая система пытается удалить количество кластеров, указанное в последнем подготовительном запросе. (Если посыпалось несколько подготовительных запросов с разными вариантами раз-

меров, решающим является последний.) Запрос `ShrinkCommit` предполагает, что механизм уплотнения завершил свою работу, и не выполняется, если в подлежащей уплотнению области остаются выделенные блоки.

- Запрос `ShrinkAbort` посыпается механизмом уплотнения или возникает в ответ на такие события, как, например, закрытие файлового дескриптора тома. Этот запрос отменяет операцию `ShrinkCommit`, возвращая разделу исходное состояние и возможность снова выделять место за пределами области уплотнения. Но при этом никуда не исчезают результаты проведенной механизмом уплотнения дефрагментации.

Если операция уплотнения прерывается перезагрузкой, NTFS возвращает целостность файловой системы через механизм восстановления метаданных, который мы рассмотрим чуть позже. Так как уплотнение не считается законченным, пока не завершатся все остальные операции, том возвращается к исходному размеру и сохраняются только результаты сброшенных на диск операций дефрагментации.

Напоследок упомянем, что уплотнение тома влияет на механизм теневого копирования (подробно он рассматривается в главе 9). Напомним, что механизм создания разностных копий позволяет VSS сохранять только изменившиеся фрагменты файла, оставляя ссылку на исходный файл данных. В случае удаленного файла эти данные будут связаны с пустым пространством, которое, скорее всего, находится в уплотняемой области. Поэтому механизму уплотнения приходится в процессе работы взаимодействовать с VSS. Таким образом, его работа сводится к копированию данных удаленного файла в область хранения резервной копии и увеличению этой области для вставки туда дополнительных данных. Это очень важная деталь, так как она накладывает еще одно ограничение на размер, до которого можно уплотнять тома, даже обладающие достаточным свободным пространством.

Драйвер файловой системы NTFS

Как отмечалось в главе 8, инфраструктура ввода-вывода в Windows устроена так, что NTFS и другие файловые системы представляют собой загружаемые драйверы устройств в режиме ядра. Они вызываются приложениями, использующими Windows или другие API-интерфейсы ввода-вывода (например, POSIX). Как показано на рис. 12.23, подсистемы окружения вызывают системные службы, которые, в свою очередь, находят и вызывают соответствующие загруженные драйверы. (Диспетчери-зация системных служб описывается в главе 3 части I.)

Драйверы разного уровня передают друг другу запросы на ввод-вывод, вызывая диспетчер ввода-вывода исполнительной системы Windows. В данном случае он используется как промежуточное звено, что обеспечивает независимость каждого из драйверов, позволяя загружать и выгружать их без последствий для других драйверов. Кроме того, NTFS-драйвер взаимодействует с тремя другими компонентами исполнительной системы, как показано на рис. 12.24. Эти компоненты тесно связаны с файловыми системами.

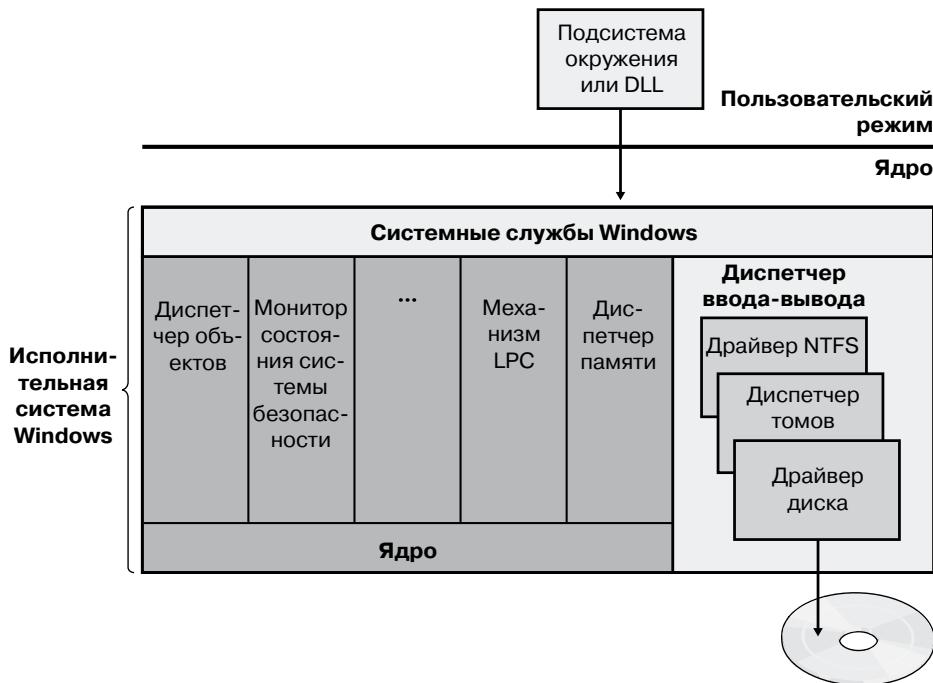


Рис. 12.23. Компоненты подсистемы ввода-вывода в Windows

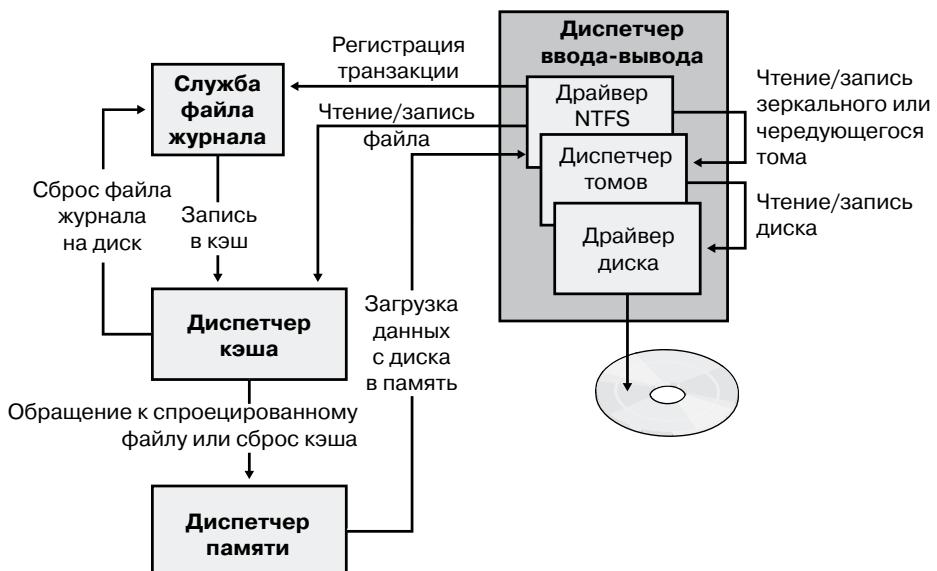


Рис. 12.24. NTFS и сопутствующие компоненты

Служба файла журнала (Log File Service, LFS) является частью файловой системы NTFS, предоставляя функциональность поддержки журнала, в котором фиксируются результаты записи на диск. Записываемый LFS-службой файл служит для восстановления отформатированных для NTFS томов в случае системного сбоя. (Эта тема подробно раскрывается в разделе «Служба файла журнала».)

Такой компонент исполнительной системы Windows, как диспетчер кэша, предоставляет общесистемные службы кэширования для NTFS-драйвера и драйверов других файловых систем, в том числе сетевых (то есть серверов и редиректоров). Все реализованные в Windows файловые системы получают доступ к кэшированным файлам, отображая их на адресное пространство системы и считывая соответствующие участки виртуальной памяти. Для этой цели диспетчер кэша снабжает диспетчер памяти специализированным интерфейсом файловых систем. Когда программа пытается обратиться к какой-либо части файла, которая не загружена в кэш, то есть имеет место *кэш-промах* (cache miss), диспетчер памяти вызывает NTFS для обращения к драйверу диска и получения с диска содержимого файла. Диспетчер кэша оптимизирует дисковый ввод-вывод, вызывая для сброса на диск содержимого кэша в фоновом режиме (асинхронная запись на диск) диспетчер памяти из программных потоков отложенной записи. (Полное описание диспетчера кэша вы найдете в главе 11.)

NTFS участвует в Windows-модели объектов, реализуя файлы в виде объектов. Эта реализация допускает совместное использование файлов и их защиту диспетчером объектов — компонентом, который управляет всеми объектами уровня исполнительной системы. (Диспетчуру объектов посвящен одноименный раздел в главе 3 части I.)

Приложение создает файлы и обращается к ним так же, как и к любым другим объектам в Windows: через дескрипторы объектов. К моменту, когда запрос на ввод-вывод достигает NTFS, диспетчер объектов и система безопасности уже видят, что вызывающий процесс имеет право на доступ к файловому объекту выбранным им способом. Чтобы убедиться в этом, система безопасности сравнивает маркер доступа вызывающего процесса с записями в списке контроля доступа для данного файлового объекта. (Списки контроля доступа рассматриваются в главе 6 части I.) Диспетчер ввода-вывода при этом преобразует дескриптор файла в указатель на файловый объект. Эту информацию из файлового объекта NTFS и использует для обращения к файлу на диске.

На рис. 12.25 представлены структуры данных, связывающие дескриптор файла со структурой файловой системы на диске.

Для перехода от файлового объекта к файлу на диске NTFS следует нескольким указателям. Как показано на рисунке, файловый объект, представляющий собой обычный вызов системной службы открытия файла, указывает на *блок управления потоком данных* (Stream Control Block, SCB) атрибута файла, который поток пытается считать или записать. На рисунке процесс открывает для файла одновременно безымянный атрибут данных и именованный поток данных. Блок управления потоком данных предоставляет индивидуальные атрибуты файла и содержит информацию о том, как обнаружить в файле определенный атрибут. Все SCB-блоки для файла указывают на общую структуру данных, которая называется *блоком управления файлом* (File Control Block, FCB). FCB содержит указатель (на самом деле, как показано в разделе «Индексы файловых записей», это индекс) на файловую запись в главной файловой таблице (MFT), которая подробно описывается в следующем разделе.

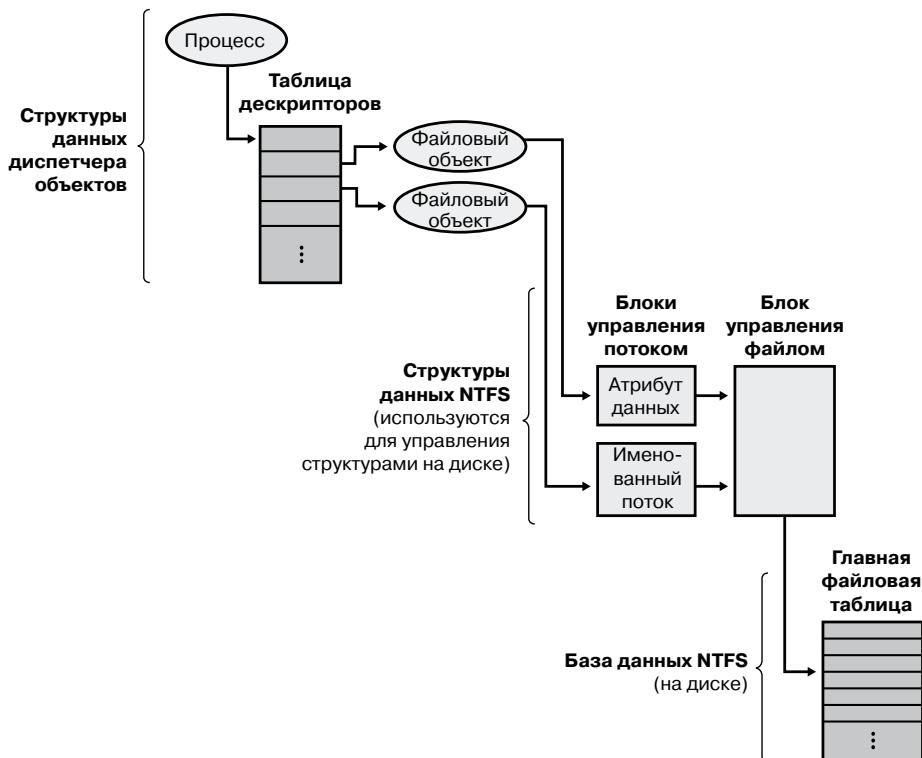


Рис. 12.25. Структуры данных в NTFS

NTFS-структура на диске

В этом разделе мы рассмотрим NTFS-строктуру тома на диске, а также способы разбиения дискового пространства и его организации в кластеры, принципы хранения на диске реальных файловых данных и информации об атрибутах, а также механизм сжатия (компрессии) данных в NTFS.

Тома

NTFS-структура начинается с тома. *Том* (volume) соответствует логическому разделу на диске и создается при форматировании диска или его части для NTFS. Оснастка *Disk Management* (Управление дисками) консоли MMC позволяет создать RAID-том, охватывающий несколько дисков. Также для этого вы можете воспользоваться командой *diskpart* (%SystemRoot%\System32\Diskpart.exe).

На диске может быть как один, так и несколько томов. Каждый том обрабатывается NTFS независимо от других. Три примера конфигурации жесткого диска объемом 150 Гбайт представлены на рис. 12.26.

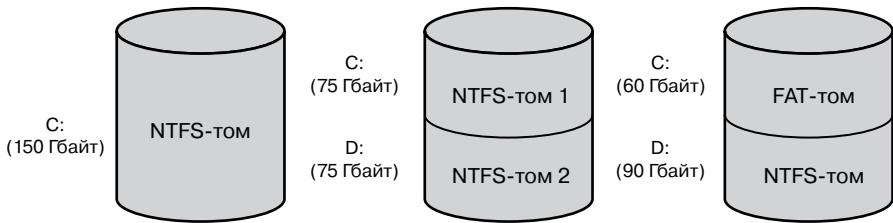


Рис. 12.26. Примеры конфигурации диска

Том состоит из набора файлов и оставшегося в разделе свободного пространства. В FAT том также содержит области, специально отформатированные для использования файловой системой. Однако в NTFS-томе все данные файловой системы, такие как битовые карты, папки и даже начальный загрузочный код, хранятся как обычные файлы.

ПРИМЕЧАНИЕ

Дисковый формат NTFS-томов в Windows 7 и Windows Server 2008 R2 относится к версии 3.1, которая используется со временем Windows XP и Windows Server 2003. Номер версии тома хранится в файле метаданных \$Volume.

Кластеры

Размер кластера на NTFS-тome, или *кластерный множитель* (cluster factor), задается при формировании тома командой `format` или через оснастку *Disk Management* (Управление дисками) консоли MMC. По умолчанию размер кластера зависит от размера тома, но всегда содержит целое число физических секторов, являющееся степенью двойки (1 сектор, 2 сектора, 4 сектора, 8 секторов и т. д.). Кластерный множитель выражается числом байтов в кластере, например 512 байт, 1 Кбайт, 2 Кбайт и т. д.

Внутренне NTFS имеет дело только с кластерами. (Но NTFS инициирует низкоуровневые операции ввода-вывода на томе, выравнивая передаваемые данные по размеру сектора и подгоняя их объем под значение, кратное размеру секторов.) NTFS использует кластер как единицу выделяемого пространства с целью обеспечения независимости от размера физического сектора. Это позволяет NTFS эффективно поддерживать очень большие диски, используя кластеры большего размера, а также недавно появившиеся диски, размер секторов которых отличается от 512 байт. (О них речь идет в главе 9.) Применение больших кластеров на больших томах снижает фрагментацию и ускоряет выделение свободного места за счет небольшой потери в эффективности расходования дискового пространства. (Если размер кластера составляет 4096 байт, а в файле только 1024, потеря составляет 3072 байт. О размерах кластеров, используемых по умолчанию, также рассказывается в главе 9.) Как команда `format` командной строки, так и команда Format (Форматировать) меню, для доступа к которой нужно раскрыть подменю All Tasks (Все задачи) меню Action (Действие) консоли *Disk Management* (Управление диском), по умолчанию выбирают кластерный множитель в зависимости от размера тома, но это значение можно поменять.

NTFS ссылается на конкретные места диска через *логические номера кластеров* (Logical Cluster Numbers, LCN). Для этого все кластеры на томе нумеруются по порядку, от начала до конца. Для преобразования LCN-номера в физический адрес на диске NTFS умножает его на кластерный множитель и получает байтовое смещение от начала тома, которое требует интерфейс драйвера диска. На данные внутри файла NTFS ссылается по *виртуальным номерам кластеров* (Virtual Cluster Numbers, VCN), присваивая кластерам конкретного файла номера от 0 до m . Виртуальные номера кластеров не обязательно должны быть физически непрерывными, но должны отображаться на любой LCN-номер на томе.

Главная таблица файлов

В NTFS все хранящиеся на томе данные содержатся в файлах. Это касается и структур данных, выделенных для поиска и выборки файлов, начального загрузочного кода и битовой карты, в которой регистрируется состояние пространства всего тома (NTFS-метаданные). Такой тип хранения позволяет системе легко находить и поддерживать данные, а также использовать для файлов дескрипторы безопасности. Кроме того, при появлении на диске битых секторов NTFS может переместить файлы метаданных, сохранив тем самым доступ к диску.

Центральное место в структуре NTFS-тома занимает главная таблица файлов (MFT). Она реализована как массив записей о файлах. Размер каждой записи фиксирован и равен 1 Кбайт независимо от размера кластера. (Структура этих записей обсуждается в разделе «Файловые записи».) Логично, что MFT содержит по одной строке на каждый файл тома, в том числе есть строка и для самой MFT-таблицы. В дополнение к MFT на каждом NTFS-томе имеется набор файлов метаданных с необходимой для реализации структуры файловой системы информацией. Имена всех файлов с метаданными начинаются со знака доллара (\$) и являются скрытыми. К примеру, имя файла для MFT — \$MFT. Остальные файлы NTFS-тома являются обычными файлами и папками, как показано на рис. 12.27.

Обычно каждая MFT-запись соответствуетциальному файлу. Но если файл обладает множеством атрибутов или сильно фрагментирован, для него может потребоваться большее количество записей. В этом случае первая MFT-запись, хранящая адреса других записей, называется *базовой файловой записью* (base file record).

При первом обращении к NTFS-тому файловая система должна его смонтировать, то есть считать с диска метаданные и сформировать внутренние структуры данных, необходимые для обработки обращений приложений к файловой системе. Чтобы смонтировать том, NTFS ищет в загрузочной записи тома (VBR), которая имеет нулевой LCN-номер, физический MFT-адрес на диске. VBR содержит такую структуру данных, как блок параметров загрузки (Boot Parameter Block, BPB). Запись о самой MFT-таблице располагается в таблице первой; вторая запись указывает на файл в середине диска (\$MFTMirr), который называется *MFT-зеркалом* (MFT mirror) и содержит копию первых четырех строк MFT-таблицы. Эта частичная копия MFT-таблицы служит для поиска файлов с метаданными в случае, когда по каким-то причинам часть MFT-файла прочитать невозможно.

0	\$MFT - MFT
1	\$MFTMirr - Зеркальная копия MFT
2	\$LogFile - Файл журнала
3	\$Volume - Файл тома
4	\$AttrDef - Таблица определения атрибутов
5	\ - Корневой каталог
6	\$BitMap - Файл распределения кластеров тома
7	\$Boot - Загрузочный сектор
8	\$BadClus - Файл поврежденных кластеров
9	\$Secure - Файл параметров безопасности
10	\$UpCase - Преобразование в прописные буквы
11	\$Extend - Каталог расширенных метаданных
12	Не используется
13	Не используется
23	Не используется
24	\$Extend\\$Quota - Информация о квотах
25	\$Extend\\$ObjId - Информ. об отслеж. измен. связей
26	\$Extend\\$Reparse - Обратн. ссылки на точки повт. обр.
27	\$Extend\\$RmMetadata - Папка метаданных DR
28	\$Extend\\$RmMetadata\\$Repair - Информ. о восст. DR
29	\$Extend\\$RmMetadata\\$TxfLog - Папка журнала TxF
30	\$Extend\\$RmMetadata\\$Txf - Папка метаданных TxF
31	\$Extend\\$RmMetadata\\$TxfLog\\$Tops - TOPS-файл
32	\$Extend\\$RmMetadata\\$TxfLog\\$TxfLog.blf - TxF BLF
33	\$TxfLogContainer00000000000000000000000000000001
34	\$TxfLogContainer00000000000000000000000000000002

Зарезервировано
для файлов
метаданных NTFS

Рис. 12.27. MFT-записи для файлов метаданных в NTFS

После того как NTFS находит запись для MFT-таблицы, файловая система получает из ее атрибута данных сведения об отображении VCN на LCN и сохраняет их в памяти. В каждом *отрезке* (run) хранится отображение VCN на LCN и длина отрезка, так как этой информации достаточно, чтобы найти LCN-номер для любого VCN-номера. (Отрезки рассматриваются в разделе «Резидентные и нерезидентные атрибуты».) Именно по ней NTFS локализует на диске отрезки, содержащие MFT. Затем NTFS обрабатывает MFT-записи для еще нескольких файлов метаданных и открывает эти файлы. Наконец, NTFS выполняет операцию восстановления файловой системы (см. далее раздел «Восстановление») и открывает остальные файлы метаданных. После этого пользователь может обращаться к тому.

ПРИМЕЧАНИЕ

Для простоты в тексте и на схемах в этой главе отрезок обозначается как сущность, содержащая VCN, LCN и длину. На самом деле NTFS сжимает эту информацию на диске в пары «LCN-номер/следующий VCN-номер». Зная начальный VCN-номер, NTFS может определить длину отрезка, просто вычтя это значение из следующего VCN-номера.

В процессе работы NTFS ведет запись в другой важный файл метаданных, *файл журнала* (log file), называемый \$LogFile. NTFS использует его для регистрации всех операций, влияющих на структуру NTFS-тома. К ним относятся, например, создание файлов и выполнение команд, влияющих на структуру папок, таких как copy. Файл журнала используется и при восстановлении NTFS-тома после краха системы, как описывается в разделе «Восстановление».

Еще одна запись в MFT зарезервирована для корневого каталога, обозначаемого как обратный слэш, например C:\. Она содержит индекс файлов и папок, хранящихся в корне NTFS-структуре каталогов. Когда NTFS в первый раз получает запрос на открытие файла, она начинает его поиск с записи корневого каталога. Открыв файл, NTFS сохраняет его номер записи в MFT, чтобы в следующий раз, когда потребуется осуществить чтение из этого файла или запись в него, можно было напрямую обратиться к его записи в MFT.

NTFS регистрирует распределение пространства тома в *файле битовой карты* (bitmap file), который называется \$BitMap. Атрибут данных для файла битовой карты содержит карту, каждый бит которой представляет собой кластер тома и идентифицирует его как свободный или выделенный.

В *файле безопасности* (security file), который называется \$Secure, хранится база данных дескрипторов безопасности, действительных на данном томе. Файлы и папки в NTFS имеют отдельно настраиваемые дескрипторы безопасности, но для экономии места NTFS хранит эти параметры в одном файле, что позволяет файлам и папкам с одинаковыми параметрами ссылаться на один и тот же дескриптор. В большинстве сред одинаковые параметры безопасности имеют целые деревья папок, и там подобная оптимизация позволяет существенно экономить место на диске.

Другой системный файл — это *загрузочный файл* (boot file) с именем \$Boot. Если мы находимся на системном томе, там хранится код начальной загрузки Windows. В остальных случаях там хранится код, выводящий сообщение об ошибке при попытках грузиться с тома. Для успешной загрузки код должен находиться в определенном месте диска, чтобы его могла обнаружить система BIOS. В процессе форматирования команда **format** определяет эту область как файл, создавая для нее запись в MFT. Все упомянутые в MFT файлы и все кластеры либо свободны, либо выделены под файлы — в NTFS нет скрытых файлов или кластеров, хотя некоторые файлы (метаданные) пользователям не видны. Загрузочный файл и файлы метаданных могут быть защищены отдельными дескрипторами безопасности, применимыми ко всем Windows-объектам. Модель «все, что присутствует на диске, является файлом» означает, что код начальной загрузки может быть изменен путем обычного ввода-вывода, хотя загрузочный файл и защищен от редактирования.

Кроме того, NTFS поддерживает *файл поврежденных кластеров* (bad-cluster file), который называется \$BadClus и содержит информацию обо всех поврежденных участ-

ках дискового тома, а также *файл тома* (volume file), который называется \$Volume и содержит имя тома, версию NTFS и номер битового флага, отмечающий состояние тома. Например, это бит, который указывает на повреждение тома и необходимость восстановить его служебной программой Chkdsk. (О ней мы поговорим чуть позже.) *Файл отображения имен на буквы верхнего регистра* (uppercase file) называется \$UpCase и содержит таблицу соответствия между символами верхнего и нижнего регистров. Поддерживается в NTFS и файл, содержащий *таблицу определения атрибутов* (attribute definition table); он называется \$AttrDef и задает типы поддерживаемых томом атрибутов, указывая, являются ли они индексируемыми, следует ли их восстанавливать в ходе операции восстановления системы и т. п.

Некоторые файлы метаданных NTFS хранят в папке расширенных метаданных \$Extend. В том числе там находятся: *файл идентификаторов объектов* (object identifier file) с именем \$ObjId, *файл квот* (quota file) с именем \$Quota, *файл журнала регистрации изменений* (change journal file) с именем \$UsnJrnl, *файл точек повторной обработки* (reparse point file) с именем \$Reparse и *используемая диспетчером ресурсов по умолчанию папка* (default resource manager directory) с именем \$RmMetadata. В этих файлах хранится информация, относящаяся к нетривиальным возможностям NTFS. Файл идентификатора объектов хранит идентификаторы файловых объектов, файл квот — величины квот и поведения томов, на которых активированы квоты, в файле журнала изменений регистрируются изменения в файлах и папках, а файл точек повторной обработки хранит список файлов и папок тома, содержащего эти точки.

Папка, по умолчанию используемая диспетчером ресурсов, содержит папки, связанные с поддержкой транзакционной версии NTFS (TxF). Среди них *папка журнала транзакций* (transaction log directory) с именем \$TxfLog, *папка изоляции транзакций* (transaction isolation directory) с именем \$Txf и *папка восстановления транзакций* (transaction repair directory) с именем \$Repair. Папка журнала транзакций содержит *файл базового TxF- журнала* (TxF base log file) с именем \$TxfLog.blf и файлы-контейнеры, количество которых зависит от размера журнала. При этом контейнеров должно быть как минимум два: один для потока данных журнала диспетчера транзакций ядра (Kernel Transaction Manager, KTM) — этот файл носит имя \$TxfLogContainer00000000000000000000000000000001, а второй для потока TxF- журнала — его файл называется \$TxfLogContainer00000000000000000000000000000002. В папке журнала транзакций также находится *поток данных старой TxF- страницы* (TxF old page stream), файл которого называется \$Tops и о котором мы поговорим позже.

ЭКСПЕРИМЕНТ: ПРОСМОТР NTFS-ИНФОРМАЦИИ

Для просмотра сведений о NTFS-томе, включая данные о расположении и размере области MFT-таблицы, можно воспользоваться встроенной утилитой командной строки Fsutil.exe:

```
C:\>fsutil fsinfo ntfsinfo c:
NTFS Volume Serial Number : 0x9a38d50e38d4ea71
Version : 3.1
Number Sectors : 0x0000000015c82ff0
Total Clusters : 0x00000000002b905fe
Free Clusters : 0x000000000013c332
```

```
Total Reserved : 0x00000000000000780
Bytes Per Sector : 512
Bytes Per Cluster : 4096
Bytes Per FileRecord Segment : 1024
Clusters Per FileRecord Segment : 0
Mft Valid Data Length : 0x0000000023db0000
Mft Start Lcn : 0x00000000000c0000
Mft2 Start Lcn : 0x0000000016082ff
Mft Zone Start : 0x000000002751f60
Mft Zone End : 0x00000000275cd60
RM Identifier: CF7234E7-39E3-11DC-BDCE-00188BDD5F49
```

Индексы файловых записей

Файл на NTFS-тome идентифицируется 64-разрядным значением, которое называется *индексом файловой записи* (file record number) и состоит из номера файла и номера последовательности. Номер файла соответствует позиции его записи в MFT минус 1 (для файлов, требующих несколько записей, это позиция базовой записи в MFT минус 1). Номер последовательности увеличивается на единицу при каждом новом использовании позиции записи в MFT. Он позволяет NTFS выполнять проверку внутренней целостности. Индекс файловой записи иллюстрирует рис. 12.28.



Рис. 12.28. Индекс файловой записи

Файловые записи

В NTFS файлы рассматриваются не просто как хранилище текстовых или двоичных данных, а как совокупность пар атрибутов и их значений, одна из которых содержит данные файла, — соответствующий атрибут называется *неименованным атрибутом данных* (unnamed data attribute). К остальным атрибутам относятся имя файла, метка времени и, возможно, дополнительные именованные атрибуты данных. MFT-запись для небольшого файла показана на рис. 12.29.

Каждый атрибут в файле хранится как отдельный поток байтов. Строго говоря, NTFS читает и записывает не файлы, а потоки атрибутов. В NTFS поддерживаются следующие операции с атрибутами: создание, удаление, чтение (как диапазон байтов) и запись (как диапазон байтов). Службы чтения и записи обычно имеют дело с неименованным атрибутом данных. Но вызывающая программа может указать другой атрибут, используя синтаксис именованных потоков данных.

В табл. 12.6 перечислены атрибуты файлов на NTFS-тome. (Не все они есть у произвольно взятого файла.)



Рис. 12.29. MFT-запись для небольшого файла

Хотя в табл. 12.6 представлены имена атрибутов, на самом деле атрибуты соответствуют числовым кодам типов, которыми NTFS пользуется для упорядочивания атрибутов в файловой записи. Файловые атрибуты в MFT-записи располагаются в порядке возрастания числовых значений кодов. При этом некоторые типы атрибутов могут встречаться более одного раза, например, если файл обладает множеством атрибутов или набором имен. Все возможные типы атрибутов (и их имена) перечислены в файле метаданных \$AttrDef.

Таблица 12.6. Атрибуты файлов в NTFS

Атрибут	Тип атрибута	Резидентность	Описание
Информация о томе	\$VOLUME_INFORMATION, \$VOLUME_NAME	Всегда	Эти атрибуты присутствуют только в файле метаданных \$Volume. Они содержат сведения о версии и метке тома
Стандартная информация	\$STANDARD_INFORMATION	Всегда	Такие атрибуты, как «только для чтения», «архивный» и т. п.; метки времени, включая время создания и последнего изменения файла
Имя файла	\$FILE_NAME	Возможна	Имя файла в виде символов Unicode 1.0. У файла может быть несколько атрибутов имени, например при наличии жесткой ссылки на него или если для файла с длинным именем было автоматически сгенерировано короткое имя для доступа приложений MS-DOS и 16-разрядной версии Windows

Атрибут	Тип атрибута	Резидентность	Описание
Дескриптор безопасности	\$SECURITY_DESCRIPTOR	Возможна	Этот атрибут служит для обратной совместимости с предыдущими версиями NTFS и в текущей версии NTFS (3.1) используется редко. NTFS хранит практически все дескрипторы безопасности в файле метаданных \$Secure, предоставляя их в пользование всем файлам и папкам с одинаковыми параметрами. Предыдущие версии NTFS хранили в каждом файле и в каждой папке закрытую информацию дескриптора безопасности. Некоторые файлы до сих пор обладают атрибутом \$SECURITY_DESCRIPTOR, например файл \$Boot
Данные	\$DATA		Содержимое файла. В NTFS файл по умолчанию имеет один неименованный атрибут данных. Кроме этого, он может обладать набором дополнительных именованных атрибутов, то есть у файла может быть несколько потоков данных. У папок отсутствует атрибут по умолчанию, но они могут обладать необязательными именованными атрибутами данных
Корень индекса, размещение индексов и битовая карта индексов	\$INDEX_ROOT, \$INDEX_ALLOCATION, \$BITMAP	Всегда, никогда, возможна	Эти три атрибута служат для реализации структуры данных Б-дерева, используемого в папках, системе безопасности, квотах и других файлах метаданных
Список атрибутов	\$ATTRIBUTE_LIST	Возможна	Список атрибутов, из которых состоит файл и индексы файловых записей в MFT, в которых находится каждый из атрибутов. Этот атрибут имеется у файлов, которым требуется более одной записи в MFT
Идентификатор объекта	\$OBJECT_ID	Всегда	16-разрядный глобальный уникальный идентификатор (GUID) файла или папки. Службы отслеживания связей назначают идентификаторы объектов ярлыкам оболочки и файлам-источникам OLE-связей. NTFS предоставляет API для открытия файлов и папок не по именам, а по идентификаторам объектов

продолжение ↴

Таблица 12.6 (продолжение)

Атрибут	Тип атрибута	Резидентность	Описание
Информация повторной обработки	\$REPARSE_POINT	Возможна	Этот атрибут хранит данные точки повторной обработки для файла. Кроме того, он присутствует в точках соединения и монтирования
Расширенные атрибуты	\$EA, \$EA_INFORMATION	Возможна, всегда	К расширенным атрибутам относятся пары имя/значение, которые обычно не используются, а нужны только для обратной совместимости с приложениями OS/2
EFS-информация	\$LOGGED.Utility_Stream	Возможна	В этом атрибуте EFS хранит данные, используемые для управления шифрованием файла, например зашифрованной версией ключа, необходимого для дешифрования файла, и списком пользователей, имеющих право доступа. Если файлы или папки принимают участие в транзакции, то TxF в атрибуте \$TxF_DATA сохраняет данные транзакции, такие как уникальный идентификатор транзакции

Каждый атрибут в файловой записи идентифицируется кодом типа атрибута, обладает значением и необязательным именем. Значение атрибута представляет собой поток байтов. К примеру, значением атрибута \$FILE_NAME является имя файла, а значение атрибута \$DATA представляет собой произвольные байты, сохраненные пользователем в файле.

Большинство атрибутов не обладают именами, хотя у атрибутов, связанных с индексом, а также атрибута \$DATA они обычно есть. Именно они позволяют различать атрибуты файла, относящиеся к одному типу. К примеру, файл с именованным потоком данных имеет два атрибута \$DATA: неименованный атрибут \$DATA, хранящий присутствующий по умолчанию неименованный поток данных, и именованный атрибут \$DATA, имя которого совпадает с именем дополнительного потока и который хранит данные этого потока.

Имена файлов

Как NTFS, так и FAT допускают для имен файлов в составе пути длину до 255 символов. Эти имена могут содержать Unicode-символы, точки и пробелы. При этом в FAT для MS-DOS длина имен файлов ограничена восемью символами (неUnicode), за которыми следует точка и состоящее из трех символов расширение. Рисунок 12.30 иллюстрирует поддерживаемые Windows пространства имен файлов и области их перекрытия.

Самого большого пространства имен среди всех сред выполнения приложений, поддерживаемых Windows, требует подсистема POSIX. И поэтому пространства

имен в NTFS и POSIX эквивалентны. Подсистема POSIX способна создавать имена, невидимые для Windows- и DOS-приложений, в том числе имена, заканчивающиеся точкой или пробелом. Обычно создание файла в большом POSIX-пространстве имен не приводит к проблемам, потому что такие файлы потом используются в подсистеме POSIX или в системе ее клиентов.

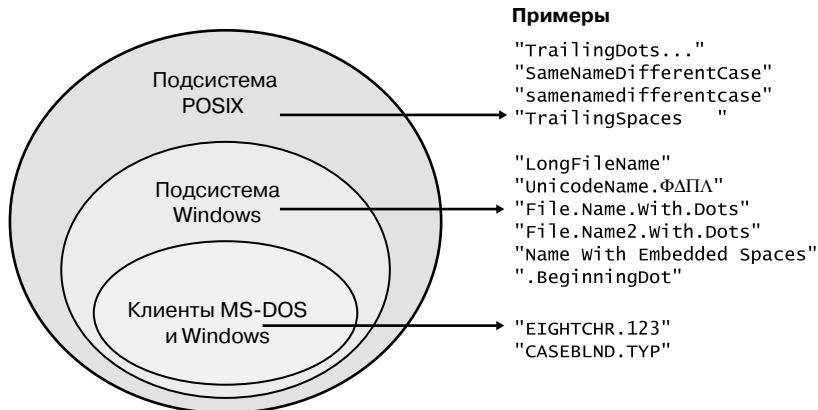


Рис. 12.30. Пространства имен файлов в Windows

Связь между 32-разрядными Windows-приложениями, DOS-программами и 16-разрядными Windows-приложениями намного теснее. Область, отведенная Windows на рис. 12.30, представляет имена файлов, которые подсистема Windows может создавать на NTFS-тome, при этом DOS-приложения и 16-разрядные Windows-приложения их не видят. В эту группу входят имена файлов, превышающие принятый в MS-DOS формат «8.3», имена с Unicode-символами (международные), имена с несколькими точками и начинающиеся с точки, а также имена с внутренними пробелами. При создании файла с таким именем NTFS автоматически генерирует для него альтернативное имя в стиле MS-DOS. Windows выводит эти короткие имена на экран, когда команда `dir` используется с параметром `/x`.

Имена файлов в MS-DOS являются полнофункциональными псевдонимами NTFS-файлов и хранятся в одной папке с длинными именами. MFT-запись для файла с автоматически сгенерированным DOS-именем показана на рис. 12.31.

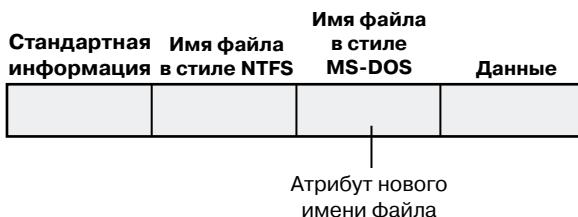


Рис. 12.31. MFT-запись с атрибутом имени файла для MS-DOS

NTFS-имя и имя, сгенерированное для MS-DOS, хранятся в одной и той же файловой записи, а значит, относятся к одному файлу. DOS-имя может применяться для открытия, чтения и копирования файла, а также для записи в него. Если пользователь меняет длинное или короткое имя файла, новое имя заменяет оба существовавших варианта. Если новое имя окажется недопустимым для MS-DOS, NTFS генерирует другой вариант DOS-имени (следует заметить, что NTFS генерирует имена в стиле MS-DOS только для первого имени файла).

ПРИМЕЧАНИЕ

Жесткие ссылки реализуются аналогичным образом. При создании жесткой ссылки на файл NTFS добавляет в MFT-запись еще один атрибут имени файла. Но эти ситуации имеют одно отличие. Когда пользователь удаляет файл, имеющий много имен (жестких ссылок), файловая запись и сам файл никуда не исчезают. Для его удаления требуется удалить все жесткие ссылки. Если же файл обладает NTFS-именем и автоматически сгенерированным DOS-именем, его можно удалить, используя любое из имен.

Вот алгоритм, которым NTFS пользуется для генерации короткого DOS-имени из длинного имени файла (он реализуется в функции `RtlGenerate8dot3Name` ядра и применяется еще и драйверами таких файловых систем, как CDFS и FAT, а также файловых систем сторонних производителей):

1. Удаляем из длинного имени все символы, недопустимые в DOS-именах, включая пробелы и Unicode-символы. Удаляем точки в начале и в конце имени. Удаляем все внутренние точки, кроме последней.
2. Урезаем часть строки перед точкой (если она есть) до шести символов. Впрочем, к этому моменту строка уже может состоять из шести и менее символов, потому что алгоритм начинает работать, когда в имени еще присутствуют недопустимые для MS-DOS символы. Если в строке осталось два символа или меньше, генерируем и добавляем строку контрольной суммы из четырех шестнадцатеричных символов. Затем добавляем строку вида «~*n*» (здесь *n* – номер начиная с 1, позволяющий различать файлы, урезание имен которых дает одинаковый результат). Строку после точки (если точка есть) урезаем до трех символов.
3. Полученный набор символов преобразуем к верхнему регистру. Операционная система MS-DOS нечувствительна к регистру символов, но данный шаг гарантирует, что NTFS не генерирует имя, отличающееся от старого только регистром.
4. Если сгенерированное имя является дубликатом имени, уже присутствующего в папке, увеличиваем на единицу строку «~*n*». При *n* больше 4 (и еще не присоединенной контрольной сумме) строку перед точкой обрезаем до двух символов, после чего генерируем и присоединим строку контрольной суммы из четырех шестнадцатеричных символов.

В табл. 12.7 представлены длинные имена, показанные на рис. 12.30, и их DOS-версии, сгенерированные в NTFS. Приведенный алгоритм и примеры на рис. 12.30 должны дать вам представление о том, как выглядят эти имена.

Таблица 12.7. Имена файлов, сгенерированные в NTFS

Длинные Windows-имена	Короткие имена, генерируемые в NTFS
LongFileName	LONGFI~1
UnicodeName. D\P\	UNICOD~1
File.Name.With.Dots	FILENA~1.DOT
File.Name2.With.Dots	FILENA~2.DOT
File.Name3.With.Dots	FILENA~3.DOT
File.Name4.With.Dots	FILENA~4.DOT
File.Name5.With.Dots	FIF596~1.DOT
Name With Embedded Spaces	NAMEWI~1
.BeginningDot	BEGINN~1
25€.two characters	255440~1.TWO
©	6E2D~1

ПРИМЕЧАНИЕ

Режим генерации коротких имен можно отключить, присвоив параметру HKLM\SYSTEM\CurrentControlSet\Control\FileSystem\NtfsDisable8dot3NameCreation реестра равное 1 значение DWORD и перезагрузив компьютер. Обычно это делать не рекомендуется из-за вероятной несовместимости с приложениями, которые работают с короткими именами.

ТУННЕЛИРОВАНИЕ

Концепция туннелирования (tunneling) используется в NTFS для совместимости с более старыми версиями программ, для которых файловая система в течение некоторого периода должна кэшировать определенные файлы метаданных даже после исчезновения файла, например его удаления или переименования. Благодаря туннелированию любой новый файл, в определенный период созданный под именем исходного, сохранит часть старых метаданных. Идея состоит в том, чтобы воспроизвести поведение DOS-программ при программировании с безопасным сохранением (safe save), когда модифицируемые данные копируются во временный файл, исходный файл удаляется, а временному присваивается его имя. При этом ожидается, что переименованный временный файл будет идентичен исходному, в противном случае время создания будет обновляться с каждой модификацией (для этого собственно и служит время модификации).

В NTFS благодаря туннелированию при удалении файла из папки его длинное и короткое имена, а также время создания сохраняются в кэше. При добавлении в папку нового файла кэш проверяется на наличие туннелированных данных, которые нужно восстановить. Так как эти операции применяются к папкам, экземпляр каждой из них обладает собственным кэшем, который удаляется при удалении самой папки. В NTFS туннелирование используется для следующих пар операций, в которых имя файла сначала удаляется, а потом создается снова:

- удаление плюс создание;
- удаление плюс переименование;
- переименование плюс создание;
- переименование плюс переименование.

По умолчанию NTFS хранит кэш туннелирования в течение 15 секунд, но время хранения можно поменять, отредактировав значение MaximumTunnelEntryAgeInSeconds в разделе HKLM\SYSTEM\CurrentControlSet\Control\FileSystem реестра. Режим туннелирования можно отключить, создав новое значение MaximumTunnelEntries и присвоив ему нулевое значение. Но это отключение может привести к несовместимости и прекращению работы старых приложений.

Увидеть туннелирование в действии можно при помощи несложного эксперимента с командной строкой:

1. Создайте файл с именем file1.
2. Подождите более 15 секунд (время ожидания кэша туннелирования, предлагаемое по умолчанию).
3. Создайте файл с именем file2.
4. Выполните команду dir /TC. Обратите внимание на время создания файлов.
5. Присвойте файлу file1 имя file.
6. Присвойте файлу file2 имя file1.
7. Выполните команду dir /TC. Обратите внимание, что теперь время создания файлов стало одинаковым.

Резидентные и нерезидентные атрибуты

Если файл имеет небольшой размер, все его атрибуты и их значения (например, содержащиеся в файле данные) поместятся в описывающей этот файл записи. Когда значение атрибута хранится непосредственно в MFT (в основной записи или в дополнительной записи, расположенной где-то внутри MFT), атрибут называется *резидентным* (resident). (К примеру, на рис. 12.31 резидентными являются все атрибуты.) Некоторые атрибуты всегда резидентны, по ним NTFS находит нерезидентные атрибуты. К этой категории относятся, например, атрибуты стандартной информации и корня индекса.

Каждый атрибут начинается со стандартного заголовка, содержащего сведения о нем, — с их помощью NTFS осуществляет базовое управление атрибутами. В заголовке, который всегда является резидентным, регистрируется факт резидентности и нерезидентности значения атрибута. Заголовок резидентных атрибутов содержит также смещение значения атрибута от начала заголовка и длину этого значения. Рисунок 12.32 иллюстрирует этот принцип для атрибута имени файла.

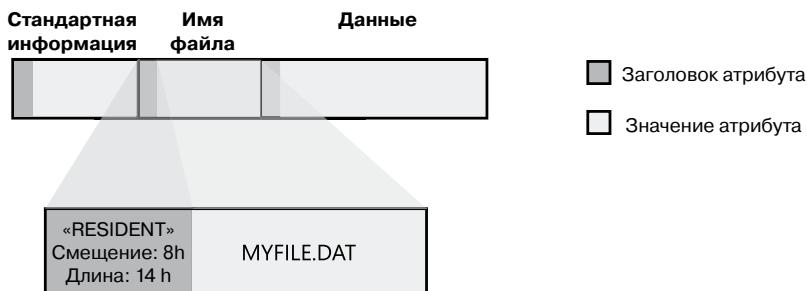


Рис. 12.32. Заголовок и значение резидентного атрибута

Когда значение атрибута хранится непосредственно в MFT, обращение к нему NTFS занимает намного меньше времени. Вместо поиска файла в таблице и считывания цепочки кластеров для нахождения файловых данных (как происходит, к примеру, в FAT), NTFS обращается к диску всего один раз и немедленно считывает данные.

Как показано на рис. 12.33, атрибуты небольшой папки и небольшого файла могут быть резидентными в MFT. Для небольшой папки атрибут корня индекса содержит индекс (построенный как Б-дерево) файловых записей для файлов (и вложенных папок) внутри данной папки.

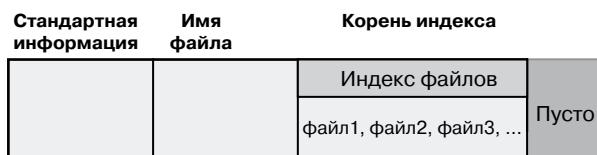


Рис. 12.33. MFT-запись для небольшой папки

Разумеется, многие файлы и папки невозможно сжать до размера MFT-записи, составляющего 1 Кбайт. Если значение какого-то атрибута слишком велико для помещения в MFT-запись, NTFS выделяет для него кластеры за пределами MFT. Непрерывная группа кластеров называется *отрезком* (run). Если значение атрибута впоследствии увеличивается (например, при добавлении в файл дополнительных данных), NTFS выделяет еще один отрезок. Атрибуты, значения которых хранятся в отрезках, а не в MFT, называются *нерезидентными* (nonresident). Файловая система сама решает, резидентным или нерезидентным будет атрибут; расположение данных прозрачно для обращающегося к ним процесса.

Если атрибут является нерезидентным, как, к примеру, атрибут данных большого файла, информация, необходимая NTFS для обнаружения значения атрибута на диске, содержится в его заголовке. Пример нерезидентного атрибута данных, хранящегося в двух отрезках, показан на рис. 12.34.

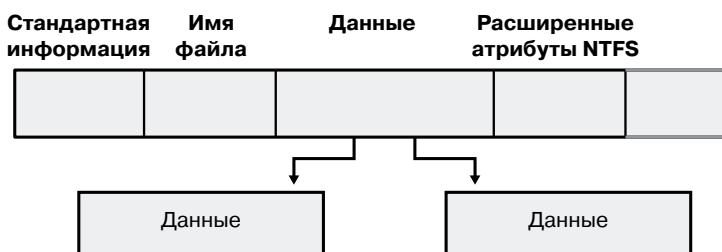


Рис. 12.34. Файловая MFT-запись для большого файла с двумя отрезками данных

Среди стандартных атрибутов нерезидентными бывают лишь те, размер которых может увеличиваться. Для файлов это атрибут данных и список атрибутов (на рис. 12.34 не показан). Атрибуты стандартной информации и имени файла всегда резидентны.

Большая папка также может обладать нерезидентными атрибутами (или частями атрибутов), как демонстрирует рис. 12.35. В этом примере в MFT-записи не хватает места для хранения Б-дерева с индексом находящихся в папке файлов. Часть индекса хранится в атрибуте корня индекса, а часть — в нерезидентных отрезках, называемых *размещениями индекса* (index allocations). Атрибуты корня индекса, размещения индекса и битовой карты в данном случае показаны упрощенно. Далее мы поговорим о них более подробно. Атрибуты стандартной информации и имени файла всегда резидентны. Для папок также резидентны заголовок и, по крайней мере, часть значения атрибута корня индекса.

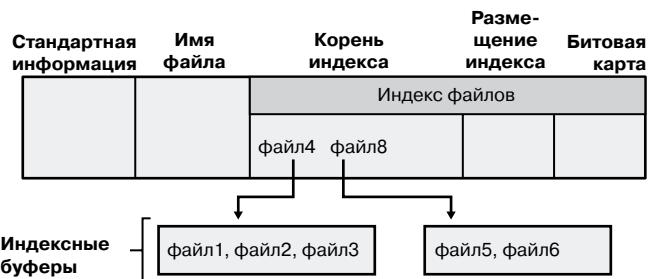


Рис. 12.35. MFT-запись для большой папки с нерезидентным индексом имени файла

Когда значение атрибута не помещается в MFT-записи и для него требуется отдельное место, NTFS следит за отрезками с помощью пар отображения VCN на LCN. LCN-номера отражают последовательность кластеров на всем томе от 0 до n , а VCN-номера нумеруют от 0 до t только кластеры, принадлежащие конкретному файлу. Пример нумерации кластеров на отрезке нерезидентного атрибута данных показан на рис. 12.36.

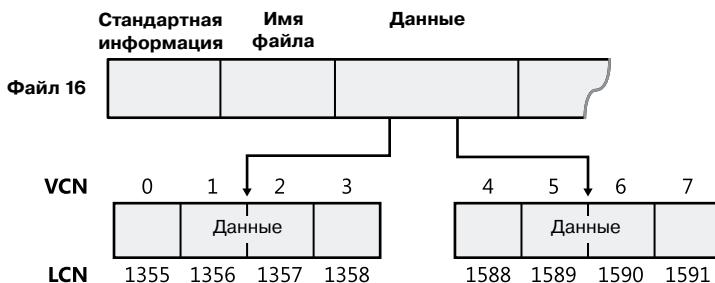


Рис. 12.36. VCN-номера для нерезидентного атрибута данных

Если бы этот файл занимал больше двух отрезков, нумерация в третьей группе начиналась бы с VCN 8. Как показано на рис. 12.37, заголовок атрибута данных содержит отображения VCN на LCN для обоих отрезков, что позволяет NTFS без труда находить на диске выделенные под них области.

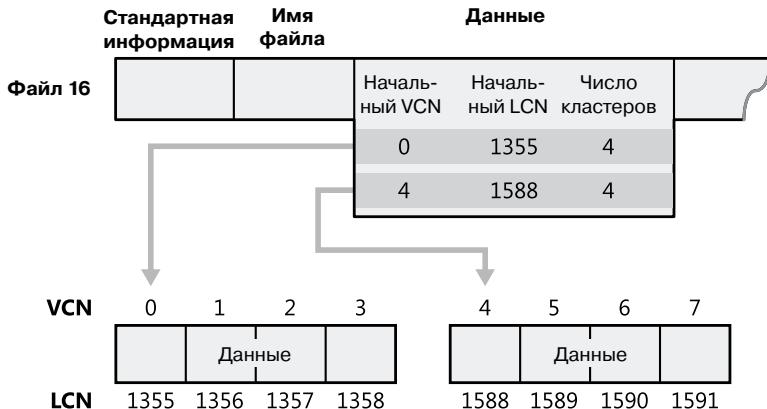


Рис. 12.37. Отображения VCN на LCN для нерезидентного атрибута данных

Хотя на рис. 12.36 показаны только отрезки для данных, в отрезках могут храниться и другие атрибуты, если в MFT-записи для них недостаточно места. Если у какого-то файла атрибутов столько, что они не помещаются в MFT-записи, для хранения дополнительных атрибутов (или их заголовков в случае нерезидентных атрибутов) выделяется вторая MFT-запись. В этом случае добавляется атрибут, называемый *списком атрибутов* (attribute list). Он содержит имя и код типа каждого атрибута файла, а также номер файла для MFT-записи, в которой располагается атрибут. Список атрибутов предназначен для случаев, когда файл становится настолько большим или фрагментированным, что одной MFT-записи уже недостаточно для хранения многочисленных отображений VCN на LCN, необходимых для поиска всех отрезков. Обычно он требуется файлам, у которых более 200 отрезков. В заключение добавим, что заголовки атрибутов всегда содержатся в файловых MFT-записях, но значение атрибута может находиться и вне MFT в одном или нескольких отрезках.

Сжатие данных и разреженные файлы

В NTFS поддерживается сжатие, или компрессия, отдельных файлов, папок и томов с применением версии алгоритма LZ77, известной как LZNT1. (Сжатию в NTFS подвергаются только пользовательские данные, метаданные системы этот процесс не затрагивает.) Определить, подвергался ли том сжатию, позволяет функция `GetVolumeInformation`. Чтобы получить реальный размер сжатого файла, воспользуйтесь функцией `GetCompressedFileSize`. Наконец, проверить или изменить параметры сжатия для файла или папки позволяет функция `DeviceIoControl`. (См. управляющие коды `FSCTL_GET_COMPRESSION` и `FSCTL_SET_COMPRESSION`.) Следует отметить, что изменение степени сжатия файла вступает в силу мгновенно, чего нельзя сказать об изменении степени сжатия папок и томов. Во втором случае заданная для папки или тома степень сжатия начинает по умолчанию использоваться для всех файлов и вложенных папок, создаваемых внутри папки или на томе. Но если выполнить сжатие папки в окне ее свойств, вызываемом через Проводник, сжатие всего дерева папки произойдет немедленно.

Далее мы рассмотрим процедуру сжатия в NTFS на примере разреженных данных. Затем перейдем к обсуждению сжатия обычных и разреженных файлов.

Сжатие разреженных данных

Разреженные данные (sparse data) обычно занимают много места, но лишь малая часть значений в них отлична от нуля. В качестве примера можно упомянуть разреженную матрицу. Как уже отмечалось, для нумерации кластеров файла NTFS использует виртуальные номера кластеров (VCN) от 0 до m . Каждый VCN-номер отображается на соответствующий логический номер кластера (LCN), определяющий положение кластера на диске. На рис. 12.38 показаны отрезки (занимаемые участки дискового пространства) обычного (несжатого) файла и его VCN-номера, а также LCN-номера, на которые они отображаются.

VCN	0	1	2	3	4	5	6	7	8	9	10	11
LCN	1355	1356	1357	1358	1588	1589	1590	1591	2033	2034	2035	2036

Рис. 12.38. Отрезки несжатого файла

Файл хранится в трех группах, каждая из которых состоит из четырех кластеров. То есть всего у нас 12 кластеров. На рис. 12.39 показана MFT-запись для этого файла. Как уже отмечалось, для экономии места на диске атрибут данных в MFT-записи содержит только одно отображение VCN на LCN для каждого отрезка, а не для каждого кластера. Тем не менее обратите внимание, что для каждого VCN-номера от 0 до 11 есть соответствующий LCN-номер. Первая запись начинается с VCN 0 и охватывает 4 кластера, вторая начинается с VCN 4 и охватывает 4 кластера и т. д. Это типичный для несжатого файла формат.

Стандартная информация	Имя файла	Данные		
		Началь- ный VCN	Началь- ный LCN	Число кластеров
		0	1355	4
		4	1588	4
		8	2033	4

Рис. 12.39. MFT-запись для несжатого файла

Один из способов сжатия файла в NTFS состоит в удалении из него длинных цепочек нулей. Если данные в файле разрежены, он обычно сжимается до размера лишь той части дискового пространства, которая требуется для его хранения в нормальном виде. При последующей записи в этот файл NTFS выделяет место только для отрезков с ненулевыми данными.

На рис. 12.40 показаны отрезки файла, содержащего разреженные данные, после его сжатия. Обратите внимание, что для некоторых диапазонов VCN-номеров (16–31 и 64–127) файлового пространства не выделено.

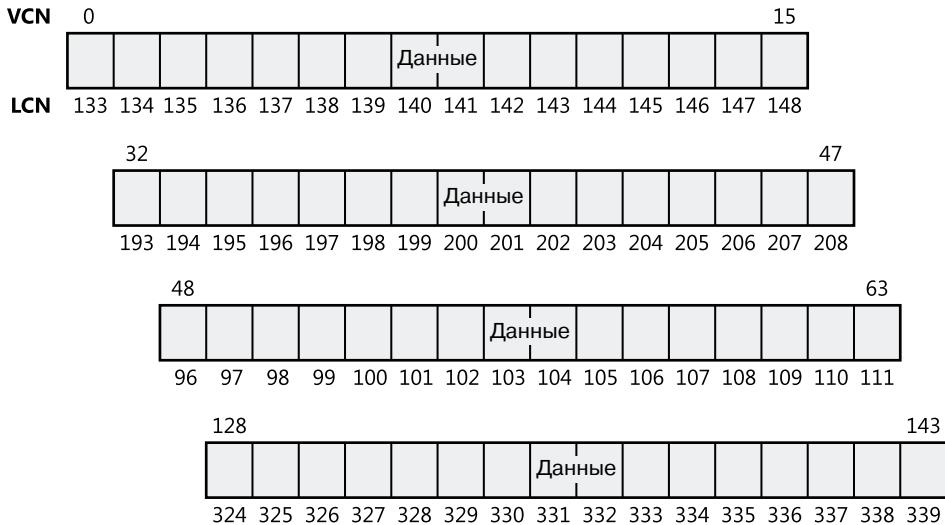


Рис. 12.40. Отрезки сжатого файла, содержащего разреженные данные

В MFT-записи для этого сжатого файла пропущены блоки VCN-номеров, содержащие нули. Именно поэтому для них не выделено файловое пространство. Так, к примеру, первый элемент данных на рис. 12.41 начинается с VCN 0 и охватывает 16 кластеров. Второй элемент пересекает на VCN 32 и охватывает еще 16 кластеров.

Стандартная информация	Имя файла	Данные		
		Началь- ный VCN	Началь- ный LCN	Число кластеров
		0	133	16
		32	193	16
		48	96	16
		128	324	16

Рис. 12.41. MFT-запись для сжатого файла, содержащего разреженные данные

Когда программа читает данные из сжатого файла, NTFS проверяет MFT-запись, чтобы выяснить наличие отображения VCN на LCN для считываемого участка файла. Если программа выполняет чтение из невыделенной «дыры» в файле, это означает, что данные этой части файла состоят из нулей. В этом случае NTFS возвращает нули,

более не обращаясь к диску. Если программа записывает в «дыру» ненулевые данные, NTFS автоматически выделяет под них дисковое пространство. Этот прием хорошо работает для разреженных файлов с большим количеством нулевых данных.

Сжатие неразреженных данных

Предыдущий пример сжатия разреженного файла до определенной степени надуман. Он описывает сжатие такого файла, целые области которого заполнены нулями. При этом на остальные данные этот процесс никак не влияет. В большинстве файлов данные не являются разреженными, но их можно сжать, применив соответствующий алгоритм.

В NTFS пользователи могут сжимать как отдельные файлы, так и целые папки. (Создаваемые в сжатой папке новые файлы сжимаются автоматически, в то время как существующие файлы нужно сжимать по отдельности, программно включив сжатие управляющим кодом `FSCTL_SET_COMPRESSION`.) Сжимая файл, NTFS разбивает его необработанные данные на *единицы сжатия* (compression units) длиной по 16 кластеров (например, для кластера в 512 байт они составляют 8 Кбайт). Определенные последовательности данных могут сжиматься не очень сильно или вообще не сжиматься; поэтому для каждой единицы сжатия NTFS определяет, будет ли в результате ее сжатия получен выигрыш хотя бы в один кластер. Если даже этого не происходит, NTFS выделяет отрезок из 16 кластеров и записывает содержащиеся в нем данные на диск без сжатия. Если же данные из 16 кластеров можно сжать до 15 и менее кластеров, NTFS выделяет ровно столько места, сколько требуется для хранения сжатых данных, и записывает их на диск. Рисунок 12.42 иллюстрирует сжатие файла, состоящего из четырех отрезков. Незакрашенные области представляют дисковое пространство, которое файл займет после сжатия. Сжимаются только первый, второй и четвертый отрезки. Но даже при наличии одного несжимаемого отрезка экономится 26 кластеров диска, то есть длина файла уменьшается на 41 %.



Рис. 12.42. Отрезки данных сжатого файла

ПРИМЕЧАНИЕ

На приведенных в этой главе схемах показаны последовательные LCN-номера, но единицы сжатия могут располагаться и на несмежных кластерах. Занимающие несмежные кластеры отрезки приводят к появлению более сложных MFT-записей, чем записи, показанные на рис. 12.42.

При записи данных в сжатый файл NTFS гарантирует, что каждый отрезок будет начинаться на виртуальной 16-кластерной границе. То есть начальный VCN-номер каждого отрезка кратен 16, а длина отрезка не превышает 16 кластеров. При работе со сжатым файлом NTFS единовременно считывает и записывает минимум одну единицу сжатия. Причем при записи сжатых данных NTFS пытается помещать единицы сжатия в физически смежные области, чтобы их можно было считывать в ходе одной операции ввода-вывода. Размер единицы сжатия в 16 кластеров выбран для уменьшения внутренней фрагментации: чем больше размер единицы сжатия, тем меньше дискового пространства нужно для хранения данных. В итоге размер единицы сжатия в 16 кластеров возник как компромисс между желанием минимизировать размер сжимаемых файлов и попыткой не слишком замедлить операцию чтения для программ с произвольным доступом к содержимому файлов. При каждом кэш-промахе приходится разворачивать эквивалент 16 кластеров. (Кэш-промахи при произвольном доступе происходят чаще.) На рис. 12.43 демонстрируется MFT-запись для сжатого файла, показанного на рис. 12.42.

Стандартная информация		Имя файла	Данные		
			Началь-ный VCN	Началь-ный LCN	Число кластеров
			0	19	4
			16	23	8
			32	97	16
			48	113	10

Рис. 12.43. MFT-запись для сжатого файла

Этот сжатый файл отличается от сжатого разреженного файла из более раннего примера тем, что в данном случае все три отрезка имеют длину менее 16 кластеров. Чтение этой информации из файловой MFT-записи позволяет NTFS понять, подвергались ли данные в файле сжатию. Любой отрезок короче 16 кластеров содержит сжатые данные — при первом считывании этих данных файловая система NTFS должна выполнить их декомпрессию. Отрезок, длина которого составляет ровно 16 кластеров, не содержит сжатых данных, а значит, не требует декомпрессии.

Если данные в отрезке были сжаты, NTFS выполняет их декомпрессию во временному буфере, копируя оттуда в буфер вызывающей программы. Кроме того, NTFS загружает данные после декомпрессии в кэш, что делает их последующее чтение из отрезка таким же быстрым, как и чтение из кэша обычных данных. В кэш NTFS записывает все обновления файла, позволяя подсистеме отложенной записи сжимать

измененные данные и записывать их на диск в асинхронном режиме. Такая стратегия гарантирует, что задержка при записи в сжатый файл будет практически такой же, как при работе с обычным файлом.

По возможности NTFS размещает сжатый файл в смежных областях диска. Как показывают LCN-номера, первые два отрезка сжатого файла с рис. 12.42 являются физически смежными, как и два последних. При смежном расположении двух и более отрезков NTFS выполняет опережающее чтение с диска, как и в случае с обычными файлами. Так как считывание и декомпрессия непрерывных файловых данных выполняются асинхронно и еще до того, как программа запросит эти данные, при последующих операциях чтения данные извлекаются непосредственно из кэша, что значительно ускоряет чтение.

Разреженные файлы

Разреженные файлы (это не то же самое, что файлы с разреженными данными), по сути, являются сжатыми файлами, неразреженные данные которых NTFS не сжимает. Однако NTFS обрабатывает данные отрезков из MFT-записи разреженного файла таким же способом, как и в случае сжатых файлов, состоящих из разреженных и неразреженных данных.

Файл журнала изменений

Файл журнала изменений `\$Extend\$UsnJrnl` представляет собой разреженный файл, в котором NTFS хранит записи изменений в файлах и папках. Такие приложения, как служба репликации файлов (File Replication Service, FRS) и служба поиска, пользуются этим журналом, чтобы реагировать на возникающие изменения в файлах и папках.

Журнал хранит записи изменений в потоке данных `$J`, а свой максимальный размер — в потоке данных `$Max`. Записи распределены по версиям и включают следующую информацию об изменениях файлов или папок:

- время изменения;
- причина изменения (табл. 12.8);
- атрибуты файла или папки;
- имя файла или папки;
- индекс файловой MFT-записи для файла или папки;
- индекс файловой ссылки родительской папки файла;
- идентификатор безопасности;
- номер последовательного обновления (Update Sequence Number, USN) записи;
- дополнительные сведения об источнике изменений (пользователь, FRS и т. п.).

Таблица 12.8. Причины изменений

Идентификатор	Причина
<code>USN_REASON_DATA_OVERWRITE</code>	Переписывание данных в файле или папке
<code>USN_REASON_DATA_EXTEND</code>	Добавление данных в файл или папку

Идентификатор	Причина
USN_REASON_DATA_TRUNCATION	Усечение данных в файле или папке
USN_REASON_NAMED_DATA_OVERWRITE	Переписывание потока данных для данных в файле
USN_REASON_NAMED_DATA_EXTEND	Расширение потока данных для данных в файле
USN_REASON_NAMED_DATA_TRUNCATION	Усечение потока данных для данных в файле
USN_REASON_FILE_CREATE	Создание нового файла или папки
USN_REASON_FILE_DELETE	Удаление файла или папки
USN_REASON_EA_CHANGE	Изменение дополнительных атрибутов файла или папки
USN_REASON_SECURITY_CHANGE	Изменение дескриптора безопасности файла или папки
USN_REASON_RENAME_OLD_NAME	Старое имя после переименования файла или папки
USN_REASON_RENAME_NEW_NAME	Новое имя после переименования файла или папки
USN_REASON_INDEXABLE_CHANGE	Изменение состояния индексирования файла или папки (вне зависимости от того, будет ли служба индексирования обрабатывать этот файл или папку)
USN_REASON_BASIC_INFO_CHANGE	Изменение атрибутов файла или папки и (или) изменение меток времени
USN_REASON_HARD_LINK_CHANGE	Добавление или удаление жесткой ссылки для файла или папки
USN_REASON_COMPRESSION_CHANGE	Изменение состояния сжатия файла или папки
USN_REASON_ENCRYPTION_CHANGE	Включение или отключение режима шифрования (EFS) для файла или папки
USN_REASON_OBJECT_ID_CHANGE	Изменение идентификатора объекта файла или папки
USN_REASON_REPARSE_POINT_CHANGE	Изменение точки повторной обработки для файла или папки, добавление новой точки повторной обработки (например, символьской ссылки), а также ее удаление
USN_REASON_STREAM_CHANGE	Добавление нового потока данных к файлу или папке, удаление или переименование существующего потока данных
USN_REASON_TRANSACTED_CHANGE	Это значение было добавлено в список для ситуаций, когда изменение является результатом недавно подтвержденной TxF-транзакции
USN_REASON_CLOSE	Был закрыт дескриптор файла или папки, указывая на завершение последнего изменения в череде операций изменения

Так как журнал является разреженным, он никогда не переполняется. Когда его размер превышает заданный максимум, NTFS просто начинает обнулять файловые данные, предшествующие текущему блоку информации об изменениях, как показано на рис. 12.44. Но для предотвращения постоянного изменения размера журнала NTFS уменьшает его, только когда он в два раза превосходит установленный максимум.

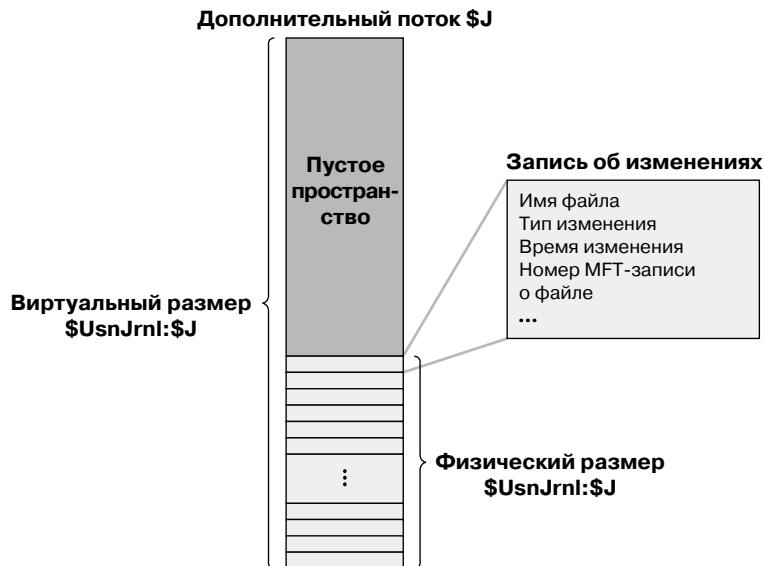


Рис. 12.44. Выделение места для журнала изменений (\$UsnJrnl)

ЭКСПЕРИМЕНТ: ЧТЕНИЕ ЖУРНАЛА ИЗМЕНЕНИЙ

Если у текущего тома есть журнал изменений, его содержимое можно увидеть с помощью утилиты командной строки Usndump.exe производства Winsider Seminars & Solutions (www.winsiderss.com/tools/usndump/usndump.htm). Кроме того, как показано далее, можно создавать и удалять журналы изменений, а также выполнять запросы к ним при помощи встроенной служебной программы Fsutil.exe:

```
C:\>fsutil usn queryjournal c:
Usn Journal ID : 0x01c89ddaec1b9648
First Usn : 0x0000000038140000
Next Usn : 0x000000003a22fa50
Lowest Valid Usn : 0x00000000000000000000
Max Usn : 0x00000fffffff0000
Maximum Size : 0x0000000002000000
Allocation Delta : 0x0000000000400000
```

Результат содержит сведения о максимальном размере журнала изменений на томе и его текущем состоянии. Проведем простой эксперимент, чтобы посмотреть, как NTFS записывает изменения в журнал. Создайте в текущей папке файл с именем Usn.txt, присвойте ему имя UsnNew.txt и выведите содержимое журнала в приложении Usndump:

```
C:\>echo hello > Usn.txt
C:\>ren Usn.txt UsnNew.txt
C:\>Usndump.exe
...
File Ref# : 0x4000000001be9
ParentFile Ref# : 0x300000000a962
USN : 0xfc54d8
SecurityId : 0x00000000
Reason : 0x00000100 (USN_REASON_FILE_CREATE)
Name (014) : Usn.txt

File Ref# : 0x4000000001be9
ParentFile Ref# : 0x300000000a962
USN : 0xfc5528
SecurityId : 0x00000000
Reason : 0x00000102 (USN_REASON_DATA_EXTEND USN_REASON_FILE_CREATE)
Name (014) : Usn.txt

File Ref# : 0x4000000001be9
ParentFile Ref# : 0x300000000a962
USN : 0xfc5578
SecurityId : 0x00000000
Reason : 0x80000102 (USN_REASON_DATA_EXTEND USN_REASON_FILE_CREATE)
Name (014) : Usn.txt

File Ref# : 0x4000000001be9
ParentFile Ref# : 0x300000000a962
USN : 0xfc55c8
SecurityId : 0x00000000
Reason : 0x00001000 (USN_REASON_RENAME_OLD_NAME)
Name (014) : Usn.txt

File Ref# : 0x4000000001be9
ParentFile Ref# : 0x300000000a962
USN : 0xfc5618
SecurityId : 0x00000000
Reason : 0x00002000 (USN_REASON_RENAME_NEW_NAME)
Name (020) : UsnNew.txt

File Ref# : 0x4000000001be9
ParentFile Ref# : 0x300000000a962
USN : 0xfc5668
SecurityId : 0x00000000
Reason : 0x80002000 (USN_REASON_RENAME_NEW_NAME)
Name (020) : UsnNew.txt
```

Записи отражают отдельные модификации, входящие в инициированные командой операции.

Индексация

В NTFS папка представляет собой просто индекс имен файлов, то есть совокупность имен файлов (с соответствующими файловыми ссылками), организованная как Б-дерево. Для создания папки NTFS индексирует атрибуты имен файлов для всех входящих в папку файлов. MFT-запись для корневой папки тома показана на рис. 12.45.

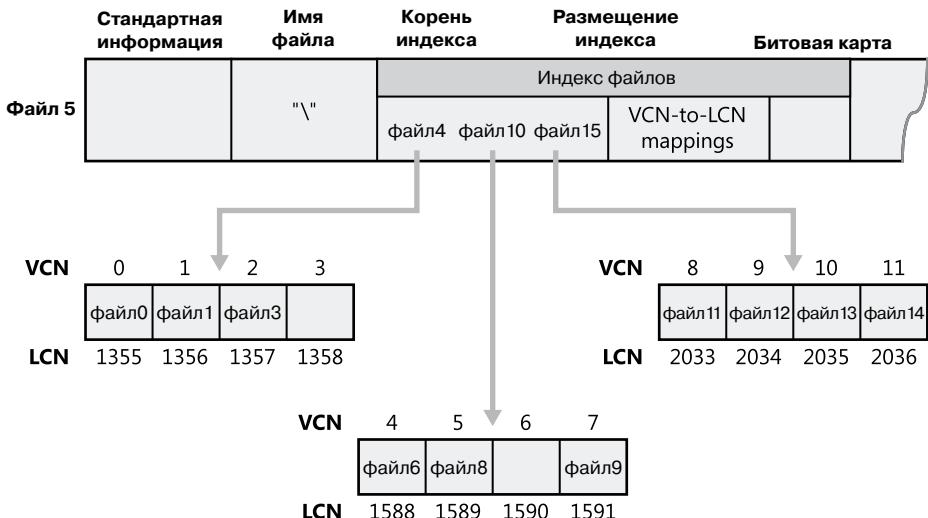


Рис. 12.45. Индекс имен файлов для корневой папки тома

По сути, MFT-запись для папки содержит в своем атрибуте корня индекса отсортированный список файлов в этой папке. Но в случае больших папок имена файлов на самом деле хранятся в буферах индексов, имеющих фиксированный размер 4 Кбайт (которые являются нерезидентным значением атрибута размещения индекса). Эти буфера реализуют такую структуру данных, как Б-дерево. Она минимизирует количество обращений к диску, необходимое для обнаружения конкретного файла, особенно в случае больших папок. Атрибут корня индекса содержит первый уровень Б-дерева (папки, вложенные в корневую папку) и ссылается на буфера индексов, содержащих следующий уровень (возможно, больше вложенных папок или файлов).

На рис. 12.45 в атрибуте корня индекса и буферах индексов показаны только имена файлов (например, file6), но каждая запись в индексе содержит также файловую ссылку на описывающую данный файл MFT-запись, метку времени и информацию о размере файла. NTFS дублирует метку времени и информацию о размере файла из файловой MFT-записи. Эта техника, используемая как в FAT, так и в NTFS, требует записи обновленной информации в два места. Но даже несмотря на это, просмотр папки значительно ускоряется, так как файловая система может выводить на экран метки времени и размеры файлов, не открывая каждый файл в папке.

Атрибут размещения индекса отображает VCN-номера отрезков буфера индекса на LCN-номера, которые показывают, в каком месте диска находятся индексные буфера, а битовая карта следит за тем, какие VCN-номера в этих буферах заняты, а какие свободны. На рис. 12.45 показана одна файловая запись на VCN-номер (то есть на кластер), но на самом деле каждый кластер содержит целый набор записей. Каждый индексный буфер размером 4 Кбайт обычно содержит 20–30 записей для имен файлов (в зависимости от длины имен файлов в папке).

Структура данных Б-дерева представляет собой разновидность сбалансированного дерева, идеально подходящую для организации отсортированных данных, хранящихся

на диске, так как она позволяет минимизировать количество обращений к диску при поиске заданного элемента. В MFT атрибут корня индекса для папки содержит несколько имен файлов, выступающих в качестве индексов для второго уровня Б-дерева. С каждым именем файла в атрибуте корня индекса связан необязательный указатель на буфер индекса. Этот индекс содержит имена файлов, лексиографические значения которых меньше его собственного. К примеру, на рис. 12.45 file4 — это элемент первого уровня Б-дерева. Он указывает на буфер индекса, содержащий имена файлов, которые меньше (лексиографически), чем его собственное имя, — это имена file0, file1 и file3. Обратите внимание, что использованные в примере имена file1, file3 и т. п. не буквальны. Они просто иллюстрируют относительное размещение файлов, лексиографически упорядоченных в соответствии с указанной последовательностью.

Хранение имен файлов в виде Б-деревьев дает ряд преимуществ. Ускоряется поиск файлов, так как их имена в папке упорядочены. Когда высокоуровневое программное обеспечение просматривает файлы в папке, NTFS возвращает уже отсортированные имена. Наконец, так как Б-деревья растут не в глубину, а в ширину, скорость поиска с увеличением размера папки не снижается.

Кроме индексации имен, NTFS обеспечивает универсальную индексацию данных, а в некоторых NTFS-механизмах (в том числе в идентификаторах объектов, системе отслеживания квот и в консолидированной системе безопасности) индексация используется для управления внутренними данными.

В NTFS индексы в виде Б-деревьев являются обобщенным инструментом и применяются для организации дескрипторов безопасности, идентификаторов безопасности, идентификаторов объектов, записей о дисковых квотах и точек повторной обработки. Папки называют *индексами имен файлов* (file name indexes), остальные же типы индексов известны как *индексы представлений* (view indexes).

Идентификаторы объектов

Кроме идентификатора объекта, назначенного файлу или папке и хранящегося в атрибуте \$OBJECT_ID MFT-записи, NTFS запоминает соответствие между идентификаторами объектов и номерами их файловых ссылок в индексе \$O файла метаданных \\$Extend\\$ObjId. Индекс собирает записи по их идентификаторам объектов (то есть GUID), позволяя NTFS быстро находить файлы. Используя эту недокументированную естественную API-функциональность, приложения могут открывать файлы или папки по идентификаторам объектов. Рисунок 12.46 демонстрирует взаимосвязь между файлом метаданных \$ObjId и атрибутами \$OBJECT_ID в MFT-записях.

Отслеживание квот

Сведения о квотах NTFS хранит в файле метаданных \\$Extend\\$Quota, который состоит из именованных атрибутов корня индекса \$O и \$Q. Структура этих индексов показана на рис. 12.47. NTFS не только присваивает каждому дескриптору безопасности уникальный внутренний идентификатор безопасности (SID), но и назначает уникальный идентификатор каждому пользователю. Когда администратор задает квоты для пользователя, NTFS выделяет идентификатор, соответствующий идентификатору безопасности этого пользователя. В индексе \$O NTFS создает запись, отображающую

SID на идентификатор пользователя, и сортирует индекс по идентификаторам безопасности. В индексе \$Q NTFS создает запись управления квотами. Она содержит данные о выделенных пользователю лимитах и об отведенном ему на томе объеме дискового пространства.

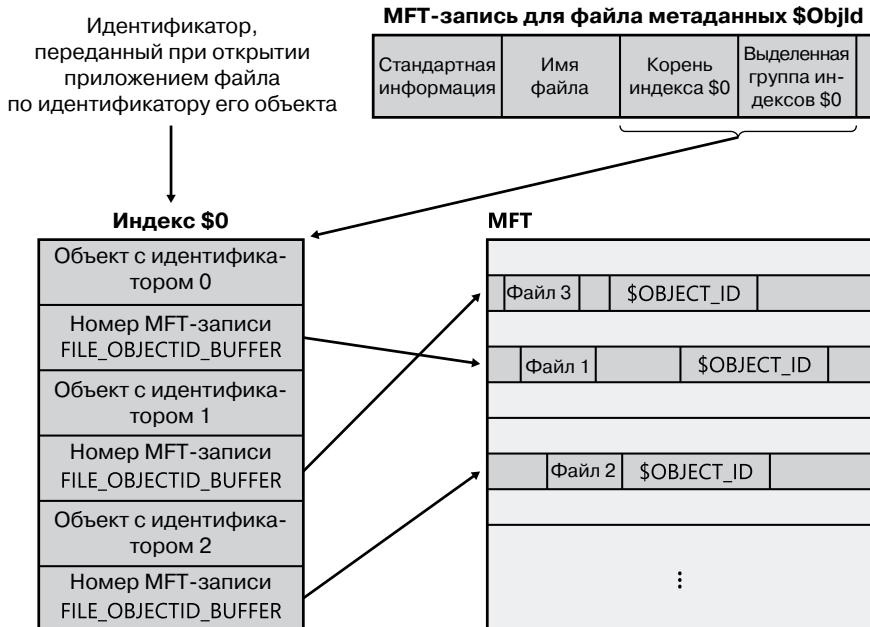


Рис. 12.46. Соотношение между \$ObjId и \$OBJECT_ID

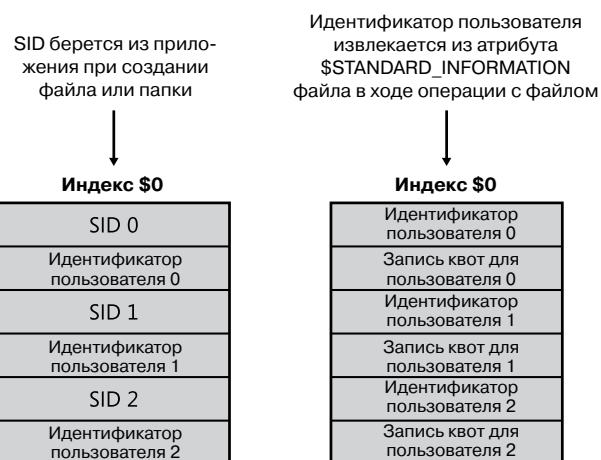


Рис. 12.47. Индексация \$Quota

Когда приложение создает файл или папку, NTFS получает SID пользователя этого приложения и ищет в индексе \$0 соответствующий идентификатор пользователя. Этот идентификатор записывается в атрибут **\$STANDARD_INFORMATION** нового файла или папки, который подсчитывает все дисковое пространство, выделенное под файл или папку относительно пользовательской квоты. Затем NTFS просматривает запись квот в индексе \$Q и определяет, не превышает ли выделенное дисковое пространство установленные для пользователя лимиты. Когда новое дисковое пространство, выделяемое пользователю, превышает пороговое значение, NTFS предпринимает соответствующие меры, например записывает событие в системный журнал или отклоняет запрос на создание файла или папки. При изменении размера файла или папки NTFS обновляет данные записи управления квотой, связанные с идентификатором пользователя и хранящиеся в атрибуте **\$STANDARD_INFORMATION**. В NTFS используется обобщенный индекс в виде Б-дерева для эффективной корреляции идентификаторов пользователей с идентификаторами безопасности учетных записей. Благодаря этому, зная идентификатор пользователя, NTFS эффективно находит сведения об управлении его квотами.

Консолидированная система безопасности

В NTFS всегда поддерживалась система безопасности, позволяющая администратору указывать, какие пользователи могут обращаться к отдельным файлам и папкам, а какие не могут. NTFS оптимизирует использование дискового пространства дескрипторами безопасности за счет применения централизованного файла метаданных **\$Secure**, в котором хранится только один экземпляр каждого дескриптора безопасности на томе.

Файл **\$Secure** содержит два индексных атрибута, **\$SDH** (Security Descriptor Hash – хэш дескриптора безопасности) и **\$SII** (Security ID Index – индекс идентификатора безопасности), а также атрибут потока данных **\$SDS** (Security Descriptor Stream – поток данных дескриптора безопасности), как показано на рис. 12.48. NTFS назначает каждому уникальному дескриптору безопасности на томе внутренний идентификатор безопасности (не путать с идентификатором безопасности, который уникально идентифицирует учетные записи компьютеров и пользователей) и хэширует дескриптор безопасности по простому алгоритму. Хэш является потенциально неуникальным «стенографическим» представлением дескриптора. Записи в индексе **\$SDH** отображают хэши дескрипторов безопасности на их местоположение внутри атрибута данных **\$SDS**, а записи индекса **\$SII** отображают NTFS-идентификаторы безопасности на местоположение дескриптора безопасности в атрибуте данных **\$SDS**.

Когда вы применяете дескриптор безопасности к файлу или папке, NTFS получает хэш этого дескриптора и просматривает индекс **\$SDH** в поисках совпадений. Далее NTFS сортирует элементы индекса **\$SDH** по хэшам соответствующих дескрипторов безопасности. При этом элементы сохраняются в виде Б-дерева. Обнаружив в индексе **\$SDH** совпадение для дескриптора, NTFS находит смещение этого дескриптора от смещения элемента и считывает его из атрибута **\$SDS**. Если хэши совпадают, а дескрипторы нет, NTFS ищет в индексе **\$SDH** следующее совпадение. После обнаружения точного совпадения файл или папка, к которым применяется дескриптор безопасности, могут ссылаться на существующий дескриптор в атрибуте **\$SDS**. Этую ссылку NTFS формирует, считывая идентификатор безопасности из записи **\$SDH** и сохраняя его в атрибуте

\$STANDARD_INFORMATION файла или папки. Присутствующий у всех файлов и папок атрибут **\$STANDARD_INFORMATION** хранит базовую информацию о файле, в том числе его атрибуты, временные метки и идентификатор безопасности.

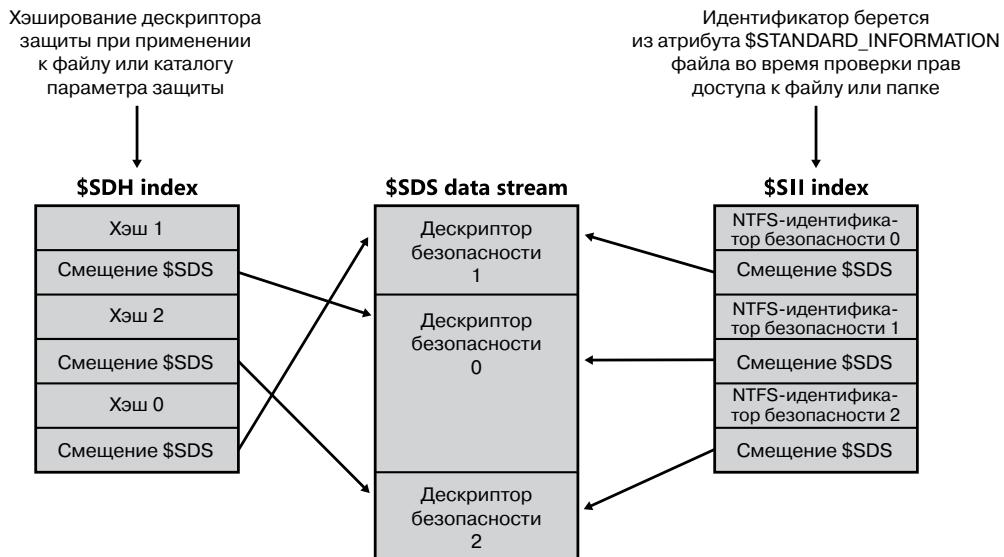


Рис. 12.48. Индексация **\$Secure**

Если NTFS не обнаруживает в индексе **\$SDH** записи с дескриптором безопасности, совпадающим с тем, который вы применяете, значит, ваш дескриптор уникоден в пределах тома, и NTFS присваивает ему новый внутренний идентификатор безопасности. Эти идентификаторы являются 32-разрядными значениями, в то время как SID-идентификаторы, как правило, превышают их в несколько раз, поэтому представление SID-идентификаторов в виде внутренних идентификаторов безопасности экономит место в атрибуте **\$STANDARD_INFORMATION**. Затем NTFS добавляет дескриптор безопасности в конец атрибута данных **\$SDS** и в записи индексов **\$SDH** и **\$SII**, которые ссылаются на смещение дескриптора в данных **\$SDS**.

При попытке приложения открыть файл или папку NTFS использует индекс **\$SII** для поиска дескриптора безопасности этого файла или папки. NTFS считывает внутренний идентификатор безопасности файла или папки из атрибута **\$STANDARD_INFORMATION** MFT-записи. Затем по индексу **\$SII** файла NTFS находит элемент с нужным идентификатором в атрибуте данных **\$SDS**. По смещению в атрибуте **\$SDS** NTFS считывает дескриптор безопасности и завершает проверку прав доступа. Последние 32 дескриптора безопасности, к которым было обращение, NTFS хранит в кэше вместе с их записями в индексе **\$SII**, чтобы впоследствии при некэшированном индексе **\$SII** иметь возможность обращаться к файлу **\$Secure**.

Записи из файла **\$Secure** NTFS не удаляет, даже если на них не ссылаются никакие файлы и папки тома. Но это не приводит к значительному увеличению размера файла,

так как на большинстве томов, даже использующихся долгое время, не так уж много уникальных дескрипторов безопасности.

Механизм индексации на основе обобщенного Б-дерева позволяет NTFS использовать для защиты файлов и папок с одинаковыми параметрами одни и те же дескрипторы безопасности. По индексу **\$SII** NTFS при проверках прав доступа быстро находит нужный дескриптор в файле **\$Secure**, а по индексу **\$SDH** определяет, хранится ли применяемый к файлу или папке дескриптор безопасности в файле **\$Secure**, и при положительном результате проверки использует его снова.

Точки повторной обработки

Как уже упоминалось, *точка повторной обработки* (reparse point) представляет собой заданный приложением блок данных повторной обработки, размер которого может доходить до 16 Кбайт, и 32-разрядный тег повторной обработки, хранящийся в атрибуте **\$REPARSE_POINT** файла или папки. Каждый раз, когда приложение создает или удаляет точку повторной обработки, NTFS обновляет файл метаданных **\\$Extend\\$Reparse**, в котором хранятся записи, идентифицирующие номера записей о файлах и папках, содержащих точки повторной обработки. Централизованное хранение записей позволяет NTFS предоставлять всем приложениям интерфейсы для просмотра всех точек повторной обработки на томе или только точек определенного типа, например точек монтирования. (Подробно точки повторной обработки описываются в главе 9.) Файл **\\$Extend\\$Reparse** использует механизм универсальной индексации по типу Б-дерева, сортируя файловые записи (в индексе с именем **\$R**) по тегам повторной обработки и по номерам файловых записей.

Поддержка транзакций

Благодаря диспетчеру транзакций ядра (Kernel Transaction Manager, KTM) и инструментам уже знакомой нам файловой системы с типовым протоколированием (CLFS) в NTFS реализована транзакционная модель, названная *транзакционной версией NTFS* (Transactional NTFS, TxF). TxF предоставляет набор API в режиме пользователя для выполнения приложениями транзакционных операций с файлами и папками, а также интерфейс управления файловой системой (File System Control, FSCTL) для управления диспетчерами ресурсов.

ПРИМЕЧАНИЕ

Поддержка TxF была добавлена к драйверу NTFS без изменения формата структур данных этой файловой системы. Именно поэтому формат NTFS версии 3.1 тот же, что был во времена Windows XP и Windows Server 2003. Обратная совместимость в TxF достигнута за счет применения имеющегося типа атрибута (**\$LOGGED.Utility_Stream**), который раньше использовался только для поддержки EFS, а не введением нового типа.

Как показано на рис. 12.49, в общую архитектуру TxF входит несколько компонентов:

- транзакционные API-интерфейсы, реализованные в библиотеке Kernel32.dll;
- библиотека для чтения TxF-журналов (%SystemRoot%\System32\Txflw32.dll);

- ❑ COM-компонент, реализующий в TxF функциональность протоколирования (%SystemRoot\System32\Txlog.dll);
- ❑ транзакционная NTFS-библиотека внутри NTFS-драйвера;
- ❑ CLFS-инфраструктура для чтения и создания записей журнала.

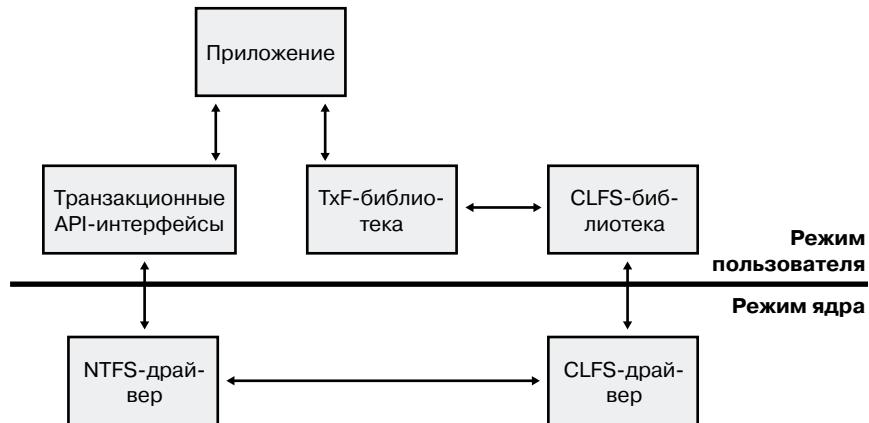


Рис. 12.49. Архитектура файловой системы TxF

Изоляция

Хотя транзакционные операции с файлами являются пакетными, как и операции с транзакционным реестром (Transactional Registry, TxR), о которых речь идет в главе 4 части I, TxF влияет на обычные, не имеющие отношения к транзакциям приложения, так как это гарантирует *изоляцию* (isolation) транзакций. К примеру, если антивирусная программа сканирует файл, в который посредством транзакции начало вносить изменения другое приложение, файловая система TxF должна гарантировать, что сканер будет читать исходную версию данных, имевшую место до начала транзакции, в то время как выполняющее транзакцию приложение будет работать с их модифицированной версией. Такая модель называется *изоляцией подтверждения по чтению* (read-committed isolation).

Изоляция подтверждения по чтению связана с такими концепциями, как *транзакционные писатели* (transacted writers) и *транзакционные читатели* (transacted readers). Первые всегда видят наиболее актуальную версию данных файла, включая все связанные в данный момент с файлом изменения в процессе транзакции. В любой момент времени может существовать только один транзакционный писатель, что означает монопольный доступ на запись. В то же время транзакционные читатели к моменту открытия ими файла имеют доступ только к подтвержденной версии данных. Следовательно, они изолированы от изменений, вносимых транзакционными писателями. Именно это позволяет им получать согласованные данные, даже когда транзакционный писатель подтверждает свои изменения. Для просмотра обновленных данных транзакционный читатель должен открыть новый дескриптор для модифицированной версии файла.

Что касается нетранзакционных писателей, то им транзакционные писатели и читатели просто не дают открыть файл. То есть они не могут вносить изменения, так как не участвуют в транзакции. Как нетранзакционные, так и транзакционные читатели видят содержимое файла, подтвержденное в момент открытия дескриптора файла. Однако нетранзакционным читателям изоляция подтверждения по чтению недоступна, поэтому им всегда предоставляется обновленная версия последних подтвержденных в транзакционном файле данных. Им не требуется для этого открывать новый дескриптор файла. Это позволяет вести себя ожидаемым образом тем приложениям, в которых наличие транзакций не учитывается.

Подведем итоги. Модель изоляции подтверждения по чтению в TxF имеет следующие характеристики:

- изменения изолированы от транзакционных читателей;
- изменения откатываются назад при откате связанной с ними транзакции, сбое компьютера или принудительном размонтировании тома;
- если транзакция подтверждается, связанные с ней изменения сбрасываются на диск.

ЭКСПЕРИМЕНТ: ПОНИМАНИЕ ТРАНЗАКЦИЙ И УПРАВЛЕНИЕ ТРАНЗАКЦИЯМИ

В этом эксперименте мы с помощью инструмента Transactdemo.exe создадим файл, добавив в него данные, реализовав этот процесс как часть транзакции, и посмотрим, как во время транзакции с файлом взаимодействуют нетранзакционные клиенты. Откройте окно командной строки и запустите программу Transactdemo.exe:

```
C:\>Transactdemo.exe
```

```
Transaction Demo v1.0  
by Mark Russinovich
```

```
Transaction created: {5CD5E900-9DA8-11DD-8379-005056C00008}
```

```
Created C:\TransactionDemo.txt.  
Pass TransDemo the GUID listed above to see the transacted file.
```

```
Rollback or commit transaction? (r/c):
```

Программа Transactdemo создает в процессе неподтвержденной транзакции файл C:\TransactionDemo.txt. Откройте второе окно командной строки и при помощи команды dir посмотрите, есть ли в наличии файл TransactionDemo.txt:

```
C:\>dir transactiondemo.txt  
Volume in drive C is OS  
Volume Serial Number is 0C30-686E
```

```
Directory of C:\  
File Not Found
```

Как видите, такого файла просто не существует. Теперь давайте симулируем нетранзакционного писателя и попытаемся добавить в файл данные командой echo:

```
C:\>echo Hello > TransactionDemo.txt  
The function attempted to use a name that is reserved for use by another trans-  
action.
```

Как и ожидалось, нетранзакционный писатель не может вносить изменения в файл.

Для работы с транзакционными операциями в этой файловой системе могут пригодиться встроенные приложения %SystemRoot%\System32\Ktmutil.exe и %SystemRoot%\System32\Fsutil.exe. К примеру, следующая команда позволяет получить список текущих транзакций в системе:

```
C:\>ktmutil tx list
TxGuid Description
-----
{5cd5e900-9da8-11dd-8379-005056c00008} Demo Transaction?
```

Обратите внимание, что GUID в данном случае совпадает с идентификатором, который был возвращен программой Transactdemo. Имея GUID, можно воспользоваться командой Fsutil и затребовать сведения о транзакции, чтобы подтвердить или откатить ее. Вот как выглядит список принимающих участие в транзакции файлов и учетная запись владельца:

```
C:\>fsutil transaction query all {5cd5e900-9da8-11dd-8379-005056c00008}
dwOutcome: 1
dwIsolationLevel: 0
dwIsolationFlags: 0
dwTimeout: -1
Owner: BUILTIN\Administrators
Number of Files: 1
---- \TransactionDemo.txt
```

Утилита Transactdemo дает возможность откатить или подтвердить текущую транзакцию, а утилита Fsutil позволяет работать с любыми активными транзакциями, к которым у вас есть доступ. Вернитесь к командной строке, из которой вы запускали Transactdemo, и нажмите клавишу С для подтверждения транзакции. После этого у вас останется обычный (нетранзакционный) файл.

Транзакционные API-интерфейсы

В TxF реализованы транзакционные версии прикладных программных интерфейсов (API) файлового ввода-вывода. В их именах присутствует суффикс «Transacted»:

- ❑ API для создания: CreateDirectoryTransacted, CreateFileTransacted, CreateHardLinkTransacted, CreateSymbolicLinkTransacted;
- ❑ API для поиска: FindFirstFileNameTransacted, FindFirstFileTransacted, FindFirstStreamTransacted;
- ❑ API для запросов: GetCompressedFileSizeTransacted, GetFileAttributesTransacted, GetFullPathNameTransacted, GetLongPathNameTransacted;
- ❑ API для удаления: DeleteFileTransacted, RemoveDirectoryTransacted;
- ❑ API для копирования и перемещения/переименования: CopyFileTransacted, MoveFileTransacted;
- ❑ API для настройки: SetFileAttributesTransacted.

Кроме того, некоторые API-интерфейсы автоматически принимают участие в транзакционных операциях, когда переданный им дескриптор файла является частью транзакции.

закции, подобно файлу, созданному при помощи API-интерфейса `CreateFileTransacted`. Перечень API-интерфейсов, поведение которых меняется, когда им приходится иметь дело с дескриптором транзакционного файла, приведен в табл. 12.9.

Таблица 12.9. API-интерфейсы, поведение которых меняет TxF

Имя API	Изменение
CloseHandle	Транзакции не подтверждаются, пока все приложения не закроют транзакционные дескрипторы файла
CreateFileMapping, MapViewOfFile	Изменения, вносимые в отображенные представления файла, участвующего в транзакции, ассоциированы с самой транзакцией
FindNextFile, ReadDirectory- Changes, GetInformationBy- Handle, GetFileSize	Если дескриптор файла является частью транзакции, к этим операциям применяются правила изоляции чтения
GetVolumeInformation	Если рассматриваемый том поддерживает TxF, функция возвращает значение <code>FILE_SUPPORTS_TRANSACTIONS</code>
ReadFile, WriteFile	Операции чтения и записи для транзакционного дескриптора файла являются частью транзакции
SetFileInformationByHandle	Если дескриптор файла является частью транзакции, изменения в классах <code>FileBasicInfo</code> , <code>FileRenameInfo</code> , <code>FileAllocationInfo</code> , <code>FileEndOfFileInfo</code> и <code>FileDispositionInfo</code> проходят как транзакционные
SetEndOfFile, SetFileShort- Name, SetFileTime	Изменения обрабатываются в рамках транзакции, если дескриптор файла является частью транзакции

Диспетчеры ресурсов

Аналогично тому, как в TxR диспетчер ресурсов (Resource Manager, RM) используется для отслеживания метаданных транзакций и файлов журналов, в TxF на каждом томе используется *предлагаемый по умолчанию диспетчер ресурсов* (default resource manager) для отслеживания состояния транзакций. Кроме того, TxF поддерживает дополнительные диспетчеры ресурсов, называемые *вспомогательными* (secondary resource managers). Эти диспетчеры могут создаваться разработчиками приложений и помещать свои метаданные в любую папку по выбору приложения, определяя собственные транзакционные рабочие единицы для операций отмены, резервного копирования, восстановления и повторения. TxF задействует предлагаемый по умолчанию диспетчер ресурсов для транзакционных API-интерфейсов и приложений, которые выполняют транзакции с *координатором распределенных транзакций* (Distributed Transaction Coordinator). Кроме того, вспомогательные диспетчеры ресурсов создают классы `System.Transaction` из .NET Framework при помощи специальных команд. Приложения могут создавать и управлять вспомогательными диспетчерами ресурсов при помощи заданных для TxF управляющих кодов, например `FSCTL_TXFS_CREATE_SECONDARY_RM`, `FSCTL_TXFS_START_RM` и `FSCTL_TXFS_SHUTDOWN_RM`. Обеспечить целостность нового вспомогательного диспетчера ресурсов должен один или несколько вызовов `FSCTL_TXFS_ROLLFORWARD_REDO`, за которыми следует вызов `FSCTL_TXFS_ROLLFORWARD_`

UNDO. Эти операции предназначены для обработки транзакций, которые записаны в журнал, но не подтверждены (например, в случае сбоя компьютера). Мы вкратце опишем процедуру восстановления для диспетчеров ресурсов. Как предлагаемый по умолчанию диспетчер, так и вспомогательные диспетчеры содержат ряд файлов метаданных, которые описывают текущее состояние:

- ❑ С папкой **\$Txf** связываются файлы, удаляемые или переписываемые во время транзакций. Если файл удаляется во время транзакции, согласно правилам изоляции чтения нетранзакционные читатели должны иметь доступ к файлу, пока операция удаления не будет реально подтверждена. Такая изоляция достигается перемещением удаляемого в процессе транзакции файла в папку **\$Txf**. После этого NTFS-драйвер отследит изоляцию, вставив временную структуру в SCB родительской папки, в которой изначально находились удаленные файлы. В результате, если родитель перечислен, файл продолжит выводиться на экран и сохранит номер записи, что позволит его открывать. После подтверждения транзакции NTFS удаляет временную структуру и файл из папки **\$Txf**. Откат транзакции сопровождается возвращением файла в папку, где он хранился изначально.
- ❑ Файл **\$Tops**, или файл потока данных старой TxF-страницы (**TxF Old Page Stream, TOPS**), содержит предлагаемый по умолчанию поток данных и дополнительный поток данных **\$T**. Предлагаемый по умолчанию поток данных содержит метаданные, связанные с диспетчером ресурсов, например его GUID, политики его CLFS-журнала и LSN-номер, с которого должно начинаться восстановление. Поток **\$T** содержит файл данных, который частично перезаписан транзакционным писателем (но это не полная перезапись, в результате которой файл переместился бы в папку **\$Txf**). В NTFS-памяти присутствует структура, отслеживающая, какие части файла претерпели изменения в процессе транзакции, что обеспечивает нетранзакционным читателям возможность доступа к неподтвержденным данным за счет перенаправления их к **\$Tops:\$T**. После подтверждения или прерывания транзакции страницы из потока **\$T** возвращаются в исходный файл или (в случае прерывания) просто уничтожаются.
- ❑ Файлы TxF-журнала представляют собой файлы CLFS-журнала, хранящие записи о транзакциях. Для предлагаемого по умолчанию диспетчера ресурсов они являются частью папки **\$TxfLog**, в то время как вспомогательные диспетчеры ресурсов могут хранить их где угодно. TxF использует мультиплексный файл базового журнала с именем **\$TxfLog.blf**. Этот файл (**\\$Extend\\$RmMetadata\\$TxfLog\\$TxfLog**) содержит два потока данных: поток **KtmLog** используется для записей метаданных диспетчера транзакций ядра, а поток **TxfLog** содержит записи TxF-журнала. Оба потока данных хранятся в контейнерах CLFS-журнала, начальным из которых является **\$TxfLogContainer**, а следующие вычисляются по уникальному увеличивающемуся идентификатору, например **00000000000000000000000000000001**. По мере роста TxF-журнала создаются дополнительные контейнеры.

Как уже упоминалось, предлагаемый по умолчанию диспетчер ресурсов хранит свои файлы в папке **\\$Extend\\$RmMetadata** на каждом томе компьютера, отформатированном для NTFS.

ЭКСПЕРИМЕНТ: ПОЛУЧЕНИЕ ИНФОРМАЦИИ О ДИСПЕТЧЕРЕ РЕСУРСОВ

Встроенная утилита командной строки %SystemRoot%\System32\Fsutil.exe позволяет получать информацию о предлагаемом по умолчанию диспетчере ресурсов, а также создавать, запускать и останавливать вспомогательные диспетчеры ресурсов, настраивать их политики протоколирования и поведения. Следующая команда посылает запрос на получение сведений о предлагаемом по умолчанию диспетчере ресурсов, который определяется корневым каталогом ():

```
C:\>fsutil resource info \
RM Identifier: CF7234E7-39E3-11DC-BDCE-00188BDD5F49
KTM Log Path for RM: \Device\HarddiskVolume3\$Extend\
$RmMetadata\$TxfLog\$TxfLog::KtmLog
Space used by TOPS: 79 Mb
TOPS free space: 100%
RM State: Active
Running transactions: 0
One phase commits: 0
Two phase commits: 1
System initiated rollbacks: 0
Age of oldest transaction: 00:00:00
Logging Mode: Simple
Number of containers: 2
Container size: 10 Mb
Total log capacity: 20 Mb
Total free log space: 14 Mb
Minimum containers: 2
Maximum containers: 20
Log growth increment: 2 container(s)
Auto shrink: Not enabled

RM prefers availability over consistency.
```

Как уже упоминалось, команда fsutil resource имеет множество параметров для настройки TxF-диспетчеров ресурсов. В числе прочего она позволяет создать вспомогательный диспетчера ресурсов в любой папке по вашему выбору. К примеру, команда fsutil resource create c:\rmtest приводит к появлению вспомогательного диспетчера ресурсов в папке Rmtest, а команда fsutil resource start c:\rmtest инициализирует его. Обратите внимание на наличие в этой папке файлов \$Tops и \$TxfLogContainer*, а также папок TxfLog и \$Txf.

Реализация на диске

Как показано в табл. 12.6, тип атрибута \$LOGGED.Utility_Stream TxF использует для хранения дополнительных данных файлов и папок, которые являются или являлись частью транзакции. Этот атрибут называется \$TxF_DATA и содержит важные сведения, позволяющие TxF поддерживать в активном состоянии автономные данные для принимающего участие в транзакции файла. Он хранится в MFT; то есть даже после того, как файл перестает быть частью транзакции, поток данных остается. Причину этого мы вкратце рассмотрим. Основные компоненты этого атрибута представлены на рис. 12.50.

Первое из показанных полей представляет собой индекс файловой записи корня диспетчера ресурсов, который отвечает за связанные с рассматриваемым файлом транзакции. Для предлагаемого по умолчанию диспетчера ресурсов этот индекс равен 5, что совпадает с индексом файловой записи корневого каталога () в MFT, как

показано на рис. 12.27. Эта информация требуется TxF при создании FCB для файла, чтобы его можно было связать с нужным диспетчером ресурсов. Последний, в свою очередь, должен создать привязку к транзакции, когда NTFS получает запрос на транзакционный файл. (Более подробно связь между диспетчером ресурсов и транзакцией рассматривается в главе 3 части I.)

Индекс файловой записи RM-корня
Флаги
TxF-идентификатор файла (TxID)
LSN для NTFS-метаданных
LSN для пользовательских данных
LSN для индекса папки
USN-индекс

Рис. 12.50. Атрибут \$TXF_DATA

Еще одним важным фрагментом данных, хранящимся в атрибуте \$TXF_DATA, является TxF-идентификатор файла (TxID). Именно из-за него нельзя удалять атрибуты \$TXF_DATA. Так как NTFS заносит в свои записи имена файлов при создании журнала транзакций, требуется способ уникальной идентификации файлов из одной папки, которые могли иметь одно и то же имя. К примеру, если в процессе транзакции из папки был удален файл sample.txt, а позднее там создан новый файл с таким же именем (причем в ходе той же самой транзакции), TxF нужно как-то различать эти два экземпляра. Для идентификации в данном случае используется TxID — уникальное 64-разрядное число, которое TxF увеличивает на единицу, когда в транзакции появляется новый файл (или новый экземпляр файла). Так как их нельзя использовать повторно, TxF-идентификаторы файлов являются фиксированными, и атрибут \$TXF_DATA из файла никогда не удаляется.

Наконец, не менее важно упомянуть, что для каждого принимающего участие в транзакции файла в CLFS сохраняются три LSN-номера. При выполнении транзакции, например во время операции создания, переименования или записи, TxF делает запись в CLFS-журнале. Каждой записи присваивается LSN-номер, который записывается в соответствующее поле атрибута \$TXF_DATA. Первый LSN-номер служит для хранения записи журнала, идентифицирующей изменения в NTFS-метаданных, касающиеся рассматриваемого файла. К примеру, если в процессе транзакции изменяются стандартные атрибуты файла, файловая система TxF должна обновить соответствующую MFT-запись. При этом сохраняется LSN для записи журнала, описывающей изменение. Второй LSN-номер TxF задействует при редактировании данных файла. Наконец, третий LSN-номер используется, когда индекс имени файла для папки требует изменений, связанных с транзакцией, в которой принимает участие файл, или когда папка является частью транзакции и получила TxID.

Еще атрибут \$TXF_DATA имеет внутренние флаги, сообщающие TxF сведения о состоянии, и индекс USN-записи, назначаемый файлу в момент подтверждения транзакции.

закции. TxF-транзакция может занимать несколько USN-записей, которые частично обновляются NTFS-механизмом восстановления (мы вкратце о нем рассказывали), поэтому индекс сообщает TxF, как много USN-записей следует применить после восстановления.

Реализация протоколирования

Как уже упоминалось, при каждом внесении изменений на диск, причиной которого становится транзакция, TxF делает запись в журнал. Для отслеживания связанных с транзакциями изменений существует множество типов записей. При этом все записи имеют универсальный заголовок, содержащий сведения о типе записи, действии, в результате которого она появилась, назначенному этой записи TxF-идентификаторе файла (TxID), а также GUID-идентификаторе KTM-транзакции, с которым ассоциирована запись.

Запись повторения (redo record) определяет, как повторно применить к тому изменившуюся часть уже подтвержденной транзакции, если транзакция не была сброшена из кэша на диск. В то же время *запись отмены* (undo record) определяет, как в ходе отката вернуть в исходное состояние изменившуюся часть неподтвержденной транзакции. Некоторые записи относятся исключительно к записям повторения, что означает отсутствие в них необходимых для отмены данных, в то время как другие записи могут содержать информацию как для отмены, так и для повторения.

Через TOPS-файл TxF поддерживает два важных фрагмента данных: *базовый LSN-номер* (base LSN) и *LSN-номер перезапуска* (restart LSN). Первый представляет собой LSN-номер первой действительной записи в журнале, а второй указывает, с какого LSN-номера должно начинаться восстановление при запуске диспетчера ресурсов. При внесении *записи перезапуска* (restart record) TxF обновляет эти два значения, указывая, что в том были внесены изменения, которые затем сбросили на диск. Это означает, что файловая система полностью согласована до нового LSN-номера перезапуска.

Также TxF вносит *компенсационные записи журнала* (Compensating Log Records, CLR). Там хранятся действия, выполнявшиеся в процессе отката транзакции (мы по-говорим о них позже). В основном они используются для хранения *следующего после отмены LSN-номера* (undo-next LSN), позволяя процессу восстановления избежать повторного выполнения операций отмены за счет пропуска уже обработанных записей, касающихся этой операции. Такая ситуация может возникнуть, если системный сбой случился в процессе восстановления, когда часть операций в рамках отмены уже выполнена. Наконец, TxF работает также с *записями подготовки* (reread records), *прерывания* (abort records) и *подтверждения* (commit records), которые описывают состояние KTM-транзакций по отношению к TxF.

Реализация восстановления

Когда диспетчер ресурсов запускается вследствие вызова FSCTL_TXFS_START_RM (а в случае предлагаемого по умолчанию диспетчера ресурсов — сразу же после монтирования тома), TxF инициирует процесс восстановления. TxF читает TOPS-файл, чтобы определить LSN-номер перезапуска, с которого должен начаться процесс восстановления, а затем начинает считывать все записи журнала в прямом направлении, выполняя так называемый *проход повторения* (redo pass). При обработке каждой записи TxF открывает

файл, на который она ссылается, и сравнивает LSN в атрибуте `$TxF_DATA` с LSN в записи. Если первый LSN-номер больше или равен второму, действие не производится, так как находящаяся на диске копия файла имеет такую же давность или новее, чем запись в журнале. Обратная же ситуация означает, что журнал содержит информацию о файле, которой никогда не было в самом файле. В этом случае TxF применяет действие, указанное в записи, и обновляет LSN в атрибуте `$TxF_DATA` значением LSN из записи.

В процессе прохода повторения TxF строит *таблицу транзакций* (transaction table), в которой описываются завершенные операции; если при этом встречаются операции прерывания или подтверждения, TxF отбрасывает связанные с ними транзакции. К концу прохода повторения TxF анализирует полученную таблицу транзакций и подключается к КТМ, чтобы проверить, что записано для транзакций — подтверждение или прерывание. (Как уже объяснялось, КТМ хранит эту информацию в потоке данных `KtmLog` мультиплексного TxF-журнала.)

После того как файловая система TxF завершает взаимодействие с КТМ, она проверяет таблицу транзакций в поиске незавершенных транзакций и начинает *проход отмены* (undo pass). Во время этого прохода TxF прерывает все незавершенные транзакции, обходя все транзакции в цепочке LSN-номеров записей отмены и применяя к каждой записи в журнале операцию отмены. В конце прохода отмены диспетчер ресурсов оказывается согласованным и инициализированным.

Этот процесс напоминает процедуру восстановления службы файла журнала, которая детально рассматривается в конце главы, довершая картину стандартных транзакционных механизмов восстановления.

Поддержка восстановления в NTFS

Поддержка восстановления в NTFS гарантирует, что в случае отключения электропитания или аварии системы ни одна операция файловой системы (транзакция) не останется незавершенной, а структура дискового тома будет сохранена без запуска программы восстановления диска. Служебная программа `Chkdsk` позволяет устранить последствия катастрофических повреждений диска, вызванных аппаратными ошибками ввода-вывода (например, аварийными секторами на диске, электрическими аномалиями или сбоями в работе диска) или ошибками в программном обеспечении. Однако благодаря наличию в NTFS средств восстановления потребность в применении программы `Chkdsk` возникает не часто.

Как уже упоминалось в разделе «Восстанавливаемость», в NTFS восстанавливаемость реализуется на базе механизма транзакций. Это гарантирует полное и исключительно быстрое (за секунды) восстановление даже самых больших дисков. NTFS ограничивает процедуры восстановления данными файловой системы, гарантируя, что пользователь, по крайней мере, не потеряет весь том из-за ее повреждения. Однако если приложение не выполняет определенных обязательных действий, например не сбрасывает на диск кэшированные файлы, NTFS не сможет гарантировать полное восстановление данных в случае сбоя. Это работа для транзакционной версии NTFS (TxF).

В следующем разделе мы подробно остановимся на схеме протоколирования транзакций, за счет которой NTFS регистрирует изменения в структурах данных файло-

вой системы. Также мы поговорим о том, каким образом NTFS восстанавливает том в случае системного сбоя.

Техническое решение

В NTFS реализован механизм *восстанавливаемой файловой системы* (recoverable file system). Такие системы гарантируют целостность тома за счет приемов протоколирования, иногда называемых *журналированием* (journaling), изначально рассчитанных на обработку транзакций. В случае сбоя операционной системы восстанавливаемая файловая система возвращает свою целостность, выполняя процедуру восстановления путем обращения к хранящейся в файле журнала информации. Так как все операции записи на диск регистрируются в журнале, восстановление занимает несколько секунд вне зависимости от размера тома (в отличие от файловой системы FAT, в которой время восстановления связано с размером тома). Процедура восстановления в восстанавливаемой файловой системе является точной и гарантирует возвращение тома в целостное состояние.

За надежность восстанавливаемой системы приходится платить. Каждая меняющая структуру тома транзакция должна сопровождаться записью в файл журнала всех составляющих ее операций. Файловая система снижает издержки протоколирования за счет объединения записей файла журнала в пакеты. То есть за одну операцию ввода-вывода в журнал добавляется несколько записей. Кроме того, восстанавливаемая файловая система может применять алгоритмы оптимизации, используемые файловыми системами с отложенной записью. Она может даже увеличить интервалы между сбросами данных из кэша на диск, так как метаданные файловой системы поддаются восстановлению, если сбой происходит до того, как изменения в кэше переписаны на диск. Производительность кэширования в файловых системах с отложенной записью часто компенсирует издержки, связанные с протоколированием транзакций.

Однако ни одна из этих файловых систем не гарантирует защиты пользовательских данных. Если сбой происходит тогда, когда приложение выполняет запись в файл, файл может быть утерян или поврежден. И что хуже всего, сбой может повредить файловую систему с отложенной записью, уничтожив существующие файлы или даже сделав недоступной всю информацию на томе.

Восстанавливаемая файловая система NTFS реализует ряд стратегий, повышающих ее надежность по сравнению с традиционными файловыми системами. Во-первых, благодаря восстанавливаемости гарантируется, что структура NTFS-тома не будет повреждена и после системного сбоя доступ ко всем файлам сохранится. Во-вторых, несмотря на отсутствие гарантий сохранения пользовательских данных в случае системного сбоя (некоторые изменения в кэше могут быть потеряны), приложения могут использовать преимущества сквозной записи и сброса кэша для записи на диск изменений файлов через подходящие интервалы времени.

Как *сквозная запись кэша* (cache write-through), то есть немедленная запись данных на диск с их дублированием в кэше, так и *сброс кэша* (cache flushing), то есть принудительная запись на диск содержимого кэша — операции вполне эффективные. NTFS не требует дополнительного ввода-вывода для сброса на диск изменений ряда структур файловой системы, так как изменения в этих структурах регистрируются в файле журнала в ходе единственной операции записи. Если в результате сбоя содержимое

кэша оказывается утраченным, изменения файловой системы могут быть восстановлены при помощи журнала. Более того, в отличие от FAT NTFS гарантирует, что сразу после сквозной записи или сброса кэша данные пользователей сохранят целостность и доступность даже при последующем сбое системы.

Протоколирование метаданных

Восстанавливаемость файловой системы NTFS обеспечивается за счет той же самой техники протоколирования, которая применяется в TxF. Ее смысл состоит в записи в журнал всех операций, меняющих метаданные файловой системы. Однако в отличие от TxF встроенный в NTFS механизм восстановления файловой системы не использует CLFS. Вместо CLFS применяется внутренняя реализация протоколирования, так называемая *служба файла журнала* (Log File Service, LFS). Это не фоновый служебный процесс, который описан в главе 4 части I. Кроме того, в отличие от файловой системы TxF, которая применяется только в случае, когда вызывающий поток данных действует в рамках транзакций, NTFS записывает все изменения метаданных, чтобы обеспечить целостность файловой системы в случае системного сбоя.

Служба файла журнала

Служба файла журнала представляет собой набор процедур режима ядра, локализованных внутри NTFS-драйвера, которые NTFS использует для доступа к файлу журнала. NTFS передает LFS указатель на открытый файловый объект, который соответствует нужному файлу журнала. LFS либо инициализирует новый журнал, либо вызывает диспетчер кэша для доступа к существующему журналу через кэш, как показано на рис. 12.51. Обратите внимание, что несмотря на сходство аббревиатур LFS и CLFS, это разные реализации протоколирования, имеющие разное назначение, хотя их функционирование во многом схоже.

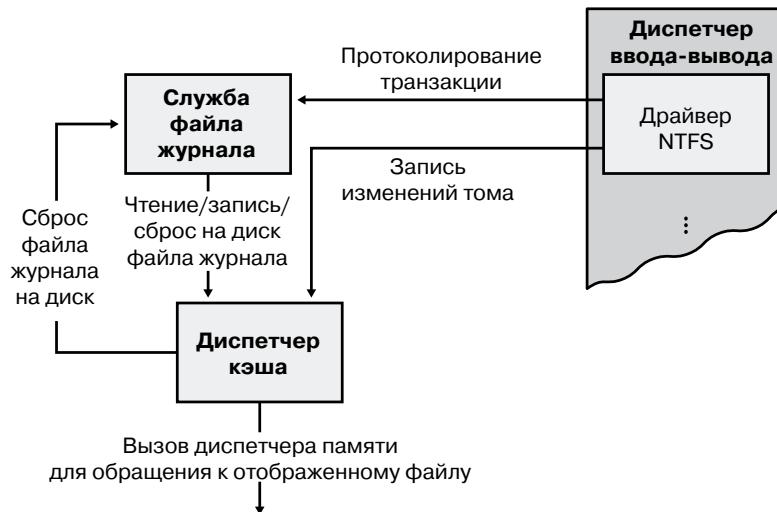


Рис. 12.51. Служба файла журнала (LFS)

Как показано на рис. 12.52, LFS делит файл журнала на две части: *область перезапуска* (restart area) и «безразмерную» *область протоколирования* (logging area).

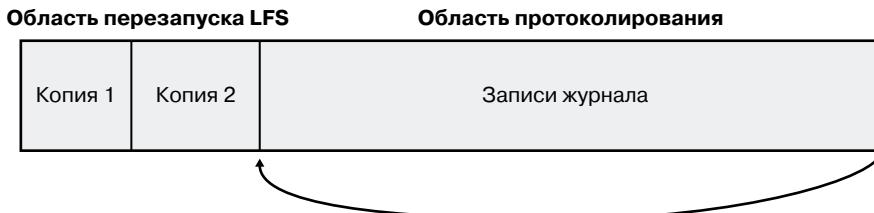


Рис. 12.52. Области файла журнала

Для чтения и записи области перезапуска NTFS вызывает LFS. В этой области NTFS хранит контекстную информацию, например место в области протоколирования, с которого начинается чтение при восстановлении системы после сбоя. На случай разрушенной или по каким-то причинам недоступной области перезапуска LFS поддерживает ее копию. Остальная часть журнала транзакций отдана под область протоколирования, в которой находятся записи транзакций, обеспечивающие восстановление тома после сбоя. За счет использования журнала транзакций в цикле LFS создает иллюзию его бесконечности (при этом нужная информация гарантированно не перезаписывается). Для идентификации помещенных в журнал записей LFS, как и CLFS, использует LSN-номера. В процессе циклического использования LSN-номера увеличиваются. Для представления LSN-номеров NTFS использует 64 бита, поэтому количество возможных LSN-номеров является практически бесконечным.

NTFS никогда непосредственно не выполняет чтение или запись транзакций. LFS предоставляет службы, которые NTFS вызывает для открытия файла журнала, помещения туда записей, чтения записей в прямом и обратном порядке, сброса записей до заданного LSN-номера или установки логического начала журнала на больший LSN-номер. В процессе восстановления NTFS вызывает LFS для выполнения операций, описанных в разделе, посвященном восстановлению TxF: проход повторения выполняется для не сбрасывавшихся на диск подтвержденных изменений, а за ним следует проход отмены для неподтвержденных изменений.

Вот как система обеспечивает восстановление тома:

1. Сначала NTFS вызывает LFS для записи в кэшируемый журнал любых транзакций, меняющих структуру тома.
2. NTFS модифицирует том (в кэше тоже).
3. Диспетчер кэша предлагает LFS сбросить на диск файл журнала. Этот сброс реализуется путем обратного вызова диспетчера кэша с указанием страниц памяти, подлежащих выводу на диск (взгляните на последовательность вызовов на рис. 12.51).
4. После сброса журнала транзакций на диск диспетчер кэша записывает туда изменения тома (собственно, результаты операций над метаданными).

Эта последовательность действий гарантирует, что если изменения в файловой системе окажутся фатальными, соответствующие транзакции можно будет считать

из журнала и повторить или отменить их в рамках процедуры восстановления файловой системы.

Восстановление файловой системы начинается автоматически при первом обращении к дисковому тому после перезагрузки. NTFS проверяет, были ли применены к тому транзакции, запротоколированные в журнале до сбоя, и если нет, повторяет их. Также NTFS гарантирует, что не полностью зафиксированные до сбоя транзакции будут отменены и никак не повлияют на состояние тома.

Типы записей журнала

Механизм восстановления NTFS использует те же типы записей журнала, что и механизм восстановления TxF: *записи обновления* (update records) соответствуют записям отмены и повторения в TxF, а *записи контрольных точек* (checkpoint records) аналогичны записям перезапуска в TxF. На рис. 12.53 показаны три записи обновления в файле журнала. Каждая запись представляет одну подоперацию в составе транзакции, создавая новый файл. Запись повторения в каждой записи обновления объясняет NTFS, каким образом следует повторно применить подоперацию к тому, а запись отмены — как откатить (отменить) эту подоперацию.

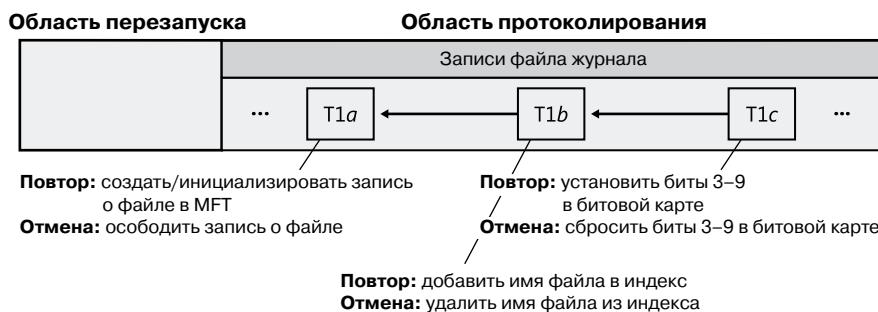


Рис. 12.53. Записи обновления в файле журнала

После протоколирования транзакции (в данном примере путем вызова LFS для внесения в журнал трех записей обновления) NTFS выполняет в кэше подоперации над самим томом. Завершив обновление кэша, NTFS помещает в журнал еще одну запись, которая указывает на завершение транзакции — данная подоперация называется *подтверждением транзакции* (committing a transaction). После подтверждения транзакции NTFS гарантирует, что результат транзакции будет виден на томе, даже если затем произойдет сбой операционной системы.

В процессе восстановления после сбоя NTFS просматривает журнал и повторяет все подтвержденные транзакции. Даже если транзакция была подтверждена до системного сбоя, NTFS не знает, успел ли диспетчер кэша сбросить изменения тома на диск. При сбое обновления могли пропасть из кэша. Поэтому, просто чтобы гарантировать актуальность состояния диска, NTFS повторно выполняет подтвержденную транзакцию.

После повторного выполнения подтвержденных транзакций NTFS ищет в журнале транзакции, которые не были подтверждены к моменту сбоя, и производит откат каждой

запротоколированной подоперации. В случае, показанном на рис. 12.53, NTFS сначала нужно отменить подоперацию T1c, а затем перейти по указателям назад и отменить подоперацию T1b. Переходы по указателям в обратном направлении и отмены должны продолжаться до достижения первой подоперации в транзакции. Следуя указателям, NTFS определяет, сколько и каких записей обновления следует отменить для отката транзакции.

Информация для повторения и отмены может быть выражена как физически, так и логически. В качестве самого нижнего уровня программного обеспечения, поддерживающего структуру файловой системы, NTFS сопровождает записи обновления физическим описанием. В этом случае обновления тома задаются в виде диапазона байтов на диске, подлежащих редактированию, перемещению и т. д. Этим NTFS отличается от файловой системы TxF, в которой применяются логические описания, выражающие обновления в терминах операций, например «удалить файл A.dat». NTFS генерирует записи обновления (обычно сразу несколько) для каждой из следующих транзакций:

- создание файла;
- удаление файла;
- увеличение файла;
- усечение файла;
- задание файловой информации;
- переименование файла;
- изменение прав доступа к файлу.

Информация для повторения и отмены в записи обновления должна быть очень точной, иначе при отмене транзакции, восстановлении после сбоя или даже в ходе нормальной работы системы NTFS может попытаться повторить уже выполненную транзакцию или, наоборот, отменить транзакцию, которая никогда не выполнялась или была отменена. Также NTFS может попытаться повторить или отменить транзакцию, состоящую из набора записей обновления, только часть из которых была записана на диск. Формат записей обновления должен гарантировать, что лишние операции повторения и отмены будут *идемпотентными* (idempotent), то есть не будут оказывать никакого эффекта. К примеру, установка бита, который уже был установлен, не оказывает никакого эффекта, чего нельзя сказать об изменении значения бита на противоположное в случае, когда эта операция уже один раз имела место. Кроме того, файловая система должна корректно обрабатывать переходные состояния тома.

Кроме записей обновления NTFS периодически помещает в журнал записи контрольных точек, как показано на рис. 12.54.

Запись контрольной точки помогает NTFS определить, какая обработка требуется для восстановления тома, если сбой произошел сразу после добавления записи в журнал. Благодаря сохраненной при этом информации NTFS, к примеру, знает, как далеко нужно переместиться по журналу, чтобы начать восстановление. После записи контрольной точки NTFS сохраняет ее LSN в области перезапуска, что позволяет при восстановлении после сбоя быстро найти самую последнюю запись. Это напоминает LSN-номер перезагрузки, которым в аналогичной ситуации пользуется TxF.

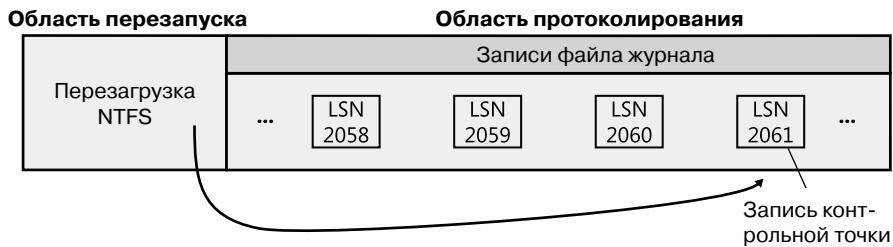


Рис. 12.54. Запись контрольной точки в файле журнала

Хотя LFS создает впечатление безразмерного журнала транзакций в NTFS, на самом деле это не так. Хотя значительный размер файла журнала и частая вставка записей контрольных точек (операция, обычно освобождающая место в файле журнала) делают вероятность переполнения крайне малой, LFS, как и CLFS, учитывает такую возможность и отслеживает ряд параметров:

- количество свободного места в журнале;
- пространство, необходимое для добавления в журнал следующей записи и отмены этого действия при необходимости;
- пространство, необходимое для отката всех активных (неподтвержденных) транзакций, если это потребуется.

Если в журнале не хватает места для суммы последних двух значений списка, LFS возвращает ошибку переполнения журнала («log file full»), а NTFS запускает исключение. Обработчик исключений в NTFS откатывает текущую транзакцию и помещает ее в очередь для последующего перезапуска.

Чтобы освободить пространство в журнале транзакций, NTFS нужно временно приостановить все дальнейшие транзакции для файлов. Для этого блокируется создание и удаление файлов, после чего NTFS запрашивает монопольный доступ ко всем системным файлам и совместный доступ ко всем файлам пользователей. Постепенно активные транзакции либо успешно завершаются, либо приводят к исключению переполнения файла журнала. В последнем случае NTFS откатывает их и помещает в очередь.

Заблокировав описанным образом выполнение транзакций для файлов, NTFS вызывает диспетчер кэша для сброса незаписанных данных на диск, в том числе данных файла журнала. После того как все успешно записано на диск, содержимое журнала для NTFS уже не требуется. NTFS устанавливает начало журнала на текущую позицию, что делает его «пустым». После этого запускаются поставленные ранее в очередь транзакции. Так что за исключением короткой паузы в обработке ввода-вывода ошибка переполнения файла журнала никак не влияет на работу программ.

Описанный сценарий является одним из примеров того, как NTFS использует файл журнала не только для восстановления файловой системы, но и для исправления ошибок, возникающих в процессе обычного функционирования. Более подробно о восстановлении мы поговорим в следующем разделе.

Восстановление

После загрузки системы при первом обращении к диску какой-либо программы NTFS автоматически производит восстановление диска. (Если восстановление не требуется, процесс становится тривиальным.) Для восстановления нужны две таблицы, которые NTFS поддерживает в памяти. Во-первых, это таблица транзакций, которая ведет себя как аналогичная таблица в TxF, во-вторых, *таблица измененных страниц* (dirty page table), в которую записывается информация о том, какие страницы кэша содержат еще не записанные на диск изменения структуры файловой системы. В процессе восстановления эти данные требуется сбросить на диск.

Каждые 5 секунд NTFS добавляет в файл журнала транзакций запись контрольной точки. Непосредственно перед этим NTFS обращается к LFS для сохранения в журнале текущей копии таблицы транзакций и таблицы измененных страниц. Затем NTFS запоминает в записи контрольной точки LSN-номера записей журнала, в которых содержатся копии таблиц. В начале процесса восстановления после сбоя NTFS обращается к LFS для поиска записей журнала транзакций, содержащих самую последнюю запись контрольной точки и самые последние копии упомянутых таблиц. Эти таблицы NTFS копирует в память.

Обычно за последней записью контрольной точки в журнале транзакций присутствует еще несколько записей обновления. Эти записи представляют изменения на томе, произошедшие после помещения в журнал последней записи контрольной точки. NTFS нужно обновить таблицу транзакций и таблицу измененных страниц, включив туда информацию из этих дополнительных записей. После обновления таблиц NTFS пользуется ими и содержимым журнала транзакций для обновления самого тома.

Чтобы выполнить восстановление тома, NTFS три раза сканирует файл журнала, при первом проходе загружая его в память, чтобы минимизировать дисковый ввод-вывод. Каждый проход имеет собственное назначение:

1. Анализ.
2. Повторение транзакций.
3. Отмена транзакций.

Анализ

В *проходе анализа* (analysis pass), схема которого показана на рис. 12.55, NTFS просматривает журнал транзакций в прямом направлении, начиная с последней операции контрольной точки, ища записи обновления и на их основе обновляя скопированные ранее в память таблицы транзакций и измененных страниц. Обратите внимание, что операция контрольной точки помещает в файл журнала три записи, среди которых могут оказаться записи обновления. Поэтому NTFS приходится начинать сканирование от начала операции контрольной точки.

Большинство записей обновления, находящихся после начала операции контрольной точки, представляют собой изменения таблицы транзакций или таблицы измененных страниц. К примеру, если запись обновления относится к подтверждению транзакции, то представляемая данной записью транзакция должна быть удалена из таблицы транзакций. Аналогично, если запись обновления относится к обновлению

страницы, меняющему структуру данных файловой системы, нужно внести соответствующую поправку в таблицу измененных страниц.



Рис. 12.55. Проход анализа

После приведения таблиц в памяти в актуальное состояние NTFS просматривает их, определяя LSN самой старой записи обновления, которая касается невыполненной дисковой операции. Таблица транзакций содержит LSN-номера неподтвержденных (незавершенных) транзакций, а таблица измененных страниц — LSN-номера записей в кэше, которые не были сброшены на диск. LSN самой старой записи, обнаруженной NTFS в этих двух таблицах, определяет, откуда начнется проход повторения. Но если более старой окажется запись контрольной точки, проход начнется именно с нее.

ПРИМЕЧАНИЕ

В TxF-модели восстановления отдельный проход анализа отсутствует. Вместо этого, как описано в соответствующем разделе, требуемые операции включаются в проход повторения.

Повторение

Во время *прохода повторения* (redo pass), схема которого показана на рис. 12.56, NTFS сканирует файл журнала в прямом направлении, начиная от LSN самой старой записи обновления, обнаруженной при проходе анализа. NTFS ищет записи обновления, содержащие модификации тома, запротоколированные до сбоя системы, но не сброшенные из кэша на диск. Эти обновления NTFS повторяет в кэше.



Рис. 12.56. Проход повторения

К моменту достижения конца файла журнала NTFS завершает обновление кэша сведениями об изменениях на томе, и принадлежащая диспетчеру кэша подсистема отложенной записи может приступить к переписыванию содержимого кэша на диск в фоновом режиме.

Отмена

Завершив проход повторения, NTFS начинает *проход отмены* (undo pass), откатывая транзакции, не подтвержденные к моменту сбоя системы. На рис. 12.57 представлены две транзакции в файле журнала; транзакция 1 была подтверждена до сбоя системы, а транзакция 2 — нет. Поэтому NTFS нужно отменить транзакцию 2.

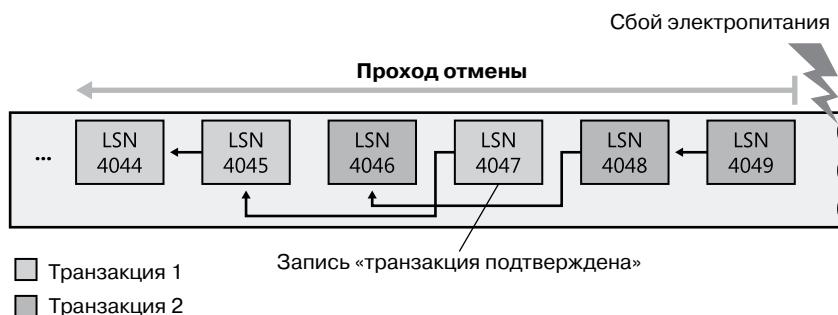


Рис. 12.57. Проход отмены

Допустим, в ходе транзакции 2 создавался файл. Эта операция состоит из трех подопераций, каждой из которых соответствует своя запись обновления. Относящиеся к одной транзакции записи обновления связываются в журнале обратными указателями, так как обычно они располагаются несмежно.

В таблице транзакций для каждой неподтвержденной транзакции NTFS хранит LSN-номер последней помещенной в журнал записи обновления. В рассматриваемом примере таблица транзакций сообщает, что для транзакции 2 это запись с LSN-номером 4049. Как показано на рис. 12.58, NTFS справа налево выполняет откат транзакции 2.



Рис. 12.58. Отмена транзакции

Локализовав LSN-номер 4049, NTFS извлекает информацию для отмены и выполняет отмену, сбрасывая в своей битовой карте биты 3–9. После этого NTFS следует по обратному указателю до LSN-номера 4048, согласно которому требуется удалить новое имя файла из соответствующего индекса имен файлов. Наконец, NTFS переходит по последнему указателю и в соответствии с указаниями из LSN-номера 4046 освобождает зарезервированную для данного файла MFT-запись. На этом откат транзакции 2 заканчивается. При наличии других неподтвержденных транзакций NTFS повторяет процедуру отката. Так как откат транзакций влияет на структуру файловой системы на томе, NTFS нужно протоколировать операции отмены в файле журнала. Ведь в процессе восстановления снова может произойти сбой электропитания, и NTFS придется повторить операции отмены!

После завершения прохода отмены том возвращается в согласованное состояние. В этот момент NTFS может сбросить на диск изменения кэша, чтобы гарантировать актуальность содержимого тома. Но перед этим NTFS выполняет обратный вызов, который TxF регистрирует для уведомлений о LFS-сбросах. Так как в TxF и NTFS применяется упреждающее протоколирование, чтобы гарантировать целостность своих метаданных, TxF нужно сбросить через CLFS свой журнал на диск до сброса NTFS-журнала. (Аналогично TOPS-файл требуется сбросить на диск до файлов журналов, управляемых CLFS.) После этого NTFS записывает «пустую» LFS-область перезапуска, показывая, что том находится в согласованном состоянии, и если немедленно после этого снова произойдет системный сбой, никакого восстановления не потребуется. На этом восстановление заканчивается.

NTFS гарантирует, что восстановление вернет том в некое существовавшее ранее целостное состояние, не обязательно непосредственно предшествующее сбою. Последнего NTFS гарантировать не может, так как с целью повышения производительности NTFS работает в соответствии с алгоритмом отложенного подтверждения. То есть журнал не сбрасывается на диск при каждом появлении записи «транзакция подтверждена», а несколько таких записей объединяются в пакет и записываются одновременно, когда диспетчер кэша вызывает LFS для сброса журнала на диск или когда LFS помещает в журнал новую запись контрольной точки (каждые 5 секунд). Есть и другая причина, по которой том не всегда возвращается к максимально актуальному состоянию. Дело в том, что в момент сбоя могли проводиться несколько параллельных транзакций, в этом случае одни записи о подтверждении транзакций могли быть перенесены на диск, а для других система могла не успеть этого сделать. Получаемое в результате восстановления согласованное состояние тома отражает только те транзакции, записи о подтверждении которых успели попасть на диск.

Файл журнала в NTFS применяется не только для восстановления тома, но и для других целей, достижение которых стало возможным благодаря протоколированию транзакций. Файловые системы обязательно содержат большой объем кода для обработки ошибок, возникающих при обычном файловом вводе-выводе. Так как NTFS протоколирует каждую влияющую на структуру тома транзакцию, можно пользоваться журналом транзакций для восстановления после ошибок файловой системы. Это существенно упрощает код обработки ошибок. Описанная ранее ошибка переполнения журнала транзакций является одним из примеров применения протоколирования транзакций для обработки ошибок.

Большинство возникающих в программах ошибок ввода-вывода не имеет отношения к ошибкам файловой системы, и NTFS не в состоянии исправить их своими силами. Например, при получении запроса на создание файла NTFS может начать с создания записи в MFT, после чего внести имя нового файла в индекс папки. Но при попытке выделить по своей битовой карте место для нового файла может оказаться, что диск уже заполнен и завершить операцию нельзя. В этом случае NTFS прибегает к информации из журнала транзакций для отмены уже выполненной части операции и освобождения зарезервированных для файла структур данных. Затем ошибка переполнения диска возвращается вызывающей программе, которая и должна предпринять соответствующие действия.

Восстановление поврежденных кластеров в NTFS

Диспетчер томов в Windows (VolMgr) может восстанавливать данные из поврежденного сектора на отказоустойчивом томе, но он не в состоянии заменить плохие секторы новыми, если жесткий диск не обеспечивает их переназначение или резервных секторов больше не осталось. (Подробно диспетчер томов рассматривается в главе 9.) Когда файловая система считывает данные из сектора, диспетчер томов восстанавливает информацию и возвращает ее с предупреждением, что это копия данных.

Файловая система FAT никак не обрабатывает это предупреждение. Более того, ни FAT, ни диспетчер томов не отслеживают поврежденные секторы, поэтому чтобы избежать постоянного восстановления диспетчером томов данных файловой системы, пользователю нужно запустить приложение Chkdsk или Format. Оба приложения удаляют поврежденные секторы из числа используемых далеко не идеально. Первое слишком долго ищет и удаляет эти секторы, а второе уничтожает все данные в форматируемом разделе.

В NTFS эквивалент механизма замены секторов динамически замещает кластер, содержащий поврежденный сектор, и ведет учет таких секторов, чтобы не допустить их использование в дальнейшем. (Вспомните, что NTFS поддерживает переносимость, адресуясь к логическим кластерам, а не к физическим секторам.) Эта функциональность в NTFS активируется, когда диспетчер томов не в состоянии заменить поврежденный сектор. Когда диспетчер томов возвращает предупреждение о наличии поврежденного сектора или когда драйвер жесткого диска сообщает о связанной с поврежденным сектором ошибке, NTFS выделяет для замены новый кластер. Восстановленные диспетчером томов данные NTFS копирует в новый кластер, чтобы снова добиться избыточности данных.

На рис. 12.59 показана MFT-запись для пользовательского файла с поврежденным кластером в одном из отрезков данных в том виде, в котором она существовала до повреждения кластера. Получив сообщение об ошибке, NTFS переназначает кластер с поврежденным сектором в файл \$BadClus, содержащий такие секторы. Это предотвращает возможность выделения этого кластера другому файлу. Затем NTFS выделяет новый кластер и таким образом меняет отображение VCN на LCN, чтобы оно указывало на новый адрес. Именно это переназначение поврежденного кластера (мы рассматривали его чуть раньше) иллюстрирует рис. 12.59. Кластер номер 1357, содержащий поврежденный сектор, следует заменить новым, неповрежденным кластером.

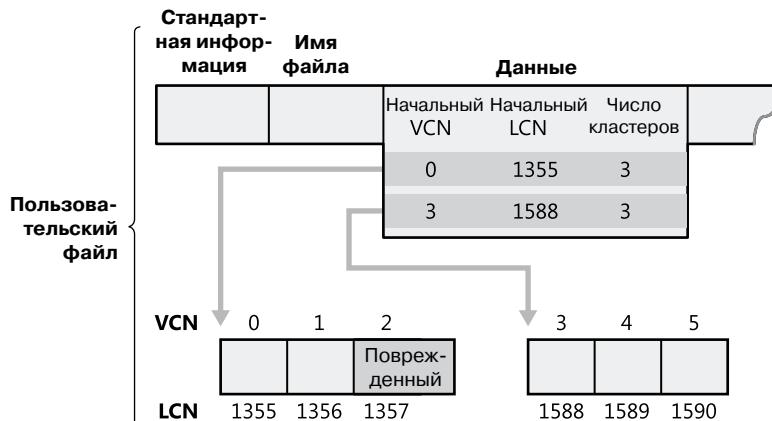


Рис. 12.59. MFT-запись для пользовательского файла с поврежденным кластером

Ошибки, связанные с поврежденными секторами, крайне нежелательны, но когда они все же появляются, лучшее решение предлагает NTFS в комбинации с диспетчером томов. Если поврежденный сектор находится на томе с избыточностью, диспетчер томов восстанавливает данные и по возможности заменяет сектор. Если замена невозможна, диспетчер томов возвращает NTFS предупреждение, и уже сама файловая система заменяет кластер с поврежденным сектором.

Если параметры тома не предполагают избыточности, восстановить данные из поврежденного сектора не удастся. В ситуации, когда том отформатирован для FAT и диспетчер томов не в состоянии восстановить данные, чтение из поврежденного сектора дает непредсказуемые результаты. Если в поврежденном секторе располагались какие-либо управляющие структуры файловой системы, может быть потеряна целая группа файлов (а потенциально и весь диск). В самом лучшем случае пропадает доступ к части данных из затронутого файла (как правило, это данные, расположенные в поврежденном секторе и за ним). Более того, скорее всего, FAT выделит поврежденный сектор под этот же или другой файл на томе, и проблема возникнет снова.

Подобно другим файловым системам, NTFS не в состоянии восстановить данные из поврежденного сектора без помощи диспетчера томов. Но NTFS значительно уменьшает наносимый таким сектором вред. Если поврежденный сектор обнаруживается в процессе чтения, NTFS выполняет переназначение кластера, как показано на рис. 12.60. Если том не сконфигурирован под избыточное хранение информации, NTFS возвращает вызывающей программе ошибку чтения. Данные из этого кластера теряются, но остаток файла и сама файловая система остаются в рабочем состоянии. В результате вызывающая программа может соответствующим образом отреагировать на потерю данных, а поврежденный кластер больше не будет использоваться при распределении пространства на томе. Если поврежденный кластер обнаруживается в процессе записи, NTFS переназначает его до выполнения записи, избегая тем самым потери данных.

Та же самая процедура восстановления применяется, когда данные файловой системы хранятся в аварийном секторе. Если он находится на томе с избыточностью, NTFS динамически заменяет соответствующий кластер, используя восстановленные диспетчером томов данные. При отсутствии на томе избыточности восстановить данные

невозможно, поэтому NTFS устанавливает в файле метаданных \$Volume бит, сигнализирующий о повреждении на томе. При перезагрузке системы служебная программа Chkdsk проверяет наличие этого бита и при его обнаружении исправляет повреждение файловой системы путем реконструкции NTFS-метаданных.

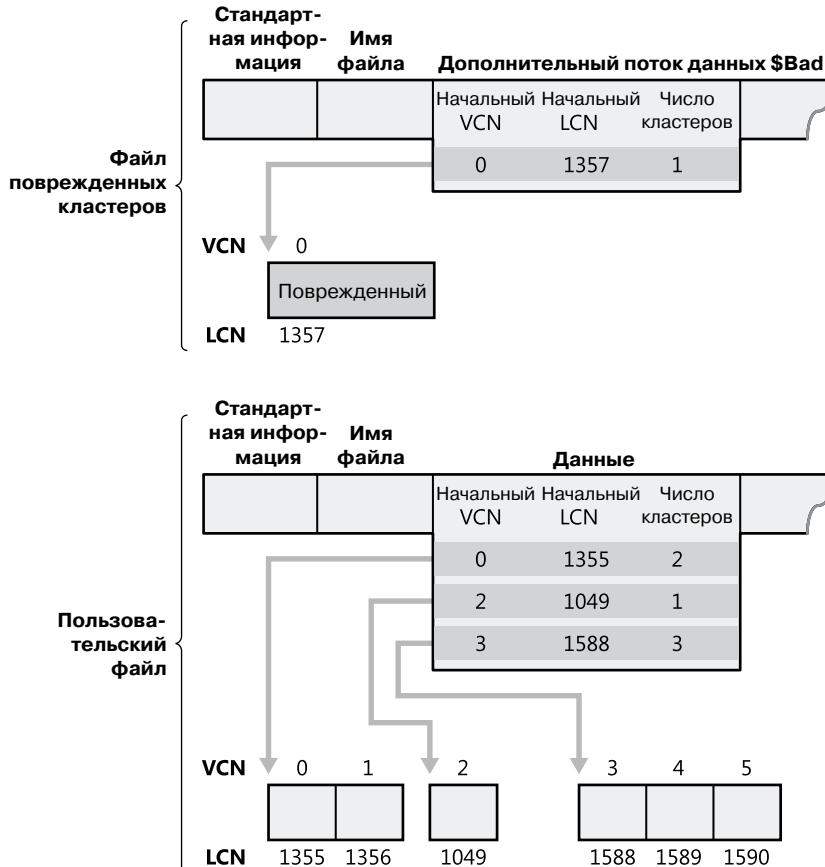


Рис. 12.60. Переназначение поврежденных кластеров

В редких случаях повреждение файловой системы может произойти даже при отказоустойчивой конфигурации диска. Двойная ошибка может разрушить как данные файловой системы, так и средства их восстановления. Если сбой системы происходит, к примеру, в момент, когда NTFS сохраняет зеркальную копию MFT-записи индекса имен файлов или журнала транзакций, эта копия обновляется не полностью. Если после перезагрузки системы связанный с поврежденным сектором ошибки возникает в том же месте основного диска, где находилась частично записанная зеркальная копия, NTFS не сможет восстановить с нее данные. Для распознавания таких повреждений данных файловой системы в NTFS реализована специальная схема. При обнаружении нарушения целостности в файле тома устанавливается бит повреждения, что

заставляет Chkdsk реконструировать NTFS-метаданные при следующей перезагрузке. Так как при отказоустойчивой конфигурации диска повреждений системных файлов практически не бывают, потребность в программе Chkdsk возникает редко. Поэтому данная программа существует в качестве вспомогательного, а не основного средства восстановления информации.

Применение программы Chkdsk в NTFS существенно отличается от ее применения в FAT. Перед записью любых данных на диск FAT устанавливает бит изменения тома, который сбрасывается после завершения модификации. Если сбой происходит в процессе ввода-вывода, бит остается, и при перезагрузке начинает работать Chkdsk. В NTFS программа Chkdsk запускается только при обнаружении неожиданных или нечитаемых данных файловой системы, которые NTFS не в состоянии восстановить с избыточного тома или избыточных структур данных файловой системы на обычном томе. Система дублирует загрузочный сектор в последний сектор тома, а также части MFT [$\$MftMirr$], необходимые для загрузки и выполнения процедуры восстановления. Эта избыточность гарантирует, что NTFS всегда сможет загрузиться и восстановить себя.

В табл. 12.10 собрана информация о том, что происходит при появлении поврежденного сектора на дисковом томе, отформатированном для одной из файловых систем Windows, в различных, рассмотренных в данном разделе ситуациях.

Таблица 12.10. Сценарии восстановления данных в NTFS

Сценарий	При наличии диска, поддерживающего переназначение поврежденных секторов, и наличии свободных секторов	При наличии диска, не поддерживающего переназначение поврежденных секторов, или при отсутствии свободных секторов
Отказоустойчивый том ¹	<ol style="list-style-type: none"> Диспетчер томов восстанавливает данные. Диспетчер томов заменяет поврежденный сектор. Файловой системе не сообщают об ошибке 	<ol style="list-style-type: none"> Диспетчер томов восстанавливает данные. Диспетчер томов отправляет файловой системе данные и ошибку «поврежденный сектор». NTFS выполняет переназначение кластера
Обычный том	<ol style="list-style-type: none"> Диспетчер томов не может восстановить данные. Диспетчер томов отправляет файловой системе ошибку «поврежденный сектор». NTFS выполняет переназначение кластера. Данные оказываются потерянными² 	<ol style="list-style-type: none"> Диспетчер томов не может восстановить данные. Диспетчер томов отправляет файловой системе ошибку «поврежденный сектор». NTFS выполняет переназначение кластера. Данные оказываются потерянными

¹ Отказоустойчивыми считаются тома, принадлежащие к типу RAID-1 (зеркальный) или RAID-5.

² При записи данные не теряются: NTFS переназначает кластер до выполнения записи.

Если том, на котором появился поврежденный сектор, сконфигурирован как отказоустойчивый — типа RAID-1 (зеркальный) или RAID-5, — а жесткий диск поддерживает замену секторов (кроме того, запас свободных секторов еще не исчерпан), не имеет значения, какой файловой системой вы пользуетесь, FAT или NTFS. Диспетчер томов заменит поврежденный сектор, не требуя вмешательства со стороны пользователя или файловой системы.

Если поврежденный сектор появился на диске, не поддерживающем замену поврежденных секторов, за переназначение такого сектора или (в случае NTFS) кластера, на котором располагается такой сектор, отвечает файловая система. В файловой системе FAT переназначение секторов или кластеров не предусмотрено. Преимущество NTFS в данном случае — это возможность исправлять поврежденные фрагменты файлов, не причиняя вреда самому файлу (или файловой системе), и выведение поврежденных кластеров из дальнейшего обращения.

Самовосстановление

Объем современных устройств хранения достигает нескольких терабайтов, поэтому отключение тома для проверки целостности может стать причиной многочасового простоя. Поскольку повреждения часто располагаются в одном файле или фрагменте метаданных, в NTFS реализована функция самовосстановления, обеспечивающая исправление повреждений без отключения тома. Обнаружив повреждение, NTFS запрещает доступ к поврежденному файлу или файлам и создает системный рабочий программный поток, вносящий исправления примерно таким же образом, как утилита Chkdsk. После завершения его работы доступ к восстановленным файлам возобновляется. Доступ к другим файлам во время этой операции сохраняется, сводя к минимуму время простоя.

Команда `fsutil repair set` позволяет просмотреть и выбрать параметры восстановления тома, перечисленные в табл. 12.11. Утилита Fsutil пользуется управляющим кодом `FSCTL_SET_REPAIR` для выбора этих параметров, хранящихся в VCB тома.

Таблица 12.11. Варианты поведения NTFS при самовосстановлении

Флаг	Поведение
<code>SET_REPAIR_ENABLED</code>	Включает самовосстановление тома
<code>SET_REPAIR_WARN_ABOUT_DATA_LOSS</code>	Если процесс самовосстановления не в состоянии полностью восстановить файл, указывает, следует ли отправить пользователю предупреждение
<code>SET_REPAIR_DISABLED_AND_BUGCHECK_ON_CORRUPTION</code>	Если значение <code>NtfsBugCheckOnCorrupt</code> из ветки реестра, управляющей поведением NTFS, равно 1 и установлен этот флаг, система прекратит работу с ошибкой STOP <code>0x24</code> , указывающей на повреждение файловой системы. Этот параметр автоматически сбрасывается при загрузке, чтобы избежать циклической перезагрузки

Во всех случаях, в том числе при отключении визуальных предупреждений (режим, предлагаемый по умолчанию), NTFS записывает все предпринятые операции в системный журнал событий.

Наряду с периодическим автоматическим самовосстановлением NTFS поддерживает циклы самовосстановления, запускаемые вручную через управляющие коды `FSCTL_INITIATE_REPAIR` и `FSCTL_WAIT_FOR_REPAIR`, которые включаются командами `fsutil repair initiate` и `fsutil repair wait`. Это позволяет пользователю инициировать восстановление определенного файла и подождать завершения процесса.

Для проверки состояния механизма самовосстановления применяется управляющий код `FSCTL_QUERY_REPAIR` или команда `fsutil repair query`:

```
C:\>fsutil repair query c:  
Self healing is enabled for volume c: with flags 0x1.  
flags: 0x01 - enable general repair  
      0x08 - warn about potential data loss  
      0x10 - disable general repair and bugcheck once on first corruption
```

Безопасность в шифрующей файловой системе

Как отмечено в главе 9, инструмент BitLocker шифрует и защищает тома от атак в автономном режиме, но после загрузки системы его работа завершается. Отдельные файлы и папки от других авторизованных пользователей защищает шифрующая файловая система (Encrypting File System, EFS). Выбирая инструмент защиты данных, между BitLocker и EFS нельзя ставить союз «или». Каждый в данной паре обеспечивает защиту от определенных — и не пересекающихся с другими — угроз. Только совместно BitLocker и EFS способны эффективно защитить данные в вашей системе.

EFS для управления доступом к файлам и папкам применяет симметричное шифрование (для шифрования и дешифрирования файлов и папок используется один ключ). Затем к ключу применяется асимметричное шифрование (открытый ключ для шифрования и закрытый ключ для дешифрирования). Детали и теоретический механизм этого процесса выходят за рамки темы данной книги; но хорошее пособие можно найти по адресу [http://msdn.microsoft.com/en-us/library/windows/desktop/aa380251\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/aa380251(v=vs.85).aspx).

Для работы с EFS применяется API-интерфейс CNG (Cryptography Next Generation — криптография нового поколения), соответственно, существует возможность воспользоваться любым поддерживаемым CNG или добавленным к ней алгоритмом. По умолчанию в EFS применяется улучшенный стандарт шифрования (Advanced Encryption Standard, AES) для симметричного шифрования (с 256-разрядным ключом) и алгоритм RSA (аббревиатура от фамилий Rivest, Shamir и Adleman) с открытым ключом для асимметричного шифрования (с 2048-разрядными ключами).

Пользователи могут шифровать файлы в Проводнике. Для этого нужно открыть для файла диалоговое окно **Properties** (Свойства), щелкнуть на кнопке **Advanced** (Другие) и установить флажок **Encrypt contents to secure data** (Шифровать содержимое для защиты данных), как показано на рис. 12.61. Файл можно зашифровать или сжать, но нельзя применить к нему обе эти процедуры одновременно. Также шифрование файлов можно осуществлять с помощью утилиты командной строки `Cipher` (`%SystemRoot%\System32\Cipher.exe`) и через такие API-интерфейсы Windows, как `EncryptFile` и `AddUsersToEncryptedFile`.

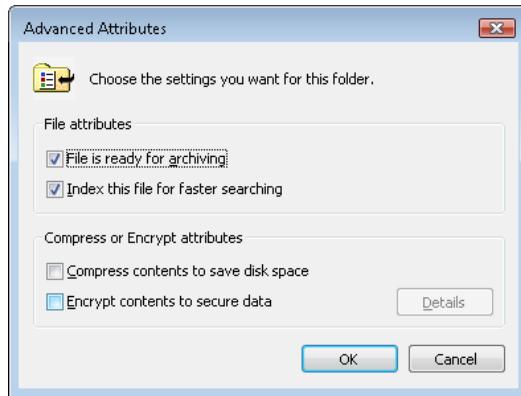


Рис. 12.61. Шифрование файла через диалоговое окно дополнительных атрибутов

Windows автоматически шифрует файлы в папках, помеченных как зашифрованные. В процессе шифрования EFS генерирует для файла случайное число, называемое *ключом шифрования файла* (File Encryption Key, FEK). В этом случае EFS применяет симметричное шифрование. Затем EFS шифрует уже сам ключ шифрования файла, используя асимметричный открытый ключ, и сохраняет результат для файла в дополнительном потоке данных \$EFS. В качестве источника открытого ключа может быть административно указан сертификат формата X.509, смарт-карта или генератор случайных чисел, который затем будет добавлен в хранилище сертификатов пользователя, доступное для просмотра через диспетчер сертификатов (%SystemRoot%\System32\Certmgr.msc). После того как EFS выполнит все эти действия, файл считается защищенным: другие пользователи не смогут получить доступ к его данным без расшифрованного ключа шифрования файла, а FEK невозможно расшифровать без открытого ключа, который имеется только у владельца файла.

Симметричные алгоритмы шифрования, как правило, работают очень быстро, что делает их крайне удобными для шифрования больших объемов данных, например большого количества файлов. Но у них есть слабое место: защиту можно обойти, получив ключ. Если несколько пользователей работают с одним файлом, защищенным только симметричным шифрованием, каждому из них требуется доступ к FEK. Очевидно, что незашифрованный ключ шифрования файла является угрозой безопасности. Но даже его шифрование не решает проблему, так как несколько человек все равно будут пользоваться одним и тем же ключом для дешифрования FEK.

Защита с помощью FEK является сложной проблемой, для решения которой в EFS используется та часть криптографической архитектуры, которая опирается на технологии шифрования с открытым ключом. Шифрование FEK на индивидуальной основе позволяет нескольким лицам совместно использовать зашифрованный файл. Зашифровать FEK для файла EFS может с помощью открытого ключа, который будет своим для каждого пользователя. Результат при этом сохраняется в потоке данных \$EFS файла. Доступ к открытому ключу пользователя есть у всех, но никто не может воспользоваться им для дешифрования закрытых им данных. Это могут делать только пользователи, обладающие закрытым ключом, с которым работает операционная

система. Закрытый ключ раскрывает нужный зашифрованный экземпляр FEK для файла. Алгоритмы на основе открытого ключа обычно работают медленно, поэтому в EFS они применяются только для шифрования FEK. Разделение ключей на открытый и закрытый слегка упрощает управление ключами по сравнению с алгоритмами симметричного шифрования и решает дилемму, связанную с защитой FEK.

Работу EFS обеспечивает взаимодействие нескольких компонентов, как демонстрирует диаграмма на рис. 12.62. Поддержка EFS встроена в NTFS-драйвер. При обнаружении зашифрованного файла NTFS выполняет встроенные EFS-функции. Они осуществляют шифрование и дешифрирование файловых данных по мере обращения приложений к зашифрованным файлам. Хотя EFS хранит FEK вместе с данными файла, FEK шифруется с помощью открытого ключа отдельного пользователя. Для шифрования и дешифрирования файловых данных EFS нужно расшифровать FEK файла с помощью CNG-служб управления ключами в режиме пользователя.

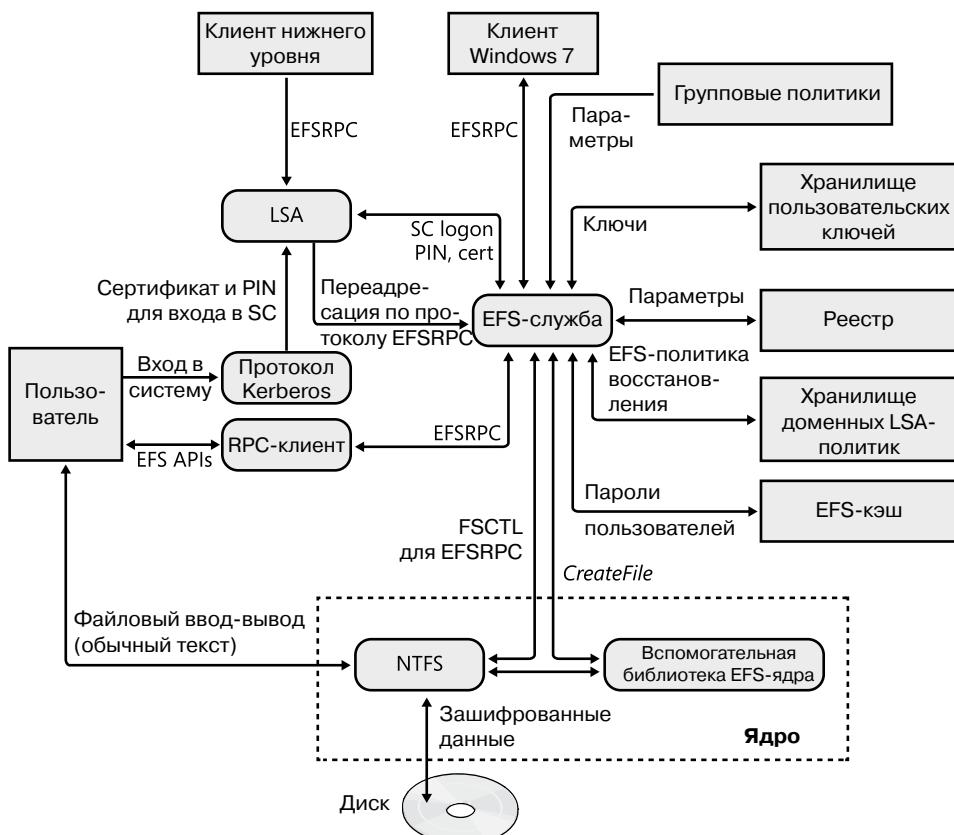


Рис. 12.62. Архитектура EFS

Подсистема локальной аутентификации (Local Security Authority Subsystem, LSASS) управляет не только сеансами входа, но и EFS-службой (%SystemRoot%\System32\lsass.exe). К примеру, когда EFS нужно расшифровать FEK для дешифрирования

данных файла, к которому обращается пользователь, NTFS отправляет запрос EFS-службе через LSASS.

Первое шифрование файла

Обнаружив зашифрованный файл, NTFS-драйвер вызывает вспомогательные EFS-функции. О состоянии шифрования файла, как и о состоянии его сжатия, сообщают его атрибуты. Для преобразования файла из незашифрованной в зашифрованную форму в NTFS существуют специальные интерфейсы, но в основном процессом управляют компоненты пользовательского режима. Как уже отмечалось, Windows позволяет шифровать файлы двумя способами: используя утилиту командной строки `cipher` или устанавливая флагок `Encrypt contents to secure data` (Шифровать содержимое для защиты данных) в окне `Advanced Attributes` (Дополнительные атрибуты), вызываемом для файла через Проводник. В обоих случаях применяется API-интерфейс `EncryptFile`, предоставляемый библиотекой `Advapi32.dll`.

В зашифрованном файле EFS сохраняет только один блок информации, который содержит записи обо всех имеющих доступ к файлу пользователях. Эти записи называются *элементами ключей* (key entries); EFS хранит их в области EFS-данных файла, которая называется полем дешифрирования данных (Data Decryption Field, DDF). Совокупность нескольких элементов ключей называется *связкой ключей* (key ring), потому что, как уже упоминалось, EFS позволяет работать с шифрованным файлом сразу нескольким пользователям.

Форматы EFS-данных файла и элементов ключа показаны на рис. 12.63. В первой части элемента ключа EFS хранит информацию, которой достаточно для точного описания открытого ключа пользователя. В нее входит идентификатор безопасности (его наличие не гарантируется) пользователя, имя контейнера, в котором хранится ключ, имя провайдера криптографических служб и хэш сертификата криптографической асимметричной пары. При дешифрировании используется только этот хэш. Вторая часть элемента ключа содержит шифрованную версию FEK. Для шифрования FEK выбранным асимметричным алгоритмом и открытым ключом пользователя EFS использует CNG.



Рис. 12.63. Форматы EFS-информации и элементов ключа

Информацию об элементах ключей восстановления EFS хранит в поле восстановления данных (Data Recovery Field, DRF). Форматы DRF- и DDF-элементов идентичны. Назначением DRF является конструирование особых учетных записей, или агентов восстановления, в случаях, когда администратору требуется доступ к пользовательским данным, например, если сотрудник компании забыл пароль для входа в систему. Администратор может сбросить его пароль, но без агентов восстановления невозможно восстановить зашифрованные данные.

Агенты восстановления задаются при помощи политики безопасности локального компьютера или домена под названием Encrypted Data Recovery Agents. Эти политики доступны через оснастку Local Security Policy (Групповая политика безопасности) консоли MMC, показанную на рис. 12.64. Щелкните правой кнопкой мыши на строке Encrypting File System и выберите команду Add Data Recovery Agent для запуска мастера добавления агента восстановления. С его помощью можно добавлять агенты восстановления и указывать, какими парами открытый/закрытый ключ (обозначенные их сертификатами) агенты могут пользоваться для восстановления EFS. Программа Lsassrv интерпретирует политику восстановления в процессе своей инициализации и при получении уведомления об изменении этой политики. Для каждого агента восстановления EFS создает DRF-элементы ключей через зарегистрированный для EFS-восстановления провайдер криптографических служб.

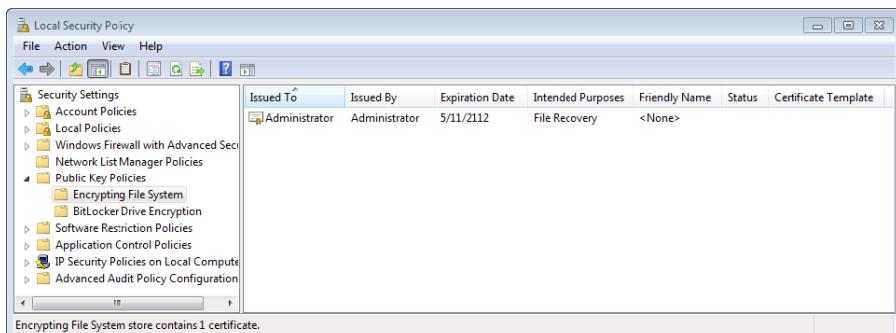


Рис. 12.64. Групповая политика Encrypted Data Recovery Agents

На завершающем этапе создания для файла EFS-информации Lsassrv вычисляет контрольную сумму для DDF и DRF, используя механизм хэширования MD5 из Base Cryptographic Provider 1.0. Эту сумму Lsassrv хранит в заголовке EFS-данных. На нее EFS ссылается при дешифровании, чтобы убедиться в отсутствии повреждения или взлома у EFS-данных файла.

Шифрование файловых данных

Когда пользователь шифрует файл, происходят следующие процессы:

1. EFS-служба открывает файл для монопольного доступа.
2. Все потоки данных в файле копируются во временный текстовый файл, расположенный во временной системной папке.

3. Случайным образом генерируется FEK и применяется для шифрования файла по алгоритму DESX или 3DES, в зависимости от эффективной политики безопасности.
4. Создается поле DDF, в которое входит зашифрованный с помощью открытого ключа пользователя ключ шифрования файла (FEK). EFS автоматически получает открытый ключ пользователя из пользовательского сертификата шифрования файла X.509 версии 3.
5. Если агент восстановления назначен через групповые политики, создается поле DRF, в которое помещаются зашифрованный по алгоритму RSA ключ шифрования файла и открытый ключ агента восстановления.

Открытый ключ агента восстановления для восстановления файла EFS автоматически получает от сертификата X.509 версии 3, который хранится в EFS-политиках восстановления. При наличии нескольких агентов восстановления копия FEK шифруется с помощью открытого ключа каждого агента, и для хранения полученных результатов создается DRF.

ПРИМЕЧАНИЕ

Свойство file recovery в сертификате является примером поля EKU (enhanced key usage — расширенное использование ключа). Расширение EKU и расширенное свойство определяют и ограничивают сферу применения сертификатов. File Recovery — одно из полей EKU, определенное Microsoft как часть инфраструктуры открытых ключей (Public Key Infrastructure, PKI).

6. Зашифрованные данные (вместе с DDF и DRF) EFS записывает обратно в файл. Так как при симметричном шифровании дополнительные данные не добавляются, увеличение размера файла после шифрования минимально. Метаданные, состоящие в основном из зашифрованных ключей шифрования файла, обычно занимают менее 1 Кбайт. Как правило, выводимый на экран размер файла до и после шифрования не меняется.
7. Удаляется временный текстовый файл.

При сохранении файла в зашифрованной папке происходит практически аналогичный процесс, просто в этом случае временный файл не создается.

Процесс дешифрирования

При обращении приложения к зашифрованному файлу происходит следующее:

1. Распознав файл как зашифрованный, NTFS посыпает запрос EFS-драйверу.
2. EFS-драйвер получает поле DDF и передает его EFS-службе.
3. EFS-служба получает из профиля пользователя закрытый ключ пользователя, с его помощью расшифровывает DDF и получает FEK.
4. EFS-служба возвращает EFS-драйверу FEK.
5. EFS-драйвер использует FEK для дешифрирования нужных приложению фрагментов файла.

ПРИМЕЧАНИЕ

Когда приложение открывает файл, расшифровывается только тот фрагмент, к которому происходит обращение в конкретный момент, так как EFS выполняет передачу зашифрованного текста поблочно. Если пользователь удаляет из файла атрибут encryption, возникает другая ситуация. В этом случае весь файл расшифровывается и переписывается как обычный текст.

- EFS-драйвер возвращает NTFS расшифрованные данные, которые затем передаются затребовавшему их приложению.

Резервное копирование шифрованных файлов

Важный аспект разработки любого механизма шифрования файлов заключается в том, что приложения не могут получить доступ к расшифрованным файлам, минуя механизм шифрования. Это ограничение особенно затрагивает программы резервного копирования, с помощью которых файлы сохраняются на архивных носителях. Эту проблему EFS решает, предоставляя программам резервного копирования механизм, с помощью которого они могут создавать резервные копии файлов и восстанавливать их в зашифрованном виде. Это избавляет от необходимости постоянно прибегать к процедурам дешифрирования и шифрования при восстановлении файла.

Для доступа к зашифрованному содержимому программы резервного копирования используют в EFS API-функции `OpenEncryptedFileRaw`, `ReadEncryptedFileRaw`, `WriteEncryptedFileRaw` и `CloseEncryptedFileRaw`. После того как программа резервного копирования в процессе создания копии открывает файл для прямого доступа, для получения файловых данных она вызывает функцию `ReadEncryptedFileRaw`.

ЭКСПЕРИМЕНТ: ПРОСМОТР EFS-ИНФОРМАЦИИ

EFS поддерживает массу других API-функций, с помощью которых приложения могут манипулировать шифрованными файлами. Например, функция `AddUsersToEncryptedFile` дает доступ к зашифрованному файлу дополнительным пользователям, а функция `RemoveUsersFromEncryptedFile`, наоборот, удаляет пользователей из списка доступа. С помощью функции `QueryUsersOnEncryptedFile` приложения получают информацию о связанных с файлом полях DDF и DRF ключа. Еще она возвращает SID, хэш сертификата и содержимое каждого из полей DDF и DRF ключа. Далее приведен результат работы программы `EFSdump` производства Sysinternals. В качестве параметра командной строки указано имя шифрованного файла:

```
C:\>efsdump test.txt
EFS Information Dumper v1.02
Copyright (C) 1999 Mark Russinovich
Systems Internals - http://www.sysinternals.com

test.txt:
DDF Entry:
DARYL\Mark:
CN=Mark,L=EFS,OU=EFS File Encryption Certificate
DRF Entry:
Unknown user:
EFS Data Recovery
```

Как видите, в файле test.txt присутствует один DDF-элемент для пользователя Mark и один DRF-элемент для агента EFS Data Recovery, который в настоящий момент является единственным зарегистрированным в системе агентом восстановления.

Копирование зашифрованных файлов

При копировании зашифрованного файла система не расшифровывает его и не подвергает повторному шифрованию в месте назначения; она просто переносит зашифрованные данные и альтернативный поток EFS-данных в указанное место. Но в месте назначения альтернативные потоки данных могут не поддерживаться, например, если файл копируется не на NTFS-том, а на FAT-том или на совместно используемый сетевой ресурс (даже если в роли этого ресурса выступает NTFS-том). В этом случае из-за потери альтернативных потоков данных копия не сможет нормально функционировать. Если копирование осуществляется при помощи Проводника, появляется диалоговое окно, информирующее пользователя о том, что выбранный им том не поддерживает шифрование, и спрашивающее, следует ли скопировать незашифрованные данные. Если пользователь соглашается, файл расшифровывается и копируется в указанное место. При копировании из командной строки команда `copy` не сработает — появится сообщение об ошибке «The specified file could not be encrypted» (указанный файл не может быть зашифрован).

Заключение

В Windows поддерживается широкий спектр форматов файловых систем, доступных как локальной системе, так и удаленным клиентам. Архитектура фильтрующего драйвера файловой системы позволяет корректно расширять и дополнять средства доступа к файловой системе, а NTFS представляет собой надежный, безопасный и масштабируемый формат. В следующей главе мы поговорим о запуске и завершении работы Windows.

Глава 13. Запуск и завершение работы системы

В этой главе рассказывается о загрузке Windows и параметрах, влияющих на запуск системы. Понимание тонкостей процесса загрузки помогает диагностировать возникающие при этом проблемы. Затем мы поговорим о том, какие ошибки могут возникнуть в процессе загрузки и как их устранить. В заключение вы узнаете, что происходит при корректном завершении работы операционной системы.

Процесс загрузки

Описание процесса загрузки Windows мы начнем с установки системы, а затем поговорим о выполнении загрузочных файлов. Важной частью процесса загрузки являются драйверы устройств, поэтому мы выясним, как они контролирует собственную загрузку и инициализацию. Далее мы опишем инициализацию компонентов исполнительной подсистемы и остановимся на том, как ядро запускает Windows в режиме пользователя через процесс Session Manager (`Smss.exe`), который начинает первые два сеанса (сесанс 0 и сесанс 1). Попутно вы узнаете, что происходит внутри системы, когда на экране в процессе загрузки появляются те или иные текстовые сообщения.

Начальные этапы процесса загрузки в операционных системах, оснащенных BIOS (Basic Input Output System — базовая система ввода-вывода) и EFI (Extensible Firmware Interface — расширяемый микропрограммный интерфейс), различаются кардинальным образом. Новый стандарт EFI в значительной степени отошел от 16-разрядного кода, который повсеместно применяется в системах на базе BIOS; он предназначен для запуска программ и драйверов начального этапа загрузки. В следующем разделе описываются детали загрузки, присущие системам на базе BIOS, а затем мы перейдем к рассмотрению этой процедуры в системах на базе EFI.

Для поддержки этих разных программных реализаций (а также стандарта EFI 2.0, известного как Unified EFI, или UEFI) в Windows существует архитектура загрузки, скрывающая от пользователей и разработчиков многочисленные различия, предлагая одинаковые среду и способ применения вне зависимости от типа программного обеспечения, применяемого в установленной системе.

Начальные этапы загрузки систем на базе BIOS

Загрузка Windows начинается не после включения электропитания или нажатия кнопки на системном блоке, а еще при установке Windows на компьютер. В какой-то момент выполнения программы Windows Setup происходит подготовка жесткого диска системы. На нем размещается код, принимающий участие в процессе загрузки.

Но перед тем как мы перейдем к рассмотрению этого кода, поговорим о том, как и в какой области жесткого диска он размещается. Стандарт разбиения жестких дисков на тома существует в системах типа x86 со времен первых версий MS-DOS.

Операционные системы производства Microsoft разбивают жесткие диски на дискретные области, называемые *разделами* (partitions). После форматирования с применением файловых систем (типа FAT и NTFS) каждые раздел образует том. Жесткий диск может содержать до четырех главных разделов. Так как данная схема разбиения ограничивает диск четырьмя томами, существует специальный тип раздела — *расширенный* (extended partition). В каждом расширенном разделе можно выделить еще до четырех дополнительных разделов. Расширенные разделы могут содержать другие расширенные разделы, которые, в свою очередь, могут иметь расширенные разделы, и т. д. Поэтому операционная система может разбить диск на практически бесконечное число томов. Пример разбиения жесткого диска показан на рис. 13.1, а в табл. 13.1 перечислены компоненты, участвующие в загрузке BIOS. (Подробно разбиение Windows на разделы рассматривается в главе 9.)

Таблица 13.1. Компоненты загрузочного процесса в системах на базе BIOS

Компонент	Режим работы процессора	Обязанности	Местоположение
Главная загрузочная запись (MBR)	16-разрядный реальный режим	Читает и загружает загрузочную запись тома (VBR)	На запоминающем устройстве
Загрузочный сектор (также называемый загрузочной записью тома)	16-разрядный реальный режим	Распознает на разделе файловую систему, по имени находит компонент Bootmgr и загружает его в память	На активном (загружаемом) разделе
Bootmgr	16-разрядный реальный режим и 32-разрядный режим без подкачки	Считывает хранилище конфигурационных данных загрузки (BCD), выводит меню загрузки и обеспечивает выполнение программ предварительной загрузки, таких как программа тестирования памяти (Memtest.exe). При загрузке 64-разрядной копии Windows перед загрузкой Winload переключается в 64-разрядный режим	В системе
Winload.exe	32-разрядный защищенный режим с подкачкой, 64-разрядный защищенный режим при загрузке Win64	Загружает программу Ntoskrnl.exe и зависящие от нее файлы (Bootvid.dll в 32-разрядных системах, Hal.dll, Kdcom.dll, Ci.dll, Clfs.sys, Psched.dll) и драйверы загрузочных устройств	В установленной копии Windows

продолжение ↗

Таблица 13.1 (продолжение)

Компонент	Режим работы процессора	Обязанности	Местоположение
Winresume.exe	32-разрядный защищенный режим, 64-разрядный защищенный режим при восстановлении копии Win64	Для возобновления работы после режима гибернации вместо обычной загрузки Windows используется файл Hiberfil.sys	В установленной копии Windows
Memtest.exe	32-разрядный защищенный режим	При выборе этого варианта в диспетчере загрузки запускается графический интерфейс для сканирования памяти и распознавания поврежденной оперативной памяти (RAM)	В системе
Ntoskrnl.exe	Защищенный режим с подкачкой	Инициализирует исполнительную подсистему, а также драйверы устройств загрузки и запуска. Готовит систему к выполнению встроенных приложений и запускает Smss.exe	В установленной копии Windows
Hal.dll	Защищенный режим с подкачкой	DLL-библиотека режима ядра, предоставляющая аппаратному обеспечению интерфейс для Ntoskrnl и драйверов. Также функционирует как драйвер для самой материнской платы, поддерживая встроенные компоненты, которые не управляются другими драйверами	В установленной копии Windows
Smss.exe	Встроенное приложение	Исходный экземпляр создает свою копию для инициализации каждого сеанса. Экземпляр сеанса 0 загружает драйвер подсистемы Windows (Win32k.sys) и запускает процесс Csrss.exe, а также процесс инициализации Windows (Wininit.exe). Экземпляры всех остальных сеансов запускают процессы Csrss и Winlogon	В установленной копии Windows
Wininit.exe	Windows-приложение	Запускает диспетчер управления службами (SCM), процесс подсистемы локальной аутентификации (LSASS) и диспетчер локальных сеансов (LSM). Инициализирует остальную часть реестра и выполняет задания инициализации в режиме пользователя	В установленной копии Windows

Компонент	Режим работы процессора	Обязанности	Местоположение
Winlogon.exe	Windows-приложение	Координирует вход в систему и безопасность пользователя, загружает LogonUI	В установленной копии Windows
Logonui.exe	Windows-приложение	Выводит интерактивное диалоговое окно для входа в систему	В установленной копии Windows
Services.exe	Windows-приложение	Загружает и инициализирует драйверы автоматического запуска устройств и Windows-службы	В установленной копии Windows

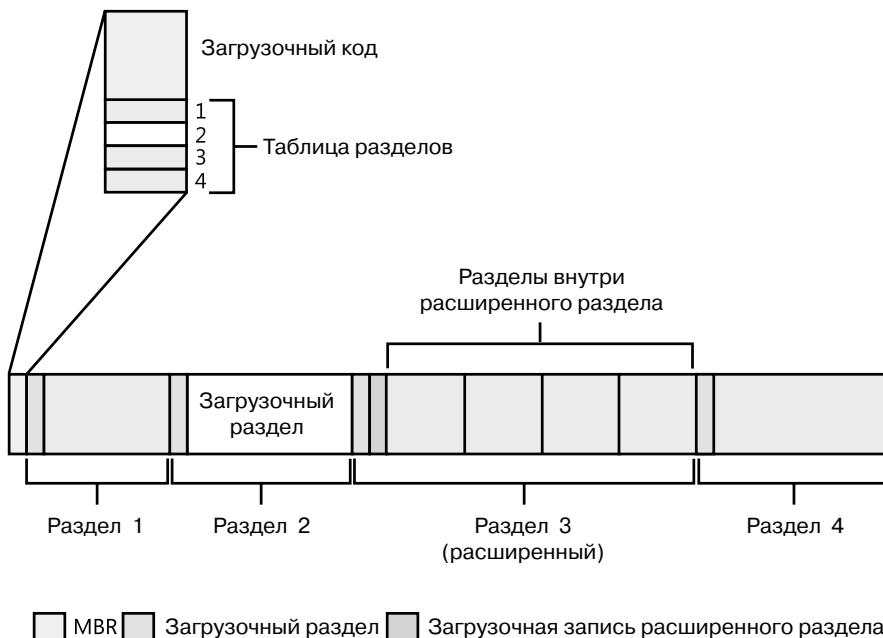


Рис. 13.1. Пример структуры разделов жесткого диска

Единицей адресации физических дисков является *сектор* (sector). Размер сектора для систем на базе BIOS составляет, как правило, 512 байт (но, как отмечено в главе 9, постепенно происходит переход к секторам размером 4096 байт). Служебные программы, подготавливающие жесткий диск к разбиению на тома, например программа Windows Setup, записывают главную загрузочную запись (MBR) в первый сектор. (Данный тип разбиения описывается в главе 9.) В MBR включено пространство, содержащее исполняемые команды, которые называются *загрузочным кодом* (boot code), и таблицу, называемую *таблицей разделов* (partition table), с четырьмя записями, опре-

деляющими местоположение основных разделов на диске. При загрузке компьютера на базе BIOS в первую очередь загружается находящаяся во флэш-памяти система BIOS. Эта система выбирает загрузочное устройство, считывает его MBR и передает управление расположенному в MBR коду.

Через аналогичный процесс чтения и передачи управления проходят главные загрузочные записи, созданные при помощи таких инструментов разбиения на разделы, как, к примеру, программа Windows Setup и оснастка Disk Management консоли MMC. Сначала MBR-код сканирует основную таблицу разделов, пока не обнаружит раздел с флагом (Active), то есть загрузочный. При обнаружении хотя бы одного такого флага MBR считывает в память первый сектор из помеченного им раздела и передает управление находящемуся в этом разделе коду. Раздел этого типа называется *системным* (system partition), а его первый сектор – *загрузочным сектором* (boot sector), или *загрузочной записью тома* (Volume Boot Record, VBR). Определенный для этого раздела том называется *системным* (system volume).

В общем случае операционные системы записывают на диск загрузочные секторы без участия пользователя. К примеру, программа Windows Setup при записи MBR одновременно создает код загрузки файловой системы (часть загрузочного сектора) в загружаемом разделе диска размером 100 Мбайт. Данный раздел помечается как скрытый, чтобы предотвратить случайное редактирование после загрузки операционной системы. Это системный том, о котором уже рассказывалось.

Перед записью в загрузочный сектор программа Windows Setup удостоверяется в том, что загрузочный раздел (то есть раздел, на который устанавливается Windows и который обычно не совпадает с системным разделом, где находятся загрузочные файлы) отформатирован для NTFS или форматирует загрузочный (и любой другой) раздел для NTFS. Файловая система NTFS – единственная, с которой может загружаться установленная на несъемный диск копия Windows. Следует заметить, что системный раздел может иметь любой формат, поддерживаемый Windows (например, FAT32). Если разделы уже корректно отформатированы, этот этап можно пропустить. После форматирования системного раздела Setup копирует туда программу Boot Manager (Bootmgr), которую Windows использует на системном томе.

Кроме того, программа Setup готовит хранилище конфигурационных данных загрузки (Boot Configuration Database, BCD), которое в системах на основе BIOS находится в файле \Boot\BCD корневого каталога системного тома. Этот файл содержит параметры начала работы устанавливаемой программой Setup версии Windows и всех ранее устанавливавшихся на этом компьютере версий. Если BCD уже существует, программа Setup просто добавляет туда строку, относящуюся к новой установленной копии. Подробно BCD описывается в главе 3 части I.

Загрузочный сектор систем на базе BIOS и Bootmgr

Перед записью загрузочного сектора программа Setup должна определить формат раздела, так как от этого будет зависеть содержимое этого сектора. Для раздела, отформатированного под NTFS, Windows записывает код NTFScapable. Этот код предоставляет Windows информацию о структуре и формате тома и осуществляет чтение из файла Bootmgr в корневом каталоге тома. Таким образом, в процессе загрузки этого

кода файловая система монтируется только для чтения. После того как файл Bootmgr окажется в памяти, код **NTFScapable** передает управление его точке входа. Если код загрузочного сектора не обнаружит Bootmgr в корневом каталоге тома, он покажет сообщение об ошибке «**BOOTMGR is missing**» (отсутствует Bootmgr).

Диспетчер загрузки представляет собой совокупность СОМ- и EXE-файлов (**Startup.com** и **Bootmgr.exe**), поэтому он появляется, когда система функционирует в реальном режиме, связанном с СОМ-файлами. В этом режиме не поддерживается преобразование виртуальных адресов памяти в физические. То есть программы, использующие адреса в памяти, интерпретируют их как физические, при этом доступен только первый мегабайт физической памяти компьютера. Именно в реальном режиме выполняются простые DOS-программы. Поэтому Bootmgr в первую очередь переключает систему в защищенный режим. На этом этапе процесса загрузки преобразование виртуальных адресов в физические по-прежнему не поддерживается, но появляется доступ к 32-разрядной адресации памяти. Полный доступ к физической памяти Bootmgr получает после перехода системы в защищенный режим. Создав достаточное количество страниц, чтобы сделать доступной память ниже 16 Мбайт, Bootmgr включает подкачку. Именно в защищенном режиме с включенной подкачкой обычно работает Windows.

После того как диспетчер Bootmgr включает защищенный режим, он становится полнофункциональным. Но для доступа к системам на базе IDE, загрузочным дискам и монитору ему по-прежнему необходимы BIOS-функции. Эти функции ненадолго переключают процессор в реальный режим, обеспечивая выполнение BIOS-служб. Затем Bootmgr при помощи встроенного кода файловой системы считывает из папки **\Boot\BCD**-файл. Как и код загрузочного сектора, Bootmgr содержит облегченную библиотеку файловой системы NTFS (кроме того, Bootmgr поддерживает и другие файловые системы, например FAT, El Torito CDFS и UDFS, а также WIM- и VHD-файлы). Но в отличие от кода загрузочного сектора, код файловой системы диспетчера Bootmgr может осуществлять чтение из вложенных папок.

ПРИМЕЧАНИЕ

Диспетчер загрузки и другие участвующие в загрузке приложения могут осуществлять запись в уже выделенные файлы на NTFS-томе, так как при этом записываются только данные, а вся работа по выделению места для файла на NTFS-томах уже сделана. Именно поэтому этим приложениям удается выполнять запись, например, в файл **bootsect.dat**.

Затем Bootmgr очищает экран. Если в Windows был включен BCD-режим, информирующий Bootmgr о пробуждении из спящего режима, процесс загрузки сокращается путем запуска файла **Winresume.exe**, считающего в память содержимое файла гибернации и передающего управление коду ядра, который осуществляет выход из спящего режима. Этот код отвечает за повторный запуск драйверов, которые были активными во время отключения системы. Файл **Hiberfil.sys** действителен только в случае, когда последнее отключение компьютера означало переход в спящий режим. После выхода оттуда файл **Hiberfil.sys** объявляется недействительным, чтобы избежать неоднократных пробуждений из одного и того же состояния. (Спящий режим рассматривался в разделе «Диспетчер электропитания» главы 8.)

При наличии в BCD нескольких записей с вариантами загрузки Bootmgr предлагает пользователю загрузочное меню (если запись всего одна, Bootmgr сразу переходит к запуску приложения Winload.exe). Выбор варианта в BCD заставляет диспетчер загрузки перейти к разделу, на котором находится системная папка (обычно это \Windows) с выбранным вариантом установки. Если текущая копия Windows получена обновлением более старой версии, этот раздел может совпадать с системным. В случае же установки «с нуля» это всегда будет скрытый раздел размером 100 Мбайт, о котором уже упоминалось.

Записи в BCD могут содержать дополнительные аргументы, интерпретируемые диспетчером загрузки, приложением Winload и другими принимающими участие в процессе загрузки компонентами. Список этих аргументов с описанием их влияния на Bootmgr представлен в табл. 13.2. Таблица 13.3 содержит список BCD-параметров для приложений загрузки, а табл. 13.4 – BCD-параметры для загрузчика Windows.

Таблица 13.2. BCD-параметры для диспетчера загрузки Windows (Bootmgr)

BCD-элемент	Тип значения	Назначение
bcdfilepath	Путь	Указывает местоположение на диске хранилища конфигурационных данных загрузки (обычно \Boot\BCD)
displaybootmenu	Логическое значение	Определяет, будет ли диспетчер загрузки выводить на экран загрузочное меню или воспользуется вариантом, предлагаемым по умолчанию
keyringaddress	Физический адрес	Указывает физический адрес, по которому располагается набор ключей для BitLocker
noerrordisplay	Логическое значение	Отключает вывод сообщений об обнаруженных диспетчером загрузки ошибках
Resume	Логическое значение	Определяет, следует ли предпринять попытку пробудить компьютер из спящего режима. Этот параметр автоматически устанавливается при переходе Windows в спящий режим
Timeout	Секунды	Количество секунд, которое должен подождать диспетчер загрузки перед выбором варианта, предлагаемого по умолчанию
Resumeobject	GUID	Идентификатор для приложения, выполняемого при выходе из спящего режима
displayorder	Список	Порядок вывода на экран пунктов меню загрузчика для нескольких операционных систем
toolsdisplayorder	Список	Порядок вывода на экран пунктов меню загрузчика для нескольких вариантов средств диагностики
bootsequence	Список	Определяет одноразовую последовательность вывода на экран для следующей загрузки

BCD-элемент	Тип значения	Назначение
Default	GUID	Запись, которую по умолчанию выбирает диспетчер загрузки по истечении тайм-аута
Customactions	Список	Определяет настраиваемые действия, которые выполняются при нажатии определенных клавиатурных комбинаций
bcddevice	GUID	Идентификатор устройства, на котором располагается BCD-хранилище

Таблица 13.3. BCD-параметры для приложений загрузки

BCD-элемент	Тип значения	Предназначение
avoidlowmemory	Целое значение	Заставляет загрузчик по возможности избегать физических адресов ниже определенного значения. Иногда требуется для устаревших устройств (таких, как ISA), где используется или видима только память ниже 16 Мбайт
badmemoryaccess	Логическое значение	Заставляет использовать страницы из списка поврежденной памяти (подробно этот список рассматривается в главе 10)
badmemorylist	Массив номеров страницного блока (Page Frame Number, PFN)	Задает список физических страниц в системе, которые известны как поврежденные из-за проблем с RAM
baudrate	Скорость передачи данных в битах в секунду	Переопределяет заданную по умолчанию скорость передачи (19200), при которой хост удаленного отладчика ядра соединяется через последовательный порт
bootdebug	Логическое значение	Включает для загрузчика удаленную отладку загрузки. Установка этого параметра позволяет использовать для соединения с загрузчиком Kd.exe или Windbg.exe
bootems	Логическое значение	Заставляет Windows включить для приложений загрузки службы аварийного управления (Emergency Management Services, EMS), предоставляющие сведения о загрузке и принимающие команды управления системой через последовательный порт
busparams	Строка	Если используется аппаратное отладочное устройство на шине PCI для обеспечения отладки FireWire или последовательного порта, задает шину PCI, функцию и номер устройства
channel	Канал между 0 и 62	Вместе с {debugtype, 1394} задает канал шины IEEE 1394, через который будет происходить обмен отладочной информацией с ядром

продолжение ↗

Таблица 13.3 (продолжение)

BCD-элемент	Тип значения	Предназначение
configaccess-policy	Default, DisallowMmConfig	Определяет, будет ли система использовать отображенный на память ввод-вывод для доступа к конфигурационному пространству PCI или произойдет переход к использованию HAL-процедур для доступа к порту ввода-вывода. Иногда помогает решить проблему с платформой устройства
debugaddress	Аппаратный адрес	Определяет аппаратный адрес последовательного (COM) порта, применяемого для отладки
debugport	Номер COM-порта	Переопределяет заданный по умолчанию последовательный порт (в системах с хотя бы двумя такими портами это обычно COM2), к которому подсоединяется хост удаленного отладчика ядра
debugstart	Active, AutoEnable, Disable	Задает параметры отладчика при включении отладки на уровне ядра. AutoEnable включает отладчик при возникновении точки останова или исключения ядра, включая сбои в работе ядра
debugtype	Serial, 1394, USB	Определяет, через какой порт – последовательный FireWire (IEEE 1394) или USB 2.0 – будет осуществляться взаимодействие при отладке на уровне ядра. (По умолчанию используется последовательный порт)
emsbaudrate	Скорость передачи данных в битах в секунду	Задает скорость передачи данных для EMS
emsport	Номер COM-порта	Задает последовательный (COM) порт для EMS
Extendedinput	Логическое значение	Включает в приложениях загрузки поддержку BIOS для расширенного ввода с консоли
firstmegabyte-policy	UseNone, UseAll, UsePrivate	Определяет, каким образом HAL использует нижний 1 Мбайт физической памяти, чтобы через BIOS уменьшить ущерб при изменении режима электропитания
fontpath	Строка	Путь к OEM-шрифту, который следует использовать в загрузочном приложении
graphicsmodedisabled	Логическое значение	Отключает графический режим в загрузочных приложениях
graphicsresolution	Разрешение	Задает разрешение графики в загрузочных приложениях

BCD-элемент	Тип значения	Предназначение
Initialconsole-input	Логическое значение	Задает начальный символ, который система вставляет во входной буфер клавиатуры PC/AT
integrityservices	Default, Disable, Enable	Включает и отключает службы целостности кода, используемые сертификатами Kernel Mode Code Signing. По умолчанию они включены
locale	Строка локализации	Задает языковые настройки для приложений загрузки (например, EN-US)
noumex	Логическое значение	Отключает исключения режима пользователя при отладке в режиме ядра. Если система зависла при загрузке в режиме отладки, попробуйте включить этот параметр
novesa	Логическое значение	Отключает использование режимов VESA-отображения
recoveryenabled	Логическое значение	Включает последовательность восстановления, если таковая имеется. Используется в новых вариантах установки Windows для предоставления интерфейса Startup And Recovery на базе PE
recoverysequence	Список	Определяет последовательность восстановления (описанную ранее)
relocatephysical	Физический адрес	Перемещает автоматически выделенную физическую память NUMA-узла на указанный физический адрес
targetname	Строка	Задает имя USB-отладчика при использовании USB2 {debugtype, usb}
testsigning	Логическое значение	Включает режим тестовых подписей, позволяющий разработчикам загружать локально подписанные 64-разрядные драйверы. Приводит к появлению на рабочем столе водяных знаков
traditionalkseg-mappings	Логическое значение	Указывает, будет ли ядро выполнять традиционное отображение KSEG0, изначально требовавшееся для поддержки MIPS. С отображением KSEG0 нижние 24 бита виртуального адресного пространства ядра отображаются на тот же самый физический адрес (то есть виртуальные 0x80800000 это 0x800000 в RAM). Отключение этого параметра увеличивает размер доступных младших адресов памяти, что может помочь работе некоторых устройств
truncatememory	Адрес в байтах	Игнорирует физическую память выше указанного физического адреса

Таблица 13.4. BCD-параметры для загрузчика Windows (Winload)

Элемент BCD	Тип значения	Предназначение
advancedoptions	Логическое значение	В случае значения <code>false</code> при сбое загрузки запускается загрузочная запись с командой автоматического восстановления; в противном случае показывает ошибку загрузки и предлагает пользователю меню дополнительных вариантов загрузки, связанное с загрузочной записью. Эквивалент нажатия клавиши F8
bootlog	Логическое значение	Заставляет Windows сохранить журнал загрузки в файле <code>%SystemRoot%\Ntbtlog.txt</code>
bootstatuspolicy	DisplayAllFailures, IgnoreAllFailures, IgnoreShutdownFailures, IgnoreBootFailures	Переопределяет поведение, заданное по умолчанию, согласно которому пользователю предлагается меню загрузки, если система не завершила предыдущую загрузку или процесс выключения
bootux	Disabled, Basic, Standard	Задает видимый пользователю во время загрузки графический интерфейс. Значение <code>Disabled</code> означает отсутствие графики (только черный экран), в то время как вариант <code>Basic</code> сделает видимым индикатор хода загрузки. В варианте <code>Standard</code> во время загрузки выводится обычный анимированный логотип Windows
clustermodead-dressing	Количество процессоров	Задает максимальное число процессоров, включенных в один кластер улучшенного программируемого контроллера прерываний (APIC)
configflags	Флаги	Задает связанные с процессором конфигурационные флаги
dbgtransport	Имя транспорта	Заставляет вместо одного из транспортов отладки ядра, предлагаемых по умолчанию (<code>Kdcom.dll</code> , <code>Kd1394</code> , <code>Kdusb.dll</code>), использовать данный файл, разрешая применять специализированный отладочный транспорт, обычно не поддерживающийся в Windows
debug	Логическое значение	Включает отладку в режиме ядра
detecthal	Логическое значение	Включает динамическое обнаружение уровня HAL
driverloadfail-urepolicy	Fatal, UseErrorControl	Определяет поведение загрузчика при сбое драйвера загрузки. Значение <code>Fatal</code> предотвращает загрузку, в то время как значение <code>UseErrorControl</code> заставляет систему использовать заданное в служебном разделе поведение, предлагаемое по умолчанию для сбояного драйвера

Элемент BCD	Тип значения	Предназначение
ems	Логическое значение	Приказывает ядру также использовать EMS. (Если применяется только компонент bootems, службу EMS будет использовать лишь загрузчик)
evstore	Строка	Хранит местоположение куста, загружающегося на этапе первоначальной загрузки
exportascd	Логическое значение	При установке этого параметра ядро будет воспринимать указанный файл ramdisk как ISO-образ, а не как файл типа Windows Installation Media (WIM) или System Deployment Image (SDI)
groupaware	Логическое значение	При ассоциировании группы с новым процессом заставляет системы задействовать группы, отличные от нулевой. Используется только в 64-разрядной Windows
groupsize	Целое	Задает максимальное количество логических процессоров, которые могут быть частью группы (максимум 64). Может применяться для создания групп в системах, для обычного функционирования которых группы не требуются. Значение должно быть степенью двойки. Используется только в 64-разрядных версиях Windows
hal	Имя HAL-образа	Переопределяет имя файла с HAL-образом (hal.dll). Этот параметр позволяет загружать систему в однопроцессорном и многопроцессорном режимах (при этом требуется указать также параметр kernel)
halbreakpoint	Логическое значение	Заставляет HAL на раннем этапе инициализации прервать работу в точке останова. В процессе своей инициализации ядро Windows прежде всего инициализирует уровень HAL, поэтому данная точка останова является самой ранней из всех возможных (если не производится отладка загрузки). Если задать данный параметр без переключателя /DEBUG, система покажет синий экран с кодом ошибки 0x00000078 (PHASE0_EXCEPTION)
hypervisorbaudrate	Скорость передачи данных в битах в секунду	Задает скорость передачи данных для последовательного отладчика гипервизора
hypervisor-channel	Номер канала от 0 до 62	При отладке гипервизора на канале шины FireWire (IEEE 1394) задает номер канала
hypervisordebug	Логическое значение	Включает режим отладки гипервизора

продолжение ⌘

Таблица 13.4 (продолжение)

Элемент BCD	Тип значения	Предназначение
hypervisorde-bugport	Номер COM-порта	При последовательной отладке гипервизора задает номер COM-порта
hypervisorde-bugtype	Serial, 1394	Задает аппаратный порт, который будет использовать отладчик гипервизора
hypervisorsidis-ableslat	Логическое значение	Заставляет гипервизор игнорировать технологию преобразования адресов второго уровня (SLAT), если она поддерживается процессором
hypervisor-launchtype	Off, Auto	Включает загрузку гипервизора в системе Hyper-V или приводит к его отключению
hypervisorpath	Имя образа двоичного файла гипервизора	Задает путь к двоичному файлу гипервизора
hypervisoruselargevtlb	Логическое значение	Заставляет гипервизор использовать больше виртуальных TLB-записей
increaseuserservo	Размер в мегабайтах	Увеличивает размер пользовательского адресного пространства с 2 Гбайт до указанной величины, максимально до 3 Гбайт (при этом уменьшается системное адресное пространство). Это позволяет повысить производительность зависящих от виртуальной памяти приложений, например сервера базы данных. (Подробно эта тема рассматривается в главе 9)
kernel	Имя образа ядра	Переопределяет имя файла с образом ядра (Ntoskrnl.exe). Этот параметр позволяет загружать систему в однопроцессорном и многопроцессорном режимах (при этом требуется указать также параметр hal)
lastknowngood	Логическое значение	Выполняет загрузку последней удачной конфигурации
loadoptions	Дополнительные параметры командной строки	Задает дополнительные параметры командной строки, которые не определяются BCD-записями. Эти параметры применяются для задания компонентов системы, которые не в состоянии пользоваться BCD (например, устаревшие компоненты)
maxgroup	Логическое значение	До максимума увеличивает число групп процессора, создаваемых при его конфигурировании. Выбор групп и соотношение с NUMA подробно рассматриваются в главе 3 части I
maxproc	Логическое значение	Принудительно ограничивает максимальное количество процессоров, добавляемых через динамическую поддержку, о чем Windows сообщает драйверам и приложениям

Элемент BCD	Тип значения	Предназначение
msi	Default, ForceDisable	Позволяет отключать поддержку прерываний, инициируемых сообщениями
nocrashauto-reboot	Логическое значение	Отключает автоматическую перезагрузку после системного сбоя (синего экрана)
nointegrity-checks	Логическое значение	Отключает выполняемую Windows проверку целостности при загрузке драйверов. Автоматически удаляется при перезагрузке
nolowmem	Логическое значение	Требует включенного режима PAE и наличия в системе более 4 Гбайт физической памяти. В этом случае версия ядра Windows с включенным режимом PAE, Ntkrnlpa.exe, не использует первые 4 Гбайт физической памяти. При этом все приложения и драйверы загружаются, а пулы памяти выделяются выше этой границы. Этот переключатель применяется при тестировании совместимости драйвера устройства с системами, обладающими большим объемом памяти
numproc	Количество процессоров	Задает количество процессоров, которые могут использоваться в многопроцессорной системе. Например, параметр /NUMPROC=2 в системе с четырьмя процессорами запретит Windows пользоваться двумя из них
nx	OptIn, OptOut, Always-Off, AlwaysOn	Этот параметр доступен только в 32-разрядных версиях Windows при наличии процессоров, поддерживающих область памяти с пометкой «только для данных» и при включенном режиме PAE. Включает режим предотвращения выполнения данных. Этот вариант защиты по умолчанию включен в 64-разрядных версиях Windows на x64-процессорах (см. главу 9)
onecpu	Логическое значение	В многопроцессорной системе заставляет Windows использовать всего один процессор
optionsedit	Логическое значение	Включает в диспетчере загрузки редактор параметров. Это дает пользователю возможность интерактивно задавать параметры и переключатели командной строки для текущей загрузки. Эквивалент нажатия клавиши F10
osdevice	GUID	Задает устройство, на котором установлена операционная система

продолжение ↴

Таблица 13.4 (продолжение)

Элемент BCD	Тип значения	Предназначение
pae	Default, ForceEnable, ForceDisable	Вариант Default позволяет загрузчику определить, поддерживает ли система режим PAE, и запустить PAE-ядро. Вариант ForceEnable принудительно задает это поведение, в то время как ForceDisable заставляет загрузчик воспользоваться версией ядра без расширения физических адресов, даже если система распознана как поддерживающая x86 PAE и в ней присутствует более 4 Гбайт физической памяти
pciexpress	Default, ForceDisable	Отключает поддержку шины и устройств PCI Express
perfmem	Размер в мегабайтах	Задает размер буфера для регистрации данных производительности. Этот параметр действует аналогично переключателю removememory, так как не дает Windows узнать размер, который указан как доступная память
quietboot	Логическое значение	Дает Windows команду не инициализировать драйвер VGA-видеоадаптера, ответственный за показ растровой графики в процессе загрузки. Этот драйвер используется для вывода данных о ходе загрузки, поэтому его отключение не даст Windows вывести эту информацию
ramdiskimage-length	Длина в байтах	Указанный размер ramdisk
ramdiskimage-offset	Смещение в байтах	Если перед виртуальной файловой системой в файле ramdisk содержатся другие данные (например, заголовок), этот параметр сообщает загрузчику, откуда начинать чтение файла ramdisk
ramdisksdipath	Имя файла образа	Определяет имя загружаемого SDI-образа файла ramdisk
ramdisktftp-blocksize	Размер блока	При загрузке WIM для файла ramdisk с сервера по протоколу Trivial FTP (TFTP) определяет размер используемых блоков
ramdisktftpclientport	Номер порта	Задает порт загрузки WIM для файла ramdisk с сервера по протоколу Trivial FTP (TFTP)
ramdisktftpwindowsize	Размер окна	Задает размер окна при загрузке WIM для файла ramdisk с сервера по протоколу Trivial FTP (TFTP)
removememory	Размер в байтах	Запрещает Windows использовать указанный объем памяти

Элемент BCD	Тип значения	Предназначение
restrictapic-cluster	Число кластеров	Задает максимальное количество APIC-кластеров, которыми будет пользоваться система
resumeobject	GUID объекта	Указывает, какое приложение будет использоваться при пробуждении из спящего режима. Обычно это программа Winresume.exe
safeboot	Minimal, Network, DsRepair	Задает параметры загрузки в безопасный режим. Вариант Minimal соответствует безопасному режиму без сетевого подключения, Network – безопасному режиму с подключением, а DsRepair – безопасному режиму со службой восстановления каталогов (безопасный режим подробно рассматривается далее)
safebootalternateshell	Логическое значение	Заставляет Windows вместо Проводника использовать программу, указанную в разделе HKLM\SYSTEM\CurrentControlSet\Control\SafeBoot\AlternateShell. Этот вариант называют запуском ОС в безопасном режиме с поддержкой командной строки
sos	Логическое значение	Заставляет Windows выводить список загружаемых драйверов, а затем – версию системы (в том числе номер сборки), количество физической памяти и число процессоров
stampdisks	Логическое значение	Заставляет Winload записывать MBR-метку диска на RAW-диск при загрузке среды Windows PE (Среда предустановки). Это может потребоваться в средах развертывания для отображения жестких дисков, пронумерованных операционной системой, на жесткие диски, пронумерованные BIOS, чтобы понять, какой из дисков будет системным
systemroot	Строка	Задает путь относительно osdevice к месту установки операционной системы
targetname	Имя	При отладке USB 2.0 назначает имя машине, на которой происходит отладка
tpmbootentropy	Default, ForceDisable, ForceEnable	Заставляет загрузчик собирать с доверенного платформенного модуля (TPM) данные об энтропии и передавать их ядру. Если используется TPM Boot Entropy, полученные от TPM данные передаются генератору случайных чисел (RNG) ядра
usefirmwarepcisettings	Логическое значение	Запрещает Windows динамически назначать PCI-устройствам ресурсы IO/IRQ. Более подробно эта тема рассматривается в статье 148501 базы знаний Microsoft

продолжение ↴

Таблица 13.4 (продолжение)

Элемент BCD	Тип значения	Предназначение
uselegacyapic-mode	Логическое значение	Принудительно включает базовую функциональность APIC, даже если микросхемы поддерживают расширенную функциональность APIC. Используется в случае аппаратных ошибок и/или несовместимости
usephysicaldestination	Логическое значение	Заставляет использовать физический APIC-контроллер
useplatformclock	Логическое значение	Заставляет использовать источник тактовых импульсов для счетчика производительности системы
vga	Логическое значение	Заставляет Windows использовать драйвер VGA-видеоадаптера вместо высокопроизводительных драйверов сторонних производителей
winpe	Логическое значение	Используется Windows PE. Этот параметр заставляет диспетчер конфигурации загружать куст реестра SYSTEM как изменяемый, в результате изменения, вносимые в него в памяти, не сохраняются в образе куста
x2apicpolicy	Disabled, Enabled, Default	Определяет, следует ли использовать расширенную функциональность APIC, если она поддерживается базовым набором микросхем. Вариант Disabled эквивалентен заданию параметра uselegacyapicmode, в то время как Enabled запускает функциональность ACPI даже при наличии ошибок. Вариант Default использует заявленные возможности микросхемы (если нет ошибок)
xsavepolicy	Целое значение	Загружает указанную политику XSAVE из драйвера политик ресурсов XSAVE (Hwpolicy.sys)
xsaveaddfeature0-7	Целое значение	Используется при тестировании поддержки XSAVE на современных процессорах Intel; позволяет имитировать наличие определенных функций, которыми процессор на самом деле не обладает. Это увеличивает размер структуры CONTEXT и подтверждает, что приложения смогут работать с дополнительными функциями, которые могут появиться в будущем
Xsaveremovefeature	Целое значение	Отменяет сообщение ядру о зарегистрированной XSAVE-функции, даже если она поддерживается процессором

Элемент BCD	Тип значения	Предназначение
Xsaveprocessorsmask	Целое значение	Битовая карта, указывающая, к каким процессорам должны применяться XSAVE-инструкции
xsavedisable	Логическое значение	Отключает функциональность XSAVE, даже если она поддерживается процессором

Инструмент `Bcdedit.exe` предоставляет удобный интерфейс для задания набора переключателей. Некоторые BCD-параметры хранятся в виде переключателей командной строки (например, `/DEBUG`) в параметре `HKLM\SYSTEM\CurrentControlSet\Control\SystemStartOptions` реестра; в противном случае они хранятся только в бинарном формате в BCD-кусте.

Если за время, указанное в BCD, пользователь не выберет свой вариант в предложенном ему меню, Bootmgr воспользуется вариантом, предлагаемым по умолчанию (при наличии в меню всего одной записи выбор осуществляется незамедлительно). После этого Bootmgr запускает связанный с выбранной записью загрузчик. Для установки Windows это `Winload.exe`.

`Winload.exe` также содержит код, который делает запрос к ACPI-интерфейсу в BIOS для получения базовых сведений об устройстве и конфигурации системы. Сюда входят:

- время и дата (хранятся в энергонезависимой CMOS-памяти);
- количество, размер и тип дисковых накопителей в системе;
- информация об устаревших устройствах, таких как шины (например, ISA, PCI, EISA, MCA), мышь, параллельные порты и видеоадаптеры, недоступных при непосредственном опросе.

Эта информация накапливается во внутренних структурах данных и позднее при загрузке сохраняется в разделе `HKLM\HARDWARE\DESCRIPTION` реестра. По большей части это унаследованные данные, такие как параметры CMOS и распознанные BIOS конфигурационные параметры диска, а также устаревшие шины, больше не поддерживаемые Windows. Вся эта информация хранится в основном для обеспечения совместимости. В настоящее время актуальные сведения об аппаратном обеспечении хранятся в PnP-диспетчере.

Затем `Winload` начинает загружать необходимые для инициализации ядра файлы с загрузочного тома (boot volume). Загрузочным называется том, на котором располагается системная папка (обычно `\Windows`) устанавливаемой операционной системы. Вот какие действия производит при этом приложение `Winload`:

1. Загрузка соответствующих образов ядра и HAL (по умолчанию это файлы `Ntoskrnl.exe` и `Hal.dll`), а также всех зависящих от них файлов. Если `Winload` не в состоянии загрузить какой-то из нужных файлов, появляется сообщение «Windows could not start because the following file was missing or corrupt» (Windows невозможно запустить, потому что отсутствует или поврежден файл), за которым следует имя файла.

2. Считывание файла VGA-шрифта (по умолчанию это файл `vgaoem.fon`). В случае неудачи появляется сообщение об ошибке, аналогичное упомянутому на предыдущем шаге.
3. Считывание NLS-файлов, отвечающих за кодировки в системе. По умолчанию это файлы `l_intl.nls`, `c_1252.nls` и `c_437.nls`.
4. Считывание раздела `\Windows\System32\Config\System` реестра, чтобы определить, какие драйверы необходимы для завершения загрузки. (Реестр подробно рассматривается в главе 4 части I.)
5. Сканирование в памяти раздела `SYSTEM` реестра с целью обнаружения всех *драйверов загрузочных устройств* (boot device drivers). Они указаны в реестре со значением `SERVICE_BOOT_START (0)`. Каждому драйверу устройства соответствует подраздел реестра в разделе `HKLM\SYSTEM\CurrentControlSet\Services`. Например, показанный на рис. 13.2 подраздел `fvevol` соответствует драйверу BitLocker. (Подробно раздел `Services` реестра рассматривается в главе 4 части I.)

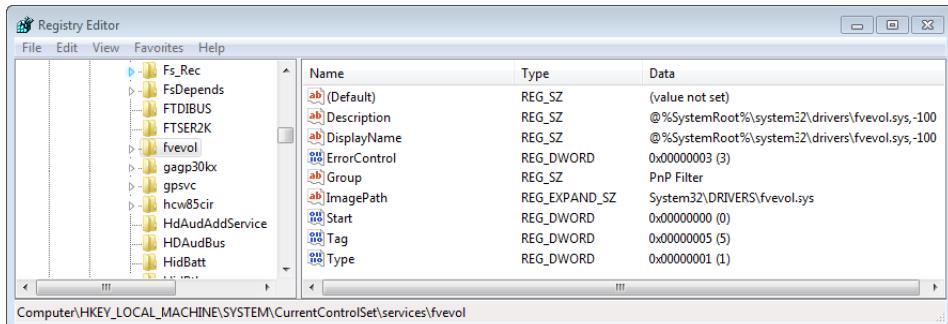


Рис. 13.2. Служебные параметры драйвера BitLocker

6. Добавление в список загрузочных драйверов драйвера файловой системы, отвечающего за реализацию кода для типа раздела (NTFS), на котором находится папка установки. Программа Winload должна загрузить этот драйвер именно сейчас. В противном случае ядро будет требовать, чтобы драйверы загрузили сами себя, и получится замкнутый круг.
7. Загрузка драйверов, обязательных для запуска системы. Чтобы обозначить процесс загрузки, Winload обновляет индикатор под текстом «Starting Windows». Если в BCD был установлен параметр `sos`, вместо индикатора Winload выводит имена файлов всех загрузочных драйверов. Имейте в виду, что драйверы на данном этапе только загружаются, их инициализация происходит позже.
8. Подготовка регистров процессора для выполнения файла `Ntoskrnl.exe`.

Кроме того, для шагов 1 и 8 Winload реализует часть инфраструктуры подписи кода в режиме ядра (Kernel Mode Code Signing, KMCS), которая описывалась в главе 3 части I, принудительно подписывая все 64-разрядные загрузочные драйверы. Кроме

того, при некорректной сигнатуре файлов ранней стадии загрузки система работать не сможет.

На этом участие программы Winload в процессе загрузки заканчивается. Для выполнения остальных процедур инициализации системы программа вызывает основную функцию `KiSystemStartup` в файле `Ntoskrnl.exe`.

Загрузка в UEFI-системах

Программное обеспечение UEFI-совместимой системы запускает код загрузчика, который программа Windows Setup поместила в энергонезависимую память (NVRAM). Загрузочный код читает данные из BCD-хранилища, которое также находится в NVRAM. Ранее упоминавшаяся утилита `Bcdedit.exe` способна абстрагироваться от связанных с программным обеспечением NVRAM-переменных в BCD, что обеспечивает полную прозрачность всего механизма.

Согласно стандарту UEFI (Unified Extensible Firmware Interface – единый расширяемый микропрограммный интерфейс), пользователю предлагается EFI-диспетчер загрузки, в котором можно выбрать подлежащую загрузке операционную систему или дополнительные приложения. Но для согласованности пользовательского интерфейса в системах на базе BIOS и UEFI Windows предлагает 2-секундную задержку для выбора EFI-диспетчера загрузки, после чего вместо него запускается EFI-версия приложения `Bootmgr` (`Bootmgfw.efi`).

Затем происходит распознавание аппаратного обеспечения, в процессе которого загрузчик через UEFI-интерфейсы определяет номера и типы следующих устройств:

- сетевые адаптеры;
- видеoadаптеры;
- клавиатуры;
- контроллеры дисков;
- накопители.

В UEFI-системах все операции и программы выполняются в штатном режиме работы процессора с включенной подкачкой, и ни один из этапов процесса загрузки Windows не выполняется в 16-разрядном режиме. Следует заметить, что хотя интерфейс EFI поддерживается как в 32-, так и в 64-разрядной системах, Windows обеспечивает поддержку EFI только на 64-разрядной платформе.

Аналогично приложению `Bootmgr` в системах x86 и x64, EFI-диспетчер загрузки после необязательной паузы выводит меню с вариантами загрузки. После выбора одного из вариантов загрузчик локализует папку, соответствующую выбранному разделу EFI System, и запускает EFI-версию загрузчика Windows (`Winload.efi`).

Согласно спецификации UEFI, в системе должен присутствовать отформатированный для файловой системы FAT раздел EFI System размером от 100 Мбайт до 1 Гбайт, что составляет менее 1 % от размера диска. Кроме того, для каждой версии Windows должна быть выделена подпапка на разделе EFI System в папке `EFI\Microsoft`.

Следует упомянуть, что благодаря унифицированному процессу загрузки и единой модели перечисленные в табл. 13.1 компоненты практически идентичны использу-

ющимся в UEFI-системах. Отличается только расширение: вместо .exe фигурирует .efi, кроме того, вместо BIOS-прерываний применяются службы и API-интерфейсы, относящиеся к EFI. Чтобы избежать ограничений, накладываемых на формат MBR-раздела (в том числе максимум из четырех разделов на диск), в UEFI-системах используется формат GPT (GUID Partition Table — таблица GUID разделов), в котором для идентификации различных разделов и их роли в системе применяются глобальные уникальные идентификаторы (GUID).

ПРИМЕЧАНИЕ

Хотя стандарт EFI доступен с 2001 года, а UEFI — с 2005 года, производители его практически не используют из-за проблем обратной совместимости и сложностей перехода от устоявшейся за 20 лет технологии к чему-то новому. Исключениями являются компьютеры на базе Itanium и Intel Macintosh от Apple.

Загрузка с iSCSI-устройств

iSCSI-устройства представляют собой подключаемые через Сеть (Internet SCSI, iSCSI) устройства с интерфейсом малых вычислительных систем (Small Computer System Interface, SCSI). Это особый вид накопителей, в которых удаленные физические диски подключаются через адаптер главной шины (Host Bus Adapter, HBA) или через Ethernet. Они отличаются от традиционных сетевых хранилищ данных (Network-Attached Storage, NAS), так как обеспечивают доступ к дискам на уровне блоков, в отличие от применяемого в NAS логического доступа через сетевую файловую систему. Соответственно, пока iSCSI-инициатор обеспечивает доступ через Ethernet, iSCSI-диск для загрузчика и операционной системы выглядит так же, как любой другой дисковый накопитель. Применение вместо локальных накопителей iSCSI-дисков позволяет экономить место, электроэнергию и охлаждение.

Хотя традиционно Windows поддерживает загрузку только с локально подсоединенными дисками, а загрузку по сети — через PXE, современные версии Windows обладают встроенной возможностью загрузки с iSCSI-устройств через механизм iSCSI-загрузки. Загрузчик (*Winload.exe*) включает в себя минимальный сетевой стек стандарта UNDI (Universal Network Device Interface — универсальный интерфейс сетевых устройств), позволяющий совместимым сетевым адаптерам (Network Interface Card, NIC) отвечать на прерывание 13h (унаследованное прерывание дискового ввода-вывода в BIOS) и преобразовывать запросы в сетевой ввод-вывод. В EFI-системах вместо этого используется предоставляемый производителем драйвер сетевого интерфейса, а вместо прерываний — API-интерфейсы EFI-устройств.

Чтобы узнать местоположение удаленного диска и путь к нему, а также получить сведения для проверки его подлинности, загрузчик читает микропрограммную таблицу iSCSI-загрузки (iSCSI Boot Firmware Table, iBFT). Доступ к этой таблице, которая должна находиться в физической памяти, обычно выполняется через ACPI (Advanced Configuration and Power Interface — усовершенствованный интерфейс управления конфигурированием и энергопотреблением). iBFT-таблицу должна также читать программа Windows Setup, определяя загружаемые iSCSI-устройства и обеспечивая их

непосредственную (без создания образа) установку. Вместе с iSCSI-инициатором это все, что нужно Windows для загрузки с iSCSI-устройства, как показано на рис. 13.3.

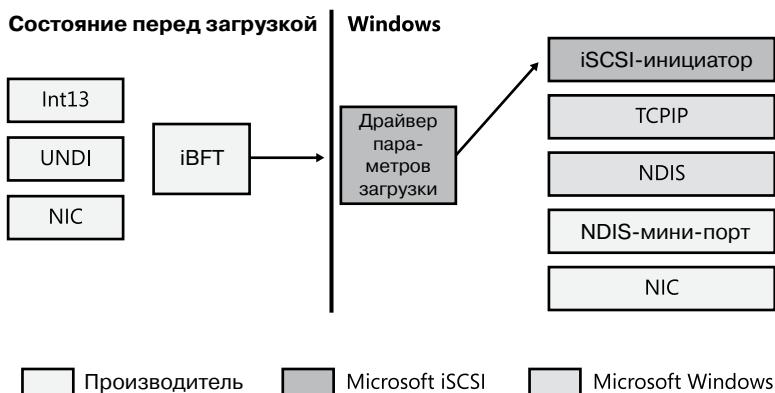


Рис. 13.3. Архитектура iSCSI-загрузки

Инициализация ядра и исполнительных подсистем

Когда программа Winload вызывает Ntoskrnl, она передает структуру данных, называемую блоком параметров загрузчика. В нем содержатся пути к системному и загрузочному разделам, указатель на таблицы памяти, которые Winload генерирует для описания физической памяти системы, древовидный список подключаемых устройств, в дальнейшем используемый для построения изменяемого раздела **HARDWARE** реестра, хранящуюся в памяти копию раздела **SYSTEM** реестра, указатель на список загруженных драйверов, запускаемых Winload, и прочую информацию, связанную с происходящим до этого момента процессом загрузки.

ЭКСПЕРИМЕНТ: БЛОК ПАРАМЕТРОВ ЗАГРУЗЧИКА

В процессе загрузки ядро сохраняет указатель на блок параметров загрузчика в переменной KeLoaderBlock. После первой фазы загрузки ядро удаляет блок параметров, поэтому единственным способом увидеть содержимое данной структуры является присоединение отладчика ядра перед загрузкой и приостановка его работы в первой точке останова. В этом случае можно воспользоваться командой dt и вывести содержимое блока:

```
0: kd> dt poi(nt!KeLoaderBlock) nt!_LOADER_PARAMETER_BLOCK
+0x000 OsMajorVersion : 6
+0x004 OsMinorVersion : 1
+0x008 Size : 0x88
+0x00c Reserved : 0
+0x010 LoadOrderListHead : _LIST_ENTRY [
0x8085b4c8 - 0x80869c70 ]
+0x018 MemoryDescriptorListHead : _LIST_ENTRY [
0x80a00000 - 0x80a00de8 ]
```

продолжение ↗

```
+0x020 BootDriverListHead : _LIST_ENTRY [
0x80860d10 - 0x8085eba0 ]
+0x028 KernelStack : 0x88e7c000
+0x02c Prcb : 0
+0x030 Process : 0
+0x034 Thread : 0x88e64800
+0x038 RegistryLength : 0x2940000
+0x03c RegistryBase : 0x80adf000 Void
+0x040 ConfigurationRoot : 0x8082d450
_CONFIGURATION_COMPONENT_DATA
+0x044 ArcBootDeviceName : 0x8082d9a0
"multi(0)disk(0)rdisk(0)partition(4)"
+0x048 ArcHalDeviceName : 0x8082d788
"multi(0)disk(0)rdisk(0)partition(4)"
+0x04c NtBootPathName : 0x8082d828 "\Windows\
+0x050 NtHalPathName : 0x80826358 \
+0x054 LoadOptions : 0x8080e1b0 "NOEXECUTE=ALWAYSON
DEBUGPORT=COM1 BAUDRATE=115200"
+0x058 NlsData : 0x808691e0 _NLS_DATA_BLOCK
+0x05c ArcDiskInformation : 0x80821408 _ARC_DISK_INFORMATION
+0x060 OemFontFile : 0x84a551d0 Void
+0x064 Extension : 0x8082d9d8
_LOADER_PARAMETER_EXTENSION
+0x068 u : <unnamed-tag>
+0x074 FirmwareInformation : _FIRMWARE_INFORMATION_LOADER_BLOCK
```

Кроме того, можно применить к полю MemoryDescriptorListHead команду !loadermemorylist и получить список диапазонов физической памяти:

```
0: kd> !loadermemorylist 0x80a00000
Base Length Type
1 00000001 HALCachedMemory
2 00000004 HALCachedMemory
...
4a32 00000023 NlsData
4a55 00000002 BootDriver
4a57 00000026 BootDriver
4a7d 00000014 BootDriver
4a91 0000016f Free
4c00 0001b3f0 Free
1fff0 00000001 FirmwarePermanent
1fff1 00000002 FirmwarePermanent
1fff3 00000001 FirmwarePermanent
1fff4 0000000b FirmwarePermanent
1ffff 00000001 FirmwarePermanent
fd000 00000800 FirmwarePermanent
fec00 00000001 FirmwarePermanent
fee00 00000001 FirmwarePermanent
ffc00 00000400 FirmwarePermanent

Summary
Memory Type Pages
Free 0001bc50 ( 113744)
LoadedProgram 0000013d ( 317)
FirmwareTemporary 000006dd ( 1757)
FirmwarePermanent 00000c37 ( 3127)
OsloaderHeap 0000022a ( 554)
```

```
SystemCode 000005dc ( 1500)
BootDriver 00000968 ( 2408)
RegistryData 00002940 ( 10560)
MemoryData 00000035 ( 53)
NlsData 00000023 ( 35)
HALCachedMemory 0000001e ( 30)
=====
Total 00020bc5 ( 134085) = ~523MB
```

После этого `Ntoskrnl` начинает фазу 0, первую в разбитом на две фазы процессе инициализации (второй является фаза 1). Большинство компонентов исполнительной подсистемы имеет инициализирующую функцию, которая принимает параметр, указывающий на текущую фазу.

В фазе 0 прерывания отключены. Эта фаза предназначена для построения начальных структур, необходимых для последующего запуска служб на фазе 1. Основная функция `Ntoskrnl` вызывает функцию `KiSystemStartup`, которая, в свою очередь, вызывает для каждого процессора функции `HalInitializeProcessor` и `KiInitializeKernel`. Последняя при запуске на процессоре загрузки производит общесистемную инициализацию ядра, инициализируя, к примеру, внутренние списки и другие общие для всех процессоров структуры данных. Также она проверяет, указана ли виртуализация в качестве BCD-параметра `hypervisorlauchtype`, а также поддерживает ли технологию аппаратной виртуализации процессор. Затем первый экземпляр `KiInitializeKernel` вызывает отвечающую за управление фазой 0 функцию `InitBootProcessor`, в то время как остальные процессоры вызывают только `HalInitSystem`.

Функция `InitBootProcessor` начинает работу с инициализации для загрузочного процессора указателей на резервный пул и с проверки и принятия BCD-параметра `burnmemory`, отбрасывая указанный объем физической памяти. Затем выполняется инициализация достаточного количества NLS-файлов, загруженных программой `Winload` (об этом уже рассказывалось), чтобы начало работать преобразование Unicode в ANSI и OEM. После этого вызывается HAL-функция `HalInitSystem`, которая предоставляет HAL возможность получить контроль над системой до того, как Windows выполнит большую часть дальнейшей инициализации. Эта функция, в частности, отвечает за подготовку к работе системного контроллера прерываний на каждом процессоре и за конфигурирование прерываний по таймеру, что требуется для учета процессорного времени. (Дополнительную информацию об учете процессорного времени вы найдете в разделе «Кванты времени» главы 5 части I.)

После того как функция `HalInitSystem` возвращает управление, функция `InitBootProcessor` переходит к вычислению обратной величины для времени срабатывания таймера. В большинстве современных процессоров обратные величины применяются для оптимизации операции деления. Дело в том, что операция умножения выполняется процессорами быстрее. А так как для определения срока истечения работы таймера Windows нужно разделить 64-разрядное значение времени, это статическое вычисление уменьшает задержку прерывания при срабатывании таймера. Затем функция `InitBootProcessor` задает корневой путь системы и ищет в образе ядра строку с сообщением о системном сбое, которая появляется на синем экране. Местоположение этой строки помещается в кэш, что позволяет избежать опасной и ненадежной операции по ее поиску во время сбоя. Затем `InitBootProcessor` инициализирует функцию

квотирования в диспетчере процессов и считывает *вектор управления* (control vector). Эта структура данных содержит более 150 настраиваемых на уровне ядра параметров, относящихся к разделу `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control` реестра. К ним относятся и лицензионная информация, и сведения об устанавливаемой версии.

В результате функция `InitBootProcessor` подготавливается к вызову процедур инициализации фазы 0 для исполнительной подсистемы, верификатора драйверов (driver verifier) и диспетчера памяти. Перечислим этапы инициализации этих компонентов.

1. Исполнительная подсистема инициализирует различные внутренние блокировки, ресурсы, списки и переменные, а также удостоверяется, что в реестре находится допустимый тип набора пакетов, препятствуя модификации реестра при обновлении Windows-пакетами, которые на самом деле не были куплены. Это только одна из множества подобных проверок в ядре.
2. Верификатор драйверов, если он включен, инициализирует различные параметры и режимы исходя из текущего состояния системы (например, проверяет, включен ли безопасный режим) и заданных параметров проверки. Также он указывает, какие драйверы будут участвовать в тестах, проводимых со случайным набором драйверов.
3. Диспетчер памяти конструирует таблицы страниц и внутренние структуры данных, необходимые для функционирования базовых служб памяти. Еще он строит и резервирует область для кэша системного файла и создает области памяти для выгружаемого и невыгружаемого пулов (они описываются в главе 10). Остальные исполнительные подсистемы, ядро и драйверы устройств используют эти два пула памяти для выделения места под свои структуры данных.

Далее функция `InitBootProcessor` вызывает функцию `HalInitializeBios` для настройки кода эмуляции BIOS, который является частью HAL. Этот код как в BIOS-, так и в EFI-системах используется для предоставления доступа (или имитации доступа) к 16-разрядным прерываниям реального режима и памяти, которые необходимы Bootvid для реализации VGA-режима экрана загрузки и экрана отладки. После возвращения управления ядро инициализирует библиотеку Bootvid и через функции `InbvEnableBootDriver` и `InbvDriverInitialize` выводит на экран сообщение о состоянии раннего этапа загрузки.

На этой стадии функция `InitBootProcessor` составляет список загруженных Winload драйверов, предназначенных для первоначальной загрузки, и вызывает функцию `DbgLoadImageSymbols`, чтобы уведомить отладчик ядра (если он присоединен) о загрузке символов для каждого из этих драйверов. Если системный отладчик настроен на прерывание загрузки символов, у отладчика ядра появляется возможность получения контроля над системой на самой ранней стадии. Далее `InitBootProcessor` вызывает функцию `Hv1InitSystem`, которая пытается подключиться к гипервизору, если Windows является виртуальной машиной в среде Hyper-V. После того как `Hv1InitSystem` возвращает управление, `InitBootProcessor` вызывает функцию `HeadlessInit` с целью инициализации последовательной консоли, если машина настроена для использования службы аварийного управления (EMS).

Затем `InitBootProcessor` конструирует информацию о версиях, которая потребуется на более поздних этапах загрузки, например номер сборки, версию пакета обновления и статус бета-версии. После этого она копирует загруженные программой Winload

NLS-таблицы в выгружаемый пул, повторно их инициализирует и, если был установлен соответствующий глобальный флаг, создает базу данных отслеживания стека ядра. (Глобальные флаги подробно рассматриваются в главе 3 части I.)

Напоследок `InitBootProcessor` вызывает диспетчер объектов, монитор безопасности, диспетчер процессов, инфраструктуру отладки приложений в режиме пользователя и PnP-диспетчер. При инициализации эти компоненты выполняют следующие действия:

1. В процессе инициализации диспетчера объектов необходимые для конструирования пространства его имен объекты определяются таким образом, что другие компоненты получают возможность вставлять туда собственные объекты. Создается таблица дескрипторов для учета ресурсов.
2. Монитор безопасности инициализирует объект типа «маркер» и использует его для создания и подготовки маркера первой учетной записи локальной системы для присвоения начальному процессу. (Учетная запись локальной системы описывается в главе 6 части I.)
3. Диспетчер процессов по большей части выполняет свою инициализацию на фазе 0, определяя такие типы объектов, как «процесс» и «программный поток», а также создавая списки отслеживания активных процессов и программных потоков. Кроме того, он создает объект процесса `Idle` для начального процесса. В завершение диспетчер процессов создает процесс `System` и системный программный поток для выполнения процедуры `Phase1Initialization`. Но этот поток не запускается, так как прерывания пока отключены.
4. Инфраструктура отладки приложений в режиме пользователя создает определение типа объекта `debug`, предназначенного для присоединения к процессу отладчика и получения его событий. (Подробно отладка в режиме пользователя рассматривается в главе 3 части I.)
5. Инициализация PnP-диспетчера сводится к простой инициализации ресурса исполнительной системы, который используется для синхронизации доступа к ресурсам шины.

После возвращения управления к функции `KiInitializeKernel` остается выделить DPC-стек для текущего процессора и область сохранения карты привилегий ввода-вывода (только в системах x86), а затем управление переходит к циклу `Idle`, который заставляет системный программный поток, созданный на третьем шаге предыдущего процесса, перейти к выполнению фазы 1. (Вспомогательные процессоры начинают свою инициализацию только на шаге 8 фазы 1.)

Фаза 1 состоит из следующих этапов:

1. Функция `Phase1InitializationDiscard`, которая, как следует из ее названия, отбрасывает код, являющийся частью секции `INIT` образа ядра, чтобы сэкономить память.
2. Программный поток инициализации присваивает себе приоритет 31 — наивысший из возможных, — чтобы не допустить вытеснения.
3. Создается топологическое соотношение NUMA/group, в котором система пытается достичь оптимального соотношения между логическими процессорами и группами

процессоров, учитывая местоположение NUMA-узлов и расстояние между ними, если эти параметры не переопределены соответствующим BCD-параметром.

4. Функция `HalInitSystem` готовит систему к получению прерываний от устройств и включает режим прерываний.
5. Вызывается драйвер видеоадаптера, показывающий графическую заставку Windows, вместо которой по умолчанию выводится черный экран с индикатором хода загрузки. Если установлен параметр `quietboot`, этот этап пропускается.
6. Если задан параметр `sos`, ядро формирует различные строки, которые выводятся на экране загрузки через `Bootvid`. Сюда входит полная информация о версии, число поддерживаемых процессоров и объем поддерживаемой памяти.
7. Начинается инициализация диспетчера электропитания.
8. Инициализируется системное время (путем вызова функции `HalQueryRealTimeClock`), текущее значение которого сохраняется как время загрузки системы.
9. В многопроцессорных системах при помощи функций `KeStartAllProcessors` и `HalAllProcessorsStarted` инициализируются остальные процессоры. Количество процессоров, которые инициализируются и поддерживаются, зависит от комбинации числа физических процессоров, лицензионной информации об установленных товарных позициях операционной системы, параметров загрузки, таких как `numproc` и `opecpu`, и того, включен ли режим динамического разбиения на разделы (только для серверов). После инициализации всех доступных процессоров обновляется привязка к ним системного процесса.
10. Диспетчер объектов создает корневой каталог пространства имен (\), каталог `\ObjectTypes` и каталог отображения имен DOS-устройств (`\Global??`). Затем им создается символьская ссылка `\DosDevices` на папку отображения имен устройств подсистем Windows.
11. Вызывается исполнительная подсистема для создания своих типов объектов, включая семафор, мьютекс, событие и таймер.
12. Вызывается диспетчер ввода-вывода для создания собственного типа объектов, в том числе объектов устройства, драйвера, контроллера, адаптера и файла.
13. Библиотека отладки ядра завершает инициализацию режимов и параметров отладки, если отладчик еще не активирован.
14. Диспетчер транзакций также создает свои типы объектов, например перечисление, диспетчер ресурсов и диспетчер транзакций.
15. Ядро инициализирует структуры данных планировщика (диспетчера) и таблицу диспетчеризации системных служб.
16. Инициализируются структуры данных библиотеки отладки пользовательского режима (`Dbgk`).
17. Если включены верификатор драйверов и режим (в зависимости от параметров верификации) проверки пула, начинается отслеживание дескриптора объекта для системного процесса.

18. Монитор безопасности создает папку \Security в пространстве имен диспетчера объектов. При включенном аудите инициализируется также аудит структур данных.
19. Создается символическая ссылка \SystemRoot.
20. Вызывается диспетчер памяти для создания объекта \Device\PhysicalMemory и системные рабочие потоки диспетчера памяти (они рассматриваются в главе 10).
21. NLS-таблицы отображаются на системное пространство, что позволяет легко отобразить их на процессы пользовательского режима.
22. На системное адресное пространство отображается модуль Ntdll.dll.
23. Диспетчер кэша инициализирует структуры данных кэша файловой системы и создает его рабочие потоки.
24. Диспетчер конфигурирования создает в пространстве имен диспетчера объектов объект раздела \Registry реестра и открывает в памяти куст SYSTEM. Затем он копирует переданные приложением Winload исходные данные о дереве устройств в куст HARDWARE.
25. Инициализируется библиотека для вывода в процессе загрузки графики с высоким разрешением. Исключением являются случаи, когда эта функция отключена через BCD или система загружается без монитора.
26. Диспетчер ошибок инициализируется и в поисках сведений об ошибках сканирует реестр и базу данных INF-файлов (файлы установки драйверов, о которых речь идет в главе 8), содержащую ошибки различных драйверов.
27. Инициализируются служба superfetch и компонент prefetcher.
28. Инициализируется диспетчер хранения.
29. Инициализируется информация о временной зоне.
30. Инициализируются структуры данных драйвера глобальной файловой системы.
31. Вызывается процедура KdDebugger-Initialize1 в такой библиотеке, как Kdcom.dll, отвечающей за связь между ядром операционной системы и отладчиком.
32. PnP-диспетчер вызывает BIOS-расширение для работы с PnP-устройствами.
33. Подсистема усовершенствованного локального вызова процедур (Advanced Local Procedure Call, ALPC) инициализирует объекты типа «ALPC-порт» и «ALPC-порт ожидания». Более старые LPC-объекты задаются в виде псевдонимов.
34. Если система загружается с протоколированием загрузки (с установленным BCD-параметром bootlog), инициализируется журнал загрузки. В безопасном режиме загрузки на экран выводится тип безопасного режима.
35. Исполнительная подсистема вызывается для проведения следующей фазы инициализации, в процессе которой в ядре выбирается конфигурация функций, связанных с лицензированием, например проверяются параметры реестра, в которых хранятся лицензионные данные. Кроме того, при наличии данных постоянного хранения

из приложений загрузки (например, результатов диагностики памяти) соответствующие файлы журналов и информация записываются на диск или в реестр.

36. Создаются разделы MiniNT/WinPE реестра, если происходит загрузка соответствующего типа. В пространстве имен создается папка NLS-объектов, которая позднее будет использоваться с целью размещения различных объектов раздела для отображенных на память NLS-файлов.

37. Вызывается для повторной инициализации диспетчер электропитания. На этот раз он настраивает механизм поддержки запросов электропитания, ALPC-канал уведомлений о яркости и механизм поддержки обратных вызовов профилей.

38. Наступает время инициализации диспетчера ввода-вывода. Это сложный этап запуска системы, занимающий изрядную долю времени загрузки.

В первую очередь диспетчер ввода-вывода инициализирует различные внутренние структуры и создает объекты типа «драйвер» и «устройство». Затем вызываются PnP-диспетчер, диспетчер электропитания и HAL, которые приступают к различным стадиям перечисления и инициализации динамических устройств. (Этот сложный и связанный с системой ввода-вывода процесс подробно рассматривается в главе 8.) Далее инициализируется подсистема инструментария управления Windows (Windows Management Instrumentation, WMI), предоставляющая поддержку драйверам WMI-устройств. (Более подробно эта тема рассматривается в разделе «Windows Management Instrumentation» главы 4 части I.) Также инициализируется система трассировки событий для Windows (Event Tracing for Windows, ETW). Затем вызываются для выполнения инициализации драйверы, отвечающие за начальный этап загрузки, после чего загружаются и инициализируются драйверы, отвечающие за запуск системы. (Об обработке в реестре информации, управляющей загрузкой драйверов, рассказывается в главе 8.) Напоследок в виде символьических ссылок в пространстве имен диспетчера объектов создаются имена устройств подсистем Windows.

39. Диспетчер транзакций настраивает WPP (Windows software trace preprocessor — препроцессор трассировки Windows-программ) и ETW, а также с помощью WMI проводит инициализацию этих систем. (Об ETW и WMI рассказывается в главе 4 части I.)

40. На этом этапе драйверы, управляющие загрузкой и запуском системы, уже загружены, и диспетчер ошибок загружает базу данных INF-файлов с ошибками драйверов, начиная ее анализировать.

41. Если компьютер загружается в безопасном режиме, это фиксируется в реестре.

42. Если эта возможность явным образом не отключена через реестр, включается подкачка кода в режиме ядра (в файле Ntoskrnl и драйверах).

43. Диспетчер конфигураций убеждается в том, что все процессоры в SMP-системе идентичны в плане поддерживаемых ими функций; в противном случае произойдет системный сбой.

44. В 32-разрядных системах инициализируется поддержка виртуальной DOS-машины (Virtual DOS Machine, VDM), для чего требуется определить, поддерживает ли процессор расширения для виртуальных машин (Virtual Machine Extensions, VME).

45. Вызывается диспетчер процессов, чтобы указать ограничение полосы пропускания для заданий, инициализировать статическую среду для защищенных процессов и найти различные определенные системой точки входа в системную библиотеку в пользовательском режиме (`Ntdll.dll`).
46. Для завершения инициализации вызывается диспетчер электропитания.
47. Инициализируется оставшаяся лицензионная информация, в том числе кэшируются сохраненные в реестре текущие параметры политик.
48. Вызывается монитор безопасности для создания взаимодействующего с LSASS программного потока сервера команд. (Способы обеспечения безопасности в Windows рассматриваются в разделе «Системные компоненты безопасности» главы 6 части I.)
49. Создается процесс `Smss` диспетчера сеансов (с ним мы познакомились в главе 2 части I). Он отвечает за создание среды пользовательского режима, обеспечивающей визуальный интерфейс Windows. Этапы инициализации этого процесса рассматриваются в следующем разделе.
50. Запрашиваются значения энтропии загрузки TPM-модуля. Этот опрос во время загрузки может выполняться только один раз, и, как правило, к этому моменту они уже затребованы системным драйвером TPM-модуля. Но если по какой-то причине этот драйвер не работает (к примеру, его отключил пользователь), незатребованные значения все еще доступны. Чтобы избежать подобной ситуации, ядро снова их запрашивает. В нормальном сценарии этот запрос остается незавершенным.
51. Освобождается память, использовавшаяся блоком параметров загрузчика и компонентами, на которые он ссылается.

Перед завершением инициализации компонентов исполнительной системы и ядра программный поток инициализации фазы 1 в течение пяти секунд ждет освобождения дескриптора процесса диспетчера сеансов. Если процесс завершается до истечения этого времени, происходит системный сбой с кодом `SESSION5_INITIALIZATION_FAILED`.

По истечении пяти секунд запуск диспетчера сеансов считается успешным, и функция инициализации фазы 1 вызывает функцию программного потока обнуления страниц диспетчера памяти (этот процедура объясняется в главе 10). Соответственно, системный поток на время существования системы становится программным потоком обнуления страниц.

Smss, Csrss и Wininit

От любого другого процесса пользовательского режима `Smss` имеет два отличия. Во-первых, Windows считает его доверенной частью операционной системы. Во-вторых, `Smss` является *встроенным* (native) приложением. В качестве доверенного компонента операционной системы `Smss` может выполнять операции, доступные лишь немногим процессам, например создавать маркеры безопасности. Будучи встроенным приложением, `Smss` вместо API-интерфейсов Windows использует только API-интерфейсы исполнительной подсистемы, известные под общим названием Windows native API. При этом API-интерфейсы из Win32 `Smss` не использует, так как в момент загрузки

этого процесса данная подсистема Windows еще не исполняется. По сути, одной из первых задач `Smss` является запуск этой подсистемы Windows.

Затем `Smss` вызывает диспетчер конфигурирования, который завершает инициализацию реестра, заполняя все его разделы. Этот диспетчер запрограммирован таким образом, что ему известно местоположение на диске всех кустов реестра, кроме кустов, содержащих пользовательские параметры. Путь ко всем загружаемым им кустам записывается в раздел `HKLM\SYSTEM\CurrentControlSet\Control\hivelist`.

Основной поток процесса `Smss` выполняет следующие операции по инициализации:

1. Помечает процесс `Smss` как критический процесс, а основной программный поток — как критический поток. Как показано в главе 5 части I, в этом случае при неожиданном завершении `Smss` происходит сбой ядра. Кроме того, `Smss` включает автоматический режим обновления родственности для поддержки динамического добавления процессоров. (Дополнительную информацию по этой теме можно найти в главе 5 части I.)
2. Создает защищенные префиксы для почтовых слотов и именованных каналов, формируя привилегированные пути для администраторов и учетные записи служб для общения по этим путям. (Подробно об этом написано в главе 7 части I.)
3. Вызывает функцию `SmpInit`, настраивающую максимальный уровень параллелизма для `Smss`, то есть максимальное количество одновременных сеансов, которые будут созданы запуском копий `Smss` в других сеансах. Это значение равно как минимум четырем, а как максимум — числу активных процессоров.
4. Затем `SmpInit` создает объект ALPC-порта (`\SmApiPort`) для получения клиентских запросов (например, загрузить новую подсистему или создать сеанс).
5. `SmpInit` вызывает функцию `SmpLoadDataFromRegistry`, которая задает для системы переменные окружения, применяемые по умолчанию, а при загрузке в безопасном режиме — переменную `SAFEBOOT`.
6. Функция `SmpLoadDataFromRegistry` вызывает функцию `SmpInitializeDosDevices`, чтобы задать символические ссылки на имена DOS-устройств (например, `COM1` и `LPT1`).
7. Функция `SmpLoadDataFromRegistry` создает папку `\Sessions` в пространстве имен диспетчера объектов (для нескольких сеансов).
8. Функция `SmpLoadDataFromRegistry` запускает программы, указанные в разделе `HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\BootExecute` параметром `SmpExecuteCommand`. Как правило, тут содержится одна команда запуска `Autochk` (версия `Chkdsk`, работающая на этапе загрузки).
9. Функция `SmpLoadDataFromRegistry` вызывает функцию `SmpProcessFileRenames` для отложенного переименования и удаления файлов, указанных в разделах `HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\PendingFileRenameOperations` и `HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\PendingFileRenameOperations2`.
10. Функция `SmpLoadDataFromRegistry` вызывает функцию `SmpCreatePagingFiles` для создания дополнительных файлов подкачки. Конфигурация файла подкачки хра-

нится в разделе `HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\PagingFiles`.

11. Функция `SmpLoadDataFromRegistry` инициализирует реестр, вызывая встроенную функцию `NtInitializeRegistry`. Остальную часть реестра достраивает диспетчер конфигурации, загружая кусты для разделов `HKLM\SAM`, `HKLM\SECURITY` и `HKLM\SOFTWARE`. Несмотря на то что в переменной `HKLM\SYSTEM\CurrentControlSet\Control\hivelist` указано местоположение файлов кустов на диске, диспетчер конфигурации ищет их в папке `\Windows\System32\Config`.
12. Функция `SmpLoadDataFromRegistry` вызывает функцию `SmpCreateDynamicEnvironmentVariables` для добавления системных переменных окружения, заданных в разделе `HKLM\SYSTEM\CurrentControlSet\Session Manager\Environment`, а также переменных окружения, связанных с процессором, таких как `NUMBER_PROCESSORS`, `PROCESSOR_ARCHITECTURE` и `PROCESSOR_LEVEL`.
13. Функция `SmpLoadDataFromRegistry` запускает все программы, определенные в разделе `HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\SetupExecute` параметром `SmpExecuteCommand`. Обычно этот параметр задается только в случае, когда Windows загружается как часть второй стадии установки, и по умолчанию имеет значение `Setupcl.exe`.
14. Функция `SmpLoadDataFromRegistry` вызывает функцию `SmpConfigureSharedSessionData` для инициализации списка подсистем, которые будут запускать в каждом сеансе (как немедленно, так и с задержкой), а также команды инициализации сеанса 0 (который по умолчанию запускает процесс `Wininit.exe`). Команду инициализации можно переопределить, создав строковый параметр `S0InitialCommand` в разделе `HKLM\SYSTEM\CurrentControlSet\Control\Session Manager` и присвоив ему путь к другой программе.
15. Функция `SmpLoadDataFromRegistry` вызывает функцию `SmpInitializeKnownDlls` для открытия известных DLL-библиотек и создает для них объекты разделов в папке `\KnownDlls` пространства имен диспетчера объектов. Список DLL-библиотек, которые считаются известными, хранится в разделе `HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\KnownDLLs`, а путь к содержащей их папке хранится в параметре `DllDirectory` этого раздела. В 64-разрядных системах 32-разрядные DLL-библиотеки, используемые как часть подсистемы Wow64, хранятся в параметре `DllDirectory32`.
16. Наконец, функция `SmpLoadDataFromRegistry` вызывает функцию `SmpTranslateSystemPartitionInformation` для преобразования значения `SystemPartition` (из раздела `HKLM\SYSTEM\Setup`, которое имеет формат пути встроенного в NT диспетчера объектов) в букву тома, хранящуюся в параметре `BootDir`. Этот раздел реестра наравне с другими компонентами позволяет приложению Windows Update определить, какой том является системным.
17. После этого функция `SmpLoadDataFromRegistry` передает управление функции `SmpInit`, которая возвращается к основной точке входа программного потока. Затем `Sms` создает набор заданных начальных сеансов (обычно это единствен-

ный сеанс 0, но можно присвоить параметру `NumberOfInitialSessions` в ранее упоминавшемся разделе реестра для `Smss` другое значение), вызывая функцию `SmpCreateInitialSession`, которая для каждого пользовательского сеанса запустит процесс `Smss`. Основной задачей этой функции является вызов функции `SmpStartCsr` для запуска в каждом сеансе процесса `Csrss`.

18. Как часть инициализации `Csrss` эта функция загружает фрагмент подсистемы Windows (`Win32k.sys`) в режиме ядра. Код инициализации в `Win32k.sys` использует видеодрайвер для переключения экрана в определенное по умолчанию в профиле разрешение. То есть именно в этот момент видеоадаптер переключается из VGA-режима, используемого загрузочным видеодрайвером, в выбранное для данной системы разрешение.
19. В то же время каждый порожденный программный поток процесса `Smss` в своем пользовательском сеансе запускает процессы других подсистем, например `Psxss`, если включена подсистема для Unix-приложений. (Подробно процессы подсистем рассматриваются в главе 3 части I.)
20. Первый процесс `Smss` из сеанса 0 выполняет команду инициализации сеанса 0 (описанную в шаге 14), по умолчанию загружая процесс инициализации Windows (`Wininit`). Остальные экземпляры `Smss` начинают процесс диспетчера интерактивного входа в систему (`Winlogon`), который в отличие от `Wininit` является встроенным. Далее мы рассмотрим начало работы процессов `Wininit` и `Winlogon`.

ОТЛОЖЕННЫЕ ДЕЙСТВИЯ ПО ПЕРЕИМЕНОВАНИЮ ФАЙЛОВ

Тот факт, что исполняемые образы и DLL-библиотеки при использовании отображаются на память, делает невозможным обновление базовых системных файлов после окончания загрузки Windows, если не использовать технологию оперативного исправления (hotpatching). API-интерфейс `MoveFileEx` позволяет указать, что перемещение файла следует отложить до следующей загрузки. Пакеты обновлений и критические исправления, призванные обновлять уже используемые файлы, отображенные на память, устанавливают заменяющие файлы во временные каталоги и применяют API-интерфейс `MoveFileEx` указанным образом. При этом `MoveFileEx` просто записывает команды в параметры `PendingFileRenameOperations` и `PendingFileRenameOperations2` в разделе `HKLM\SYSTEM\CurrentControlSet\Control\Session Manager` реестра. Эти параметры принадлежат к типу `MULTI_SZ`, и каждая операция задается парами имен файлов: первое имя — источник, второе — приемник. При удалении вместо приемника указывается пустая строка. Для просмотра зарегистрированных отложенных команд переименования и удаления служит программа `Pendmoves`, разработанная в Windows Sysinternals (<http://www.microsoft.com/technet/sysinternals>).

После выполнения указанных этапов инициализации основной программный поток процесса `Smss` бесконечно ожидает дескриптор процесса `Winlogon`, в то время как остальные программные ALPC-потоки ждут сообщений для создания новых сеансов или подсистем. Так как процессы `Wininit` и `Csrss` помечены как критические, их неожиданное завершение приводит к краху системы. При неожиданном завершении процесса `Winlogon` происходит выход из связанного с ним сеанса.

Затем процесс `Wininit` продолжает инициализацию, выполняя такие операции, как создание начального оконного терминала и объектов рабочего стола. Также он

настраивает перехватчик окон сеанса 0, при помощи которого служба обнаружения интерактивных служб (`UI0Detect.exe`) обеспечивает обратную совместимость с интерактивными службами. (Службы подробно рассматриваются в главе 4 части I.) После этого `Wininit` создает SCM-процесс (`%SystemRoot%\System32\Services.exe`), который загружает все службы и драйверы устройств, помеченные как предназначенные для автоматического запуска, а также запускает LSASS-процесс (`%SystemRoot%\System32\Lsass.exe`). Напоследок загружается диспетчер локальных сеансов (`%SystemRoot%\System32\Lsm.exe`). В сеансе 1 и следующих вместо этого запускается процесс `Winlogon`, который загружает зарегистрированные провайдеры учетных данных (по умолчанию провайдер учетных данных Microsoft поддерживает вход в систему на основе пароля и на основе смарт-карты) в дочерний процесс `LogonUI` (`%SystemRoot%\System32\Logonui.exe`), отвечающий за вывод на экран интерфейса входа в систему. (Более подробно последовательность запуска процессов `Wininit`, `Winlogon` и LSASS описывается в разделе «Инициализация `Winlogon`» главы 6 части I.)

После того как SCM-процесс инициализирует автоматически запускаемые службы и драйверы, а пользователь успешно войдет в систему через консоль, загрузка считается завершенной. Реестр последней удачной конфигурации (указанный в разделе `HKLM\SYSTEM\Select\LastKnownGood`) обновляется для приведения в соответствие с параметром `\CurrentControlSet`.

ПРИМЕЧАНИЕ

Так как на неинтерактивных серверах не бывает интерактивного входа, не обновляется раздел `LastKnownGood`, в котором содержится набор параметров, обеспечивающих успешную загрузку. Определение успешной загрузки можно переопределить. Для этого параметру `HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\ReportBootOk` присваивается значение 0 и пишется программа, проверяющая успешность загрузки, которая при положительном результате проверки вызывает API-интерфейс `NotifyBootConfigStatus`, а путь к этой программе указывается в параметре `HKLM\SYSTEM\CurrentControlSet\Control\BootVerificationProgram`.

После запуска SCM `Winlogon` ждет уведомления об интерактивном входе от провайдера учетных данных. Получив такое уведомление и проверив вход в систему (подробно этот процесс рассматривается в разделе «Этапы входа пользователя в систему» главы 6 части I), `Winlogon` загружает куст реестра из профиля авторизующегося пользователя и отображает его на `HKCU`. Затем он настраивает переменные окружения этого пользователя, хранящиеся в разделе `HKCU\Environment`, и направляет уведомление о входе компонентам, зарегистрированным в разделе `HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\Notify`.

Далее `Winlogon` вызывает оболочку, запуская исполняемый файл или файлы, указанные в параметре `HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\WinLogon\Userinit` (если исполняемых файлов несколько, их имена перечисляются через запятую). По умолчанию этот параметр указывает на `\Windows\System32\Userinit.exe`. Файл `Userinit.exe` выполняет следующие действия:

1. Обрабатывает пользовательские сценарии, указанные в разделе `HKCU\Software\Policies\Microsoft\Windows\System\Scripts`, и машинные сценарии входа, заданные в разделе `HKLM\SOFTWARE\Policies\Microsoft\Windows\System\Scripts`. (Машинные

сценарии запускаются после пользовательских, а значит, могут переопределять параметры последних.)

2. Если групповая политика задает для профиля пользователя квоту, Userinit.exe запускает %SystemRoot%\System32\Proquota.exe.
3. Запускает оболочку или оболочки, указанные через запятую в разделе HKCU\Software\Microsoft\Windows NT\CurrentVersion\Winlogon\Shell. При отсутствии данного параметра Userinit.exe запускает оболочку или оболочки, указанные в разделе HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\Shell. По умолчанию это Explorer.exe.

Затем Winlogon уведомляет зарегистрированные сетевые провайдеры о входе пользователя. Сетевой провайдер от Microsoft, маршрутизатор сетевого доступа (%SystemRoot%\System32\Mpr.dll), восстанавливает заданные пользователем постоянные подключения к сетевому диску и принтерам. Эти варианты отображения хранятся в параметрах HKCU\Network и HKCU\Printers соответственно. На рис. 13.4 показано дерево процессов в приложении Process Monitor после входа пользователя (при протоколирования загрузки). Обратите внимание на недоступные процессы Smss (это означает их завершение к моменту просмотра). Они относятся к порожденным копиям, которые инициализировали каждый из сеансов.

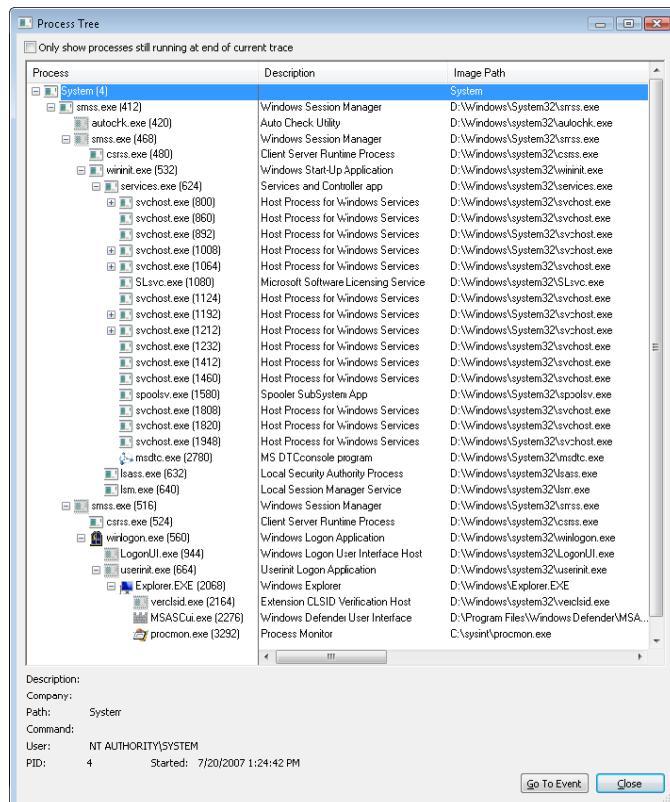


Рис. 13.4. Дерево процессов при входе

ReadyBoot

Если в системе установлено менее 700 Мбайт оперативной памяти, Windows во время загрузки использует стандартную логическую предварительную выборку (описанную в главе 10), в случае же большего количества оперативной памяти для оптимизации процесса загрузки используется встроенный в оперативную память кэш. Размер этого кэша зависит от общего объема доступной памяти; он достаточно велик, чтобы обеспечить эффективное кэширование, но оставляет при этом достаточно свободной памяти для нормального выполнения процедуры загрузки системы.

После каждой загрузки системы служба ReadyBoost (она рассматривается в главе 10) в моменты простоя процессора планирует кэширование для следующей загрузки системы. Она анализирует информацию об обращениях к файлам за пять предыдущих загрузок, определяя, к каким файлам производились обращения и где эти файлы расположены на диске. Обработанная информация об обращениях сохраняется в папке %SystemRoot%\Prefetch\Readyboot в виде файлов с расширением .fx, а план кэширования сохраняется в разделе HKLM\SYSTEM\CurrentControlSet\Services\Rdyboost\Parameters реестра в виде параметров REG_BINARY с именами, соответствующими именам внутренних дисков.

Кэширование реализуется тем же самым драйвером устройства, что и в службе ReadyBoost (Ecache.sys), но управление заполнением кэша во время загрузки осуществляется в соответствии с планом, ранее сохраненным в реестре. Хотя кэш загрузки сжимается таким же образом, как и кэш ReadyBoost, существует одно отличие в управлении этими двумя кэшами. Дело в том, что в режиме ReadyBoot кэш не шифруется. Служба ReadyBoost удаляет кэш через 50 секунд после начала работы, или (при других требованиях к памяти) статистика кэша записывается в раздел HKLM\SYSTEM\CurrentControlSet\Parameters\Ecache\ReadyBootStats реестра, как показано на рис. 13.5.

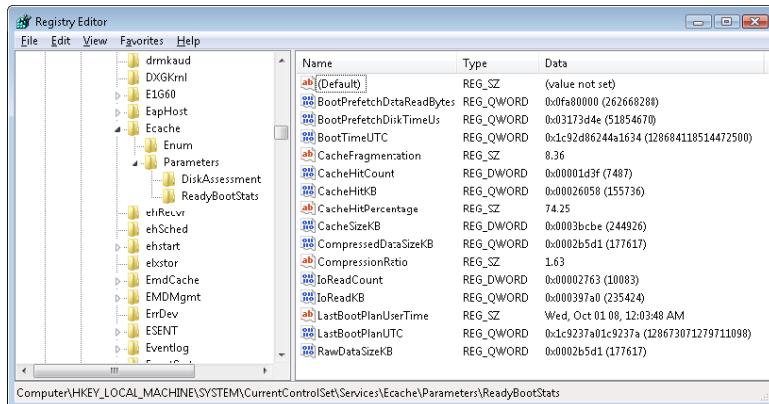


Рис. 13.5. Статистика службы ReadyBoot

Автоматически запускаемые образы

Помимо параметров Userinit и Shell в разделе Winlogon реестра находится много других разделов и папок, проверяемых и обрабатываемых системными компонентами

для автоматического запуска процессов при загрузке и входе. Служебная программа Msconfig (%SystemRoot%\System32\Msconfig.exe) выводит на экран образы, сконфигурированные в нескольких местах. Но разработанный в Sysinternals инструмент Autoruns анализирует больше мест, чем Msconfig, и показывает больше сведений о настроенных на автоматический запуск образах (рис. 13.6). По умолчанию Autoruns показывает только те места, в которых задан автоматический запуск хотя бы одного образа, но команда Include Empty Locations меню Options заставляет программу показать все проверяемые места. В меню Options также присутствует команда, позволяющая скрыть элементы от Microsoft, но ее всегда нужно применять в паре с командой Verify Image Signatures. В противном случае заодно можно скрыть вредоносные программы, содержащие фальшивые сведения о своем производителе.

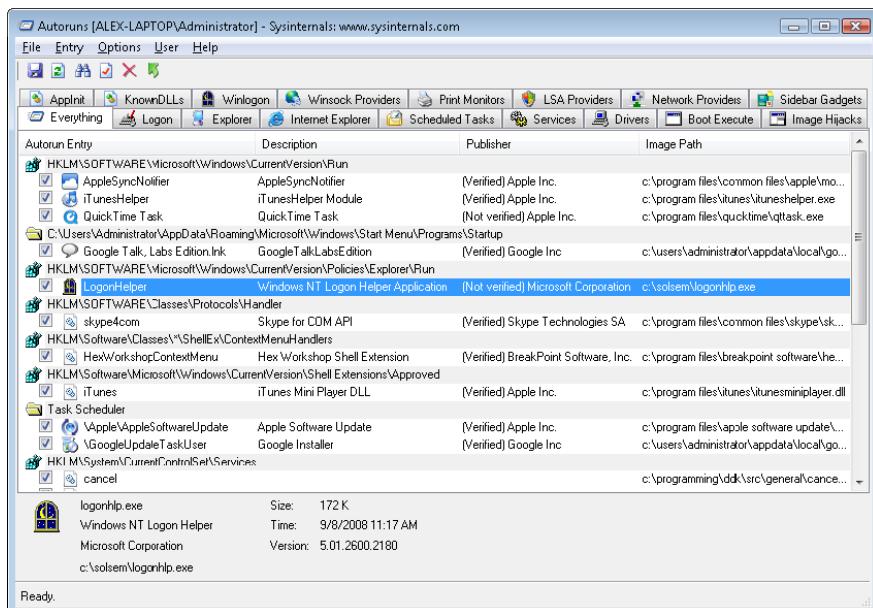


Рис. 13.6. Программа Autoruns от Sysinternals

ЭКСПЕРИМЕНТ: ПРОГРАММА AUTORUNS

Многие пользователи даже не представляют, сколько программ выполняется во время их входа в систему. Изготовители оборудования часто настраивают свои системы при помощи дополнительных служебных программ, которые выполняются в фоновом режиме, используя параметры реестра или папки файловой системы, предназначенные для автоматического запуска, и поэтому обычно их не видно. Чтобы посмотреть, какие программы автоматически запускаются на вашем компьютере, воспользуйтесь программой Autoruns от Sysinternals. Сравните полученный в Autoruns список с тем, что показывает Msconfig, и обратите внимание на различия. Затем попробуйте понять назначение каждой из программ.

Анализ проблем при загрузке и запуске системы

В этом разделе представлены подходы к решению проблем, которые иногда возникают при запуске Windows из-за повреждения жесткого диска, повреждения файлов, отсутствия файлов и ошибок в драйверах сторонних производителей. Сначала мы опишем три подхода к восстановлению Windows в случае проблем с загрузкой: возвращение к последней удачной конфигурации, загрузка в безопасном режиме и использование среды восстановления Windows (Windows Recovery Environment, WinRE). Затем мы поговорим о наиболее распространенных проблемах, возникающих при загрузке, их причинах и способах их устранения. В качестве решений предлагаются возвращение к последней удачной конфигурации, безопасный режим, WinRE и другие средства, доступные в Windows.

Последняя удачная конфигурация

Последняя удачная конфигурация (Last Known Good, LKG) представляет собой механизм возвращения рухнувшей в процессе загрузки системы в загружаемое состояние. Так как параметры системной конфигурации хранятся в разделе `HKLM\SYSTEM\CurrentControlSet\Control`, а параметры драйверов и служб — в разделе `HKLM\SYSTEM\CurrentControlSet\Services`, изменения этих фрагментов реестра могут сделать систему незагружаемой. Например, если вы установили драйвер устройства с ошибкой, которая становится причиной системного сбоя, в момент загрузки можно нажать клавишу F8 и выбрать в появившемся меню последнюю удачную конфигурацию. Контрольный набор, который использовался при загрузке, система отмечает как неудачный, устанавливая в разделе `HKLM\SYSTEM\Select` параметр `Failed` и заменяя значение параметра `HKLM\SYSTEM\Select\Current` значением `HKLM\SYSTEM\Select\LastKnownGood`. Обновляется также символическая ссылка `HKLM\SYSTEM\CurrentControlSet`, которая начинает указывать на контрольный набор `LastKnownGood`. Так как в разделе `Services` у контрольного набора `LastKnownGood` отсутствует подраздел для нового драйвера, система без проблем загружается.

Безопасный режим

Пожалуй, чаще всего загрузку Windows осуществить не удается из-за драйверов устройств, приводящих к системному сбою в процессе загрузки. Так как со временем программино-аппаратная конфигурация системы меняется, латентные ошибки драйверов могут проявиться в любой момент. Администратор может решить эту проблему путем загрузки в *безопасном режиме* (safe mode). Этим термином называется конфигурация загрузки, состоящая из минимального набора драйверов устройств и служб. Используя только те драйверы и службы, без которых загрузка невозможна, Windows избегает других драйверов, которые могут стать причиной системного сбоя.

Нажатие клавиши F8 в начале загрузки открывает специальное меню с вариантами загрузки в безопасном режиме. Обычно предоставляется три варианта: *Safe Mode*

(Безопасный режим), Safe Mode With Networking (Безопасный режим с загрузкой сетевых драйверов) и Safe Mode With Command Prompt (Безопасный режим с поддержкой командной строки). Стандартный безопасный режим включает в себя минимум необходимых для успешной загрузки драйверов и служб. В безопасном режиме с сетевой поддержкой дополнительно загружаются сетевые драйверы и службы. Наконец, безопасный режим с поддержкой командной строки отличается от стандартного тем, что вместо обычной Windows-оболочки, возникающей после включения графического режима, операционная система запускает оболочку командной строки (Cmd.exe).

В Windows существует и четвертый безопасный режим — Directory Services Restore (Восстановление службы каталогов). Он применяется для загрузки системы с отключенной службой каталогов доменного контроллера Active Directory и без открытия ее базы данных. Это позволяет администратору исправить поврежденную базу данных или восстановить ее из резервной копии. В данном режиме загружаются все драйверы и службы, кроме Active Directory. Он предназначен для устранения неполадок в случае, когда войти в систему невозможно из-за повреждения базы данных Active Directory.

Загрузка драйверов в безопасном режиме

Откуда Windows знает, какие драйверы устройств и службы требуются для загрузки в стандартном безопасном режиме, а какие — в безопасном режиме с сетевой поддержкой? Ответ следует искать в разделе HKLM\SYSTEM\CurrentControlSet\Control\SafeBoot реестра. Этот раздел содержит подразделы Minimal и Network, каждый из которых, в свою очередь, содержит подразделы с именами драйверов, служб или их групп. Например, подраздел vga.sys определяет драйвер видеоадаптера VGA, входящий в стандартную конфигурацию. Этот драйвер поддерживает стандартный набор графических сервисов для любого PC-совместимого видеоадаптера. Он используется в безопасном режиме вместо драйверов, позволяющих задействовать преимущества более совершенных видеоадаптеров, но могущих помешать успешной загрузке системы. Каждому подразделу в разделе SafeBoot соответствует параметр, предлагаемый по умолчанию и описывающий, что именно идентифицирует подраздел. Например, по умолчанию в подразделе vga.sys используется параметр Driver.

Для подраздела Boot файловой системы по умолчанию используется параметр Driver Group. Создавая сценарий установки драйвера устройства (файл .inf), разработчик может указать, что этот драйвер относится к какой-либо группе. Определенные в системе группы перечислены в параметре List раздела HKLM\SYSTEM\CurrentControlSet\Control\ServiceGroupOrder. Разработчик вносит драйвер в определенную группу, чтобы показать Windows, на каком этапе загрузке следует его запускать. Раздел ServiceGroupOrder в основном предназначен для определения порядка загрузки драйверов. Некоторые типы драйверов загружаются до или после других типов. Параметр Group в разделе реестра со сведениями о конфигурации драйвера ассоциирует его с определенной группой.

Разделы со сведениями о конфигурации драйверов и служб находятся внутри раздела HKLM\SYSTEM\CurrentControlSet\Services. Взглянув на его содержимое, вы обнаружите раздел VgaSave для драйвера видеоадаптера VGA, который входит в группу Video Save. Все драйверы файловой системы, необходимые Windows для обращения к системному диску, загружаются автоматически, если поместить их в группу Boot file

system. Другие драйверы файловой системы входят в группу File system, которая также включена в конфигурации Save Mode и Save Mode with Networking.

При загрузке в безопасном режиме загрузчик (Winload) передает ядру (Ntoskrnl.exe) параметр командной строки вместе с другими параметрами, указанными в BCD для текущей загрузки. Если вы загружаетесь в безопасном режиме, Winload задает BCD-параметр safeboot, значение которого соответствует типу выбранного вами режима. Для стандартного безопасного режима это значение minimal, а для режима с загрузкой сетевых драйверов – значение network. Если нужно загрузиться в безопасном режиме с поддержкой командной строки, Winload добавляет к параметру minimal параметр safebootalternateshell. При загрузке в режиме восстановления службы каталогов применяется параметр dsrepair.

Ядро Windows на ранней стадии загрузки и во время выполнения функции `InitSafeBoot` сканирует параметры загрузки в поисках спецификаторов безопасного режима и в зависимости от результатов поиска присваивает внутренней переменной `InitSafeBootMode` значение, которое также записывается в раздел `HKLM\SYSTEM\CurrentControlSet\Control\SafeBoot\Option\OptionValue`. Именно это позволяет компонентам пользовательского режима, таким как SCM, определять режим загрузки системы. Кроме того, при загрузке в безопасном режиме с поддержкой командной строки ядро присваивает параметру `HKLM\SYSTEM\CurrentControlSet\Control\SafeBoot\Option\UseAlternateShell` значение 1. Параметры, которые ядру передает Winload, записываются в параметр `HKLM\SYSTEM\CurrentControlSet\Control\SystemStartOptions`.

При загрузке драйверов устройств, указанных в разделе `HKLM\SYSTEM\CurrentControlSet\Services`, диспетчер ввода-вывода выполняет функцию `IopLoadDriver`. А когда PnP-диспетчер обнаруживает новое устройство и хочет динамически загрузить для него драйвер, он вызывает функцию `PipCallDriverAddDevice`. Обе эти функции перед загрузкой драйвера обращаются к функции `IopSafebootDriverLoad`, которая проверяет значение переменной `InitSafeBootMode` и определяет, можно ли загрузить данный драйвер. Например, при загрузке в стандартном безопасном режиме функция `IopSafebootDriverLoad` ищет в подразделе `Minimal` группу этого драйвера, если таковая существует. Обнаружив ее, `IopSafebootDriverLoad` уведомляет вызывающую функцию о том, что драйвер можно загрузить. Если же группы там не оказывается, `IopSafebootDriverLoad` ищет в том же подразделе имя драйвера. Если оно есть в списке, драйвер может быть загружен. В противном случае его загрузка запрещается. При загрузке системы в безопасном режиме с сетевой поддержкой функция `IopSafebootDriverLoad` выполняет поиск в подразделе `Network`. Если осуществляется загрузка в нормальном режиме, `IopSafebootDriverLoad` разрешает загрузку всех драйверов.

ПРИМЕЧАНИЕ

Существуют драйверы, которые не загружаются в безопасном режиме. В отличие от ядра Winload загружает все драйверы, у которых в соответствующих разделах реестра значение параметра Start равно 0. Но Winload не проверяет раздел SafeBoot, так как предполагает, что для успешной загрузки системы требуются все драйверы, у которых значение параметра Start равно 0. Соответственно, Winload загружает все загрузочные драйверы, которые впоследствии запускаются Ntoskrnl.

Программы с поддержкой безопасного режима

Проводя инициализацию при загрузке, такой компонент пользовательского режима, как диспетчер управления службами (SCM), реализуемый Services.exe, проверяет параметр HKLM\SYSTEM\CurrentControlSet\Control\SafeBoot\Option\OptionValue, чтобы определить, происходит ли загрузка в безопасном режиме. В случае положительного результата проверки SCM воспроизводит действия функции `IoSafebootDriverLoad`. SCM обрабатывает все службы, перечисленные в разделе HKLM\SYSTEM\CurrentControlSet\Services, но выбирает из них лишь отмеченные в соответствующем подразделе реестра для загрузки в безопасном режиме. Подробно процесс инициализации SCM описывался в разделе «Службы» главы 4 части I.

Инициализирующий среду для пользователя при входе в систему компонент Userinit (%SystemRoot%\System32\Userinit.exe) является еще одним компонентом пользовательского режима, которому нужно знать, загружается ли система в безопасном режиме. Он проверяет значение параметра HKLM\SYSTEM\CurrentControlSet\Control\SafeBoot\Option\UseAlternateShell. Если оно установлено, Userinit запускает в качестве пользовательской оболочки не Explorer.exe, а программу, указанную в параметре HKLM\SYSTEM\CurrentControlSet\Control\SafeBoot\AlternateShell. В процессе установки Windows присваивает параметру AlternateShell значение Cmd.exe, после чего командная строка Windows становится оболочкой, предлагаемой по умолчанию для безопасного режима с командной строкой. Впрочем, даже в этом случае можно запустить Проводник, воспользовавшись командой Explorer.exe. Аналогичным образом из командной строки можно запустить любую другую программу с графическим интерфейсом.

Как приложения узнают о загрузке системы в безопасном режиме? Для этого они вызывают функцию `GetSystemMetrics(SM_CLEANBOOT)`. Выполняющие некоторые действия по загрузке системы в безопасном режиме сценарии проверяют наличие переменной окружения `SAFEBOOT_OPTION`, так как система определяет ее только при загрузке в безопасном режиме.

Протоколирование загрузки в безопасном режиме

При загрузке системы в безопасном режиме Winload передает ядру вместе с параметрами, требуемыми данным режимом, строку, заданную параметром `bootlog`. В процессе инициализации ядро проверяет наличие параметра `bootlog` вне зависимости от того, задан ли безопасный режим. Если ядро обнаруживает строку журнала загрузки, оно начинает протоколировать свои действия при загрузке каждого драйвера. К примеру, если функция `IoSafebootDriverLoad` запрещает диспетчеру ввода-вывода загрузку какого-либо драйвера, диспетчер вызывает функцию `IoBootLog` для фиксации этого факта. Аналогично, после того как функция `IoLoadDriver` успешно загрузит драйвер, входящий в конфигурацию безопасного режима, она вызывает функцию `IoBootLog` для внесения соответствующей записи в журнал. Изучив файлы протокола загрузки, можно выяснить, какие драйверы являются частью данной конфигурации.

Так как ядро хочет избежать редактирования диска до запуска Chkdsk, который происходит на более позднем этапе загрузки, функция `IoBootLog` не может просто сбрасывать записи в файл журнала. Вместо этого она заносит их в параметр реестра HKLM\SYSTEM\CurrentControlSet\BootLog. Первый загружаемый компонент пользовательского режима, диспетчер сеансов (%SystemRoot%\System32\Smss.exe), для проверки целост-

ности системного диска запускает Chkdsk, а затем завершает инициализацию реестра, вызывая функцию `NtInitializeRegistry`. Это становится для ядра сигналом, что можно безопасно открыть файл журнала, вызвав функцию `IopCopyBootLogRegistryToFile`. Она создает в системной папке Windows (%SystemRoot%) файл `Ntbtlog.txt` и копирует туда содержимое параметра `BootLog`. Также функция `IopCopyBootLogRegistryToFile` устанавливает флаг, сигнализирующий функции `IopBootLog` о том, что появилась возможность записи непосредственно в файл журнала. Вот пример протокола загрузки:

```
Microsoft (R) Windows (R) Version 6.1 (Build 7601)
10 4 2012 09:04:53.375
Loaded driver \SystemRoot\system32\ntkrnlpa.exe
Loaded driver \SystemRoot\system32\hal.dll
Loaded driver \SystemRoot\system32\kdcom.dll
Loaded driver \SystemRoot\system32\mcupdate_GenuineIntel.dll
Loaded driver \SystemRoot\system32\PSHED.dll
Loaded driver \SystemRoot\system32\BOOTVID.dll
Loaded driver \SystemRoot\system32\CLFS.SYS
Loaded driver \SystemRoot\system32\CI.dll
Loaded driver \SystemRoot\system32\drivers\Wdf01000.sys
Loaded driver \SystemRoot\system32\drivers\WDFLDR.SYS
Loaded driver \SystemRoot\system32\drivers\acpi.sys
Loaded driver \SystemRoot\system32\drivers\WMILIB.SYS
Loaded driver \SystemRoot\system32\drivers\msisadrv.sys
Loaded driver \SystemRoot\system32\drivers\pci.sys
Loaded driver \SystemRoot\system32\drivers\volmgr.sys
Loaded driver \SystemRoot\system32\DRIVERS\compbatt.sys
Loaded driver \SystemRoot\system32\DRIVERS\BATT.CSYS
Loaded driver \SystemRoot\System32\drivers\mountmgr.sys
Loaded driver \SystemRoot\system32\drivers\intelide.sys
Loaded driver \SystemRoot\system32\drivers\PCIINDEX.SYS
Loaded driver \SystemRoot\system32\DRIVERS\pciide.sys
Loaded driver \SystemRoot\System32\drivers\volmgrx.sys
Loaded driver \SystemRoot\system32\drivers\atapi.sys
Loaded driver \SystemRoot\system32\drivers\ataport.SYS
Loaded driver \SystemRoot\system32\drivers\fltmgr.sys
Loaded driver \SystemRoot\system32\drivers\fileinfo.sys
...
Did not load driver @battery.inf,%acpi\acpi0003.devicedesc%;Microsoft AC Adapter
Did not load driver @battery.inf,%acpi\pnp0c0a.devicedesc%;Microsoft ACPI-Compliant
Control Method Battery
Did not load driver @oem46.inf,%nvidia_g71.dev_0297.1%;NVIDIA GeForce Go 7950 GTX
Did not load driver @oem5.inf,%nic_mpclie%;Intel(R) PRO/Wireless 3945ABG Network
Connection
Did not load driver @netb57vx.inf,%bcm5750a1clnahkd%;Broadcom NetXtreme 57xx Gigabit
Controller
Did not load driver @sdbus.inf,%pci\cc_080501.devicedesc%;SDA Standard Compliant
SD Host Controller
...
```

Среда восстановления Windows

Безопасный режим дает возможность восстановить систему, которую не удается загрузить из-за сбоя какого-либо драйвера. Но бывают ситуации, когда это не помогает. Например, если приводящий к сбою драйвер входит в группу `Safe`, загрузиться в безо-

пасном режиме не получится. Не помогает загрузка в безопасном режиме и при сбое в драйвере стороннего производителя, например в драйвере антивирусного сканнера. (Он относится к загрузочным драйверам, которые стартуют независимо от режима загрузки.) Аналогичная ситуация возникает при повреждении файлов системных модулей, драйверов, являющихся частью конфигурации безопасного режима, а также главной загрузочной записи (MBR).

Справиться с этими проблемами позволяет среда восстановления Windows (Windows Recovery Environment, WinRE). Она предоставляет целый ассортимент инструментов и автоматизированных технологий восстановления для решения наиболее распространенных проблем загрузки. Вот пять основных инструментов:

- ❑ **Startup Repair** (Восстановление запуска). Автоматизированный инструмент, распознающий наиболее распространенные проблемы запуска Windows и автоматически пытающийся их решать.
- ❑ **System Restore** (Восстановление системы). Позволяет вернуться в точку восстановления в случаях, когда Windows невозможно загрузить даже в безопасном режиме.
- ❑ **System Image Recover** (Восстановление системного образа). В предыдущих версиях Windows аналогом этого инструмента были Complete PC Restore и ASR (Automated System Recovery — автоматическое восстановление системы). Он позволяет восстанавливать Windows из полной резервной копии, а не из точки восстановления, что снижает вероятность потери данных.
- ❑ **Windows Memory Diagnostic Tool** (Средство диагностики памяти в Windows). Анализирует память компьютера на наличие аппаратных проблем. Дефекты RAM могут стать причиной периодических сбоев в работе ядра и приложений и нестабильного поведения системы.
- ❑ **Command Prompt** (Командная строка). В случаях, когда для решения проблем требуется ручное вмешательство (например, при копировании файлов с другого диска или при манипуляциях с BCD), можно воспользоваться командной строкой как оболочкой Windows, позволяющей при наличии требуемых зависимостей запускать практически любые программы. Этим командная строка отличается от применявшейся в предыдущих версиях консоли восстановления, которая поддерживала ограниченный набор специальных команд.

При загрузке системы с дистрибутивного компакт-диска Windows или с загрузочных дисков приложение Windows Setup предоставляет выбор между установкой новой версии и восстановлением уже существующей. При выборе второго варианта появляется диалоговое окно **System Recovery Options**, показанное на рис. 13.7.

Последние версии Windows также устанавливают WinRE в раздел восстановления на жестком диске. В этом случае доступ к WinRE можно получить нажатием клавиши F8. В результате появится меню дополнительных вариантов загрузки. Если вы видите там пункт **Repair Your Computer**, значит, у вас есть локальная копия жесткого диска вашего компьютера. Если же по каким-то причинам этот пункт меню отсутствует, воспользуйтесь инструкциями из блога Microsoft WinRE (<http://blogs.msdn.com/winre>) и установите WinRE из дистрибутива Windows и комплекта автоматической установки Windows (Windows Automated Installation Kit, AIK).

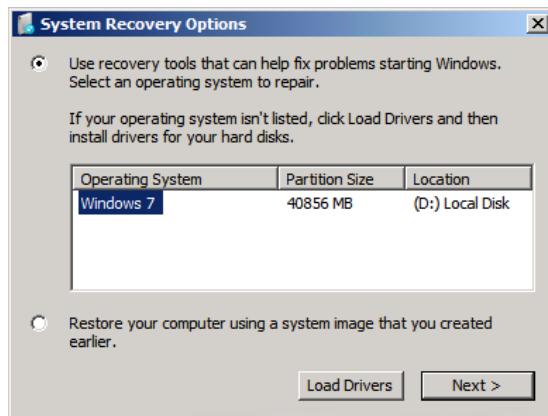


Рис. 13.7. Диалоговое окно System Recovery Options

При выборе первого пункта меню появляется диалоговое окно с различными вариантами восстановления, показанное на рис. 13.8. В то же время выбор второго варианта эквивалентен выбору в меню, показанном на рис. 13.8, пункта **System Image Recovery**.



Рис. 13.8. Усовершенствованное диалоговое окно System Recovery Options

Кроме того, если система не смогла загрузиться из-за поврежденных файлов или по другой понятной причине, приложение Bootmgr получит инструкцию автоматически запустить WinRE при следующей перезагрузке. Вместо показанного на рис. 13.8 диалогового окна среда восстановления автоматически запустит программу Startup Repair (рис. 13.9).

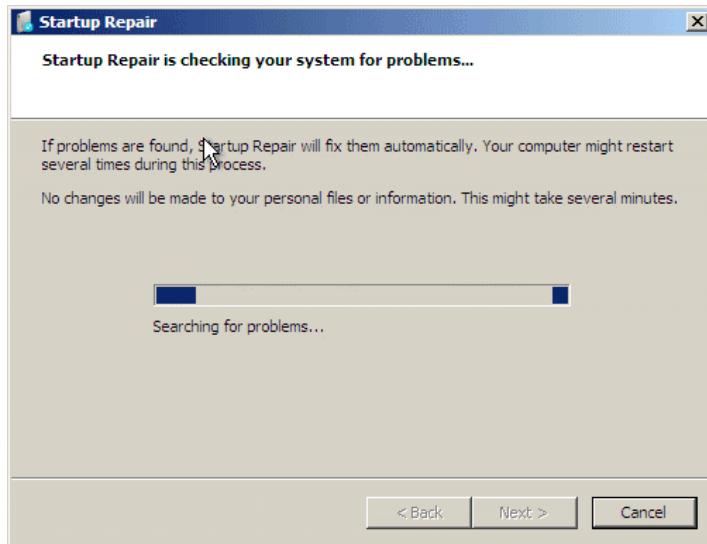


Рис. 13.9. Программа Startup Repair

В конце цикла сканирования и восстановления эта программа пытается автоматически устранить все обнаруженные неполадки, в числе прочего замещая системные файлы с установочного диска. Можно щелкнуть на расположенной снизу ссылке и узнать, какие именно проблемы были устранены. К примеру, как показано на рис. 13.10, программа Startup Repair исправила поврежденный загрузочный сектор.

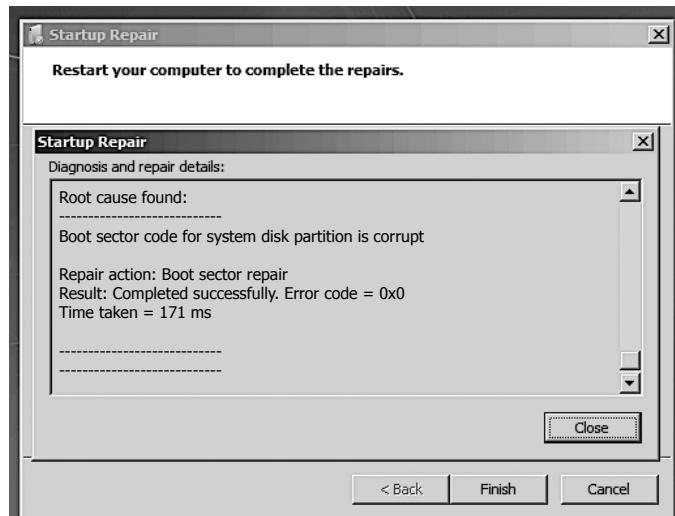


Рис. 13.10. Просмотр деталей работы Startup Repair

Если инструмент Startup Repair не в состоянии автоматически устраниить неполадку или вы прервали операцию, у вас появляется возможность попробовать другие варианты в диалоговом окне System Recovery Options.

ФАЙЛ СОСТОЯНИЯ ЗАГРУЗКИ

В Windows файл состояния загрузки (%SystemRoot%\Bootstat.dat) используется для фиксации прохождения через различные стадии жизненного цикла системы, включая загрузку и выключение. Именно он позволяет диспетчеру загрузки, загрузчику Windows и программе Startup Repair распознать проблемы на стадии выключения системы и предложить варианты восстановления, такие как возвращение к последней удачной конфигурации и загрузка в безопасном режиме. Этот бинарный файл содержит информацию, посредством которой система отчитывается об успешном завершении следующих фаз:

- загрузка (определение успешной загрузки аналогично упомянутому ранее определению последней удачной конфигурации);
- выключение;
- выход из спящего режима или восстановление после приостановки работы.

Также файл состояния загрузки указывает, была ли распознана проблема при последней попытке пользователя загрузить операционную систему, сопровождавшейся появлением параметров восстановления. Последние означают, что пользователь в курсе проблемы и предпринял некие действия. API-интерфейсы библиотеки времени выполнения (Runtime Library, Rtl) в файле Ntdll.dll содержат закрытые интерфейсы, посредством которых Windows осуществляет чтение из файла и запись в файл. Они, как и BCD, недоступны для редактирования пользователями.

Решение распространенных проблем загрузки

В этом разделе мы поговорим о проблемах, которые могут возникнуть в процессе загрузки, их симптомах, причинах и подходах к их решению. Для простоты поиска они перечислены в порядке возникновения в процессе загрузки. Следует заметить, что в большинстве случаев вам нужно просто загрузиться в среде восстановления Windows и запустить инструмент Startup Repair для анализа и выполнения автоматизированных процедур восстановления.

Повреждение MBR

□ Симптомы

Система с поврежденной главной загрузочной записью (MBR) пройдет выполняемый BIOS тест самодиагностики при включении (Power-On Self Test, POST), выведет на экран предоставленную BIOS информацию о версии или модели компьютера, но затем экран станет черным и компьютер зависнет. В зависимости от типа повреждения MBR может появиться одно из следующих сообщений: «Invalid partition table» (Недопустимая таблица разделов), «Error loading operating system» (Ошибка загрузки операционной системы) или «Missing operating system» (Операционная система не найдена).

□ Причина

Причиной повреждения MBR могут быть ошибки жесткого диска, повреждение диска в результате сбоя одного из драйверов в процессе работы Windows или деструктивная деятельность вируса.

□ Решение

Загрузите среду восстановления Windows, выберите вариант Command Prompt и выполните команду `bootrec /fixmbr`. Она перепишет исполняемый код в MBR.

Повреждение загрузочного сектора**□ Симптомы**

Повреждение загрузочного сектора может выглядеть как повреждение MBR, при котором появляется черный экран, и система зависает после того, как BIOS завершает тест POST. Или же на черном экране могут появиться сообщения «A disk read error occurred» (Возникла ошибка чтения диска), «BOOTMGR is missing» (Отсутствие BOOTMGR) или «BOOTMGR is compressed» (Сжатие BOOTMGR).

□ Причина

Причиной повреждения загрузочного сектора могут стать ошибки жесткого диска, повреждение диска в результате сбоя одного из драйверов в процессе работы Windows или деструктивная деятельность вируса.

□ Решение

Загрузите среду восстановления Windows, выберите вариант Command Prompt и выполните команду `bootrec /fixboot`. Она перепишет загрузочный сектор на указанном вами томе. Если на вашем компьютере системный диск отличается от загрузочного, следует выполнить эту команду для обоих дисков.

Неправильная конфигурация BCD**□ Симптомы**

После того как BIOS завершает тест POST, появляется сообщение, которое начинается так: «Windows could not start because of a computer disk hardware configuration problem» (Не удалось запустить Windows из-за проблемы с конфигурацией дискового устройства), «Could not read from selected boot disk» (Не удалось прочитать данные с выбранного загрузочного диска) или «Check boot path and disk hardware» (Проверьте путь к диску и загрузочное устройство).

□ Причина

BCD-хранилище было удалено, повреждено или больше не ссылается на загрузочный том из-за добавления раздела, которое стало причиной изменения имени тома.

□ Решение

Загрузите среду восстановления Windows, выберите вариант Command Prompt и выполните команды `bootrec /scanos` и `bootrec /rebuildbcd`. Это заставит систему проанализировать все тома в поисках установленных копий Windows. При обнаружении первой из них появится вопрос, следует ли добавить ее в BCD в качестве варианта загрузки и под каким названием она должна присутствовать в загрузочном меню. Для остальных вариантов повреждений, связанных с BCD, можно воспользоваться

программой `Bcdedit.exe` и построить «с нуля» новое BCD-хранилище или продублировать существующую корректную копию.

Повреждение системных файлов

□ Симптомы

Повреждение системных файлов (к ним относятся исполняемые файлы, драйверы и DLL) может проявляться по-разному. Например, после того как BIOS завершит тест POST, может появиться сообщение на черном экране: «Windows could not start because the following file is missing or corrupt» (Не удалось запустить Windows из-за отсутствия или повреждения следующего файла). Далее выводится имя файла и запрос на его переустановку. Кроме того, может появиться синий экран с текстом: «STOP: 0xC0000135 {Unable to Locate Component}».

□ Причины

Поврежден том, на котором находится системный файл, или удалены либо повреждены несколько системных файлов.

□ Решение

Загрузите среду восстановления Windows, выберите вариант **Command Prompt** и выполните команду `chkdsk`. Программа `Chkdsk` попытается устранить повреждение тома. Если она не обнаружит проблем, вам потребуется резервная копия нужного системного файла. Такие копии можно найти, например, в папке `%SystemRoot%\winsxs\Backup`, где Windows хранит копии многих системных файлов для доступа к технологии WRP (см. далее врезку «Защита ресурсов Windows»). Если в этой папке нужной копии не оказалось, поищите ее на других компьютерах сети. Следует заметить, что резервная копия файла должна принадлежать тому же пакету обновления или критическому исправлению, что и заменяемый файл.

Бывают случаи, когда поврежденными или удаленными оказывается сразу несколько системных файлов, тогда процесс восстановления потребует неоднократной перезагрузки, пока все они не будут заменены. Если вы считаете, что поврежденных файлов слишком много, подумайте о восстановлении системы из резервного образа, например, сгенерированного программой **Backup and Restore** (Архивация и восстановление), или из точки восстановления.

Запустив программу **Backup and Restore** (Архивация и восстановление), которая доступна в подменю **Maintenance** (Обслуживание) меню **Start** (Пуск), можно сгенерировать образ восстановления системы. В него войдут все файлы на системном и загрузочном дисках, а также содержимое дискеты, на которой хранятся сведения о системных дисках и томах. Для восстановления системы из этого образа нужно загрузиться с дистрибутива Windows и выбрать подходящий вариант в предложенном меню (или воспользоваться описанной ранее средой восстановления).

Если резервной копии у вас нет, остается последнее средство – запуск программы установки Windows в режиме исправления. Загрузите Windows дистрибутива и следуйте указаниям мастера. При появлении вопроса, хотите ли вы исправить существующую систему или установить новую, выберите первый вариант. После этого программа установки переустановит все системные файлы, сохранив данные ваших приложений и параметры реестра.

ЗАЩИТА РЕСУРСОВ WINDOWS

Технология защиты ресурсов Windows (Windows Resource Protection, WRP) направлена на сохранение целостности множества вовлеченных в процесс загрузки компонентов, как и других важных файлов, библиотек и приложений. Она реализована через списки контроля доступа (ACL), защищающие важные системные файлы. Доступ к ней осуществляется через запускаемый с помощью утилиты Sfc.exe API-интерфейс (он находится в файлах %SystemRoot%\System32\Sfc.dll и %SystemRoot%\System32\Sfc_os.dll). С его помощью вы можете вручную проверить файлы на наличие повреждений и восстановить их.

При необходимости WRP защищает целые папки, блокируя доступ к ним даже для администратора (без редактирования списка контроля доступа к папке). Единственным поддерживаемым средством редактирования файлов, защищенных с помощью WRP, является служба Modules Installer, запускаемая в учетной записи TrustedInstaller. Она служит для установки обновлений и исправлений. Эта учетная запись имеет доступ к различным защищенным файлам, и, как следует из ее названия (дословно — «установщик модулей»), система позволяет ей редактировать важные файлы и заменять их. Еще WRP защищает важные разделы реестра и может блокировать целые кусты, если все входящие в них параметры и подразделы считаются важными для работы системы.

Для защищенных файлов, папок или разделов реестра WRP создает ACL, позволяя редактировать или удалять файлы только учетной записи TrustedInstaller. Разработчики приложений могут использовать API-интерфейс SfcIsFileProtected или SfcIsKeyProtected для проверки блокировки файлов и разделов реестра.

Для обратной совместимости некоторые установщики считаются известными. Существует модификатор совместимости приложений, обходящий ошибку доступа, которая возникает при попытке редактирования определенными установщиками ресурсов, защищенных с помощью WRP. Вместо этого установщик получает фальшивый код, свидетельствующий об успешном завершении, но редактирование не производится. Эта виртуализация аналогична технологии управления учетными записями пользователей (UAC), о которой шла речь в главе 6, но применима и к операциям записи. Она используется при следующих обстоятельствах:

- приложение является устаревшим, то есть не содержит файл манифеста, совместимый с заданным значением requestedExecutionLevel;
- приложение пытается редактировать защищенный с помощью WRP ресурс (файл или раздел реестра содержит идентификатор безопасности TrustedInstaller);
- приложение запускается под управлением учетной записи администратора (что всегда имеет место в системах с включенным режимом UAC благодаря автоматическому обнаружению программы-установщика).

WRP копирует необходимые для перезагрузки Windows файлы в папку кэша, расположенную по адресу %SystemRoot%\winsxs\Backup. Важные файлы, не требующиеся для перезагрузки, сюда не попадают. Размер этой папки и список копируемых в нее файлов изменить нельзя. Для восстановления файла из этой папки можно воспользоваться утилитой System File Checker (Sfc.exe), которая сканирует систему в поисках отредактированных защищенных файлов и восстанавливает их из корректной копии.

Повреждение куста System

□ Симптомы

Если куст реестра **System** (он рассматривается в разделе «Реестр» главы 4) отсутствует или поврежден, после того как BIOS завершит тест POST, Winload покажет на черном экране сообщение: «Windows could not start because the following file is missing or corrupt: \WINDOWS\SYSTEM32\CONFIG\SYSTEM» (Не удалось запустить Windows из-за отсутствия или повреждения следующего файла: \WINDOWS\SYSTEM32\CONFIG\SYSTEM).

□ Причины

Удален или поврежден куст реестра **System**, содержащий необходимые для загрузки системы данные о конфигурации.

□ Решение

Загрузите среду восстановления Windows, выберите вариант **Command Prompt** и выполните команду **chkdsk**. Если это не решит проблему, возьмите резервную копию куста **System**. Копии кустов реестра Windows делает каждые 12 часов (при этом предыдущая копия немедленно получает расширение **.OLD**) и сохраняет их в папке **%SystemRoot%\System32\Config\RegBack**. Поэтому вам нужно скопировать файл **System** в **%SystemRoot%\System32\Config**.

Если у вас доступен инструмент восстановления системы (он рассматривался в главе 9), можно получить более раннюю резервную копию кустов реестра из наиболее актуальной точки восстановления. Выберите вариант **System Restore** в среде восстановления Windows для восстановления реестра из последней точки восстановления.

Сбой или зависание после вывода экранной заставки

□ Симптомы

К данной категории относятся проблемы, возникающие в Windows после вывода экранной заставки, появления рабочего стола или входа в систему. Они могут проявляться в виде синего экрана или зависания, при котором либо замораживается вся система, либо сохраняется возможность перемещать по экрану указатель мыши, но при этом система ни на что не реагирует.

□ Причины

Эти проблемы практически всегда являются следствием ошибки в драйвере устройства, но иногда они возникают из-за повреждения куста реестра, отличного от куста **System**.

□ Решение

Существует несколько способов устранения данной проблемы. Прежде всего следует попробовать вернуться к последней удачной конфигурации (LKG). Эта конфигурация состоит из такого контрольного набора параметров реестра, который позволил в последний раз успешно загрузить систему. Так как в этот набор входит базовая системная конфигурация и регистрационная база данных драйверов устройств и служб, там не отражаются последние изменения в составе драйверов или служб, что зачастую помогает обойти источник проблемы. Для доступа к последней удачной конфигурации следует нажать клавишу **F8** на ранней стадии процесса загрузки и в появившемся меню выбрать нужный пункт.

В этой главе уже упоминалось, что при загрузке LKG система сохраняет тот контрольный набор параметров, от которого вы отказываетесь, и помечает его как неудачный. Если LKG загружается, можно экспорттировать содержимое текущего и неудачного контрольных наборов в файлы с расширением .reg и, сравнив их, определить, что стало причиной неудачной загрузки. Для этого используйте поддержку экспорта в редакторе реестра в меню File (Файл):

1. Запустите редактор реестра и выберите раздел HKLM\SYSTEM\CurrentControlSet.
2. Выберите в меню File (Файл) команду Export (Экспорт) и сохраните содержимое в файле good.reg.
3. Откройте раздел HKLM\SYSTEM>Select, посмотрите значение параметра Failed и выберите подраздел HKLM\SYSTEM\ControlXXX, где XXX – значение параметра Failed.
4. Экспортируйте содержимое этого набора в файл bad.reg.
5. Воспользуйтесь приложением WordPad для замены всех вхождений CurrentControlSet в файле good.reg значением ControlSet.
6. Затем замените все вхождения ControlXXX в файле bad.reg значением ControlSet.
7. Запустите программу Windiff из набора Support Tools и сравните два файла.

Различия между удачным и неудачным контрольным набором могут быть весьма значительными, поэтому следует сосредоточиться на изменениях в подразделах Control и Parameters всех драйверов и служб, зарегистрированных в разделе Services. Различия в подразделах Enum разделов для драйверов в ветви Services управляющего набора игнорируйте.

Последняя удачная конфигурация не загрузится, если проблема вызвана драйвером или службой, которые присутствовали в системе до последней успешной загрузки. Не поможет она и в ситуации, если изменившийся проблемный параметр конфигурации находится вне контрольного набора или изменен до последней успешной загрузки. В подобных случаях нужно попробовать загрузиться в безопасном режиме. Если у вас получится это сделать и вы знаете, какой драйвер стал причиной предыдущего сбоя, этот драйвер следует отключить в диспетчере устройств (доступ к нему осуществляется через Панель управления). Для этого нужно выделить проблемный драйвер и выбрать в меню Action (Действие) команду Disable (Отключить). Если вы недавно обновляли этот драйвер и считаете, что ошибка появилась как следствие обновления, можно совершить откат к предыдущей версии, что также осуществляется в диспетчере устройств. Для этого нужно дважды щелкнуть на имени устройства, чтобы открыть для него диалоговое окно Properties (Свойства), и щелкнуть на кнопке Roll Back Driver (Откатить) на вкладке Driver (Драйвер).

В системах с доступным инструментом восстановления системы предлагается откат всей системы к предыдущей точке восстановления (заданной через программу восстановления системы). В безопасном режиме программа распознает наличие точек восстановления и, обнаружив их, спрашивает, хотите ли вы войти в систему и вручную выполнить диагностику и исправление или же предпочитаете загрузить мастер восстановления системы. Попытка сделать систему загружаемой с помощью программы восстановления системы является хорошим вариантом в случае, когда вы

знаете причину проблемы и хотите автоматически ее устраниить или когда причина вам не известна, но вы не желаете тратить время на ее выявление.

Если программа восстановления системы вас не устраивает или вы хотите определить причину сбоя при нормальной загрузке, если в безопасном режиме система успешно загружается, попытайтесь получить журнал неудачной загрузки, нажав клавишу F8 для входа в специальное меню и выбрав там соответствующий пункт. Как уже упоминалось, диспетчер сеансов (%SystemRoot%\System32\Smss.exe) сохраняет в файле %SystemRoot%\ntbtlog.txt журнал загрузки с записями как о драйверах устройств, которые были загружены, так и о тех, которые система предпочла не загружать. Поэтому получить данный журнал можно только в случае, когда сбой или зависание происходит после инициализации диспетчера сеансов. После перезагрузки в безопасном режиме система добавит новые записи в существующий журнал загрузки. Скопируйте фрагменты журнала, относящиеся к неудачной попытке и к загрузке в безопасном режиме, в разные файлы. Удалите строки с текстом «Did not load driver» и сравните файлы, например, при помощи программы Windiff. Поочередно отключайте драйверы, которые загружались при нормальной загрузке, но не в безопасном режиме, пока система снова не начнет функционировать. (Затем подключите драйверы, не связанные с проблемой.)

Если вам не удается получить журнал с результатом нормальной загрузки (например, когда сбой системы происходит до инициализации диспетчера сеансов), а система, к тому же, не в состоянии загрузиться в безопасном режиме, или если сравнение двух фрагментов журнала не выявило существенных различий, попробуйте воспользоваться программой Driver Verifier и проанализировать аварийный дамп. (Эта тема рассматривается в главе 14.)

Завершение работы

Если в системе авторизован какой-то пользователь, а некий процесс инициирует завершение работы системы, вызывая функцию `ExitWindowsEx`, процесс Csrss текущего сеанса получает команду завершить работу. В свою очередь, Csrss выступает в роли вызывающей функции и в режиме RPC (Remote Procedure Call — удаленный вызов процедур) отправляет сообщение приложению Winlogon, требуя завершить работу системы. Затем Winlogon от лица авторизованного в данный момент пользователя (который может иметь или не иметь такой же контекст защиты, как пользователь, инициировавший прекращение работы) вызывает функцию `ExitWindowsEx` с набором специальных внутренних флагов. В результате процесс Csrss получает еще одно сообщение с запросом на выключение системы.

На этот раз Csrss видит, что запрос поступил от Winlogon, и перебирает все процессы в сеансе интерактивного пользователя (еще раз отметим, что это не тот же пользователь, который запросил завершения работы) в порядке, обратном их *уровню завершения* (shutdown level). Уровень завершения показывает, когда процессу следует завершиться относительно других процессов. Его задают при помощи функции `SetProcessShutdownParameters`. Допустимые уровни лежат в диапазоне от 0 до 1023, а уровень, используемый по умолчанию, равен 640. Например, приложение Explorer имеет уровень завершения 2, а диспетчер задач — 1. Каждому процессу, владеющему

окном верхнего уровня, для всех его программных потоков Csrss посыпает сообщение `WM_QUERYENDSESSION` с циклом выборки Windows-сообщений. Если поток возвращает значение `TRUE`, процесс завершения работы системы продолжается. После этого Csrss посыпает программному потоку сообщение `WM_ENDSESSION` с требованием завершить работу и ожидает его выполнения в течение времени, указанного в параметре `HKEY_CURRENT_USER\Control Panel\Desktop\HungAppTimeout`. (По умолчанию оно составляет 5000 миллисекунд.)

Если в течение этого времени поток не завершается, Csrss выводит экран, показанный на рис. 13.11. (Его можно отключить, создав параметр `HKEY_CURRENT_USER\Control Panel\Desktop\AutoEndTasks` реестра и присвоив ему значение 1.) На этом экране перечисляются работающие в данный момент программы с информацией об их состоянии, если таковая доступна. Windows отмечает программы, которые не завершаются в течение заданного времени, и предлагает пользователю убить соответствующий процесс или прервать процесс завершения работы компьютера. (У этого экрана отсутствует тайм-аут, поэтому на данном этапе запрос на завершение может длиться бесконечно.) Кроме того, могут добавлять информацию о своем состоянии приложения сторонних производителей. Например, средство виртуализации может вывести на экран количество работающих виртуальных машин.

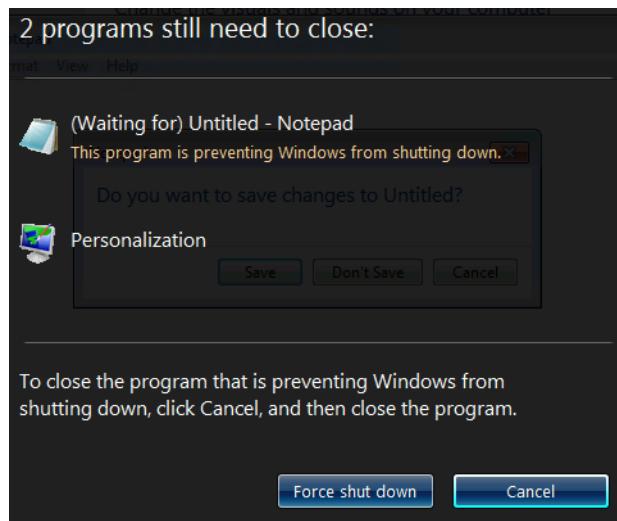
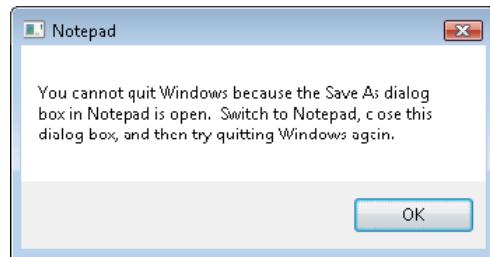


Рис. 13.11. Экран «зависших» программ

ЭКСПЕРИМЕНТ: ПРОВЕРКА ПАРАМЕТРА HUNGAPPTIMEOUT

Можно проверить, как используется параметр `HungAppTimeout` реестра, открыв Блокнот, введя в него текст и выйдя из системы. По истечении времени, указанного в `HungAppTimeout`, Csrss.exe спросит, хотите ли вы закрыть незавершенный процесс Notepad. Этот процесс ждет, когда вы сообщите ему, следует ли сохранить введенный текст. После щелчка на кнопке `Cancel` процесс Csrss.exe отменяет завершение работы системы.

В качестве второго эксперимента попробуйте снова завершить работу системы (не закрывая диалоговое окно с запросом о завершении процесса Notepad). Блокнот выведет на экран собственное диалоговое окно с сообщением, информирующим о невозможности завершения работы. Но это окно является информационным и предназначено для помощи пользователям. Процесс Csrss.exe при этом считает, что приложение Notepad «зависло», и предоставляет пользовательский интерфейс, позволяющий завершить неотвечающий процесс.



Если поток завершается до истечения тайм-аута, Csrss продолжает посыпать пары сообщений WM_QUERYENDSESSION и WM_ENDSESSION другим программным потокам в процессах, обладающих собственными окнами. После завершения всех потоков в процессе Csrss завершает этот процесс и переходит к следующему процессу в рамках интерактивного сеанса.

Обнаружив консольное приложение, Csrss вызывает обработчик консоли, посылая событие CTRL_LOGOFF_EVENT. (Это событие при завершении работы системы посыпается только служебным процессам.) Если обработчик возвращает значение FALSE, Csrss убивает процесс. При получении значения TRUE или при отсутствии ответа в течение указанного в параметре HKCU\Control Panel\Desktop\WaitToKillAppTimeout времени (по умолчанию оно составляет 20 000 миллисекунд), Csrss выводит экран, показанный на рис. 13.11.

Затем Winlogon вызывает функцию ExitWindowsEx, чтобы заставить Csrss завершить все СОМ-процессы, являющиеся частью интерактивного сеанса пользователя.

К этому моменту выполнение всех процессов в сеансе уже закончено. После этого Wininit снова вызывает функцию ExitWindowsEx, на этот раз в контексте системного процесса. Это заставляет Wininit отправить сообщение процессу Csrss как части сеанса 0, где располагаются службы. Затем Csrss смотрит на все принадлежащие системному контексту процессы и отправляет сообщения WM_QUERYENDSESSION и WM_ENDSESSION GUI-потокам (как раньше). Но консольным приложениям с зарегистрированными обработчиками вместо CTRL_LOGOFF_EVENT он посыпает сообщение CTRL_SHUTDOWN_EVENT. Обратите внимание, что диспетчер управления службами (SCM) — это консольная программа, которая регистрирует обработчик. При получении запроса на завершение она, в свою очередь, рассыпает соответствующие сообщения всем службам, зарегистрированным для получения уведомлений о завершении работы. Более подробно темы завершения служб (а также тайм-аута для SCM) рассматриваются в главе 4 части I.

Хотя в данном случае Csrss работает с теми же тайм-аутами, что и при завершении пользовательских процессов, никаких диалоговых окон этот процесс не выводит и не

завершает процессы принудительно. (Параметры реестра, задающие тайм-аут системных процессов, берутся из профиля пользователя, предлагаемого по умолчанию.) Эти тайм-ауты просто позволяют системным процессам корректно завершиться до выключения системы. Соответственно, в момент выключения системы многие из них еще выполняются, например Smss, Wininit, Services и LSASS.

Как только Csrss заканчивает уведомление системных процессов о завершении, Winlogon вызывает функцию `NtShutdownSystem` исполнительной подсистемы. Она, в свою очередь, вызывает функцию `PoSetSystemPowerState`, управляющую завершением драйверов и прочих компонентов исполнительной подсистемы (PnP-диспетчера, диспетчера электропитания, диспетчера ввода-вывода, диспетчера конфигурации и диспетчера памяти).

К примеру, `PoSetSystemPowerState` вызывает диспетчер ввода-вывода для рассылки пакетов завершения ввода-вывода всем драйверам устройств, запросившим уведомление о завершении работы системы. Это позволяет драйверам подготовить свои устройства к завершению работы Windows. Подгружается стек рабочих потоков, диспетчер конфигурации сбрасывает на диск все измененные данные реестра, а диспетчер памяти записывает все измененные страницы с файловыми данными обратно в соответствующие файлы. Если включен режим очистки файла подкачки при завершении работы, диспетчер памяти выполняет эту операцию. Затем снова вызывается диспетчер ввода-вывода, чтобы проинформировать системные драйверы о завершении работы системы. Действия диспетчера электропитания зависят от пользовательских параметров. Это может быть выключение компьютера, перезагрузка или отключение электропитания.

Заключение

В этой главе подробно рассмотрены этапы загрузки и завершения работы Windows как в нормальном режиме, так и при возникновении сбоев. Мы изучили общую структуру Windows и базовые системные механизмы, обеспечивающие запуск, работу и, в конечном счете, выключение системы. Последняя глава посвящена действиям, которые следует предпринимать в случае неожиданного завершения работы — при крахе системы.

Глава 14. Анализ аварийного дампа

Почти каждый пользователь Windows слышал о так называемом «синем экране смерти» или даже видел его. Этим зловещим термином называют синий экран, возникающий при прекращении работы Windows из-за катастрофического сбоя или внутренней ошибки, препятствующей работе системы.

В этой главе рассматриваются основные причины сбоев в Windows. Также рассказывается об информации, выводимой на «синий экран», и о различных конфигурационных параметрах, управляющих созданием *аварийного дампа* (*crash dump*) — снимка системной памяти в момент прекращения работы, помогающего определить, какой из компонентов стал причиной сбоя и почему. В этой главе мы *не будем* в деталях изучать информацию, которую можно получить путем анализа аварийного дампа. Тем не менее вы узнаете, как идентифицировать сбойный драйвер или вышедший из строя компонент. Базовый анализ аварийного дампа требует минимальных усилий и выполняется за несколько минут. Даже если выявить проблемный драйвер с первой попытки не удается, имеет смысл прибегнуть к анализу, так как он позволяет избежать потери данных или простой системы.

Почему в Windows случаются сбои?

Существуют различные причины сбоев в Windows и появления синего экрана. Часто такой причиной становится ссылка на адрес в памяти, нарушающая права доступа, например попытка записи на страницу, имеющую атрибут «только для чтения», или попытка чтения по еще не отображеному на память адресу. Другой распространенной причиной являются непредвиденные исключения или прерывания. Сбои также случаются, когда одна из подсистем ядра (такая, как диспетчер памяти или диспетчер электропитания) или драйвер (например, USB-драйвер или драйвер дисплея) обнаруживают несогласованность выполняемых им операций.

Когда в режиме ядра драйвер устройства или подсистема вызывают недопустимое исключение, Windows попадает в затруднительное положение. Выясняется, что часть системы, имеющая доступ к аппаратному обеспечению и любым областям памяти, выполнила недопустимые действия.

Но почему это обязательно должно сопровождаться остановкой работы Windows? Разве нельзя просто проигнорировать исключение, позволив драйверу устройства или подсистеме работать дальше? Ведь существует вероятность, что ошибка носила локальный характер и компонент сумеет восстановиться. Впрочем, гораздо вероятнее, исключение связано с более серьезными проблемами, например с повреждением памяти или со сбоями в работе оборудования. Если система при этом продолжит свою работу, это может кончиться появлением дополнительных исключений и повреждением данных на диске или других периферических устройствах, что слишком рискованно.

Поэтому Windows предпочитает политику *быстрой остановки*, пытаясь предотвратить распространение на диск повреждений в RAM.

Синий экран

По какой бы причине ни произошел сбой системы, во всех случаях она вызывается функцией `KeBugCheckEx`, описанной в Windows Driver Kit (WDK). Эта функция принимает *стоп-код* (stop code), или *контрольный код ошибки* (bugcheck code), и четыре параметра, которые интерпретируются в зависимости от этого кода. Функция `KeBugCheckEx` маскирует все прерывания на всех процессорах системы, а затем переключает видеоадаптер в графический режим VGA с низким разрешением (который поддерживается всеми совместимыми с Windows видеокартами) и выводит стоп-код на синем экране, сопровождая его рекомендациями по дальнейшим действиям пользователя. Напоследок `KeBugCheckEx` вызывает все зарегистрированные (через функцию `KeRegisterBugCheckCallback`) функции обратного вызова драйверов устройств при ошибке, предоставляя драйверам возможность остановить управляемые ими устройства. Затем активируются функции обратного вызова, проверяющие причину сбоя (они зарегистрированы через функцию `KeRegisterBugCheckReasonCallback`), благодаря которым драйверы получают возможность добавить данные в аварийный дамп или вывести сведения из этого дампа на альтернативное устройство.

Первая строка из раздела «Technical information» в показанном на рис. 14.1 примере синего экрана содержит стоп-код и четыре дополнительных параметра, передаваемые в функцию `KeBugCheckEx`. Стока в верхней части экрана представляет собой текстовый эквивалент числового идентификатора стоп-кода. В данном примере стоп-код `0x0000000D1` соответствует ошибке `DRIVER_IRQL_NOT_LESS_OR_EQUAL`. Если параметр содержит адрес части операционной системы или кода драйвера устройства (как на рис. 14.1), Windows выводит на экран базовый адрес модуля, в котором произошла ошибка, отметку даты и имя файла драйвера устройства. Этой информации может быть вполне достаточно для идентификации проблемного компонента.

Существует больше 300 уникальных стоп-кодов, но большинство из них встречается крайне редко или вообще никогда не встречается. Большая часть сбоев в Windows сопровождается несколькими типичными стоп-кодами. Кроме того, стоп-код определяет смысл четырех дополнительных параметров (при этом не для всех стоп-кодов предусматривается расширенная информация, передаваемая через эти параметры). Тем не менее анализ стоп-кода и значений параметров (если эти значения имеют смысл) способен помочь в диагностике некорректно работающего компонента (или устройства, приведшего к сбою).

Необходимая для интерпретации стоп-кодов информация содержится в разделе «Bug Checks (Blue Screens)» файла помощи Debugging Tools for Windows. (Этот файл подробно рассматривается в главе 1 части I.) Кроме того, имеет смысл воспользоваться поиском по стоп-коду имени устройства или драйвера в базе знаний Microsoft (<http://support.microsoft.com>). Там можно найти сведения об обходных путях и пакетах обновлений, позволяющих устранить существующую проблему. Файл `Bugcodes.h` из WDK содержит полный список из более чем 300 стоп-кодов с описанием причин

возникновения некоторых из них. Ну и, наконец, стоп-коды перечислены и документированы на странице [http://msdn.microsoft.com/en-us/library/windows/hardware/hh406232\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/hardware/hh406232(v=vs.85).aspx).

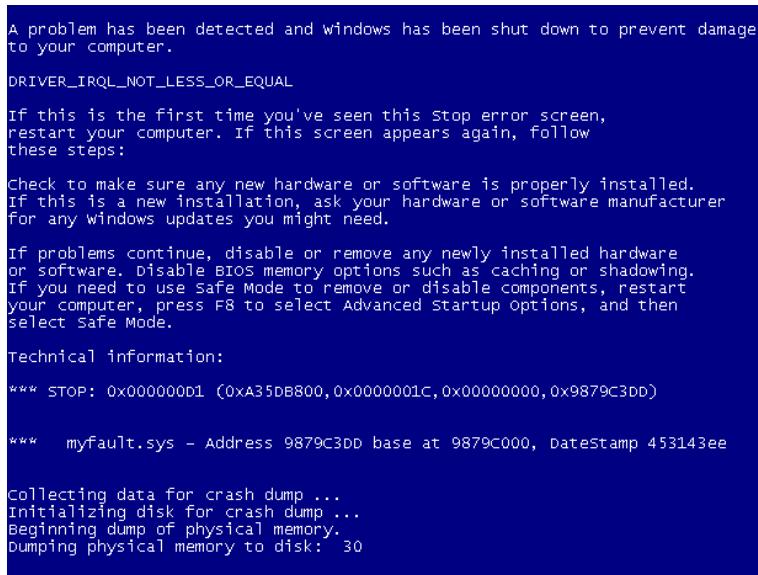


Рис. 14.1. Пример синего экрана

Причины сбоев в Windows

На основе данных, собранных в Windows версий с 7 по 7 SP1, двадцать чаще всего встречающихся стоп-кодов, которые являются причиной 91 % системных сбоев, можно разбить на следующие категории:

- **Обращение к отсутствующей странице.** Обращение к данным выгруженной из памяти страницы при IRQL-уровне DPC/dispatch или выше, когда диспетчеру памяти приходится ждать операции ввода-вывода. Но ядро при таком IRQL-уровне не в состоянии ждать или менять порядок программных потоков. (Об IRQL рассказывается в главе 3 части I.) Стоп-коды в этом случае:
 - 0xA — IRQL_NOT_LESS_OR_EQUAL;
 - 0xD1 — DRIVER_IRQL_NOT_LESS_OR_EQUAL.
- **Управление электропитанием.** Драйвер устройства или функция операционной системы в режиме ядра находятся в несогласованном или недопустимом состоянии энергопотребления. Чаще всего это связано с неспособностью какого-то компонента завершить за заданный по умолчанию период 10 минут запрос ввода-вывода на управление электропитанием. Стоп-код:
 - 0x9F — DRIVER_POWER_STATE_FAILURE.

- ❑ **Исключения и системные прерывания.** Драйвер устройства или функция операционной системы в режиме ядра сталкиваются с неожиданным исключением или прерыванием. Стоп-коды:
 - 0x1E — KMODE_EXCEPTION_NOT_HANDLED;
 - 0x3B — SYSTEM_SERVICE_EXCEPTION;
 - 0x7E — SYSTEM_THREAD_EXCEPTION_NOT_HANDLED;
 - 0x7F — UNEXPECTED_KERNEL_MODE_TRAP;
 - 0x8E — KERNEL_MODE_EXCEPTION_NOT_HANDLED с параметром P1, не равным 0xC0000005 STATUS_ACCESS_VIOLATION.
- ❑ **Нарушение прав доступа.** Драйвер устройства или функция операционной системы в режиме ядра сталкиваются со сбоем при доступе к ячейке памяти, вызванным попыткой осуществить запись на предназначенную только для чтения страницу или попыткой прочитать еще не отображенный на память и поэтому недействительный адрес в памяти. Стоп-коды:
 - 0x50 — PAGE_FAULT_IN_NONPAGED_AREA;
 - 0x8E — KERNEL_MODE_EXCEPTION_NOT_HANDLED с параметром P1, равным 0xC0000005 STATUS_ACCESS_VIOLATION.
- ❑ **Дисплей.** Драйвер дисплея обнаруживает, что не может больше контролировать графический процессор. Это означает, что попытка переустановить драйвер дисплея окончилась неудачей. Стоп-код:
 - 0x116 — VIDEO_TDR_FAILURE.
- ❑ **Пул.** Диспетчер пула в режиме ядра обнаруживает поврежденный заголовок пула или недействительную ссылку на пул. Стоп-коды:
 - 0x19 — BAD_POOL_HEADER;
 - 0xC2 — BAD_POOL_CALLER;
 - 0xC5 — DRIVER_CORRUPTED_EXPOOL.
- ❑ **Управление памятью.** Диспетчер памяти в режиме ядра обнаруживает повреждение управляющих памятью структур данных или некорректный запрос на управление памятью. Стоп-коды:
 - 0x1A — MEMORY_MANAGEMENT;
 - 0x4E — PFN_LIST_CORRUPT.
- ❑ **Аппаратное обеспечение.** Ошибка аппаратного контроля или немаскируемого прерывания (Non-Maskable Interrupt, NMI). Входят в эту категорию и сбои диска при попытках диспетчера памяти читать данные, чтобы компенсировать ошибки страниц. Стоп-коды:
 - 0x7A — KERNEL_DATA_INPAGE_ERROR;
 - 0x124 — WHEA_UNCORRECTABLE_ERROR.

- ❑ **USB.** При операции на универсальной последовательной шине (Universal Serial Bus, USB) возникает неустранимая ошибка. Стоп-код:
 - 0xFE — BUGCODE_USB_DRIVER.
- ❑ **Важный объект.** Критическая ошибка в потоке или процессе, без которого невозможна работа Windows. Стоп-код:
 - 0xF4 — CRITICAL_OBJECT_TERMINATION.
- ❑ **Файловая система NTFS.** Критическая ошибка обнаружена файловой системой NTFS. Стоп-код:
 - 0x24 — NTFS_FILE_SYSTEM.

На рис. 14.2 показана диаграмма распределения стоп-кодов по этим категориям для Windows 7 и Windows 7 SP1 на май 2012 года.

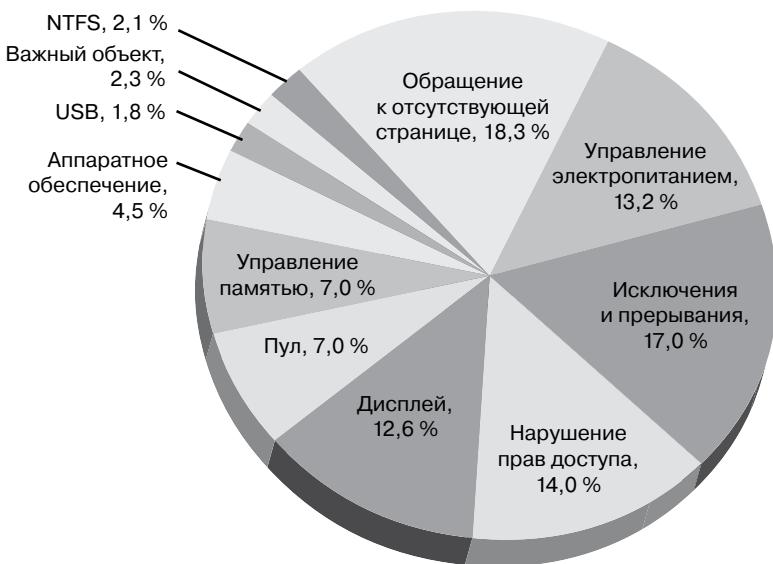


Рис. 14.2. Распределение по категориям двадцати самых распространенных стоп-кодов в операционных системах Windows 7 и Windows 7 SP1 на май 2012

Устранение проблем при сбоях

Синий экран часто появляется после установки нового программного или аппаратного обеспечения. Если после установки дополнительного драйвера и перезагрузки на ранней стадии инициализации системы вы получили синий экран, перезагрузите

систему еще раз и при появлении соответствующей инструкции нажмите клавишу F8, а затем выберите в меню вариант Last Known Good Configuration (Последняя удачная конфигурация). После этого Windows использует копию раздела реестра, в котором были зарегистрированы драйверы устройств (HKLM\SYSTEM\CurrentControlSet\Services) при последней удачной загрузке (до установки нового драйвера). Удачной считается загрузка, при которой полностью загружаются все службы и драйверы и выполняется хотя бы один вход в систему. (Детали см. в главе 13.)

При перезагрузке после сбоя диспетчер загрузки (Bootmgr) автоматически определит некорректное завершение работы Windows и выведет сообщение о восстановлении Windows после ошибки, аналогичное показанному на рис. 14.3. Этот экран позволяет загрузиться в безопасном режиме и отключить или удалить компонент программного обеспечения, который может являться источником проблемы.

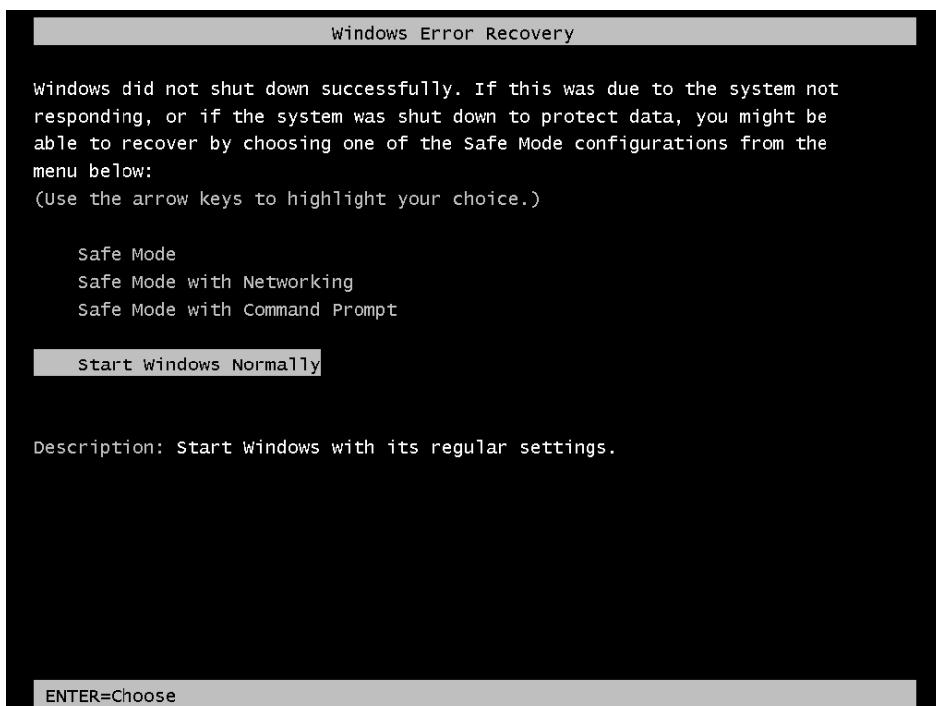


Рис. 14.3. Пример сообщения о восстановлении Windows после ошибки

Если даже после этого при попытке загрузиться вы видите синий экран, очевидно, следует удалить компонент, после установки которого возникли проблемы. Если с момента установки прошло некоторое время или же вы добавили одновременно несколько компонентов, следует обратить внимание на фигурирующие на синем экране имена драйверов. Если вы обнаружите там ссылку на недавно добавленный компонент (например, Storport.sys после установки нового SCSI-диска), скорее всего, именно он и является причиной сбоя.

Часто драйверы устройств имеют неочевидные имена, но существует возможность сопоставить их с устройством или программным компонентом. Для этого нужно открыть раздел `HKLM\SYSTEM\CurrentControlSet\Services` реестра, в котором Windows хранит регистрационную информацию обо всех драйверах системы, найти там имя вашего драйвера и ассоциированную с ним службу. Описание драйвера содержится в параметрах `DisplayName` и `Description`, более того, здесь же фигурирует предназначение некоторых драйверов. Например, строка "Virus Scanner" в параметре `DisplayName` указывает на принадлежность драйвера к антивирусной программе. Список драйверов позволяет получить программа `System Information` (Сведения о системе) — в меню `Start` (Пуск) следует выбрать команду `All Programs` (Все программы) ▶ `Accessories` (Стандартные) ▶ `System Tools` (Служебные) ▶ `System Information` (Сведения о системе). В программе нужно раскрыть узел `Software Environment` (Программная среда) и выбрать строку `System Drivers` (Системные драйверы). Приложение `Process Explorer` также позволяет увидеть все загруженные в данный момент драйверы с номерами версий и адресами загрузки (в разделе `DLL` процесса `System`). Еще можно для драйвера открыть диалоговое окно `Properties` (Свойства) и исследовать содержимое вкладки `Details` (Подробно). Здесь часто содержится описание драйвера и сведения о производителе. Следует помнить, что данные реестра и описание файла предоставляются производителем драйвера, поэтому гарантировать их точность невозможно.

Но зачастую стоп-кода и четырех связанных с ним параметров недостаточно для поиска и устранения причины сбоя. Для выяснения точного имени сбойного драйвера может понадобиться, например, анализ стека вызовов в режиме ядра. Кроме того, так как Windows после сбоя автоматически перезагружается, вам просто не хватит времени на изучение синего экрана. Поэтому по умолчанию Windows пытается сбросить на диск сведения о сбое для последующего анализа. Это так называемые файлы аварийного дампа, о которых пойдет речь в следующем разделе.

Файлы аварийного дампа

По умолчанию все системы семейства Windows настроены на запись информации о состоянии на момент сбоя. Для просмотра соответствующих параметров откройте через панель управления диалоговое окно `System Properties` (Свойства системы), войдите в раздел `System` (Система) и щелкните в левой панели на ссылке `Advanced System Settings` (Дополнительные параметры системы). В этом окне вам понадобится вкладка `Advanced` (Дополнительно), где нужно щелкнуть на кнопке `Settings` (Параметры) в разделе `Startup And Recovery` (Загрузка и восстановление). Варианты настройки Windows, предлагаемые по умолчанию, показаны на рис. 14.4.

Записываемые при сбое сведения делятся по объему на три уровня:

- Полный дамп памяти.** Включает в себя все содержимое физической памяти на момент сбоя. Для записи такого дампа нужно, чтобы размер файла подкачки был равен как минимум размеру физической памяти плюс 1 Мбайт для заголовка. Драйверы устройств могут добавлять к дополнительным данным дампа памяти до 256 Мбайт, поэтому на всякий случай рекомендуется увеличить размер файла подкачки еще на

256 Мбайт. Данный вариант используется реже всего, так как в системах с большим объемом памяти файл подкачки получается слишком большим. При наличии более 2 Гбайт оперативной памяти режим создания полного дампа просто отключается, но его можно включить вручную, введя в командную строку с повышенными привилегиями следующую команду:

```
wmic recoveros set DebugInfoType=1
```

При включении режима создания полного дампа через приложение Wmic.exe WMI-компонент Win32 Provider присваивает параметру CrashDumpEnabled в разделе HKLM\SYSTEM\CurrentControlSet\Control\CrashControl значение 1. В процессе инициализации Windows проверяет, поместится ли полный дамп в файл подкачки, и при отрицательном результате проверки автоматически переключается на создание малого дампа памяти.

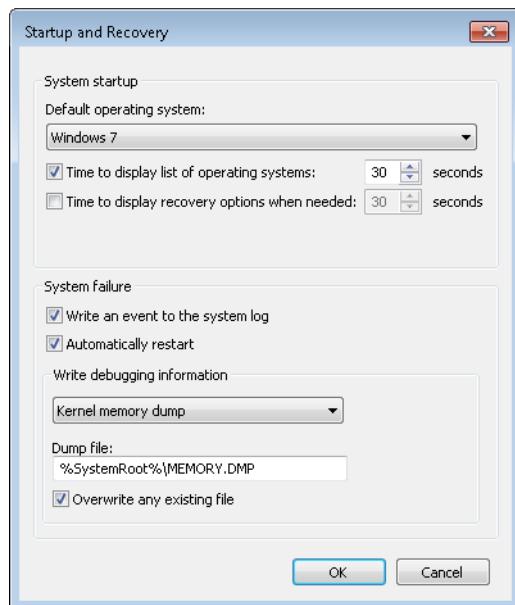


Рис. 14.4. Параметры аварийного дампа

- **Дамп памяти ядра.** Включает в себя только находившиеся в физической памяти на момент сбоя страницы в режиме ядра, место под которые выделено операционной системой и драйверами устройств. Страницы, связанные с пользовательскими процессами, сюда не входят. Но так как только код в режиме ядра может стать непосредственной причиной сбоя, вряд ли пользовательские страницы пригодятся при анализе причин сбоя. Кроме того, все используемые при анализе аварийного дампа структуры данных, в том числе список работающих процессов, стек текущего программного потока и перечень загруженных драйверов, хранятся в невыгружаемой памяти, содержимое которой входит в *дамп памяти ядра* (kernel memory dump).

Заранее предсказать объем дампа данного вида невозможно, так как он зависит от объема памяти в режиме ядра, выделенного операционной системой и существующими драйверами. Данный вариант по умолчанию используется как в клиентских, так и в серверных версиях Windows.

- **Малый дамп памяти.** Обычно его размер составляет от 128 Кбайт до 1 Мбайт, поэтому его называют *минимальным дампом* (triage dump). Он включает в себя стоп-код и параметры, список загруженных драйверов устройств, структуры данных, описывающие текущие процесс и программный поток (рассматривавшиеся в главе 5 *EPROCESS* и *ETHREAD*), стек ядра для ставшего причиной сбоя программного потока, и дополнительную память, которая потенциально может помочь при анализе аварийного дампа. В последнюю категорию попадают, например, страницы, содержащие адреса памяти и добавленные драйверами данные о сбое, на которые ссылаются регистры процессора.

ПРИМЕЧАНИЕ

Драйверы устройств могут зарегистрировать процедуру обратного вызова дополнительных данных дампа, вызвав функцию *KeRegisterBugCheckReasonCallback*. Ядро активирует эти обратные вызовы после сбоя, позволяя процедуре вставлять в дамп дополнительные сведения для упрощения отладки, например об аппаратной памяти устройства. Все драйверы системы могут совокупно добавить до 256 Мбайт данных. Точная величина зависит от того, сколько места требуется для хранения дампа, и от размера файла, в который он записывается. Каждый обратный вызов добавляет, по меньшей мере, одну восьмую доступного дополнительного пространства. После того как оно будет израсходовано, все остальные вызываемые драйверы не смогут добавлять свою информацию.

Отладчик отмечает, что доступная ему информация ограничена, когда он загружает минимальный дамп, а команде *!process* не хватает требуемых данных. Вот пример команды *!process* для минимального дампа:

```
Microsoft (R) Windows Debugger Version 6.12.0002.633 AMD64
Copyright (c) Microsoft Corporation. All rights reserved.
```

```
Loading Dump File [C:\Windows\Minidump\100911-22965-01.dmp]
Mini Kernel Dump File: Only registers and stack trace are available
...
0: kd> !process 0 0
**** NT ACTIVE PROCESS DUMP ****
GetPointerFromAddress: unable to read from fffff800030c5000
Error in reading nt!_EPROCESS at 0000000000000000
```

Больше информации содержит дамп памяти ядра, но переход к выводу пространства адресов другого процесса не сработает, так как требуемые данные в файле дампа попросту отсутствуют. Вот пример загрузки в отладчик дампа памяти ядра после попытки перейти в другое пространство адресов:

```
Microsoft (R) Windows Debugger Version 6.12.0002.633 AMD64
Copyright (c) Microsoft Corporation. All rights reserved.
```

```
Loading Dump File [C:\Windows\MEMORY.DMP]
```

продолжение ↗

```
Kernel Summary Dump File: Only kernel address space is available
...
0: kd> !process 0 0 explorer.exe
PROCESS ffffffa8009b47540 ...

0: kd> .process ffffffa8009b47540
Process ffffffa80`09b47540 has invalid page directories
```

Полный дамп памяти представляет собой расширенную версию двух предыдущих вариантов, но у него есть существенный недостаток. Его размер зависит от объема физической памяти в системе и, следовательно, может оказаться слишком большим. В большинстве случаев код и данные пользовательского режима для анализа причин сбоя не требуются (так как сбои возникают из-за проблем в памяти ядра, там же располагаются системные структуры данных), поэтому большая часть информации из полного дампа памяти бесполезна и только понапрасну занимает место. Кроме того, размер файла подкачки должен равняться объему физической памяти в системе, плюс 1 Мбайт для заголовка дампа, плюс еще 256 Мбайт для дополнительных данных о сбое. Так как требуемый размер файла подкачки в общем случае обратно пропорционален объему имеющейся физической памяти, указанное условие делает его неоправданно большим. Поэтому имеет смысл рассмотреть преимущества, предлагаемые малым дампом памяти или дампом памяти ядра.

Преимуществом минимального дампа является его небольшой размер, благодаря которому этот дамп удобно передавать по электронной почте. При каждом сбое в папке %SystemRoot%\Minidump появляется файл с уникальным именем, состоящим из даты, количества миллисекунд, прошедшего с момента запуска системы, и порядкового номера (например, 040712-24835-01.dmp). При возникновении конфликта имен система пытается создать дополнительное имя файла, вызвав функцию `GetTickCount` для получения новой временной метки и одновременно увеличив порядковый номер. По умолчанию Windows хранит последние 50 мини-дампов. Но можно изменить их количество, изменив параметр `MinidumpsCount` в разделе `HKLM\SYSTEM\CurrentControlSet\Control\CrashControl`.

К сожалению, для анализа мини-дампов требуются именно те образы, которые использовала сгенерировавшая их система. (Даже для самого простого анализа нужна как минимум копия `Ntoskrnl.exe`.) Если анализировать дамп предстоит не в той системе, где он был создан, это может стать проблемой. Однако на сервере символов Microsoft есть образы (и символы) для всех последних версий Windows, поэтому в отладчике можно указать путь к символу на сервере, после чего нужные образы загружаются автоматически. (Разумеется, образы для драйверов сторонних производителей там отсутствуют.)

Еще более существенным недостатком является ограниченный объем данных, сохраняемый в таком дампе, что зачастую становится помехой анализу. При этом воспользоваться минимальным дампом можно, даже если система настроена на генерацию дампа памяти ядра или полного дампа. Достаточно открыть имеющийся дамп в приложении `WinDbg` и воспользоваться для извлечения мини-дампа командой `.dump /m`. То есть мини-дамп автоматически создается, даже если вы указали, что нужно создавать дамп другой разновидности.

ПРИМЕЧАНИЕ

Команда .dump в программе LiveKd позволяет сгенерировать образ памяти работающей системы, который можно анализировать в автономном режиме без остановки работы. Это нужно в ситуации, когда проблему приходится решить без остановки работы. Так как различные области памяти извлекаются в разные моменты времени, дампы могут оказаться несогласованными. Для борьбы с этим явлением в программе LiveKd поддерживается флаг -m. Возможность получения зеркала дампа дает согласованный снимок памяти режима ядра. Для этого применяется соответствующий API-интерфейс диспетчера памяти, фиксирующий моментальный снимок системы. О способе использования инструмента LiveKd с виртуальными машинами Hyper-V читайте далее в описании эксперимента «Вывод списка виртуальных машин с помощью LiveKd» на с. 657.

Дамп памяти ядра является золотой серединой. Он содержит всю физическую память режима ядра, предоставляемую для анализа тот же самый объем информации, что и полный дамп, но при этом имеет значительно меньший размер, так как в него не включаются код и данные пользовательского режима. Например, в 64-разрядной версии Windows с 4 Гбайт RAM объем дампа памяти ядра составляет 294 Мбайт.

Когда вы задаете параметры дампа памяти ядра, система проверяет размер файла подкачки. Общие рекомендации даны в табл. 14.1, но это всего лишь оценочные размеры, так как точно предсказать величину будущего дампа невозможно. Дело в том, что он зависит от объема используемой операционной системой памяти в режиме ядра и установленных на момент сбоя драйверов.

Соответственно, на момент сбоя файл подкачки может оказаться слишком маленьким для дампа ядра, поэтому система генерирует минимальный дамп. Чтобы узнать размер дампа ядра вашей системы, инициируйте системный сбой. Настройте систему таким образом, чтобы это можно было сделать через консоль или при помощи программы Notmyfault. (Работу с этой программой мы рассмотрим чуть позже.) После перезагрузки вы сможете проверить, сгенерирован ли дамп памяти ядра, и по его размеру оценить, каким должен быть файл подкачки. Для надежности в 32-разрядных системах его размер может составлять 2 Гбайт плюс 256 Мбайт, так как именно 2 Гбайт составляет максимальный размер адресного пространства режима ядра (если, конечно, загрузка не осуществляется с параметром increaseuserserv, когда этот размер сокращается до 1 Гбайт). Если на загрузочном томе недостаточно места для файла Memory.dmp, его можно поместить на любой другой жесткий диск через диалоговое окно, показанное на рис. 14.4.

Таблица 14.1. Минимальные размеры файла подкачки для дампов ядра

Размер системной памяти, Гбайт	Минимальный размер файла подкачки, Мбайт
< 4	200
8	400
≥ 8	800

Чтобы ограничить место, занимаемое аварийными дампами, Windows требуется понять, следует ли сохранять последнюю копию полного дампа или дампа ядра. Сообщив

об ошибке ядра (об этом мы поговорим чуть позже), Windows применяет следующий алгоритм, чтобы определить, оставлять ли файл `Memory.dmp`. В серверных системах он всегда сохраняется. В клиентских системах он по умолчанию сохраняется только на машинах, находящихся в домене. В остальных случаях сохранение происходит только при наличии более 25 Гбайт свободного места на соответствующем томе. Если из-за нехватки места на диске система не может сохранить копию аварийного дампа, в журнале событий `System` появляется запись об удалении файла дампа, как показано на рис. 14.5. Это поведение можно переопределить, создав параметр `DWORD` в разделе `HKLM\SYSTEM\CurrentControlSet\Control\CrashControl\AlwaysKeepMemoryDump` и присвоив ему значение 1. В этом случае Windows будет сохранять аварийный дамп вне зависимости от объема свободного места.

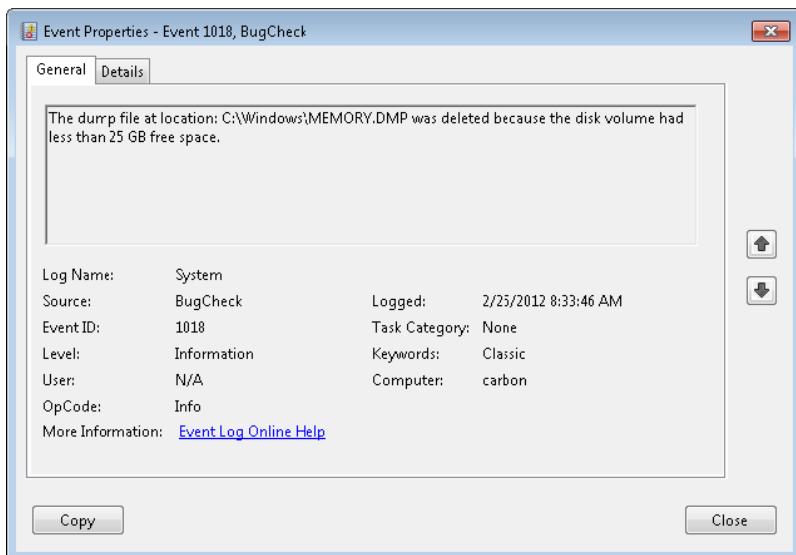


Рис. 14.5. Запись в журнале об удалении файла дампа

ЭКСПЕРИМЕНТ: ПРОСМОТР ИНФОРМАЦИИ В ФАЙЛЕ ДАМПА

Каждый файл аварийного дампа содержит заголовок, в котором указывается стоп-код и его параметры, тип сбойной системы (в том числе данные о ее версии) и список указателей на необходимые для анализа структуры режима ядра. Также в заголовок входят сведения о типе записанного аварийного дампа и информация, характерная для дампа такого типа. Для просмотра заголовка воспользуйтесь командой `.dumpdebug` отладчика. К примеру, вот результат действия этой команды в системе, настроенной на создание дампа ядра:

```
0: kd> .dumpdebug
----- 64 bit Kernel Summary Dump Analysis

DUMP_HEADER64:
MajorVersion 00000000
```

```
MinorVersion 00001db1
KdSecondaryVersion 00000000
DirectoryTableBase 00000001`ad6a2000
PfnDataBase ffffffa80`00000000
PsLoadedModuleList ffffff800`02a47670
PsActiveProcessHead ffffff800`02a29350
MachineImageType 00008664
NumberProcessors 00000002
BugCheckCode 000000d1
BugCheckParameter1 ffffff8a0`027475c0
BugCheckParameter2 00000000`00000002
BugCheckParameter3 00000000`00000000
BugCheckParameter4 ffffff880`0343a361
KdDebuggerDataBlock ffffff800`029f30a0
SecondaryDataState 00000000
ProductType 00000001
SuiteMask 00000110

SUMMARY_DUMP64:
DumpOptions 504d4453
HeaderSize 00049000
BitmapSize 00230000
Pages 000151f0
Bitmap.SizeOfBitMap 00230000

KiProcessorBlock at ffffff800`02ab1c40
2 KiProcessorBlock entries:
ffffff800`029f4e80 ffffff880`009ec180
```

Команда .enumtag позволяет получить все дополнительные данные аварийного дампа. Выводятся тег, размер данных и сами данные (в байтах и в формате ASCII) для каждого обратного вызова дополнительных данных. Разработчики при помощи API-интерфейса Debugger Extension могут создавать расширения отладчика, предназначенные, в том числе, для чтения дополнительных данных дампа. (Более подробно об этом рассказывается в разделе Debugging Tools файла помощи Windows.)

```
0: kd> .enumtag
{270A33FD-3DA6-460D-BA93C1BAE21E39B} - 0xfc8 bytes
09 00 00 00 00 00 00 48 00 00 00 13 00 00 00 .....H.....
48 08 00 00 14 00 00 00 C8 0F 00 00 15 00 00 00 H.....
C8 0F 00 00 17 00 00 00 00 00 00 00 00 00 00 00 .....
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00 00 00 00 00 00 00 EF B2 01 00 00 00 00 00 00 00 .....
...
```

Генерация аварийного дампа

При загрузке система получает параметры аварийного дампа из раздела HKLM\SYSTEM\CurrentControlSet\Control\CrashControl реестра. Если генерация дампа задана, создается копия драйвера мини-порта диска, через которую том записывается в память. Этой копии присваивается имя мини-порта с приставкой «dump». Также система запрашивает значение DumpFilters для всех необходимых при записи тома фильтрующих драйверов. Например, для драйвера BitLocker Drive Encryption Crashdump Filter это Dumpfve.sys.

(Шифрование диска с помощью BitLocker рассматривалось в главе 9.) Кроме того, система собирает сведения, относящиеся к участвующим в записи аварийного дампа компонентам, к которым в числе прочего относятся имя драйвера мини-порта диска, необходимые для записи дампа структуры диспетчера ввода-вывода и местоположение файла подкачки на диске. Две копии этих данных сохраняются в структурах, находящихся в контексте дампа.

При системном сбое драйвер аварийного дампа (%SystemRoot%\System32\Drivers\Crashdump.sys) проверяет целостность двух находящихся в контексте дампа структур, полученных в процессе загрузки путем сравнения памяти. При отсутствии совпадений аварийный дамп не записывается, так как запись, скорее всего, окажется безуспешной или даже приведет к повреждению диска. Если же проверка проходит успешно, Crashdump.sys вместе с драйвером мини-порта диска и всеми необходимыми фильтрующими драйверами записывают дамп непосредственно в занятые файлом подкачки секторы диска, в обход драйвера файловой системы и стека драйверов внешней памяти (которые могут быть повреждены и являться причиной сбоя).

ПРИМЕЧАНИЕ

Файл подкачки становится доступным для аварийного дампа на ранних стадиях загрузки системы, поэтому там можно найти сведения о большинстве сбоев, вызванных ошибками в инициализации запускающих систему драйверов. Если же сбой возникает в таком компоненте, как HAL, или при инициализации драйверов загрузки, файл подкачки еще недоступен, поэтому единственным способом проанализировать аварийный дамп является использование дополнительного компьютера для отладки процесса запуска операционной системы. (Эта процедура рассматривается далее в эксперименте «При соединение отладчика ядра».)

В процессе загрузки операционной системы диспетчер сеансов (Smss.exe) проверяет параметр HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\ExistingPageFiles реестра на наличие списка существующих файлов подкачки от предыдущей загрузки. (О файлах подкачки рассказывается в главе 10.) Затем для каждого из файлов списка вызывается функция SmpCheckForCrashDump, проверяющая наличие внутри данных аварийного дампа. В заголовках файлов подкачки она ищет сигнатуры PAGEDUMP и PAGEDU64 для 32-разрядной и 64-разрядной систем соответственно. Обнаружение такой сигнатуры указывает на наличие информации из аварийного дампа. В этом случае диспетчер сеансов считывает из раздела HKLM\SYSTEM\CurrentControlSet\Control\CrashControl реестра набор параметров сбоя, в число которых входит и имя целевого файла дампа (обычно это %SystemRoot%\Memory.dmp, если в параметрах не указан другой вариант).

Затем диспетчер сеансов проверяет, находится ли файл дампа и файл подкачки на разных томах. Если это так, он проверяет наличие на целевом томе свободного места (размер, необходимый для аварийного дампа, указан в заголовке дампа в файле подкачки), а затем обрезает файл подкачки до размера аварийного дампа и присваивает ему имя временного дампа. (Позднее, когда диспетчер сеансов вызовет функцию NtCreatePagingFile, будет создан новый файл подкачки.) Имя файла временного дампа имеет формат DUMPxxxx.tmp, где xxxx – текущее младшее слово для счетчика тактов системы. (Для поиска значения, не вступающего в конфликт с уже существующими,

система делает до 100 попыток.) После переименования файла подкачки система убирает из него атрибуты `hidden` и `system` и задает дескрипторы безопасности для защиты аварийного дампа.

Затем диспетчер сеансов создает изменяемый раздел `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\CrashControl\MachineCrash` реестра и сохраняет имя временного файла дампа в параметре `DumpFile`. После этого `DWORD` записывается в параметр `TempDestination`, указывая, что местоположение файла дампа является временным. Если же файл подкачки и окончательный файл аварийного дампа находятся на одном и том же томе, временный файл дампа не создается. Вместо этого файл подкачки обрезается и непосредственно переименовывается в файл дампа. В этом случае параметр `DumpFile` будет иметь в качестве значения итоговый файл дампа, а параметр `TempDestination` будет равен нулю.

На более поздних стадиях загрузки процесс `Wininit` проверяет наличие раздела `MachineCrash`, и если он существует, `Wininit` запускает процесс `WerFault` (о нем мы поговорим в следующем разделе), который считывает параметры `TempDestination` и `DumpFile`. Если параметр `TempDestination` имеет значение 1, что указывает на использование временного файла, `WerFault` перемещает временный файл в то место, где он должен оказаться в итоге, и обеспечивает его безопасность, оставив доступ только для учетной записи `System` и локальной группы `Administrators`. Затем процесс `WerFault` записывает окончательное имя файла дампа в переменную `FinalDumpFileLocation` раздела `MachineCrash`. Схематично все эти шаги представлены на рис. 14.6.

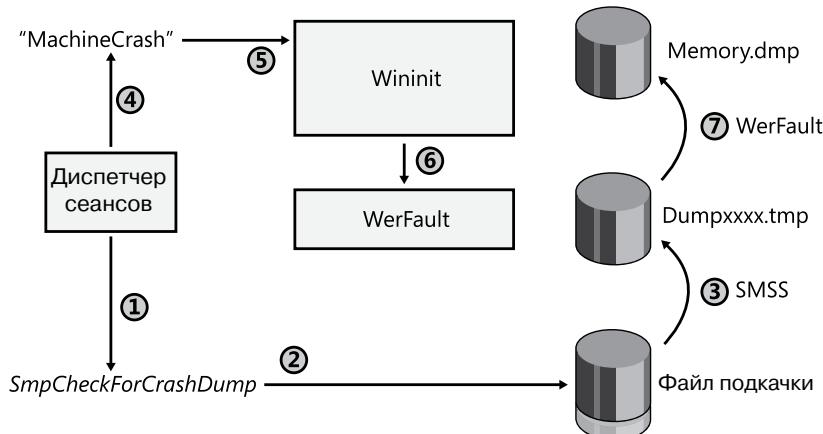


Рис. 14.6. Генерация файла аварийного дампа

Для дополнительного контроля над местом записи файла дампа, например в системах, загружающихся с SAN, или в системах, у которых на томе с файлом подкачки недостаточно свободного места, Windows поддерживает *выделенный файл дампа памяти* (*dedicated dump file*), настраиваемый в параметрах `DedicatedDumpFile` и `DumpFileSize` раздела `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\CrashControl`. В этом случае драйвер аварийного дампа создает файл указанного размера и записывает данные о сбое сюда,

а не в файл подкачки. При отсутствии параметра DumpFileSize Windows создает выделенный файл дампа такого размера, чтобы там можно было сохранить полный дамп. Требуемый размер Windows вычисляет как размер общего числа физических страниц памяти в системе плюс место под заголовок дампа (одна страница в 32-разрядных системах и две в 64-разрядных), плюс максимальный размер дополнительных данных аварийного дампа, то есть еще 256 Мбайт. Если в конфигурации указан полный дамп или дамп памяти, но на целевом томе недостаточно места для создания выделенного файла дампа требуемого размера, система запишет минимальный дамп.

Передача в Microsoft отчетов об ошибках

Как упоминалось в главе 3 части I, в Windows входит система отчетов об ошибках в Windows (Windows Error Reporting, WER), автоматически передающая данные о сбоях процессоров и системы (таких, как прекращение работы или зависание) в Microsoft (или на внутренний сервер отчетов об ошибках) для анализа. Эта система по умолчанию включена, но такое поведение можно изменить. Ведь WER при перезагрузке после сбоя проверяет, задан ли в параметрах системы режим отправления аварийного дампа в Microsoft (или на закрытый сервер, о котором мы поговорим в разделе «Анализ сбоев через Интернет»). Основная страница Problem Reporting Settings (Параметры отчетов о проблемах), для перехода к которой нужно выбрать в Панели управления раздел Action Center (Центр поддержки) и перейти по ссылке Change Action Center Settings (Настройки центра поддержки), показана на рис. 14.7. Именно здесь задаются параметры для системы отчетов об ошибках в Windows.

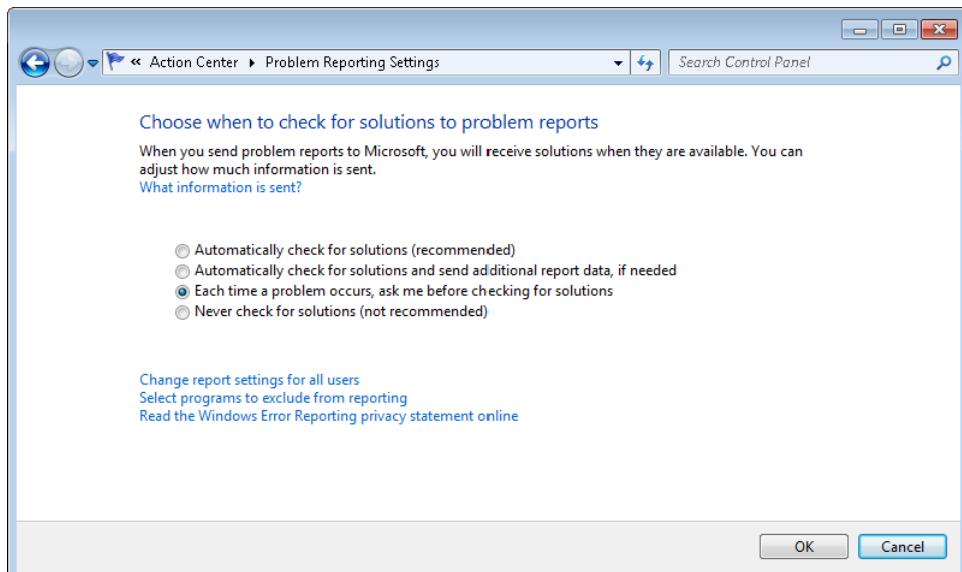


Рис. 14.7. Страница настройки системы отчетов об ошибках

Как уже упоминалось, если процесс Wininit.exe обнаруживает раздел HKLM\SYSTEM\CurrentControlSet\Control\CrashControl\MachineCrash, он запускает WerFault.exe с флагами -k, -c (флаг k указывает на отчет о проблемах ядра, а флаг c означает, что полный дамп и дамп ядра следует преобразовать в минимальный дамп), заставляя его проверять файл аварийного дампа в режиме ядра. Вот как процесс WerFault готовится к отправке отчета о сбое на поддерживаемый Microsoft сайт анализа сбоев через Интернет (Online Crash Analysis, OCA) или, если это указано в параметрах, на внутренний сервер отчетов об ошибках:

1. Если генерированный дамп не является минимальным, из файла извлекается минидамп и сохраняется в заданном по умолчанию каталоге %SystemRoot%\Minidump. Место его сохранения можно изменить через параметр MinidumpDir раздела HKLM\SYSTEM\CurrentControlSet\Control\CrashControl.
2. Имя мини-дампа записывается в параметр HKLM\SOFTWARE\Microsoft\Windows\Windows Error Reporting\KernelFaults\Queue.
3. Команда запуска процесса WerFault (%SystemRoot%\System32\WerFault.exe) с флагами -k, -qr (флаг qr указывает на использование очереди отчетов и перезапуск WerFault) добавляется в раздел HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\RunOnce, чтобы процесс WerFault для отправки отчета об ошибках запускался при первом входе пользователя в систему.

Анализ сбоев через Интернет

Когда во время входа в систему запускается служебная программа WerFault, она конфигурируется и запускается снова с флагами -k, -q (флаг q указывает на режим постановки отчетов в очередь), а работа предыдущего экземпляра программы завершается. Это делается, чтобы оболочка Windows не ждала, пока процесс WerFault по возможности быстро вернет управление процессу RunOnce. После нового запуска WerFault проверяет раздел HKLM\SOFTWARE\Microsoft\Windows\Windows Error Reporting\KernelFaults\Queue реестра на наличие в очереди отчетов, которые могли быть добавлены при преобразовании предыдущего дампа. Также проверяется наличие неотправленных отчетов о сбоях из предыдущих сеансов. При наличии таковых WerFault.exe генерирует два файла в формате XML:

- ❑ Первый файл содержит базовое описание системы, включая версию операционной системы, а также списки установленных драйверов и устройств.
- ❑ Во втором файле находятся метаданные, используемые службой ОСА, в том числе тип события, инициирующего запуск WER, и дополнительные сведения о конфигурации, например данные о производителе системы.

Если в соответствии с настройкой требуется ввод данных пользователем (этот вариант настройки предлагается по умолчанию), появляется показанное на рис. 14.8 диалоговое окно с вопросом, хочет ли пользователь поискать решение проблемы в Интернете. Если пользователь отвечает положительно и это не противоречит групповым политикам, WerFault отправляет копию двух XML-файлов и мини-дампа на

сайт <https://oca.microsoft.com>, откуда данные пересылаются ферме серверов для автоматического анализа.

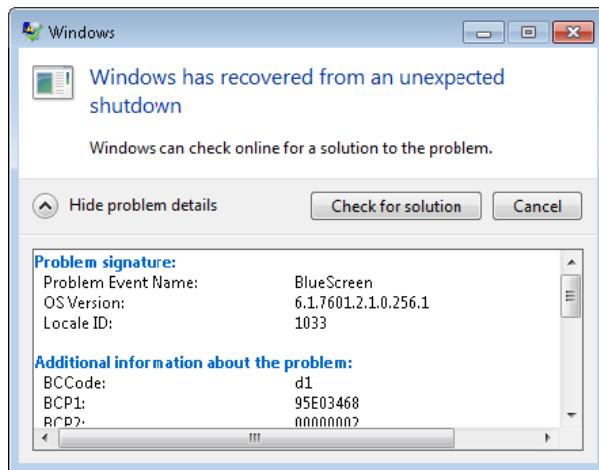


Рис. 14.8. Диалоговое окно, предлагающее отправить отчет об ошибке

На ферме серверов для автоматического анализа используется тот же механизм, что и в отладчиках ядра производства Microsoft, в которые можно загрузить аварийный дамп (об этом мы еще поговорим). При анализе генерируется так называемый *идентификатор корзины* (bucket ID), который представляет собой сигнатуру, идентифицирующую определенный тип сбоя. Ферма серверов отправляет запрос к базе данных, пытаясь по идентификатору найти решение проблемы, ставшей причиной сбоя. Затем процессу WerFault отправляется URL-адрес WER-сайта (<https://wer.microsoft.com>). Все решения доступны на странице Action Center (Центр поддержки) Панели управления в разделе System And Security (Безопасность). Когда происходит поиск решения, центр поддержки использует для открытия WER-сайта с отчетом о предварительном анализе сбоя окно интернет-браузера. Если решение найдено, на странице появляется инструкция для получения исправления, обновления или драйвера стороннего производителя.

Базовый анализ аварийного дампа

Если сайт анализа сбоев через Интернет (ОСА) не сможет найти решение проблемы или у вас не будет возможности отправить туда аварийный дамп, проанализировать дамп можно самостоятельно. Как уже упоминалось, после загрузки аварийного дампа в отладчик WinDbg или Kd анализ выполняется по тому же механизму, что и на сайте ОСА, и иногда этого оказывается вполне достаточно для определения источника проблемы. То есть если вам повезет, решение найдется при автоматическом анализе. Впрочем, даже в случае неудачи остаются простые методики, позволяющие найти причину сбоя.

В этом разделе показано, как выполнить базовый анализ аварийного дампа и использовать инструмент Driver Verifier (см. главу 8) для перехвата сбойных драйверов и определения их местоположения при анализе аварийного дампа.

ПРИМЕЧАНИЕ

Автоматизированный анализ на сайте ОСА с большой вероятностью покажет причину сбоя, но не предоставит информации о том, какой драйвер подозревается в сбое. Дело в том, что в базе данных ОСА находятся только отчеты о причинах сбоя с идентификатором корзины. Новые записи в базу добавляются только после проверки причины специалистами Microsoft. При отсутствии в отчете идентификатора корзины ОСА отвечает, что причиной сбоя стал «неизвестный драйвер».

Программа Notmyfault

Различные сбои, о которых рассказывается в этой главе, можно вызвать при помощи программы Notmyfault, созданной в Windows Sysinternals (<http://technet.microsoft.com/en-us/sysinternals/bb963901>). Она состоит из исполняемого файла Notmyfault.exe и драйвера Myfault.sys. После запуска этот файл загружает драйвер и выводит показанное на рис. 14.9 диалоговое окно, позволяющее различными способами прекратить работу системы или вызвать ее зависание, а также заставить драйвер инициировать утечку памяти из выгружаемого или невыгружаемого пула. Предлагаются наиболее распространенные (по статистике службы поддержки Microsoft) типы сбоев. После выбора варианта и щелчка на кнопке Crash, Hang, Leak Paged или Leak Nonpaged исполняемый файл через API DeviceIoControl укажет драйверу, ошибка какого типа должна произойти.

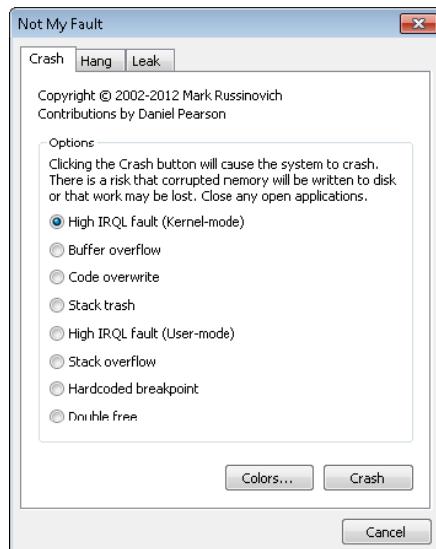


Рис. 14.9. NotMyFault

ПРИМЕЧАНИЕ

Инициировать при помощи приложения Notmyfault сбои желательно на тестовой системе или на виртуальной машине, так как существует незначительный риск сбрасывания поврежденной памяти на диск, что ведет к повреждению файла или диска.

ПРИМЕЧАНИЕ

Имена исполняемого файла и драйвера Notmyfault подчеркивают (дословно — «не моя вина»), что режим пользователя не может быть непосредственной причиной сбоя. Исполняемый файл Notmyfault вызывает сбой только путем загрузки драйвера, который выполняет запрещенную операцию в режиме ядра.

Базовый анализ

Самый простой сбой в приложении Notmyfault вызывается установкой переключателя High IRQL Fault (Kernel-Mode) и щелчком на кнопке Crash. После этого драйвер выделяет страницу в пуле подкачиваемой памяти, увеличивает IRQL-уровень до DPC/dispatch и обращается к освобожденной странице. (Об IRQL речь идет в главе 3 части I.) Если это не приводит к сбою, процесс продолжает считывать память после конца страницы, пока не произойдет сбой из-за обращения к недействительной странице. В результате драйвер выполняет несколько недопустимых операций:

1. Сылается на память, которая ему не принадлежит.
2. Обращается к пулу подкачиваемой памяти при IRQL-уровне DPC/dispatch и выше, что недопустимо, так как на этих уровнях ошибки страниц не разрешаются.
3. Выходит за границы выделенной области памяти и пытается обратиться к памяти, которая потенциально может быть недействительной.

Первое обращение к странице не всегда приводит к сбою, потому что освобожденная драйвером страница может остаться в системном рабочем наборе. (Системные рабочие наборы рассматриваются в главе 10.)

После загрузки сгенерированного этим сбоем аварийного дампа в отладчик WinDbg вы увидите примерно такой результат:

```
Microsoft (R) Windows Debugger Version 6.12.0002.633 AMD64
Copyright (c) Microsoft Corporation. All rights reserved.
```

```
Loading Dump File [C:\Windows\MEMORY.DMP]
Kernel Complete Dump File: Full address space is available
```

```
Symbol search path is: srv*c:\symbols*http://msdl.microsoft.com/download/symbols
Executable search path is:
Windows 7 Kernel Version 7601 (Service Pack 1) MP (2 procs) Free x86 compatible
Product: WinNt, suite: TerminalServer SingleUserTS
Built by: 7601.17514.x86fre.win7sp1_rtm.101119-1850
Machine Name:
Kernel base = 0x82814000 PsLoadedModuleList = 0x8295e850
Debug session time: Wed Mar 21 08:12:50.194 2012 (UTC - 7:00)
System Uptime: 8 days 8:54:38.580
Loading Kernel Symbols
```

```
.....  
Loading User Symbols  
.....  
Loading unloaded module list  
.....  
*****  
* *  
* Bugcheck Analysis *  
* *  
*****  
Use !analyze -v to get detailed debugging information.  
  
BugCheck D1, {946ae800, 2, 0, 91df15ab}  
  
*** ERROR: Module load completed but symbols could not be loaded for myfault.sys  
Probably caused by : myfault.sys ( myfault+5ab )  
  
Followup: MachineOwner  
-----
```

Прежде всего, следует заметить, что WinDbg сообщает об ошибках при загрузке символов для драйвера Myfault.sys. Этого можно было ожидать, так как файлы символов для него хранятся не по заданному в параметрах пути (который указывает на сервер символов, принадлежащий Microsoft). Аналогичные ошибки возникают для драйверов сторонних производителей, не поставляющихся вместе с операционной системой.

Текст с результатами анализа достаточно краток. Показан числовой стоп-код и параметры проверки ошибок, за которым следует строка «Probably caused by». В ней указан драйвер, который с точки зрения анализирующего механизма является наиболее вероятной причиной сбоя. В нашем случае указывается драйвер Myfault.sys, поэтому нет нужды проводить анализ вручную.

Строка «Followup», как правило, не несет полезной информации. Представленные там данные используются в Microsoft, когда отладчик ищет имя модуля в файле Triage.ini, который находится в папке Triage установочного каталога Debugging Tools for Windows. В версии этого файла для внутреннего использования Microsoft перечислены разработчики или группы, ответственные за обработку сбоев в отдельных драйверах. Именно их имена отладчик выводит в строке «Followup».

Детальный анализ

После того как при базовом анализе в программе Notmyfault идентифицируется сбойный драйвер, нужно провести более детальное исследование, выполнив в отладчике команду:

```
!analyze -v
```

Первым бросающимся в глаза отличием детального анализа является описание стоп-кода и его параметров. Вот результат действия указанной команды для того же самого дампа:

```
DRIVER_IRQL_NOT_LESS_OR_EQUAL (d1)  
An attempt was made to access a pageable (or completely invalid) address at an  
interrupt request level (IRQL) that is too high. This is usually caused by drivers  
продолжение ↴
```

```
using improper addresses.
If kernel debugger is available get stack backtrace.
Arguments:
Arg1: 946ae800, memory referenced
Arg2: 00000002, IRQL
Arg3: 00000000, value 0 = read operation, 1 = write operation
Arg4: 91df15ab, address which referenced memory
```

Это избавляет вас от необходимости открывать для получения данной информации файл помощи. Кроме того, иногда выводимый текст содержит рекомендации по устранению неисправностей. Пример такого текста приведен в следующем разделе, в котором рассматривается углубленный анализ дампов.

Другой выводимой при детальном анализе потенциально полезной информацией являются трассировочные данные стека программного потока, выполнявшегося в момент сбоя. Вот как они выглядят для того же самого полного дампа:

```
STACK_TEXT:
93cdbb3c 91df15ab badb0d00 84f3e380 946ad800 nt!KiTrap0E+0x2cf
WARNING: Stack unwind information not available. Following frames may be wrong.
93cdbbb8 91df19db 86d77900 93cdbbfc 91df1b26 myfault+0x5ab
93cdcbc4 91df1b26 85e38488 00000001 00000000 myfault+0x9db
93cdbbbfc 8284b593 86c9a510 86d77900 86d77900 myfault+0xb26
93cdcbc14 82a3f99f 85e38488 86d77900 86d77970 nt!IoCallDriver+0x63
93cdcbc34 82a42b71 86c9a510 85e38488 00000000 nt!IoSyncrhonousServiceTail+0x1f8
93cdbcd0 82a893f4 86c9a510 86d77900 00000000 nt!IoPxxxControlFile+0x6aa
93cdcbd04 828521ea 000000c4 00000000 00000000 nt!NtDeviceIoControlFile+0x2a
93cdcbd04 77af70b4 000000c4 00000000 00000000 nt!KiFastCallEntry+0x12a
0009f370 77af5864 75cb989d 000000c4 00000000 ntdll!KiFastSystemCallRet
0009f374 75cb989d 000000c4 00000000 00000000 ntdll!NtDeviceIoControlFile+0xc
0009f3d4 77a1a671 000000c4 83360018 00000000 KERNELBASE!DeviceIoControl+0xf6
0009f400 00c421f9 000000c4 83360018 00000000 kernel32!DeviceIoControlImplementation+
0x80
0009f4a0 7749c4e7 000201ec 00000111 000003f9 NotMyfault+0x21f9
```

Как демонстрирует приведенный стек, показанный внизу образ исполняемого файла **Notmyfault** вызвал функцию **DeviceIoControlImplementation** в **Kernel32.dll**, которая, в свою очередь, вызвала функцию **DeviceIoControl** в **Kernelbase.dll** и так далее, пока при выполнении инструкции в образе **Myfault** не случился системный сбой. Подобные этому стеки вызовов могут оказаться полезными, так как иногда причиной сбоя является передача одному драйверу другим неверно отформатированных или поврежденных данных или недопустимых параметров. Передавший некорректные данные драйвер может стать причиной сбоя, и именно на него укажет анализ, в то время как стек покажет, что в дело замешан еще один драйвер. В данном простом примере в стеке фигурирует только драйвер **Myfault** (модуль **nt** — это **Ntoskrnl**).

Если выявленный при анализе драйвер вам неизвестен, воспользуйтесь командой **lm** (аббревиатура от **list modules** — список модулей) для просмотра сведений о его версии. Укажите после имени драйвера параметры **k** (модули ядра), **v** (подробный анализ) и **m** (совпадение):

```
0: kd> lm kv m myfault
start end module name
91df1000 91df2880 myfault (no symbols)
Loaded symbol image file: myfault.sys
```

```
Image path: \??\C:\Windows\system32\drivers\myfault.sys
Image name: myfault.sys
Timestamp: Sat Apr 07 09:34:40 2012 (4F806CA0)
CheckSum: 00003871
ImageSize: 00001880
File version: 4.0.0.0
Product version: 4.0.0.0
File flags: 0 (Mask 3F)
File OS: 40004 NT Win32
File type: 3.7 Driver
File date: 00000000.00000000
Translations: 0409.04b0
CompanyName: Sysinternals
ProductName: Sysinternals Myfault
InternalName: myfault.sys
OriginalFilename: myfault.sys
ProductVersion: 4.0
FileVersion: 4.0 (sysinternals.com)
FileDescription: Crash Test Driver
LegalCopyright: Copyright © 2002-2012 Mark Russinovich
```

Перед тем как приступить к дальнейшему анализу сбоев, убедитесь, что у вас установлены последние версии ядра и драйверов, воспользовавшись службой Windows Update и сайтами поддержки драйверов сторонних производителей.

Описание поможет вам понять предназначение драйвера, а номера файла и версии покажут, установлена ли у вас самая актуальная версия. Если номер версии отсутствует (так как на момент сбоя он может быть удален из физической памяти), посмотрите свойства образа драйвера через Проводник.

Чтобы заставить Windows Update найти более новую версию драйвера, откройте диспетчер устройств и найдите устройство, с которым связан нужный вам драйвер. Щелкните на его имени правой кнопкой мыши и выберите в появившемся меню команду **Update Driver Software** (Обновить драйверы). Если служба Windows Update сообщает об отсутствии более новой версии драйвера, имеет смысл проверить сайт производителя оборудования. А если и там вы не найдете последней версии драйвера, посетите сайт производителя драйвера.

Инструменты устранения сбоев

Вызванный нами в предыдущем разделе при помощи приложения Notmyfault сбой был без проблем проанализирован отладчиком. К сожалению, в большинстве случаев проанализировать сбой сложно или даже невозможно. Существует несколько уровней проверки, позволяющих извлечь информацию из непригодного для анализа дампа, причем степень сложности уровня пропорциональна падению производительности системы. Если после настройки системы в соответствии с требованиями одного уровня и перезагрузки вам не удалось обнаружить причину сбоя, переходите на следующий уровень.

- Если вы считаете причиной сбоя один или несколько драйверов, так как они были установлены или обновлены относительно недавно или же на них указывают обстоятельства сбоя, проверьте их инструментом Driver Verifier, используя все

варианты, кроме имитации нехватки ресурсов. (Инструмент Driver Verifier рассматривается в главе 8.)

- Если на компьютере установлена 32-разрядная версия Windows, включите указанный в предыдущем пункте уровень проверки для всех неподписанных драйверов. (В 64-разрядной системе все драйверы должны быть подписаными, хотя это ограничение можно снять в процессе загрузки, нажав клавишу F8 и выбрав в меню вариант Disable Driver Signature Enforcement.)
- Включите этот же уровень проверки для всех драйверов системы. Для сохранения приемлемой производительности можно разбить драйверы на группы и в промежутках между перезагрузками запускать Driver Verifier для очередной группы.

ПРИМЕЧАНИЕ

Если обнаруженная инструментом Driver Verifier ошибка драйвера сделала систему незагружаемой, загрузите ее в безопасном режиме (в котором проверка отключена), запустите Driver Verifier и удалите параметры проверки.

Далее вы увидите, каким образом инструмент Driver Verifier превращает дампы, непригодные для отладки, в дампы, доступные для анализа.

Переполнение буфера, повреждение памяти и особый пул

Одной из самых распространенных причин сбоя в Windows является повреждение пула. Обычно оно возникает, когда драйвер сталкивается с ошибкой переполнения или опустошения буфера, в результате которой данные записываются до начала или после конца буфера, выделенного в выгружаемом или невыгружаемом пуле. Структуры управления пулами исполнительной системы располагаются по сторонам от буфера и отделяют один буфер от другого. То есть подобные ошибки приводят к повреждению структур управления пулами или буферов других драйверов либо к тому и другому одновременно. Отследить переполнение буфера часто можно, исследовав окружающие теги пула командой !pool. Найдите адрес повреждения и выполните команду !pool *адрес_повреждения*. Это позволит вывести на экран варианты распределения пула, расположенные на одной странице с повреждением. Посмотрите на левый столбец, определите диапазон поврежденных адресов, обратите внимание на все, что выделено до них, и определите все теги пула. Скорее всего, это и будет причиной переполнения буфера. Можно воспользоваться файлом Pooltag.txt из папки Triage, расположенной в установочном каталоге Debugging Tools for Windows для поиска драйвера, которому принадлежит данный тег пула, или применить для этой цели программу Strings, созданную в Sysinternals.

Повреждения пула также возникают при записи драйвером в освобожденный пул, владельцем которого он раньше являлся. Это называется *ошибкой использования освобожденной памяти* (use after free bug), а ее причиной обычно становится состояние гонок в драйвере. Такие ошибки особенно сложно обнаружить, так как повредивший память драйвер больше не имеет прослеживаемой привязки к этой памяти, такой как

соседние теги пула в случае переполнения буфера. Другой распространенной причиной повреждения памяти является *прямой доступ к памяти* (Direct Memory Access, DMA). Он возникает, когда устройство осуществляет запись непосредственно в RAM, минуя драйвер, хотя именно драйвер отвечает за координирование этого процесса, выделяя устройству память под запись и программируя аппаратные регистры устройства деталями операции. При наличии в драйвере ошибки, из-за которой память освобождается до того, как устройство успеет осуществить запись, эту память может получить другой драйвер или даже пользовательское приложение, которое, разумеется, не ожидает обращений со стороны устройства.

Сбои, вызванные повреждением пула, практически невозможно отследить, так как авария происходит в момент обращения к поврежденным данным, а не в момент их повреждения. Но есть методы, позволяющие хотя бы приблизительно догадаться, что стало причиной повреждения памяти. Прежде всего следует попытаться определить размер поврежденного участка, исследовав некорректные данные. Если поврежден один бит, скорее всего, причиной являются проблемы с RAM или процессором. Небольшие повреждения могут возникать как из-за аппаратного, так и из-за программного обеспечения, и найти точную причину в этом случае практически невозможно. А при значительных повреждениях нужно искать повторяющиеся элементы, такие как строки (например, информация из HTTP-пакетов, содержимое текстовых файлов и т. п.).

ПРИМЕЧАНИЕ

Чтобы облегчить поиск повреждений пула, Windows проверяет целостность структур управления пулем и непосредственных соседей буфера при каждом выделении и освобождении памяти. Соответственно, переполнение буфера, скорее всего, будет распознано практически сразу после повреждения и ассоциировано со сбоем, стоп-код которого BAD_POOL_HEADER (0x19).

Сбой, связанный с переполнением буфера, можно сымитировать, запустив программу Notmyfault и установив переключатель Buffer Overflow. В этом случае драйвер Myfault выделит память под буфер и перепишет следующие за буфером 48 байт. Между щелчком на кнопке Crash и сбоем может пройти довольно много времени, возможно, вам потребуется даже задействовать пул, запустив какие-либо приложения. Это еще раз подчеркивает, что повреждение влияет на стабильность системы отнюдь не мгновенно. Проведенный при такой ошибке анализ аварийного дампа почти всегда показывает, что проблема связана с Ntoskrnl или другим драйвером. То есть наглядно демонстрируется польза подробного анализа с его описанием стоп-кодов:

DRIVER_CORRUPTED_EXPPOOL (c5)

An attempt was made to access a pageable (or completely invalid) address at an interrupt request level (IRQL) that is too high. This is caused by drivers that have corrupted the system pool. Run the driver verifier against any new (or suspect) drivers, and if that doesn't turn up the culprit, then use gflags to enable special pool.

Arguments:

Arg1: 4f4f4f53, memory referenced

Arg2: 00000002, IRQL

Arg3: 00000000, value 0 = read operation, 1 = write operation

Arg4: 829234a7, address which referenced memory

В описании дается совет запустить для всех новых или подозрительных драйверов Driver Verifier или активировать особый пул с помощью отладчика Gflags. В обоих случаях преследуется одна цель: выявить потенциальное повреждение в момент его возникновения и вызвать системный сбой таким образом, чтобы автоматический анализ показал проблемный драйвер.

Если в инструменте Driver Verifier включен режим особого пула, проверяемые драйверы будут пользоваться им вместо выгружаемого и невыгружаемого пулов при выделении памяти для буферов, размер которых слегка меньше размера страницы. Буфер, память под который выделена из особого пула, находится между двумя недействительными страницами и по умолчанию выровнен по верхней границе страницы. Кроме того, программы управления особым пулом случайным образом (на основе счетчика тактов системы) занимают неиспользованное пространство страницы, внутри которой находится буфер. Подробно особый пул рассматривается в главе 10.

Система распознает любые переполнения буфера, размер которого меньше размера страницы, в момент их возникновения. Ведь они приводят к ошибкам страницы, так как происходит обращение к недействительной странице, расположенной за буфером. Подпись нужна, чтобы перехватить опустошение буфера в момент, когда драйвер освобождает выделенную под буфер память, так как целостность заполняющих буфер элементов, помещенных туда в момент выделения памяти, при этом будет нарушена.

ЭКСПЕРИМЕНТ: ВКЛЮЧЕНИЕ ОСОБОГО ПУЛА ПРИ ПОМОЩИ DRIVER VERIFIER

Чтобы посмотреть, каким образом особый пул вызывает системный сбой, легко диагностируемый механизмом анализа, запустите программу Driver Verifier Manager (Диспетчер проверки драйверов) для настройки особого пула. Эта программа позволяет активировать большинство проверочных механизмов без перезагрузки системы. Вот как включается особый пул:

1. В меню Start (Пуск) введите команду verifier и нажмите клавишу Enter для запуска диспетчера проверки драйверов.
2. Установите переключатель Display Information About The Currently Verified Drivers (Вывести сведения о текущих проверенных драйверах) и щелкните на кнопке Next (Далее).
3. Щелкните на кнопке Change (Изменить), установите флажок Special Pool (Особый пул) и щелкните на кнопке OK. (В столбце Enabled? (Разрешено?) останется вариант No (Нет), пока вы не выберете драйвер для проверки.)
4. Щелкните на кнопке Add (Добавить), затем введите в поле File Name (Имя файла) myfault.sys и щелкните на кнопке Open (Открыть). (Искать драйвер Myfault.sys не нужно, просто введите его имя.)
5. Щелкните на кнопке Next (Далее) для перехода к странице, на которой диспетчер проверки драйверов выводит список глобальных счетчиков для всех проверенных на текущий момент драйверов. Следующий щелчок на кнопке Next (Далее) покажет список счетчиков для каждого проверенного драйвера. В списке должен фигурировать драйвер Myfault.sys.
6. Щелкните на кнопке Finish (Готово) для завершения работы мастера.

Проверенные при помощи механизма No Reboot инструмента Driver Verifier драйверы не контролируются так тщательно, как драйверы, загружаемые после перезагрузки. По возможности выполните проверку вашего драйвера и перезагрузите систему. Ввод следующей команды в командную строку с расширенными полномочиями заставит Driver Verifier сохранить после перезагрузки параметры проверки:

```
C:\>verifier /flags 0x1 /driver myfault.sys
```

New verifier settings:

```
Special pool: Enabled  
Pool tracking: Disabled  
Force IRQL checking: Disabled  
I/O verification: Disabled  
Deadlock detection: Disabled  
DMA checking: Disabled  
Security checks: Disabled  
Force pending I/O requests: Disabled  
Low resources simulation: Disabled  
IRP Logging: Disabled  
Miscellaneous checks: Disabled
```

Verified drivers:

```
myfault.sys
```

You must restart this computer for the changes to take effect.

При запуске программы Notmyfault произойдет переполнение буфера, что немедленно вызовет системный сбой. Вот что покажет анализ дампа:

```
Probably caused by : myfault.sys ( myfault+61d )
```

При подробном анализе стоп-код описывается следующим образом:

```
DRIVER_PAGE_FAULT_BEYOND_END_OF_ALLOCATION (d6)  
N bytes of memory was allocated and more than N bytes are being referenced.  
This cannot be protected by try-except.  
When possible, the guilty driver's name (Unicode string) is printed on the bug-  
check screen and saved in KiBugCheckDriver.  
Arguments:  
Arg1: beb50000, memory referenced  
Arg2: 00000001, value 0 = read operation, 1 = write operation  
Arg3: 9201161d, if non-zero, the address which referenced memory.  
Arg4: 00000000, (reserved)
```

Благодаря особому пуллу трудноуловимая ошибка вышла на поверхность, и анализ стал тривиальным.

Перезапись кода и защита системного кода от записи

Драйвер, который содержит ошибку, становится причиной повреждения памяти или неверной интерпретации его собственных структур. В этом случае драйвер, воспри-

нимая поврежденные данные как указатель на область памяти, может обращаться к памяти, которая ему не принадлежит. В виртуальном адресном пространстве он может быть нацелен куда угодно, в том числе на принадлежащие другим драйверам данные, недействительные страницы памяти или код других драйверов и ядра. Как и в случае с переполнением буфера, к моменту, когда повреждение приводит к системному сбою, уже, как правило, невозможно установить, какой драйвер стал его причиной. Использование особого пула повысит шансы обнаружения ошибок, связанных с некорректными указателями, но не поможет найти повреждения кода.

Если в программе Notmyfault установить переключатель Code Overwrite, драйвер Myfault повредит точку входа функции `NtReadFile` ядра. Далее возможны два варианта развития событий. Если в системе 2 Гбайт или меньше физической памяти, произойдет сбой, анализ которого укажет на `Myfault.sys`. Выводимое при детальном анализе описание стоп-кода покажет, что драйвер Myfault пытался выполнить запись в память, предназначенную только для чтения:

```
ATTEMPTED_WRITE_TO_READONLY_MEMORY (be)
An attempt was made to write to readonly memory. The guilty driver is on the stack
trace (and is typically the current instruction pointer).
When possible, the guilty driver's name (Unicode string) is printed on the bugcheck
screen and saved in KiBugCheckDriver.
Arguments:
Arg1: 826a023c, Virtual address for the attempted write.
Arg2: 026a0121, PTE contents.
Arg3: 90f83b4c, (reserved)
Arg4: 0000000b, (reserved)
```

Но если объем памяти превышает 2 Гбайт, возникает сбой другого типа, так как попытку повреждения памяти засечь не удается. Так как функция `NtReadFile` — это часто используемая системная функция, сбой Windows произойдет практически сразу же, как только какой-либо программный поток попытается выполнить поврежденный код и сгенерирует ошибку неверной команды. В этом случае анализ дампов даст неверные результаты. Обычно он указывает, что наиболее вероятными источниками ошибки являются файлы `Win32k.sys` и `Ntoskrnl.exe`. Вот описание ошибки для такого сбоя:

```
KERNEL_MODE_EXCEPTION_NOT_HANDLED (8e)
This is a very common bugcheck. Usually the exception address pinpoints the driver/
function that caused the problem. Always note this address as well as the link date of
the driver/image that contains this address.
Some common problems are exception code 0x80000003. This means a hard coded breakpoint
or assertion was hit, but this system was booted /NODEBUG. This is not supposed to
happen as developers should never have hardcoded breakpoints in retail code, but ...
If this happens, make sure a debugger gets connected, and the system is booted /DEBUG.
This will let us see why this breakpoint is happening.
Arguments:
Arg1: c0000005, The exception code that was not handled
Arg2: 826a0240, The address that the exception occurred at
Arg3: 978eb9c4, Trap Frame
Arg4: 00000000
```

Причина разного поведения различных конфигураций связана с механизмом защиты системного кода от записи (system code write protection). Если такая защита включена, диспетчер памяти выполняет отображение `Ntoskrnl.exe`, `HAL` и загрузочных

драйверов с помощью стандартных физических страниц (4 Кбайт в x86 и x64 и 8 Кбайт в IA64). Так как защита образа осуществляется с точностью до размера страницы, диспетчер памяти может защищать содержащие код страницы от записи. В результате попытка их отредактировать приведет к ошибке доступа (что мы и наблюдаем в первом случае). Но при отключенной защите системного кода от записи в системах с более чем 2 Гбайт RAM диспетчер памяти для отображения Ntoskrnl.exe и HAL использует большие страницы (4 Мбайт в x86 и 16 Мбайт в IA64 и x64).

Если при отключенной защите системного кода от записи анализ аварийного дампа сообщает о маловероятных причинах сбоя или вы считаете, что код был поврежден, включите защиту. Проще всего это сделать, проверив хотя бы один драйвер инструментом Driver Verifier. Кроме того, можно добавить параметр в раздел `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management`. Следует указать максимально возможное значение для объема памяти, при котором диспетчер памяти использует для отображения Ntoskrnl.exe большие страницы вместо стандартных. Поэтому создайте параметр `LargePageMinimum` типа `DWORD` и присвойте ему значение `0xFFFFFFFF`. Перезагрузите машину, чтобы изменения вступили в силу.

Углубленный анализ аварийных дампов

В предыдущем разделе мы узнали, как при помощи инструмента Driver Verifier создать аварийный дамп, доступный для автоматического анализа средствами отладчика. Тем не менее вы все равно можете столкнуться с ситуацией, когда невозможно получить от системы легко анализируемый дамп. В этом случае можно попытаться провести анализ вручную. Вот примеры основных команд, позволяющих получить представление о том, что произошло во время сбоя. Их полное описание находится в файле помощи Debugging Tools for Windows. Там же вы найдете другие команды, а также примеры их применения при анализе сбоев:

- ❑ Команда `!cpuinfo` выводит список процессоров, которые может использовать система.
- ❑ Команда `k` с идентификатором процессора, например `1k`, позволяет увидеть трассировку стека каждого процессора в системе. Убедитесь, что вы знаете все перечисленные модули и у вас установлены их последние версии.
- ❑ Команда `!thread` выводит сведения о текущем программном потоке на каждом процессоре. Для смены текущего процессора используется команда `~s` с идентификатором процессора, например `~1s`. Обратите внимание на незавершенные пакеты запросов на ввод или вывод (о них мы поговорим в следующем разделе).
- ❑ Команда `.time` позволяет получить сведения о системном времени, в том числе о моменте сбоя и продолжительности работы. Короткий период рабочего времени указывает на частое возникновение проблем.
- ❑ Команда `!m` с параметрами `k, t` (флаг `t` позволяет получить данные о временных метках, так как время компиляции файла может отличаться от показанного в файловой системе) выводит список загруженных драйверов режима ядра. Убедитесь,

что вы понимаете назначение всех драйверов сторонних производителей и у вас используются самые последние версии этих драйверов.

- ❑ Команда `!vm` показывает, не исчерпана ли виртуальная память системы, а также выгружаемый и невыгружаемый пулы. При исчерпании виртуальной памяти объем подтвержденных страниц окажется близким к лимиту подтверждения. В этом случае попытайтесь выявить потенциальную утечку памяти. Посмотрите список процессов и выберите те, которые потребляют много памяти. При исчерпании выгружаемого и невыгружаемого пулов (то есть при слишком к максимальному объему занятой памяти) обратитесь к эксперименту «Поиск и устранение утечек пулла» в главе 10.
- ❑ Команда `!process 0 0` отладчика позволяет увидеть работающие процессы. Убедитесь, что понимаете назначение каждого из них. Попробуйте завершить или удалить ненужные приложения и службы.

Существуют и другие отладочные команды, которые могут оказаться полезными в данном случае, но для их применения требуются более глубокие знания. Такой командой является, например, команда `!irp`. В следующем разделе вы увидите, как с ее помощью выявить подозрительные драйверы.

Засорение стека

Переполнение или засорение стека обычно возникает из-за переполнения или опустошения буфера или по причине передачи драйвером адреса буфера, расположенного в стеке, драйверу, находящемуся в стеке на более низком уровне. Последний при этом начинает работать в асинхронном режиме.

В случае переполнения или опустошения стека интересующий нас буфер находится не в пуле, как в случае с ошибкой переполнения буфера, инициируемого программой Notmyfault, а в стеке программного потока, выполняющего ошибочный код. Борьба с ошибками данного типа также представляет собой крайне сложный процесс, так как стек играет важную роль при анализе аварийного дампа.

При передаче буферов драйверам, расположенным ниже в стеке, может возникнуть ситуация, когда целевой драйвер возвращает управление вызывающей процедуре не синхронно, а немедленно, так как использует процедуру завершения. В этом случае задействуется ранее переданный адрес стека, который на этот момент может соответствовать другому состоянию стека вызывающей функции, что становится причиной повреждения стека.

После запуска программы Notmyfault и выбора варианта **Stack Trash** драйвер Myfault переполняет буфер, память под который выделена в стеке драйвера программного потока, исполняющего код. При попытке драйвера Myfault вернуть управление вызвавшей его функции `Ntoskrnl` он считывает из стека адрес возврата, то есть адрес, с которого должно продолжаться выполнение. Но этот адрес при переполнении буфера стека повреждается, поэтому программный поток продолжает выполнение с какого-то другого адреса, может быть, даже не содержащего кода. При попытке потока выполнить недопустимую инструкцию процессора или обратиться к недействительной области памяти генерируется непредусмотренное исключение и происходит системный сбой.

Анализ аварийного дампа, проводимый при переполнении стека, каждый раз будет указывать на разные драйверы, но стоп-код практически всегда останется одним и тем же: KERNEL_MODE_EXCEPTION_NOT_HANDLED (0x8E) в 32-разрядной системе и KMODE_EXCEPTION_NOT_HANDLED (0x1E) в 64-разрядной. Вот как выглядит результат трассировки стека после детального анализа:

```
STACK_TEXT:
9569b6b4 828c108c 0000000e c0000005 00000000 nt!KeBugCheckEx+0x1e
9569badc 8284add6 9569baf8 00000000 9569bb4c nt!KiDispatchException+0x1ac
9569bb44 8284ad8a 00000000 00000000 badb0d00 nt!CommonDispatchException+0x4a
9569bbfc 82843593 853422b0 86b99278 86b99278 nt!Kei386EoiHelper+0x192
00000000 00000000 00000000 00000000 nt!IofCallDriver+0x63
```

Следует заметить, что вызов функции `IofCallDriver` немедленно ведет к вызову `Kei386EoiHelper` и исключению, а не к процедуре отправки IRP-пакета драйвером. Это согласуется с информацией о повреждении стека. И именно процедура отправки IRP-пакета вызывает исключение, ведь при попытке вернуть управление вызывающей функции она ссылается на поврежденный адрес возврата. К сожалению, такие механизмы, как особый пул и защита системного кода от записи, не помогают выявлять ошибки данного типа. Приходится анализировать дамп вручную, чтобы косвенным образом установить, какой из драйверов работал в момент повреждения стека. Можно исследовать IRP-пакеты, с которыми работал выполнявшийся в момент засорения стека программный поток. При передаче потоком запроса на ввод-вывод диспетчер ввода-вывода заносит указатель на соответствующий IRP-пакет в список пакетов, хранящийся в структуре `ETHREAD` программного потока. Команда отладчика `!thread` выводит дамп этого списка для заданного потока. (Если адрес объекта потока не указывается, команда `!thread` выводит дамп текущего программного потока.) После этого поток можно исследовать при помощи команды `!irp`:

```
0: kd> !thread
THREAD 8527fa58 Cid 0d0c.0d10 Teb: 7ffdf000 Win32Thread: fe4ec4f8 RUNNING on processor
0
IRP List:
 86b99278: (0006,0094) Flags: 00060000 Mdl: 00000000
Not impersonating
...
0: kd> !irp 86b99278
Irp is active with 1 stacks 1 is current (= 0x86b992e8)
No Mdl: No System Buffer: Thread 8527fa58: Irp stack trace.
  cmd flg cl Device File Completion-Context
>[ e, 0] 5 0 853422b0 85e3aed8 00000000-00000000
  \Driver\MYFAULT
  Args: 00000000 00000000 83360010 00000000
```

Этот листинг показывает, что текущий и единственный фрагмент IRP-стека (обозначенный префиксом `>`) принадлежит драйверу `Myfault`. Если бы речь шла о реальном сбое, далее следовало бы убедиться, что установлена последняя версия драйвера, и если это не так, установить последнюю версию. Затем осталось бы включить для него инструмент `Driver Verifier` (с включением всех режимов, кроме имитации недостатка памяти).

ПРИМЕЧАНИЕ

Новейшие драйверы, созданные при помощи WDK, по умолчанию используют флаг /GS компилятора (проверка переполнения буфера). При включенной проверке переполнения буфера компилятор резервирует в стеке пространство перед обратным адресом, которое при выполнении функции заполняется так называемыми cookie-данными безопасности (security cookie). После завершения функции эти данные проверяются. Несовпадение исходных и конечных cookie-данных безопасности указывает на возможную перезапись стека. В этом случае сгенерированный компилятором код вызывает функцию KeBugCheckEx и передает ей стоп-код DRIVER_OVERRAN_STACK_BUFFER (0xF7).

При сбоях данного типа зачастую лучше всего использовать ручной анализ. Обычно он включает создание дампа текущего регистра указателя стека (например, esp и gsp в системах x86 и x64 соответственно). Но так как вызвавший системный сбой код может изменить стек таким способом, что проанализировать его будет крайне сложно, ответственный за сбой системы процессор предоставляет резервное хранилище для данных стека. Оно называется KiPreBugcheckStackSaveArea и содержит копию стека до выполнения кода в функции KeBugCheckEx.

При помощи команды dps отладчика (создание дампа указателя с символами) можно получить дамп этой области (вместо регистра указателя стека процессора) и проанализировать символы, пытаясь найти те, которые могут относиться к трассировке стека. В данном случае вот что в конце концов показал дамп области стека в 32-разрядной системе:

```
0: kd> dps KiPreBugcheckStackSaveArea KiPreBugcheckStackSaveArea+3000
81d7dd20 881fcc44
81d7dd24 98fcf406 myfault+0x406
81d7dd28 badb0d00
```

Среди множества других функций эти данные привлекли внимание упоминанием функции в драйвере Myfault, который, как мы видели, в настоящее время работает с не показанным в стеке IRP-пакетом. Более подробно о ручном анализе стека можно почитать в файле помощи Debugging Tools for Windows.

Зависание, или отсутствие отклика

Если система перестает отвечать (то есть не реагирует на ввод с клавиатуры или при помощи мыши), указатель мыши не перемещается или же вы можете его двигать, но система не реагирует на щелчки мыши, говорят, что система *зависла* (*hung*). Для такого поведения существует несколько причин:

- ❑ После процедуры обработки прерываний (Interrupt Service Routine, ISR) или отложенного вызова процедуры (Deferred Procedure Call, DPC) управление не вернулось к драйверу устройства.
- ❑ Выполняемый в режиме реального времени программный поток с высоким приоритетом вытеснил программные потоки ввода данных драйвера подсистемы управления окнами.
- ❑ При выполнении кода в режиме ядра произошла взаимная блокировка (каждый из двух программных потоков или процессоров удерживает ресурсы, необходимые другому, и ни один из них не освобождает свой ресурс).

Найти взаимные блокировки позволяет механизм *выявления взаимных блокировок* (deadlock detection) инструмента Driver Verifier. Он проверяет спин-блокировки, обычные и быстрые мьютексы, выявляя закономерности, которые могут привести к взаимной блокировке. (Информация об этих и других примитивах синхронизации дана в главе 3 части I.) При обнаружении подобной ситуации Driver Verifier вызывает системный сбой, указывая, какой драйвер является причиной взаимной блокировки. Простейшая взаимная блокировка возникает в ситуации, когда каждый из двух программных потоков удерживает нужный другому потоку ресурс и не освобождает его, ожидая, пока это сделает другой поток. Значит, первое, что нужно сделать для устранения зависаний системы, – включить для подозрительных драйверов режим обнаружения взаимных блокировок. Затем это нужно сделать для неподписанных драйверов, а далее для всех драйверов, пока не произойдет системный сбой, который позволит выявить вызывающий взаимную блокировку драйвер.

Существует два подхода к исследованию зависшей системы, позволяющие найти драйвер или компонент, являющийся источником проблемы. Во-первых, можно аварийно завершить работу зависшей системы, надеясь на получение доступного для анализа дампа. Во-вторых, можно исследовать систему с помощью отладчика ядра. Но при обоих подходах требуется предварительная настройка и перезагрузка. Для выявления причины зависания в обоих случаях выполняется одно и то же исследование состояния системы.

Чтобы вручную вызвать аварийное завершение работы зависшей системы, добавьте в раздел `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\i8042prt\Parameters\CrashOnCtrlScroll` реестра параметр типа `DWORD` со значением 1. После перезагрузки драйвер порта i8042, который является драйвером порта ввода с PS/2-клавиатуры, в своей ISR-процедуре отслеживает нажатия клавиш (эта тема рассматривается в главе 3 части I), ожидая двукратного нажатия клавиши `Scroll Lock` при нажатой правой клавише `Control`. Обнаружив такую последовательность нажатий, драйвер вызывает функцию `KeeBugCheckEx` со стоп-кодом `MANUALLY_INITIATED_CRASH (0xE2)`, указывающим на тот факт, что сбой был инициирован вручную. После перезагрузки системы откройте аварийный дамп и с помощью описанных ранее приемов попробуйте определить, почему зависла система (например, узнайте, какой программный поток выполнялся в этот момент, что показывает стек ядра и т. п.). Следует заметить, что данный подход работает в большинстве сценариев зависания, но не годится, если не выполняется ISR-процедура драйвера порта i8042. (Последнее происходит, когда все процессоры зависли из-за того, что их IRQL-уровень превышает IRQL-уровень ISR-процедуры, либо из-за того, что повреждение системных структур данных затронуло код или данные, используемые для обработки прерываний.)

ПРИМЕЧАНИЕ

При использовании USB-клавиатур вызвать аварию зависшей системы вручную, воспользовавшись функциональностью драйвера порта i8042, невозможно. Этот подход работает только с PS/2-клавиатурами. О том, как включить поддержку USB-клавиатур, читайте на странице <http://msdn.microsoft.com/en-us/library/windows/hardware/ff545499.aspx>.

Также аварийное завершение работы можно вызвать, воспользовавшись встроенной кнопкой «crash». (На некоторых качественных серверах она встроена в материнскую

плату или доступна через интерфейс дистанционного управления.) В этом случае материнская плата генерирует немаскируемое прерывание (Non-Maskable Interrupt, NMI). Для применения этой возможности присвойте значение 1 параметру типа `DWORD` реестра из раздела `HKLM\SYSTEM\CurrentControlSet\Control\CrashControl\NMICrashDump`. В этом случае при нажатии кнопки аварийного перезапуска в системе будут генерироваться немаскируемые прерывания, и обработчик немаскируемых прерываний ядра вызовет функцию `KeBugCheckEx`. Этот подход более универсален, чем применение драйвера порта i8042, так как IRQL-уровень немаскируемых прерываний всегда выше, чем у прерываний драйвера порта i8042. Более подробно об этом можно почитать на странице <http://support.microsoft.com/kb/927069>.

Если сгенерировать аварийный дамп вручную нельзя, попытайтесь проникнуть в зависшую систему. Для начала загрузите ее в отладочном режиме. Это можно сделать двумя способами. Можно нажать клавишу `F8` во время загрузки и выбрать вариант `Debugging Mode` или создать параметр `debuggingmode` в BCD, скопировав существующую запись загрузки и добавив параметр `debug`. При нажатии клавиши `F8` система будет использовать соединение, предлагаемое по умолчанию (последовательный порт COM1 и скорость 115 200 бод). Но можно нажать клавишу `F10` для перехода к экрану `Edit Boot Options`, на котором осуществляется редактирование связанных с отладкой параметров загрузки. После включения режима отладки нужно будет настроить механизм соединения между системой, на которой выполняется отладчик ядра, и целевой системой, загружаемой в отладочном режиме, сконфигурировав параметры транспорта в соответствии с типом соединения. Доступны три типа соединения. Нуль-модемный кабель для последовательных портов, кабель IEEE 1394 (FireWire) для порта 1394 в каждой из систем и аппаратный ключ USB 2.0 между хостами, использующий USB-порт в каждой из систем. Подробно настройка хост-системы и системы для отладки ядра описаны в файле помощи `Debugging Tools for Windows` и в эксперименте «Присоединение отладчика ядра» на с. 660.

При загрузке в отладочном режиме система загружает отладчик ядра и готовит его для соединения с отладчиком ядра, выполняемом на другом компьютере, подключение к которому осуществляется через последовательный кабель, IEEE 1394-кабель или аппаратный ключ USB 2.0 между хостами. Следует заметить, что наличие отладчика ядра не влияет на производительность. Когда система зависнет, запустите на подключенной системе отладчик `WinDbg` или `Kd`, установите между отладчиками соединение и проникните в зависшую систему. Однако при отключенных прерываниях или поврежденном коде отладчика ядра эта методика не работает.

ПРИМЕЧАНИЕ

Загрузка системы в отладочном режиме не влияет на производительность, если нет связи с другой системой. Кроме того, если система, загруженная в отладочном режиме, настроена на автоматическую перезагрузку после сбоя, она не будет ждать соединения со стороны второй системы, когда не подсоединен отладчик.

Во время выполнения анализа систему можно не оставлять в остановленном состоянии. Воспользуйтесь командой `dump` отладчика для создания файла аварийного дампа на хост-машине. После чего зависшая система перезагружается и дамп анализируется

в автономном режиме (или отправляется в Microsoft). Следует иметь в виду, что если вы используете последовательный нуль-модемный кабель или соединение по USB 2.0 (по сравнению с более скоростным соединением 1394), процесс может занять долгое время. Поэтому у вас может появиться желание ограничиться мини-дампом. В этом случае используйте команду `.dump /m`. В качестве альтернативы, в случае, когда целевой компьютер способен записать аварийный дамп, можно это сделать из отладчика командой `.crash`. Дамп появится на локальном жестком диске целевого компьютера, и его можно будет исследовать после перезагрузки системы.

ЭКСПЕРИМЕНТ: ВЫВОД СПИСКА ВИРТУАЛЬНЫХ МАШИН С ПОМОЩЬЮ LIVEKD

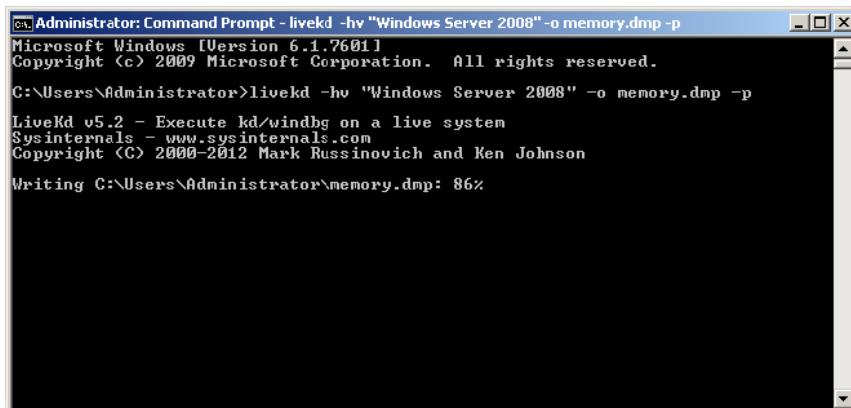
Программа LiveKd позволяет не только применять команду `.dump` к работающим системам, но и создавать аварийный дамп виртуальных машин Hyper-V. Для вывода списка виртуальных машин воспользуйтесь командой `-hv1`. LiveKd покажет как их имена, так и GUID-идентификаторы их разделов:

```
C:\Users\Administrator>livekd -hv1

LiveKd v5.2 - Execute kd/windbg on a live system
Sysinternals - www.sysinternals.com
Copyright (C) 2000-2012 Mark Russinovich and Ken Johnson

Partition GUID Name
7EB669F2-EB6E-405D-94EA-21CB2ABD0A52 Windows Server 2008
D57D7601-D154-473B-847D-C3C77413AD0B Windows Server 2003
```

Полученное имя виртуальной машины или GUID раздела передается в LiveKd при помощи параметра `-hv` в комбинации с переключателем `-o`, указывающим место записи файла аварийного дампа. Программа LiveKd записывает полный дамп, для которого требуется место на томе, равное объему памяти, выделенному под виртуальную машину. Так как виртуальная машина Hyper-V при этом работает, программа LiveKd может включиться в момент, когда структуры данных редактируются системой и потому находятся в несогласованном состоянии. Чтобы избежать подобной ситуации, в программу LiveKd добавлена возможность приостановки виртуальной машины Hyper-V перед записью аварийного дампа. Эта возможность реализуется через параметр `-p`.



Кроме того, LiveKd добавляет к заголовку файла аварийного дампа комментарий, оповещающий о снятии дампа с работающей системы и предупреждающий выполняющего анализ пользователя о возможной несогласованности. Готовый дамп можно анализировать в любом отладчике ядра, применяя описанные в предыдущих разделах приемы. Если ранее виртуальная машина Hyper-V работала, LiveKd автоматически вернет ее в рабочее состояние.

Зависание можно вызвать, выбрав в приложении Notmyfault вариант **Hang With DPC**. В этом случае драйвер Myfault поставит DPC в очередь на каждом из процессоров системы, выполняяющим бесконечный цикл. Так как при выполнении DPC-функций IRQL-уровень такого процессора равен **DPC/dispatch**, ISR клавиатуры будет отвечать сбоем на нажатие определенной клавиатурной комбинации.

После того как вы приступили к отладке зависшей системы или загрузили в отладчик вручную генерированный для нее дамп, обычно следует команда **!analyze** с параметром **-hang**. Отладчик в этом случае анализирует блокировки системы и пытается определить наличие взаимной блокировки и найти вовлеченные в нее драйверы. Но в случае сбоя, вызванного программой Notmyfault с заданным параметром **Hang With DPC**, эта команда не поможет.

Поэтому следует выполнить команды **!thread** и **!process** в контексте дампа каждого процессора, чтобы понять, чем занят каждый из процессоров. (Переключение контекста процессора осуществляется командой **~s**, например команда **~1s** выполняет переключение на контекст процессора 1.) Если вызвавший зависание системы программный поток выполняет бесконечный цикл на IRQL-уровне **DPC/dispatch** и выше, в трассировочной информации стека, выводимой командой **!thread**, содержится имя драйвера, в котором выполняется процесс. Вот пример трассировочной информации стека для аварийного дампа системы, зависание которой спровоцировано с помощью программы Notmyfault:

```
STACK_TEXT:
8078ae30 8cb49160 000000e2 00000000 00000000 nt!KeBugCheckEx+0x1e
8078ae60 8cb49768 00527658 010001c6 00000000 i8042prt!I8xProcessCrashDump+0x251
8078aeac 8287c7ad 851c8780 855275a0 8078aed8 i8042prt!I8042KeyboardInterruptService+0x
2ce
8078aeac 91d924ca 851c8780 855275a0 8078aed8 nt!KiInterruptDispatch+0x6d
WARNING: Stack unwind information not available. Following frames may be wrong.
8078afa4 828a5218 82966d20 86659780 00000000 myfault+0x4ca
...

```

Верхние несколько строк относятся к нажатию клавиш, при котором драйвер порта i8042 вызывает системный сбой. Присутствие драйвера Myfault показывает, что именно он может быть причиной сбоя. Также вам может оказаться полезной команда **!locks**, выводящая состояние всех блокировок исполнительной системы. По умолчанию выводятся только спорные ресурсы, то есть ресурсы, на обладание которыми претендуют как минимум два программных потока. Стеки потоков, владеющих такими ресурсами, исследуются при помощи команды **!thread**, показывающей, в каком драйвере они выполняются. Иногда можно обнаружить, что владелец одной из блокировок ожидает завершения IRP-пакета (список связанных с программным потоком IRP-пакетов выводится командой **!thread**). В подобном случае сложно

сказать, почему с IRP-пакетом ничего не происходит. (До завершения IRP-пакеты обычно ставятся в очередь скрыто управляемого драйвера.) Команда `!irp` позволяет найти драйвер, задерживающий такой IRP-пакет, — в выводимой информации рядом с его стеком будет фигурировать слово «pending» (ожидающий обработки). Получив имя драйвера, используйте команду `!stacks` для просмотра других программных потоков, которыми он может управлять. Часто именно это дает представление о том, что делает владеющий блокировкой драйвер. В большинстве случаев оказывается, что он находится в состоянии взаимной блокировки.

Если аварийный дамп отсутствует

В этом разделе мы поговорим о том, как устранять неполадки в системах, которые по каким-либо причинам не записывают аварийный дамп. Это случается, к примеру, когда файл подкачки не в состоянии вместить в себя дамп. В такой ситуации вопрос решается созданием файла подкачки требуемого размера. Второй причиной может быть повреждение при сбое кода ядра и структур данных, необходимых для записи дампа. Как уже упоминалось, эти данные фиксируются при загрузке системы, и если проведенная при сбое проверка целостности показывает разницу контрольных сумм, система даже не пытается сохранять аварийный дамп (чтобы не рисковать данными на диске). В этом случае нужно пытаться отследить момент сбоя и определить его причину.

Подобная ситуация возникает также, когда дисковая подсистема не в состоянии обработать запросы на запись (это само по себе может вызвать системный сбой). Причиной в данном случае может быть аппаратный сбой контроллера дисков или повреждение кабеля жесткого диска.

Еще одна возможность возникает при наличии в системе драйверов с зарегистрированными обратными вызовами, которые использовались до записи аварийного дампа. При обратном вызове возможен, к примеру, некорректный доступ драйвера к структурам данных в выгружаемой памяти, что приведет к новому сбою. При сбое в процессе обратного вызова дополнительного дампа файл аварийного дампа в системе сохранится, а вот данные дополнительного дампа могут отсутствовать или оказаться неполными.

Проще всего при этом отключить режим `Automatically Restart` (Выполнить автоматическую перезагрузку) в меню `Startup And Recovery` (Загрузка и восстановление), чтобы при сбое можно было изучить синий экран с консоли. Но представленный на нем текст позволяет установить причины только самых очевидных сбоев.

Для более глубокого анализа следует исследовать поведение системы во время сбоя при помощи отладчика ядра. Для этого загрузите систему в отладочном режиме, как описано в предыдущих разделах. В этом случае при сбое вместо синего экрана и попытки записать дамп вас ожидает соединение с отладчиком ядра, работающим на хост-системе. Вы можете воспользоваться упоминавшимися ранее командами отладчика для проведения базового анализа и выявления причин сбоя. Команда `.dump` сохраняет копию пространства памяти системы, позволяя вам перезагрузиться и выполнить отладку в автономном режиме.

ЭКСПЕРИМЕНТ: ПРИСОЕДИНЕНИЕ ОТЛАДЧИКА ЯДРА

Для присоединения отладчика ядра к работающей системе вам потребуется второй компьютер. Анализируемая система должна быть загружена в отладочном режиме. Для этого в процессе загрузки нужно нажать клавишу F8 и выбрать в меню вариант Debugging Mode. Кроме того, можно отредактировать базу конфигурационных данных загрузки через командную строку с расширенными полномочиями, воспользовавшись утилитой BCDEDit:

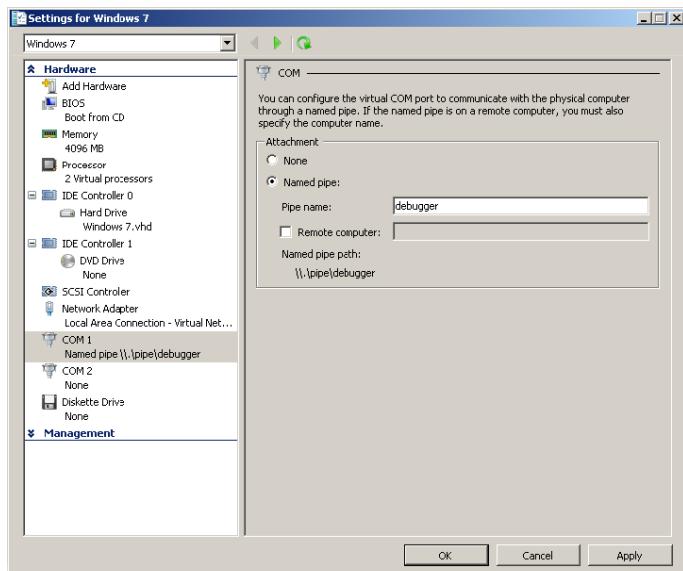
```
bcdedit /debug on
```

Если не задавать другие параметры, система воспользуется последовательным портом COM1 и скоростью передачи данных 115200. В хост-системе — на компьютере, где запущен отладчик, — следует указать символьный путь, чтобы отладчик мог найти нужные файлы. Это можно сделать через системную переменную окружения _NT_SYMBOL_PATH. Системные переменные позволяют другим приложениям, таким как Process Explorer и Process Monitor, использовать символьный путь без дополнительной настройки. Кроме того, этот путь можно задать через командную строку с повышенными привилегиями, воспользовавшись следующей командой:

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

Переключатель /m указывает, что переменная должна быть общесистемной. По умолчанию же она задается только для активного пользователя. Напоследок нужно настроить транспортный уровень. Если вы работаете на двух компьютерах, соедините их последовательные порты нуль-модемным кабелем.

В следующем примере в качестве целевой системы фигурирует виртуальная машина Hyper-V. Система виртуализации Hyper-V (как и другие подобные технологии) позволяет настроить виртуальный последовательный порт для обмена данными с физическим компьютером через именованный канал. В случае набора именованных каналов у каждого из них должно быть уникальное имя, чтобы избежать конфликта.



Перед перезагрузкой целевой системы на хост-системе следует указать именованный канал, который будет служить транспортом для отладчика. При соединении с виртуальной машиной Hyper-V в представленной далее команде следует использовать параметры resets=0 и reconnect. (Параметры других средств виртуализации вы найдете в файле помощи Debugging Tools for Windows.) Эта команда запускает отладочный сеанс на виртуальной машине, работающей на одном компьютере с отладчиком:

```
windbg -k com:pipe,port=\.\pipe\debugger,resets=0,reconnect
```

Должно появиться окно отладчика WinDbg с сообщением об ожидании соединения:

```
Microsoft (R) Windows Debugger Version 6.12.0002.633 AMD64  
Copyright (c) Microsoft Corporation. All rights reserved.
```

```
Waiting for pipe \\.\pipe\debugger  
Waiting to reconnect...
```

В этот момент следует перезагрузить целевую систему. Через небольшой промежуток времени произойдет соединение систем по именованному каналу. Вывод следующей информации подтверждает, что хост соединен с целевой системой через отладчик ядра:

```
Connected to Windows 7 7601 x86 compatible target at  
(Mon Mar 12 19:34:01.295 2012 (UTC - 7:00)), ptr64 FALSE  
Kernel Debugger connection established.  
Symbol search path is: srv*c:\symbols*http://msdl.microsoft.com/download/symbols  
Executable search path is:  
Windows 7 Kernel Version 7601 (Service Pack 1) MP (1 procs) Free x86 compatible  
Built by: 7601.17514.x86fre.win7sp1_rtm.101119-1850  
Machine Name:  
Kernel base = 0x82813000 PsLoadedModuleList = 0x8295d850  
System Uptime: not available
```

Чтобы убедиться, что в случае сбоя система войдет в отладчик, воспользуйтесь параметром /bugcheck приложения Notmyfault, вызывающим системный сбой. Как и в случае других инструментов этого приложения, управляющий код отправляется драйверу Myfault.sys. Этот код указывает, что следует вызвать процедуру KeBugCheckEx и передать ей ссылку на стоп-код. Вот пример применения стоп-кода, заданного пользователем:

```
notmyfault /bugcheck 0xdeaddead
```

При наличии присоединенного к системе отладчика в случае сбоя управление передается этому отладчику до появления синего экрана и любых обратных вызовов, направленных на поиск ошибок. Это позволяет как перейти к дальнейшему анализу, так и задать точки останова:

```
*** Fatal System Error: 0xdeaddead  
(0x00000000,0x00000000,0x00000000,0x00000000)
```

```
Break instruction exception - code 80000003 (first chance)
```

```
A fatal system error has occurred.  
Debugger entered on first try; Bugcheck callbacks have not been invoked.
```

```
A fatal system error has occurred.  
...
```

Системный код и структуры данных, обрабатывающие исключения, могут оказаться поврежденными так, что это приведет к повторяющимся ошибкам. Примером такой ситуации является, скажем, повреждение обработчика прерываний, приводящее к ошибке страницы. В результате активируется обработчик ошибок страниц, который тоже будет функционировать некорректно, и система безнадежно зависает. Для предотвращения подобной ситуации в процессор встроен механизм защиты от рекурсивных ошибок, задающий предел рекурсии. На большинстве процессоров x86 глубина вложенности ошибок равна двум. При возникновении третьей рекурсивной ошибки процессор перезагружает систему. Это называется *тройной ошибкой* (triple fault), которая случается в том числе при наличии неисправного аппаратного компонента. В такой ситуации не удается активировать даже отладчик ядра. Впрочем, иногда данный факт подтверждает, что источником проблемы являются недавно добавленное устройство или драйверы.

ПРИМЕЧАНИЕ

Для вызова тройной ошибки нужно установить в отладчике ядра точку останова в процедуре диспетчеризации KiDispatchException. В результате диспетчер исключений вызовет исключение точки останова, что приведет к вызову диспетчера исключений и т. д.

Анализ распространенных стоп-кодов

В этом разделе вы найдете пошаговое руководство для анализа стоп-кодов, чаще всего присыаемых в службу Online Crash Analysis. Анализ каждого кода начинается с подробного результата действия команды `!analyze -v`.

Причины сбоев могут различаться, соответственно, для их анализа применяются разные команды и приемы. Более подробно процесс анализа стоп-кодов освещен в файле помощи Debugging Tools for Windows, а также на сайтах, ссылки на которые вы найдете в конце данной главы.

Код 0xD1 — DRIVER_IRQL_NOT_LESS_OR_EQUAL

Стоп-код `DRIVER_IRQL_NOT_LESS_OR_EQUAL` (0xD1) является результатом попытки драйвера устройства обратиться по выгруженному или недействительному адресу при слишком высоком уровне запроса на прерывание. Фактически, драйвер устройства использует некорректные адреса.

`DRIVER_IRQL_NOT_LESS_OR_EQUAL (d1)`

An attempt was made to access a pageable (or completely invalid) address at an interrupt request level (IRQL) that is too high. This is usually caused by drivers using improper addresses.

If kernel debugger is available get stack backtrace.

Arguments:

Arg1: a0a91660, memory referenced

Arg2: 00000002, IRQL

Arg3: 00000000, value 0 = read operation, 1 = write operation

Arg4: 85701579, address which referenced memory

Трассировка стека выполнявшегося в момент сбоя программного потока выявит драйвер устройства, который обращается к выгруженной или недействительной памяти:

STACK_TEXT:

```
8b94bb3c 85701579 badb0d00 84f40600 a0a4f660 nt!KiTrap0E+0x2cf
WARNING: Stack unwind information not available. Following frames may be wrong.
8b94bbb8 85701849 86ffe5d8 8b94bbfc 857018ac myfault+0x579
8b94bbc4 857018ac 850d6890 00000001 00000000 myfault+0x849
8b94bbc4 8283e593 86efaa98 86ffe5d8 86ffe648 myfault+0x8ac
8b94bc14 82a3299f 850d6890 86ffe5d8 86ffe648 nt!ofCallDriver+0x63
8b94bc34 82a35b71 86efaa98 850d6890 00000000 nt!IopSynchronousServiceTail+0x1f8
8b94bcd0 82a7c3f4 86efaa98 86ffe5d8 00000000 nt!IopXxxControlFile+0x6aa
8b94bd04 828451ea 000000b8 00000000 00000000 nt!NtDeviceIoControlFile+0x2a
8b94bd04 776f70b4 000000b8 00000000 00000000 nt!KiFastCallEntry+0x12a
0012f994 00000000 00000000 00000000 00000000 0x776f70b4
```

Анализирующий механизм отладчика может локализовать и вывести на экран фрейм системного прерывания, созданный при появлении исключения, ставшего причиной сбоя. Этот фрейм содержит данные о состоянии компьютера, в том числе регистры процессора, которые выполнялись программным потоком ядра. Регистр указателя команд (eip для процессоров x86 и rip для x64) содержит адрес команды, выполнение которой привело к прерыванию. В нижней строке данных, выводимых командой `.trap` в отладчике, указывается адрес ставшей причиной сбоя команды, ее двоичный код, мнемоника языка ассемблера и подробные данные:

```
TRAP_FRAME: 8b94bb3c -- (.trap 0xffffffff8b94bb3c)
ErrCode = 00000000
eax=a0a91660 ebx=86ffe5f0 ecx=00200073 edx=84f40600 esi=a0a4f660 edi=00000000
eip=85701579 esp=8b94bbb0 ebp=8b94bbb8 iopl=0 nv up ei ng nz na pe nc
cs=0008 ss=0010 ds=0023 es=0023 fs=0030 gs=0000 efl=00010286
myfault+0x579:
85701579 8b08    mov     ecx,dword ptr [eax]    ds:0023:a0a91660=??
mov    ecx,dword ptr [eax]
```

Первый параметр отладки стоп-кода `DRIVER_IRQL_NOT_LESS_OR_EQUAL` (`0xD1`) указывает на адрес в памяти, по которому обращается драйвер устройства. Если отладчик не в состоянии показать этот адрес (так как он недействителен или отсутствует в файле аварийного дампа), на экран выводится набор вопросительных знаков. В показанном фрейме системного прерывания отладчик оказался не в состоянии разрешить адрес обращения драйвера устройства.

Просмотр результатов применения к этому адресу команды `!pte` подтверждает, что для соответствующего элемента таблицы страниц не был задан *бит достоверности* (*valid bit*). То есть адрес не отображается на страницу в физической памяти:

```
0: kd> !pte a0a91660
VA a0a91660
PDE at C0602828      PTE at C0505488
contains 000000010BE6863 contains 00007A1800000000
pfn 10be6    ---DA--KWEV not valid
              PageFile: 0
              Offset: 7a18
              Protect: 0
```

Код 0x8E — KERNEL_MODE_EXCEPTION_NOT_HANDLED

Стоп-код KERNEL_MODE_EXCEPTION_NOT_HANDLED (0x8E) имеет место, когда программный поток ядра генерирует необрабатываемое исключение. Первый параметр отладки задает код исключения, для которого не был найден обработчик. Типичные коды исключений в этом случае STATUS_BREAKPOINT (0x80000003) и STATUS_ACCESS_VIOLATION (0xC0000005).

KERNEL_MODE_EXCEPTION_NOT_HANDLED (8e)

This is a very common bugcheck. Usually the exception address pinpoints the driver/function that caused the problem. Always note this address as well as the link date of the driver/image that contains this address.

Some common problems are exception code 0x80000003. This means a hard coded breakpoint or assertion was hit, but this system was booted /NODEBUG. This is not supposed to happen as developers should never have hardcoded breakpoints in retail code, but ... If this happens, make sure a debugger gets connected, and the system is booted /DEBUG. This will let us see why this breakpoint is happening.

Arguments:

Arg1: 80000003, The exception code that was not handled
 Arg2: 92c70a78, The address that the exception occurred at
 Arg3: 9444fb4c, Trap Frame
 Arg4: 00000000

Трассировка стека сбойного потока указывает на драйвер или функцию, послужившую причиной сбоя. Если ничто не вызывает подозрений, следует обратить внимание на адрес, по которому появилось исключение. Вот как выглядит результат трассировки стека системы, в которой произошел сбой:

```
STACK_TEXT:
9444f6b4 828ba08c 0000008e 80000003 92c70a78 nt!KeBugCheckEx+0x1e
9444fadc 82843dd6 9444faf8 00000000 9444fb4c nt!KiDispatchException+0x1ac
9444fb44 82844678 9444fb4c 92c70a79 badb0d00 nt!CommonDispatchException+0x4a
9444fb44 92c70a79 9444fb4c 92c70a79 badb0d00 nt!KiTrap03+0xb8
WARNING: Stack unwind information not available. Following frames may be wrong.
9444fb4c 92c70b1c 8730f980 00000001 00000000 myfault+0xa79
9444fbfc 8283c593 87314a08 87279950 87279950 myfault+0xb1c
9444fc14 82a3099f 8730f980 87279950 872799c0 nt!IoCallDriver+0x63
9444fc34 82a33b71 87314a08 8730f980 00000000 nt!IopSynchronousServiceTail+0x1f8
9444fc0 82a7a3f4 87314a08 87279950 00000000 nt!IopXxxControlFile+0x6aa
9444fd04 828431ea 00000004 00000000 00000000 nt!NtDeviceIoControlFile+0x2a
9444fd04 772c70b4 00000004 00000000 00000000 nt!KiFastCallEntry+0x12a
0012f2ac 00000000 00000000 00000000 00000000 0x772c70b4
```

Второй параметр отладки задает местоположение исключения в памяти. В случае исключения STATUS_BREAKPOINT анализ адреса подтвердит наличие команды останова. Команда INT 3 процессора вызывает прерывание команд отладчика. При выполнении команды INT 3 система вызывает обработчик исключений отладчика.

Если отладчик присоединен к компьютеру, система войдет в него.

```
0: kd> u 92c70a78
myfault+0xa78:
92c70a78 cc          int     3
...
```

Обычно в продаваемых версиях драйверов устройств точек останова быть не должно. Команда `lm` иногда позволяет определить, для какого окружения предназначен

драйвер устройства. При компиляции готового драйвера (если разработчик не сделает по-другому) устанавливается флаг, указывающий на тип выпуска. Наличие в значении свойства **File flags** слова «Debug» означает, что драйвер был создан в среде отладки:

```
0: kd> lm kv m myfault
start      end          module name
92c70000  92c71880    myfault (no symbols)
Loaded symbol image file: myfault.sys
Image path: \??\C:\Windows\system32\drivers\myfault.sys
Image name: myfault.sys
Timestamp:        Sat Apr 07 09:34:40 2012 (4F806CA0)
CheckSum:         00004227
ImageSize:        00001880
File version:     4.0.0.0
Product version: 4.0.0.0
File flags:       1 (Mask 3F) Debug
File OS:          40004 NT Win32
...
...
```

Точка останова в отладочной версии драйвера может указывать на ошибку макроса **ASSERT**. Если отладчик ядра присоединен к системе, появится сообщение, после чего пользователю будут предложены варианты действий, связанные со сбоем диагностики.

Код 0x7F — UNEXPECTED_KERNEL_MODE_TRAP

Стоп-код **UNEXPECTED_KERNEL_MODE_TRAP** (0x7F) означает, что процессор сгенерировал прерывание, которое ядро Windows не может обработать. Причиной этому может быть *связанное прерывание* (bound trap), то есть прерывание, которое ядру нельзя перехватывать, или *двойная ошибка* (double fault), то есть ошибка, возникающая в процессе обработки ядром предыдущей ошибки. Первый параметр отладки определяет тип прерывания.

```
UNEXPECTED_KERNEL_MODE_TRAP (7f)
This means a trap occurred in kernel mode, and it's a trap of a kind that the kernel
isn't allowed to have/catch (bound trap) or that is always instant death (double
fault). The first number in the bugcheck params is the number of the trap (8 = double
fault, etc)
Consult an Intel x86 family manual to learn more about what these traps are. Here is a
*portion* of those codes:
If kv shows a taskGate
    use .tss on the part before the colon, then kv.
Else if kv shows a trapframe
    use .trap on that value
Else
    .trap on the appropriate frame will show where the trap was taken
    (on x86, this will be the ebp that goes with the procedure KiTrap)
Endif
kb will then show the corrected stack.
Arguments:
Arg1: 00000008, EXCEPTION_DOUBLE_FAULT
Arg2: 801db000
Arg3: 00000000
Arg4: 00000000
```

Большинство прерываний в этой категории возникают в результате неисправной или некорректно работающей аппаратуры. Если недавно вы добавляли новое устройство, попробуйте удалить его и посмотреть, возникнет ли наблюдаемая проблема снова. Удалите любое оборудование, которое может работать некорректно, заменив его новым. Также рекомендуется воспользоваться поставляемыми производителями инструментами диагностики для выявления сбойных компонентов.

Впрочем, бывают и прерывания, причиной которых являются программные ошибки. Просмотр сгенерированного фрейма прерывания или шлюза задачи (в зависимости от типа прерывания) позволяет найти команду, ставшую причиной прерывания:

```
TSS: 00000028 -- (.tss 0x28)
eax=8336001c ebx=86d57388 ecx=83360044 edx=00000000 esi=86d57388 edi=00000000
eip=96890918 esp=92985000 ebp=92987bc4 iopl=0 nv up ei pl zr na pe nc
cs=0008 ss=0010 ds=0023 es=0023 fs=0030 gs=0000 efl=00010246
myfault+0x918:
96890918 e8f9ffffff call myfault+0x916 (96890916)
```

Описанный ранее тип прерывания `EXCEPTION_DOUBLEFAULT` обычно возникает по двум причинам: переполнение стека ядра и неисправное аппаратное обеспечение. Переполнение стека ядра имеет место при обращении программного потока ядра к защитной странице как результат исчерпания в стеке памяти, выделенной под текущий поток. Ядро пытается поместить фрейм прерывания в стек, но для него там нет места, что приводит к двойной ошибке.

Применение команды `!thread` для проверки пределов стека выполняющегося программного потока подтвердит, было ли причиной двойной ошибки именно переполнение стека:

```
0: kd> !thread
THREAD 850e3918 Cid 0fb8.0fb8 Teb: 7ffdde000 Win32Thread: fe4f0dd8 RUNNING on processor
0
IRP List:
 86d57370: (0006,0094) Flags: 00060000 Mdl: 00000000
Not impersonating
DeviceMap           8fa3b8e8
Owning Process     85100670      Image: NotMyfault.exe
Attached Process   N/A          Image: N/A
Wait Start TickCount 21664 Ticks: 0
Context Switch Count 461
UserTime            00:00:00.000
KernelTime          00:00:00.046
Win32 Start Address 0x00fe27ff
Stack Init 92987fd0 Current 92987af8 Base 92988000 Limit 92985000 Call 0
Priority 12 BasePriority 8 UnusualBoost 0 ForegroundBoost 2 IoPriority 2 PagePriority 5
ChildEBP RetAddr Args to Child
00000000 96890918 00000000 00000000 00000000 nt!KiTrap08+0x75 (FPO: TSS 28:0)
WARNING: Stack unwind information not available. Following frames may be wrong.
92987bc4 96890b1c 87015038 00000001 00000000 myfault+0x918
92987bfc 82845593 85154158 86d57370 86d57370 myfault+0xb1c
92987c14 82a3999f 87015038 86d57370 86d573e0 nt!IoCallDriver+0x63
92987c34 82a3cb71 85154158 87015038 00000000 nt!IoPynchronousServiceTail+0x1f8
92987cd0 82a833f4 85154158 86d57370 00000000 nt!IoPxxxControlFile+0x6aa
92987d04 8284c1ea 00000004 00000000 00000000 nt!NtDeviceIoControlFile+0x2a
92987d04 779a70b4 00000004 00000000 00000000 nt!KiFastCallEntry+0x12a (FPO: [0,3]
TrapFrame @ 92987d34)
0012f424 00000000 00000000 00000000 00000000 0x779a70b4
```

Нас интересуют два значения: база стека и границы стека. Сравнение границы стека со значением, хранящимся в регистре указателя стека (в данном случае `esp`) показанного ранее сегмента состояния задачи, подтверждает, что достигнут нижний предел. (Оба местоположения содержат одно и то же значение.)

Чтобы понять, какой компонент использовал память, выделенную в стеке программному потоку ядра, потребуются два значения, полученных нами ранее: база стека и границы стека. Применив к ним команду `dps`, мы получим стек программного потока, описываемый символами, разрешающими имена функций:

```
0: kd> dps 92985000 92988000
92985000 9689091d myfault+0x91d
92985004 9689091d myfault+0x91d
92985008 9689091d myfault+0x91d
...

```

В данном случае повторяющийся адрес соответствует драйверу `Myfault.sys`. Это согласуется с идеей о драйвере устройства, который рекурсивно вызывает сам себя. Каждый вызов функции помешает в стек обратный адрес, увеличивая размер стека и приближая его к пределу, выделенному для данного программного потока. Обратный адрес убирается из стека только после того, как функция вернет управление. Но в данном случае драйвер (или функция) рекурсивно вызывает сам себя, поэтому возвращения управления не происходит.

Код 0xC5 — DRIVER_CORRUPTED_EXPOOL

Выявить причины повреждения пула сложно, а иногда практически невозможно без применения дополнительных инструментов. Для поиска неисправностей данного типа рекомендуется через инструмент Driver Verifier подключить для новых или подозрительных драйверов особый пул. Но сначала имеет смысл потратить несколько минут на анализ сбоя. Это может дать интересные результаты.

Причиной появления стоп-кода `DRIVER_CORRUPTED_EXPOOL` (`0xC5`) является попытка обращения к выгруженной или недействительной памяти при слишком высоком `IRQL`-уровне. Данный стоп-код изначально возникает в ядре как `IRQL_NOT_LESS_OR_EQUAL` (`0xA`). Внутри функции `KeBugCheck2` ядра (для которой `KeBugCheckEx` — всего лишь заглушка) система проверяет значение стоп-кода. При его равенстве `IRQL_NOT_LESS_OR_EQUAL` (`0xA`) система отправляет запрос четвертому параметру отладки, то есть к адресу в памяти, обращение к которому и стало причиной сбоя. Если он находится между областями памяти, содержащими функции пула исполнительной системы Windows, система меняет стоп-код на `DRIVER_CORRUPTED_EXPOOL` (`0xC5`). Это должно подчеркнуть, что причиной сбоя стала не ошибка в одной из процедур пула, а повреждение одной из управляемых этими процедурами структур.

```
DRIVER_CORRUPTED_EXPOOL (c5)
An attempt was made to access a pageable (or completely invalid) address at an
interrupt request level (IRQL) that is too high. This is caused by drivers that have
corrupted the system pool. Run the driver verifier against any new (or suspect)
drivers, and if that doesn't turn up the culprit, then use gflags to enable special
pool.
Arguments:
Arg1: 4f4f4f53, memory referenced
```

продолжение ↗

```
Arg2: 00000002, IRQL
Arg3: 00000000, value 0 = read operation, 1 = write operation
Arg4: 829234a7, address which referenced memory
```

При повреждениях пула трассировка стека практически всегда показывает на программу Ntoskrnl или другой драйвер устройства как на вероятную причину сбоя. В следующем примере трассировка стека программного потока, выполнявшегося в момент сбоя, содержит только функции операционной системы Windows:

```
STACK_TEXT:
8b8e3554 829234a7 badb0d00 00000000 91470d90 nt!KiTrap0E+0x2cf
8b8e3610 8288d2c6 00000000 00000280 76615358 nt!ExAllocatePoolWithTag+0x49d
8b8e3620 8288d19d 00000001 00000053 8b8e38a8 nt!KeAllocateXStateContext+0x25
8b8e3644 8288d6b5 00000003 00000000 8b8e37b4 nt!KeSaveExtendedProcessorState+0x104
8b8e3658 9139b443 8b8e37b4 fe7b8010 8288d038 nt!KeSaveFloatingPointState+0x14
8b8e3864 9139bfdb fe8af408 ffbabd540 00000000 win32k!EngAlphaBlend+0x230
8b8e38d0 9139c394 fe7b8010 fe1c0010 win32k!SURFREFDC::vUnlock+0x1e5
8b8e3974 913a4a2f fe7b8010 fe989010 00000000 win32k!SURFREFDC::vUnlock+0x59e
8b8e39d4 913a4981 fe7b8010 fe989010 00000000 win32k!EngNineGrid+0x6e
8b8e3a34 913a4847 fe7b8010 fe989010 00000000 win32k!EngDrawStream+0x109
8b8e3aa8 913a13a3 8b8e3ba4 00000000 fe989000 win32k!NtGdiDrawStreamInternal+0x232
8b8e3bd4 913a0e09 3a010231 00000000 fe9ef140 win32k!GreDrawStream+0x557
8b8e3d20 828401ea 3a010231 00000060 0012f628 win32k!NtGdiDrawStream+0x8c
8b8e3d20 774570b4 3a010231 00000060 0012f628 nt!KiFastCallEntry+0x12a
0012f49c 75c973a5 75c9738f 3a010231 00000060 ntdll!KiFastSystemCall1Ret
0012f4a0 75c9738f 3a010231 00000060 0012f628 GDI32!NtGdiDrawStream+0xc
0012f5a4 74243efa 3a010231 00000060 0012f628 GDI32!GdiDrawStream+0x432
```

Сгенерированный при попытке обратиться к выгруженной или недействительной памяти фрейм системного прерывания выводит выполнявшиеся команды процессора и значения регистров процессора, на котором работал программный поток. Отладчик с помощью символьного файла для образа ядра может показать имя сбойной функции, используя в качестве ссылки указатель команды:

```
TRAP_FRAME: 8b8e3554 -- (.trap 0xffffffff8b8e3554)
eax=8b8e35f8 ebx=82939940 ecx=4f4f4f4f edx=00000000 esi=82939da8 edi=82939944
eip=829234a7 esp=8b8e35c8 ebp=8b8e3610 iopl=0 ov up ei ng nz na po cy
cs=0008 ss=0010 ds=0023 es=0023 fs=0030 gs=0000 efl=00010a83
nt!ExAllocatePoolWithTag+0x49d:
829234a7 8b4104    mov    eax,dword ptr [ecx+4] ds:0023:4f4f4f53=?????????
```

Как и в предыдущих примерах, набор вопросительных знаков говорит о недействительном адресе, который отладчик просто не в состоянии вывести на экран. В случае данной команды процессор считывает адрес, хранящийся в регистре `ecx`, добавляет к нему значение 4 и пытается обратиться к памяти по этому адресу (для сохранения в регистре `eax`). Но полученный в результате адрес недействителен, что приводит к генерации процессором исключения.

Чтобы понять, почему в регистре `ecx` был сохранен недействительный адрес, проанализируем команды, выполнявшиеся перед сбоем. Вот результат дизассемблирования команд сбойного программного потока в обратном направлении от текущего указателя команды:

```
0: kd> ub 829234a7
nt!ExAllocatePoolWithTag+0x479:
...
829234a5 8b0e    mov    ecx,dword ptr [esi]
```

Анализ показывает, что адрес в регистр `esx` был записан командой, считающей значение, на которое указывает регистр `esi`. Применение команды `dc` к адресу, хранящемуся в регистре `esi` фрейма системного прерывания, показывает, откуда взялось значение `4f4f4f4f`. В выводимых данных интересно то, что каждый адрес появляется в виде пары значений, причем первое (содержащее недействительный адрес) не совпадает с соседним:

```
0: kd> dc 82939da8
82939da8 4f4f4f4f 85045810 82939db0 82939db0 0000.X.....
82939db8 82939db8 82939db8 86f749f8 86f749f8 .....I...I..
82939dc8 82939dc8 82939dc8 82939dd0 82939dd0 .....
82939dd8 82939dd8 82939dd8 82939de0 82939de0 .....
82939de8 82939de8 82939de8 82939df0 82939df0 .....
...
...
```

Заподозрив, что эти значения представляют собой пары адресов, причем первое значение недействительно, выведем адреса, следующие за поврежденными значениями, — это позволит понять причину повреждения. В кодировке ASCII значение `4f4f4f4f` представляет собой `0000`, что подтверждается показанным результатом:

```
0: kd> dc 85045810
85045810 4f4f4f4f 4f4f4f4f 4f4f4f4f 4f4f4f4f 0000000000000000
85045820 4f4f4f4f 4f4f4f4f 4f4f4f4f 4f4f4f4f 0000000000000000
85045830 46524556 00574f4c 00000000 00000000 VERFLOW....
85045840 00000000 00000000 00000000 00000000 .....
85045850 00000000 00000000 00000000 00000000 .....
...
...
```

Проверка выделенных в пуле адресов командой `!pool` подтверждает, что выделенная память вместе с заголовками пула повреждена:

```
0: kd> !pool 85045810
Pool page 85045810 region is Nonpaged pool
 85045000 size: 808 previous size: 0 (Allocated) None
85045808 is not a valid large pool allocation, checking large session pool...
85045808 is freed (or corrupt) pool
Bad previous allocation size @85045808, last size was 101
```

Следует иметь в виду, что обнаруженное повреждение памяти может не являться непосредственной причиной анализируемого нами сбоя. Поэтому любое выявленное повреждение пула требует дальнейшего изучения. В противном случае остается риск дальнейших системных сбоев или даже повреждения дисковых данных.

Далее в сведениях о выделенной в поврежденном пуле памяти нас будет интересовать ссылка на строку "OVERFLOW". Команда `!for_each_module` позволяет искать вхождения нужной строки в каждом из загруженных модулей. Следующая команда отладчика выводит на экран имена всех загруженных драйверов, содержащих строку поиска:

```
0: kd> !for_each_module .foreach (address {s -[1]a @#Base @#End "OVERFLOW"}) {1m 1m a
address}
BTHUSB
CLASSPNP
CLASSPNP
rfcomm
rfcomm
rfcomm
...
myfault
```

Дальнейший анализ аварийного дампа, который на первый взгляд казался совершенно непригодным для диагностики, сужает список подозреваемых драйверов. На этом этапе имеет смысл подключить для перечисленных в списке драйверов устройств особый пул в инструменте Driver Verifier.

Отказы аппаратуры

Еще одним типом сообщений об ошибках является экран аппаратного сбоя. Он появляется при проверке процессором состояния оборудования. Пример такого экрана показан на рис. 14.10. В зависимости от вызвавшей его появление причины экран может содержать дополнительную информацию, указывающую на источник ошибки. Экран аппаратного сбоя не исчезает, так как в этом случае система игнорирует параметр AutoReboot в разделе HKLM\SYSTEM\CurrentControlSet\Control\CrashControl.

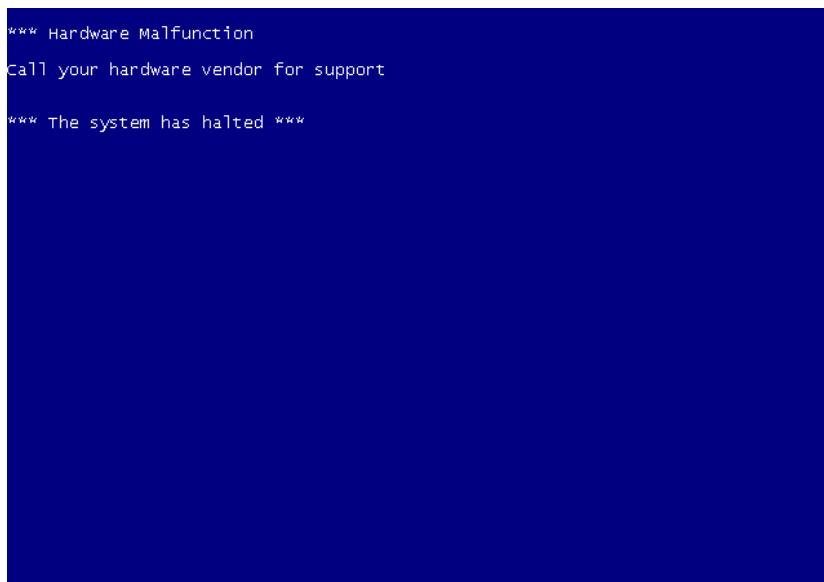


Рис. 14.10. Пример экрана аппаратного сбоя

При любых сообщениях об ошибках, причиной которых, на ваш взгляд, стали неполадки с оборудованием, нужно запустить инструменты диагностики, предоставленные производителем, и определить, есть ли в системе неисправные компоненты и какие. Если вы установили новое аппаратное обеспечение, уберите его и посмотрите, не исчезнет ли проблема. Замените любое оборудование, которое могло выйти из строя.

К появлению экрана аппаратного сбоя приводит и сигнал о генерации материнской платой немаскируемого прерывания (NMI) при отсутствии параметра HKLM\SYSTEM\CurrentControlSet\Control\CrashControl\NMICrashDump. Если вы собирались при помощи кнопки немаскируемого прерывания вручную генерировать аварийный дамп для анализа в автономном режиме, убедитесь в правильности параметра NMICrashDump.

ЭКСПЕРИМЕНТ: ЭКРАННАЯ ЗАСТАВКА В ВИДЕ СИНЕГО ЭКРАНА

Отличным напоминанием о том, как выглядит синий экран, и возможностью пошутить над друзьями и коллегами является экранная заставка Sysinternals Blue Screen, созданная в Sysinternals. Она точно воспроизводит синий экран для установленной на компьютере версии Windows, генерируя соответствующую действительности системную информацию, например список загруженных драйверов. Кроме того, она имитирует автоматическую перезагрузку, показывая экран запуска Windows. При этом в отличие от других экранных заставок, исчезающих при движениях указателя мыши, Blue Screen требует нажатия клавиши.

Утилита PsExec производства Sysinternals позволяет запустить эту экранную заставку на другой системе. Это делается при помощи команды:

```
psexec \\computername -c -f -i -d "SysInternalsBluescreen.scr" -s -accepteula
```

Для выполнения данной команды требуются административные привилегии на удаленной системе. (Переключатели `-u` и `-p` позволяют указать альтернативные учетные данные.) Проверьте, обладают ли ваши коллеги чувством юмора!

Заключение

Представленные в этой главе приемы позволяют проанализировать изрядную часть системных сбоев, но часто для анализа требуются более изощренные приемы, описание которых выходит за рамки темы данной книги. Вот дополнительные ресурсы, которые будут полезны желающим изучить нетривиальные приемы анализа аварийных дампов:

- ❑ Блог группы Microsoft Platforms Global Escalation Services на сайте <http://blogs.msdn.com/ntdebugging>. Здесь вы найдете множество советов, приемов и реальных сценариев, с которыми пришлось столкнуться членам группы.
- ❑ Сайт <http://www.dumpanalysis.org> содержит сотни готовых рецептов, сценариев углубленного анализа и практических рекомендаций.

Об авторах



Марк Руссинович — научный сотрудник подразделения Windows Azure компании Microsoft, занимается облачными операционными системами. Он автор кибертриллера *Zero Day* (Thomas Dunne Books, 2011) и соавтор книги *Windows Sysinternals Administrator's Reference* (Microsoft Press, 2011). Марк присоединился к Microsoft в 2006 году после приобретения Microsoft компанией Winternals Software, соучредителем которой он был с 1996 года, и компании Sysinternals, где он разработал и опубликовал десятки популярных утилит по администрированию и диагностике Windows. Он выступал с докладами на крупных отраслевых конференциях.

Марка можно найти в Твиттере (@markrussinovich) и на Facebook (<http://facebook.com/markrussinovich>).



Дэвид Соломон — президент семинаров для специалистов (www.solsem.com), занимается объяснением внутреннего устройства линейки операционных систем Microsoft Windows NT начиная с 1992 года. Он ведет курсы Windows Internals, известные тысячам разработчиков и IT-специалистов всего мира. Среди его клиентов все основные компании, производящие аппаратное и программное обеспечение, включая Microsoft. Дэвид номинировался на звание Microsoft Most Valuable Professional в 1993 году и с 2005 по 2008 годы.

До того как основать собственную компанию, Дэвид на протяжении девяти лет работал руководителем проектов и разработчиком в компании Digital Equipment Corporation в группе по созданию операционной системы VMS. Его первой книгой была *Windows NT for Open VMS Professionals* (Digital Press/Butterworth Heinemann, 1996), посвященная Windows NT и адресованная программистам и системным администраторам, знающим VMS. Его вторая книга, *Inside Windows NT, Second Edition* (Microsoft Press, 1998), рассказывала о внутреннем устройстве Windows NT 4.0. Начиная с третьего издания (*Inside Windows 2000*) Дэвид работает над этой серией книг в соавторстве с Марком Руссиновичем.

Кроме организации семинаров и преподавания Дэвид регулярно выступает на научных конференциях, таких как Microsoft TechEd и Microsoft PDC. Также он выполнял обязанности научного руководителя на нескольких последних конференциях по Windows NT. Когда Дэвид не занят исследованием Windows, он плавает под парусом, читает и смотрит сериал «Звездный путь».