ВНУТРЕННЕЕ УСТРОЙСТВО

Microsoft Windows

ПИТЕР®

Оглавление

Введение

Шестое издание книги «Внутреннее устройство Windows» предназначено для подготовленных специалистов в области компьютерной техники (для разработчиков и системных администраторов), желающих разобраться во внутренней работе основных компонентов операционных систем Microsoft Windows 7 и Windows Server 2008 R2. Обладая этими знаниями, разработчики смогут лучше понять обоснование проектных решений при создании приложений, предназначенных для платформы Windows. Эти знания также помогут разработчикам вести отладку сложных проблем. Пользу от этой информации могут также получить и системные администраторы, поскольку понимание принципов работы операционной системы «под капотом» помогает понять вопросы поведения системы при стремлении поднять ее производительность и облегчить решение вопросов диагностики системы при возникновении сбоев в ее работе. Прочитав эту книгу, вы сможете лучше разобраться в работе Windows и в том, каковы причины того или иного ее поведения.

Структура книги

Впервые книга «Внутреннее устройство Windows» была разделена авторами на две части.

Эта книга представляет собой первую часть и начинается двумя главами, в которых определяются ключевые понятия, дается представление об инструментальных средствах, используемых в книге, и рассматривается общая архитектура и компоненты системы. В следующих двух главах дается представление о ключевых основополагающих системных и управляющих механизмах. Также охватываются три основных компонента операционной системы: процессы, потоки и задания; безопасность и работа в сети.

Оригинальное издание второй части было опубликовано издательством Microsoft Press осенью 2012 года и охватывает остальные основные подсистемы: ввод-вывод, хранение данных, управление памятью, диспетчер кэша и файловые системы. Рассмотрены процессы запуска и завершения работы и дано описание анализа аварийного дампа.

Глава 1. Общие представления и инструментальные средства

В этой главе будут даны ключевые понятия и термины операционной системы Microsoft Windows, которые будут использоваться по всей книге. Это Windows API, процессы, потоки, виртуальная память, режим ядра и пользовательский режим, объекты, дескрипторы, система безопасности и реестр. Будут также показаны инструментальные средства, которые могут использоваться для исследования таких внутренних компонентов Windows, как отладчик ядра, монитор производительности и ключевой инструментарий из состава Windows Sysinternals (www.microsoft.com/technet/sysinternals). Кроме этого вы узнаете, как пользоваться наборами инструментов для создания драйверов — Windows Driver Kit (WDK) — и для разработки программного обеспечения — Windows Software Development Kit (SDK) — в качестве ресурсов для поиска дополнительной информации о внутренних компонентах Windows.

Материал этой главы нужно обязательно усвоить, потому что это необходимо для понимания остальных глав книги.

Версии операционной системы Windows

В этой книге рассматриваются самые последние клиентские и серверные версии операционных систем Microsoft Windows: Windows 7 (32- и 64-разрядные версии) и Windows Server 2008 R2 (только 64-разрядная версия). Если нет специальных оговорок, предоставленная информация относится ко всем версиям. В табл. 1.1 перечислены названия продуктов Windows, их внутренние номера версий и даты выпусков.

Таблица 1.1. Выпуски операционной системы Windows

Название продукта	Внутренний номер версии	Дата выпуска
Windows NT 3.1	3.1	Июль 1993 г.
Windows NT 3.5	3.5	Сентябрь 1994 г.
Windows NT 3.51	3.51	Май 1995 г.
Windows NT 4.0	4.0	Июль 1996 г.
Windows 2000	5.0	Декабрь 1999 г.
Windows XP	5.1	Август 2001 г.
Windows Server 2003	5.2	Март 2003 г.
Windows Vista	6.0 (сборка 6000)	Январь 2007 г.
Windows Server 2008	6.0 (сборка 6001)	Март 2008 г.
Windows 7	6.1 (сборка 7600)	Октябрь 2009 г.
Windows Server 2008 R2	6.1 (сборка 7600)	Октябрь 2009 г.

ПРИМЕЧАНИЕ

Цифра «7» в имени продукта «Windows 7» не относится к внутреннему номеру версии, она просто является признаком поколения. В действительности, чтобы свести к минимуму проблемы совместимости приложений, для Windows 7 настоящим номером версии считается 6.1, что и показано в табл. 1.1. Это дает возможность тем приложениям, которые проводят проверку на основной номер версии, продолжать вести себя в Windows 7 точно так же, как это делалось в Windows Vista. Фактически, и у Windows 7, и у Server 2008 R2 одинаковые номера версий и сборок, потому что они были созданы из одной и той же кодовой основы Windows.

Основные термины и понятия

В этой книге будут встречаться ссылки на некоторые структуры и понятия, которые могут быть неизвестны отдельным читателям. В этом разделе мы дадим определения тем терминам, которые будут встречаться по всей книге. Ознакомьтесь с ними прежде, чем перейти к следующим главам.

Windows API

Интерфейс прикладного программирования Windows API (application programming interface) является интерфейсом системного программирования в пользовательском режиме для семейства операционных систем Windows. До выхода 64-разрядных версий Windows программный интерфейс для 32-разрядных версий операционных систем Windows назывался Win32 API, чтобы его можно было отличить от исходной 16-разрядной версии Windows API (которая служила интерфейсом программирования для начальных 16-разрядных версий Windows). В данной книге термин *Windows API* будет относиться как к 32-, так и к 64-разрядным интерфейсам программирования в среде Windows.

ПРИМЕЧАНИЕ

Описание Windows API можно найти в документации по набору инструментальных средств разработки программного обеспечения — Windows Software Development Kit (SDK). (См. далее в этой главе раздел «Windows Software Development Kit».) Эта документация доступна на веб-сайте www.msdn.microsoft.com. Она также включена со всеми уровнями подписки в сеть Microsoft Developer Network (MSDN), предназначенную для разработчиков. Дополнительную информацию можно найти по адресу www.msdn.microsoft.com. Великолепное описание процесса программирования с помощью базового Windows API дано в книге «Windows via C/C++», пятое издание, написанной Джеффри Рихтером (Jeffrey Richter) и Кристофером Назарре (Christophe Nasarre) (Microsoft Press, 2007).

Windows API состоит из нескольких тысяч вызываемых функций, которые разбиты на следующие основные категории:

- Базовые службы (Base Services).
- Службы компонентов (Component Services).
- Службы пользовательского интерфейса (User Interface Services).

- Графические и мультимедийные службы (Graphics and Multimedia Services).
- Обмен сообщениями и совместная работа (Messaging and Collaboration).
- Сеть (Networking).
- Веб-службы (Web Services).

В этой книге основное внимание уделяется внутренним составляющим ключевых базовых служб: процессам и потокам, управлению памятью, вводу-выводу и безопасности.

А ЧТО МОЖНО СКАЗАТЬ НАСЧЕТ ТЕХНОЛОГИИ .NET?

Microsoft .NET Framework состоит из библиотеки классов под названием Framework Class Library (FCL) и управляемой среды выполнения кода — Common Language Runtime (CLR). CLR обладает функциями своевременной компиляции, проверки типов, сборки мусора и обеспечения безопасности доступа к коду. Предлагая эти функции, CLR предоставляет среду разработки, повышающую производительность работы программистов и сокращающую количество наиболее распространенных ошибок программирования. Отличное описание .NET Framework и архитектуры ядра этой платформы можно найти в книге «CLR via C#», третье издание, написанной Джеффри Рихтером (Jeffrey Richter) (Microsoft Press, 2010).

Среда CLR реализована, как классический COM-сервер, код которого находится в стандартной Windows DLL-библиотеке, предназначенной для работы в пользовательском режиме. Фактически все компоненты .NET Framework реализованы как стандартные Windows DLL-библиотеки пользовательского режима, наложенные поверх неуправляемых функций Windows API. (Ни один из компонентов .NET Framework не работает в режиме ядра.) Взаимоотношения между этими компонентами показаны на рис. 1.1.

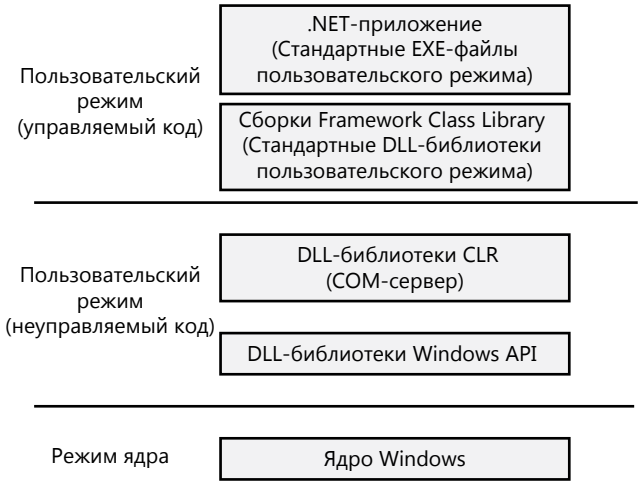

Рис. 1.1. Взаимоотношения между компонентами .NET Framework

Службы, функции и стандартные программы

Некоторые термины в пользовательской и программной документации Windows в разных контекстах имеют разные значения. Например, слово *служба* может

относиться к вызываемой в операционной системе стандартной подпрограмме, драйверу устройства или к обслуживающему процессу. Что именно означают те или иные термины в этой книге, показано в следующем списке:

- **Функции Windows API.** Документированные, вызываемые подпрограммы в Windows API. Например, CreateProcess, CreateFile и GetMessage.
- **Собственные системные службы** (или системные вызовы). Недокументированные базовые службы в операционной системе, вызываемые при работе в пользовательском режиме. Например, NtCreateUserProcess является внутренней службой, которую функция Windows CreateProcess вызывает для создания нового процесса. Определение системного вызова дано в разделе «Диспетчеризация системных служб» (см. главу 3).
- **Функции поддержки ядра** (или подпрограммы). Подпрограммы внутри операционной системы Windows, которые могут быть вызваны только из режима ядра (определение которому будет дано далее в этой главе). Например, ExAllocatePoolWithTag является подпрограммой, вызываемой драйверами устройств для выделения памяти из системных динамически распределяемых областей Windows (называемых пулами).
- **Службы Windows.** Процессы, запускаемые Диспетчером управления службами (Windows service control manager). Например, служба Диспетчер задач запускается в виде процесса, работающего в пользовательском режиме, в котором поддерживается команда at (аналогичная UNIX-командам at или cron). (Примечание: хотя в реестре драйверы устройств Windows определены как «службы», в этой книге в таком качестве они не упоминаются.)
- **Библиотеки DLL** (dynamic-link libraries — динамически подключаемые библиотеки). Набор вызываемых подпрограмм, связанных вместе в виде двоичного файла, который может быть загружен в динамическом режиме приложениями, которые используют эти подпрограммы. В качестве примера можно привести Msvcrt.dll (библиотеку времени выполнения для приложений, написанных на языке C) и Kernel32.dll (одну из библиотек подсистемы Windows API). DLL-библиотеки широко используются компонентами и приложениями Windows, которые работают в пользовательском режиме. Преимущество, предоставляемое DLL-библиотеками по сравнению со статическими библиотеками, заключается в том, что они могут использоваться сразу несколькими приложениями, и Windows обеспечивает наличие в памяти только одной копии кода DLL-библиотеки для тех приложений, в которых имеются ссылки на эту библиотеку. Следует заметить, что неисполняемые .NET-сборки компилируются как DLL-библиотеки, но без каких-либо экспортируемых подпрограмм. CLR анализирует скомпилированные метаданные для доступа к соответствующим типам и элементам классов.

ИСТОРИЯ WIN32 API

Интересно, что Win32 не планировался в качестве исходного интерфейса программирования для той системы, которая в ту пору называласьWindows NT. Поскольку проект Windows NT запускался в качестве замены для OS/2 версии 2, первоначальным интерфейсом программирования был 32-разрядный OS/2 Presentation Manager API. Но через год после запуска проекта произошел взлет поступившей в продажу Microsoft Windows 3.0. В резуль-

тате этого Microsoft сменила направление и сделала Windows NT будущей заменой семейства продуктов Windows, а не заменой OS/2. В связи с этим и возникла необходимость выработать спецификацию Windows API — до этого, в Windows 3.0, API существовал только в виде 16-разрядного интерфейса.

Хотя в Windows API намечалось введение множества новых функций, недоступных в Windows 3.1, Microsoft решила сделать новый API, по возможности, максимально совместимым по именам, семантике и используемым типам данных с 16-разрядным Windows API, чтобы облегчить бремя переноса существующих 16-разрядных Windows-приложений в Windows NT. Этим, собственно, и объясняется тот факт, что многие имена функций и интерфейсов могут показаться непоследовательными: так нужно было для обеспечения совместимости нового Windows API со старым 16-разрядным Windows API.

Процессы, потоки и задания

Хотя при поверхностном взгляде программы и процессы похожи друг на друга, на самом деле они в корне различаются. Программа — это статическая последовательность инструкций, в то время как процесс — это контейнер для набора ресурсов, используемых при выполнении экземпляра программы. На самом высоком уровне абстракции Windows-процесс включает в себя следующее:

- закрытое виртуальное адресное пространство, являющееся набором адресов виртуальной памяти, которым процесс может воспользоваться;
- исполняемую программу, определяющую исходный код и данные, и отображаемую на виртуальное адресное пространство процесса;
- перечень открытых дескрипторов (описателей) различных системных ресурсов — семафоров, коммуникационных портов и файлов, доступных всем потокам процесса;
- связанную с процессом среду безопасности, называемую маркером доступа, идентифицирующим пользователя, группы безопасности, права доступа, виртуализированное состояние системы управления учетными записями пользователей — User Account Control (UAC), сессию и ограниченное состояние учетной записи пользователя;
- уникальный идентификатор, называемый идентификатором процесса, — process ID (внутренняя часть идентификатора называется идентификатором клиента — client ID);
- как минимум один поток выполнения (хотя возможен и абсолютно бесполезный «пустой» процесс).

Каждый процесс также указывает на свой родительский процесс или процесс-создатель. Если родительского процесса больше нет, эта информация не обновляется. Поэтому процесс может указывать на уже несуществующий родительский процесс. Это не создает никаких проблем, поскольку от актуальности этой информации ничего не зависит. При использовании программы исследования процессов ProcessExplorer берется в расчет время запуска родительского процесса, чтобы избежать присоединения дочернего процесса на основе повторно используемого идентификатора процесса. Такое поведение иллюстрируется экспериментом, описанным ниже.

ЭКСПЕРИМЕНТ: ПРОСМОТР ДЕРЕВА ПРОЦЕССОВ

Одним из уникальных атрибутов, относящихся к процессу и неотображаемых большинством инструментальных средств, является идентификатор родительского процесса или процесса-создателя (parent or creator process ID). Это значение можно извлечь с помощью Системного монитора (Performance Monitor) или программным способом, путем запроса Creating Process ID. Дерево процессов может показать такое средство, как Tlist.exe (из состава предназначенных для Windows средств отладки Debugging Tools), используемое с ключом /t. Рассмотрим пример вывода, полученного в результате выполнения команды tlist /t:

```
C:\>tlist /t
System Process (0)
System (4)
  smss.exe (224)
csrss.exe (384)
csrss.exe (444)
  conhost.exe (3076) OleMainThreadWndName
winlogon.exe (496)
wininit.exe (504)
  services.exe (580)
    svchost.exe (696)
    svchost.exe (796)
    svchost.exe (912)
    svchost.exe (948)
    svchost.exe (988)
    svchost.exe (244)
      WUDFHost.exe (1008)
      dwm.exe (2912) DWM Notification Window
    btwdins.exe (268)
    svchost.exe (1104)
    svchost.exe (1192)
    svchost.exe (1368)
    svchost.exe (1400)
    spoolsv.exe (1560)
    svchost.exe (1860)
    svchost.exe (1936)
    svchost.exe (1124)
    svchost.exe (1440)
    svchost.exe (2276)
    taskhost.exe (2816) Task Host Window
    svchost.exe (892)
  lsass.exe (588)
  lsm.exe (596)
explorer.exe (2968) Program Manager
  cmd.exe (1832) Administrator: C:\Windows\system32\cmd.exe - "c:\tlist.exe"  /t
    tlist.exe (2448)
```

Чтобы показать взаимоотношения каждого процесса с его родительскими и дочерними процессами, применяются отступы. Процессы, родители

которых прекратили свое существование, выровнены по левому краю (как Explorer.exe в предыдущем примере), поскольку, даже при наличии прародительского процесса, способов обнаружения связи с ним просто не существует. Windows сохраняет только идентификатор процесса-создателя и не дает ссылок на создателя этого создателя и т. д.
Чтобы продемонстрировать тот факт, что Windows не отслеживает более одного идентификатора родительского процесса, выполните следующие действия:

1. Откройте окно командной строки.
2. Наберите title Parent, чтобы изменить заголовок окна на «Parent» (родительский).
3. Наберите start cmd (что приведет к запуску второго окна командной строки).
4. Наберите во втором окне командной строки title Child, чтобы изменить заголовок окна на «Child» (дочерний).
5. Откройте Диспетчер задач.
6. Наберите во втором окне командной строки mspaint (команду, запускающую Microsoft Paint).
7. Снова обратитесь ко второму окну командной строки и наберите exit. (Заметьте, что Paint остается в рабочем состоянии.)
8. Перейдите в Диспетчер задач.
9. Щелкните на вкладке Приложения.
10. Щелкните правой кнопкой мыши на задаче Parent и выберите пункт Перейти к процессу.
11. Щелкните правой кнопкой мыши на процессе cmd.exe и выберите пункт Завершить дерево процессов.
12. В окне подтверждения Диспетчера задач щелкните на кнопке Завершить дерево процессов.

Первое окно командной строки исчезнет, но по-прежнему можно будет наблюдать окно программы Paint, поскольку оно было потомком во втором поколении завершенного процесса командной строки. Поскольку промежуточный процесс (родительский по отношению к Paint) был завершен, связь между родительским процессом и его потомком во втором поколении была утрачена. ■

Для просмотра процессов и информации о них (а также для внесения изменений) используется несколько инструментальных средств. В описанных ниже экспериментах иллюстрируются различные виды информации о процессах, которые могут быть получены с помощью некоторых из этих средств. Большинство этих средств входит в состав самой Windows, а также в состав отладочного комплекта Debugging Tools for Windows и в состав Windows SDK, но есть и другие автономные средства под маркой Sysinternals. Многие из этих средств показывают частично совпадающие поднаборы сведений об основных процессах и потоках, которые иногда идентифицируются по-разному.

Наверное, самым востребованным средством изучения активности процессов является Диспетчер задач[1]. В ходе следующего эксперимента будет показана

[1] Поскольку в отношении ядра Windows такое понятие, как «задача», не используется, название этого средства — Диспетчер задач — кажется немного странным.

разница между тем, что в списках Диспетчера задач называется приложениями, и процессами.

ЭКСПЕРИМЕНТ: ПРОСМОТР ИНФОРМАЦИИ О ПРОЦЕССАХ С ПОМОЩЬЮ ДИСПЕТЧЕРА ЗАДАЧ

Встроенный в Windows Диспетчер задач предоставляет краткий список процессов, идущих в системе. Запустить этот диспетчер можно одним из четырех способов: 1) нажатием клавиш Ctrl+Shift+Esc, 2) щелчком правой кнопки мыши на панели задач с последующим выбором пункта Запустить диспетчер задач, 3) нажатием клавиш Ctrl+Alt+Delete с последующим щелчком на кнопке Запустить диспетчер задач, или 4) запуском исполняемой программы Taskmgr.exe. Чтобы увидеть перечень процессов, нужно после запуска Диспетчера задач щелкнуть на вкладке Процессы. Обратите внимание на то, что процессы идентифицируются по именам тех образов, экземплярами которых они являются. В отличие от некоторых объектов Windows, процессам нельзя давать глобальные имена. Чтобы посмотреть дополнительную информацию, выберите в меню Вид пункт Показать столбцы и отметьте те столбцы, которые нужно добавить.

Вполне очевидно, что на вкладке Процессы Диспетчера задач показывается список процессов, а вот о содержимом вкладки Приложения с такой же долей очевидности судить нельзя. Там показывается список видимых окон верхнего уровня на всех Рабочих столах интерактивного оконного терминала, к которому вы подключены. (По умолчанию есть только один интерактивный Рабочий стол, но приложение может создать и больше, если воспользуется функцией Windows CreateDesktop, как это сделано в средстве Sysinternals Desktops.) В столбце Состояние показывается, находится ли поток, являющийся владельцем окна, в состоянии ожидания сообщения от этого окна. Состояние «Работает» означает, что поток ожидает поступления в это окно какого-нибудь ввода, а состояние «Не отвечает» означает, что поток не ждет поступления ввода в окно (например, поток может быть запущен или же находиться в ожидании ввода-вывода или в ожидании изменения состояния какого-нибудь объекта синхронизации Windows).

На вкладке Приложения можно сопоставить задачу тому процессу, который является владельцем потока, являющегося владельцем окна задачи. Для этого нужно щелкнуть правой кнопкой мыши на имени задачи и выбрать пункт Перейти к процессу, как показано в предыдущем эксперименте со средством tlist. ■

Инструментальное средство Process Explorer из арсенала Sysinternals предоставляет больше подробностей о процессах и потоках, чем другие доступные средства, поэтому вы увидите его в ряде экспериментов, рассматриваемых в данной книге. Можно упомянуть следующие, допускаемые Process Explorer уникальные возможности показа той или иной информации или выполнения действий:

- демонстрация маркера доступа к процессу (в виде перечня групп, прав доступа и состояния виртуализации);
- выделение изменений в списке процессов и потоков;
- вывод списка служб внутри процессов, являющихся их хозяевами, включая демонстрацию имен и описаний этих служб;
- показ процессов, являющихся частью задания и подробностей задания;
- показ процессов, являющихся хозяевами .NET-приложений и специфичных для .NET-технологии подробностей (таких как список экземпляров класса AppDomain, загруженных сборок и счетчиков производительности CLR);
- показ времени запуска процессов и потоков;
- показ полного списка отображенных на память файлов (не только DLL-библиотек);
- приостановка процесса или потока;
- завершение отдельного потока;
- простота определения тех процессов, которые потребляли за определенный период времени основную часть времени центрального процессора. (Системный монитор (Performance Monitor) может показывать процесс использования центрального процессора для заданного набора процессов, но он не будет автоматически показывать процессы, созданные после запуска сеанса мониторинга производительности, это может сделать только ручная трассировка в двоичном формате вывода.)

Process Explorer также обеспечивает легкий доступ к информации, собранной в одном месте. Вы можете просмотреть:

- дерево процессов (с возможностью свертывать части дерева);
- открытые в процессе дескрипторы (включая безымянные дескрипторы);
- список имеющихся в процессе DLL-библиотек (и отображенных на память файлов);
- активность потоков в процессе;
- стеки потоков пользовательского режима и режима ядра (включая отображение адресов на имена с использованием библиотеки Dbghelp.dll, поставляемой с отладочным пакетом Debugging Tools для Windows);
- более точный процентный показатель использования центрального процессора, вычисляемый с помощью счетчика циклов потока (как объясняется в главе 5 «Процессы, потоки и задания», по сравнению с обычным представлением, это еще более точное представление об активности центрального процессора);
- уровень целостности;
- подробности управления памятью, такие как зафиксированная пиковая нагрузка (peak commit charge), страницы памяти ядра (kernel memory paged) и лимиты резидентного пула (nonpaged pool limits) — другие инструментальные средства показывают только текущий размер.

А теперь проведем начальный эксперимент с использованием средства Process Explorer.

ЭКСПЕРИМЕНТ: ПРОСМОТР ПОДРОБНОСТЕЙ ПРОЦЕССА С ПОМОЩЬЮ PROCESS EXPLORER

Загрузите самую последнюю версию Process Explorer с сайта Sysinternals и запустите ее. При первом запуске вы получите сообщение о неправильной настройке символов. При правильной настройке Process Explorer может получить доступ к информации о символах для вывода символического имени стартовой функции потока и функций в стеке вызовов потока (доступном после двойного щелчка на имени процесса и щелчка на вкладке Threads (Потоки)). Это поможет определить, что делают потоки внутри процесса. Для доступа к символам у вас должен быть установлен пакет средств для отладки Debugging Tools для Windows (его описание будет дано чуть позже). После его установки щелкните на пункте Options (Дополнительные параметры), выберите пункт Configure Symbols (Настройка символов) и заполните поле пути к библиотеке Dbghelp.dll в папке Debugging Tools, а также укажите правильный путь к символам. Например, для 64-разрядной системы вполне подойдет следующая настройка.

В предыдущем примере для доступа к символам и копирования файлов символов с целью хранения этих файлов на локальной машине в каталоге c:\symbols используется сервер востребованных символов. Более подробная информация по настройкам использования сервера символов дана на сайте http://msdn.microsoft.com/en-us/windows/hardware/gg462988.aspx.

При запуске средства Process Explorer оно по умолчанию показывает дерево процессов. У него также имеется дополнительная нижняя панель, в которой могут быть показаны открытые дескрипторы или отображенные на память DLL-библиотеки и другие, отображенные на память файлы. (Все это исследуется в главе 3 «Системные механизмы».) Также это средство дает подсказки для некоторых видов хост-процессов (hosting processes):

- При перемещении указателя мыши над именем хост-процесса служб (Svchost.exe) показываются службы внутри этого процесса.
- Показываются задачи COM-объектов внутри процесса Taskeng.exe (запускаемого Диспетчером задач (Task Scheduler)).
- Показывается целевой объект процесса Rundll32.exe (используется для таких объектов, как элементы Панели управления (Control Panel)).
- Показывается COM-объект, размещающийся внутри процесса Dllhost.exe.
- Показываются процессы вкладок Internet Explorer.
- Показываются хост-процессы консоли.

Оценить некоторые основные возможности Process Explorer вам помогут следующие действия:

1. Обратите внимание на то, что хост-процессы служб по умолчанию выделены розовым фоном. Ваши собственные процессы выделены синим фоном. (Цвета можно настроить по-другому.)
2. Проведите указателем мыши над отображениями имен процессов и обратите внимание на то, что в окне подсказки показывается полное пу-

тевое имя. Как уже говорилось, для процессов конкретного типа в подсказке выводятся и дополнительные сведения.

3. Щелкните на пунктах View (Вид), Select Columns (Выбрать столбцы) и во вкладке Process Image (Отображение процесса) установите флажок Image Path (Отображение пути).
4. Отсортируйте информацию, щелкнув на заголовке столбца Process (Процесс), и обратите внимание на то, что отображение дерева исчезло. (Доступен либо просмотр дерева, либо просмотр информации, отсортированной по любому из показанных столбцов.) Щелкните еще раз для сортировки в порядке следования имен от Z до A. Затем щелкните еще раз, чтобы вернуться к просмотру дерева процессов.
5. Чтобы просматривать только свои процессы, щелкните на пункте View (Вид) и снимите флажок Show Processes From All Users (Показывать процессы от всех пользователей).
6. Щелкните на пунктах Options (Дополнительные параметры), Difference Highlight Duration (Продолжительность различных подсветок) и измените значение на 5 секунд. Затем запустите новый (любой) процесс и обратите внимание на то, что новый процесс будет выделен зеленым фоном в течение 5 секунд. Завершите этот новый процесс и обратите внимание, что он остается выделенным красным фоном в течение 5 секунд, пока не исчезнет из перечня. Этот эффект помогает заметить в вашей системе созданные и завершенные процессы.

И наконец, щелкните дважды на имени процесса и исследуйте различные вкладки, доступные в окне свойств процесса. (Ссылки на эти вкладки будут встречаться в различных экспериментах, предлагаемых в данной книге, там же будут даны объяснения, касающиеся выводимой в них информации.) ■

Поток, выполнение которого планируется операционной системой Windows, является составляющей того или иного процесса. Без него программа, запустившая процесс, не сможет работать. Поток включает в себя следующие основные компоненты:

- Содержимое набора регистров центрального процессора, отображающее его состояние.
- Два стека — один, используемый потоком при выполнении кода в режиме ядра, и один, используемый при выполнении кода в пользовательском режиме.
- Закрытую область хранения, называемую локальным хранилищем потока — thread-local storage (TLS) для использования подсистемами, библиотеками времени выполнения и DLL-библиотеками.
- Уникальный идентификатор, называемый идентификатором потока — *thread ID* (часть внутренней структуры, называемая идентификатором клиента — *client ID* — идентификаторы процессов и идентификаторы потоков образуются из одного и того же пространства имен, поэтому они никогда не перекрываются).
- Иногда у потоков имеется свой собственный контекст безопасности, или маркер (token), часто используемый многопоточными серверными приложениями, который является олицетворением контекста безопасности обслуживаемых этими приложениями клиентов.

Подвергаемые изменениям регистры, стеки и закрытая область хранения называются контекстом потока. Поскольку эта информация для каждой машинной

архитектуры, в среде которой запущена Windows, бывает разной, структура, если это необходимо, также зависит от той или иной архитектуры. Доступ к этой, специфической для конкретной архитектуры, информации (называемой блоком CONTEXT) предоставляется Windows-функцией GetThreadContext.

ПРИМЕЧАНИЕ

В потоках 32-разрядных приложений, запущенных под управлением 64-разрядной версии Windows, будут содержаться блоки как 32-разрядного, так и 64-разрядного контекста, которые Wow64 будет использовать при необходимости для переключения приложения из работы в 32-разрядном режиме в работу в 64-разрядном режиме. У этих потоков будут два стека для работы в пользовательском режиме и два CONTEXT-блока, а обычные функции Windows API будут возвращать 64-разрядный контекст. Но функция Wow64GetThreadContext будет возвращать 32-разрядный context. Дополнительная информация о Wow64 дана в главе 3.

Волокна и потоки планировщика пользовательского режима

Поскольку при переключении выполнения с одного потока на другой задействуется планировщик ядра, такая операция может обойтись весьма дорого, особенно если два потока часто переключаются между собой. В Windows реализованы два механизма сокращения «накладных расходов»: волокна и использование планировщика пользовательского режима — user-mode scheduling (UMS).

Волокна позволяют приложению осуществлять планирование работы своих собственных «потоков», не полагаясь на встроенный в Windows механизм планирования на основе приоритетов. Волокна часто называют «облегченными» потоками, и, с точки зрения планирования работы, ядру они не видны, поскольку реализуются в пользовательском режиме с помощью библиотеки Kernel32.dll. Чтобы воспользоваться волокнами, сначала нужно вызвать Windows-функцию преобразования потока в волокно — ConvertThreadToFiber. Эта функция преобразует поток в запущенное волокно. Впоследствии только что созданное волокно может с помощью функции CreateFiber создавать дополнительные волокна. (У каждого волокна может быть свой собственный набор волокон.) Но в отличие от потока волокно не приступает к выполнению своего кода, пока оно не будет выбрано с помощью функции SwitchToFiber. Новое волокно выполняется до тех пор, пока оно существует, или до тех пор, пока в нем не будет вызвана функция SwitchToFiber, которая выберет для выполнения другое волокно. Дополнительные сведения можно найти в документации Windows SDK по функциям волокон.

UMS-потоки, доступные только на 64-разрядных версиях Windows, имеют те же основные преимущества, что и волокна, но избавлены от многих недостатков, присущих волокнам. UMS-потоки имеют свое собственное состояние потоков ядра, и поэтому они видимы ядру, которое позволяет нескольким UMS-потокам выдавать блокирующие системные вызовы, совместно использовать и вести борьбу за ресурсы и иметь состояние для каждого потока. Но когда двум и более UMS-потокам требуется работать только в пользовательском режиме, они могут периодически переключать контексты выполнения (за счет уступок одного потока другому) без участия планировщика: переключение контекста происходит

в пользовательском режиме. С точки зрения ядра ничего не меняется и продолжается выполнение все того же потока. Когда UMS-поток выполняет операцию, требующую входа в ядро (например, системный вызов), он переключается на выделенный ему поток режима ядра (это называется непосредственным переключением контекста — directed context switch). Дополнительная информация по UMS дана в главе 5.

Хотя у потоков имеется свой собственный контекст выполнения, каждый поток внутри какого-нибудь процесса использует общее виртуальное адресное пространство этого процесса (вдобавок ко всем остальным ресурсам, принадлежащим процессу). Таким образом, все потоки в процессе имеют полноправный доступ к виртуальному адресному пространству процесса. Но потоки не могут случайно сослаться на адресное пространство другого процесса, пока этот другой процесс не сделает часть своего закрытого адресного пространства общим разделом памяти[1], или пока у одного процесса не будет прав на открытие другого процесса для использования таких функций памяти, касающихся обоих процессов, как ReadProcessMemory и WriteProcessMemory.

Как показано на рис. 1.2, кроме закрытого адресного пространства и одного или нескольких потоков, у каждого процесса есть контекст безопасности и список открытых дескрипторов таких объектов ядра, как файлы, общие разделы памяти или один из объектов синхронизации из разряда мьютексов, событий или семафоров.

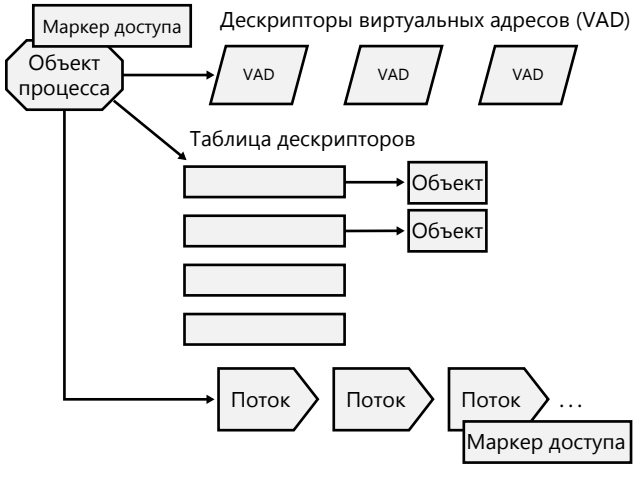

Рис. 1.2. Процесс и его ресурсы

Контекст безопасности каждого процесса хранится в объекте, который называется *маркером доступа* (access token). Маркер доступа процесса содержит идентификацию безопасности и полномочия процесса. По умолчанию потоки не имеют своего собственного маркера доступа, но они могут получить такой маркер, позволяющий отдельным потокам имитировать контекст безопасности другого процесса, включая процессы на удаленной системе Windows, не оказывая

[1] В Windows API этот раздел памяти называется объектом, проецируемым на файл.

при этом никакого влияния на другие потоки процесса. (Более подробно вопросы безопасности процессов и потоков рассмотрены в главе 6 «Безопасность».)

Дескрипторы виртуального адресного пространства — virtual address descriptors (VAD) являются структурами данных, используемых диспетчером памяти для отслеживания виртуальных адресов, используемых процессом.

Windows предоставляет расширение модели процесса, называемое *заданием* (job). Основная функция объектов заданий заключается в том, чтобы управлять группами процессов как единым целым и осуществлять на них одновременное воздействие. Объект задания позволяет управлять конкретными атрибутами и предоставляет ограничения для процесса или процессов, связанных с заданием. Он также записывает основную учетную информацию для всех процессов, связанных с заданием, а также для всех процессов, которые были связаны с заданием ранее, но на данный момент уже завершены. Некоторым образом объект задания компенсирует в Windows отсутствие структурированного дерева процесса, но во многих отношениях он является более мощным средством, чем дерево процесса в UNIX-стиле.

Внутренняя структура заданий, процессов и потоков, механизмы создания процессов и потоков и алгоритмы планирования работы потоков более подробно рассмотрены в главе 5.

Виртуальная память

Windows реализует систему виртуальной памяти на основе плоского (линейного) адресного пространства, предоставляя каждому процессу иллюзию наличия его собственного, большого, закрытого адресного пространства. Виртуальная память предоставляет логическое представление памяти, которое может не соответствовать ее физическому расположению. Во время работы диспетчер памяти при содействии оборудования переводит, или отображает виртуальные адреса на физические, по которым и хранятся данные. Управляя защитой и отображением, операционная система может обеспечивать отсутствие столкновений процессов или перезаписи своих данных. На рис. 1.3 показаны три виртуально-последовательные страницы, отображенные на три непоследовательные страницы физической памяти.

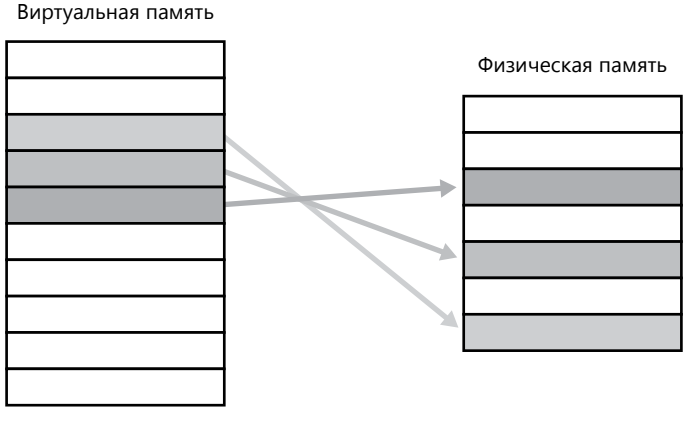

Рис. 1.3. Отображение виртуальной памяти на физическую

Поскольку у большинства систем физической памяти намного меньше общего количества виртуальной памяти, использующейся работающими процессами, диспетчер памяти переводит или осуществляет постраничный перенос части содержимого памяти на диск. Постраничный перенос данных на диск освобождает физическую память, чтобы она могла использоваться другими процессами или самой операционной системой. Когда поток обращается к памяти по виртуальному адресу той страницы, которая была перенесена на диск, диспетчер виртуальной памяти загружает информацию обратно с диска в память. Использование страничной подкачки не требует каких-либо изменений в приложениях, поскольку диспетчер памяти благодаря аппаратной поддержке справляется с этим, не ставя в известность процессы или потоки и не пользуясь их содействием.

Размер виртуального адресного пространства зависит от конкретной аппаратной платформы. На 32-разрядных системах x86 общее виртуальное адресное пространство имеет теоретический максимальный объем, равный 4 Гбайт. По умолчанию Windows распределяет половину этого адресного пространства (нижнюю половину 4-гигабайтного виртуального адресного пространства с адресами от 0x00000000 до 0x7FFFFFFF) между процессами для их уникальных закрытых хранилищ и использует другую половину (верхнюю, с адресами от 0x80000000 до 0xFFFFFFFF) в качестве своей собственной защищенной памяти операционной системы. Отображения нижней половины изменяются в соответствии с виртуальным адресным пространством текущих выполняемых процессов, а отображения верхней части всегда состоят из виртуальной памяти операционной системы. Windows поддерживает параметры загрузки[1], которые дают процессам, выполняющим специально помеченные программы[2], возможность использования до 3 Гбайт закрытого адресного пространства (оставляя 1 Гбайт для операционной системы). Это параметр позволяет таким приложениям, как серверы баз данных, хранить более крупные части баз данных в адресном пространстве процесса, сокращая тем самым потребности в отображении представлений подмножеств базы данных. На рис. 1.4 показаны две типичные схемы виртуальных адресных пространств, поддерживаемых 32-разрядной Windows. (Параметр increaseuserva позволяет помеченным приложениям использовать от 2 до 3 Гбайт адресного пространства.)

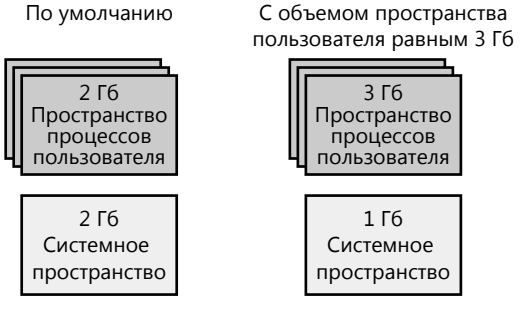

Рис. 1.4. Типичные схемы адресных пространств для 32-разрядной версии Windows

[1] Спецификатор increaseuserva в базе данных конфигурации загрузки — Boot Configuration Database.

[2] В заголовке исполняемого образа должен быть установлен флаг признака большого адресного пространства.

Несмотря на то что 3 Гбайт иметь лучше, чем 2 Гбайт, для отображения очень больших (многогигабайтных) баз данных этого объема все же не хватает. Чтобы отвечать их потребностям на 32-разрядных системах, Windows предоставляет механизм под названием Address Windowing Extension (AWE), позволяющий 32-разрядным приложениям выделять до 64 Гбайт физической памяти, а затем проецировать представления, или окна, на свое 2-гигабайтное виртуальное адресное пространство. Использование AWE возлагает на программиста обязанности по управлению отображением виртуальной памяти на физическую. Этот механизм удовлетворяет потребность в непосредственном доступе к большему объему физической памяти, чем тот, который может в любой заданный момент времени быть отображен на адресное пространство 32-разрядного процесса.

64-разрядная версия Windows предоставляет процессам намного более обширное адресное пространство: 7152 Гбайт на системах IA-64 и 8192 Гбайт на системах x64. На рис. 1.5 показано упрощенное представление структуры адресного пространства 64-разрядной системы. Следует заметить, что показанные объемы памяти не отображают архитектурных ограничений этих платформ. Шестьдесят четыре разряда адресного пространства позволяют адресовать более 17 миллиардов гигабайт, но имеющееся в настоящее время 64-разрядное оборудование ограничивает этот объем более скромным значением. А ограничения, связанные с реализацией текущей 64-разрядной операционной системы Windows, снижают доступное адресное пространство еще больше, сводя его к 8192 Гбайт (8 Тб).

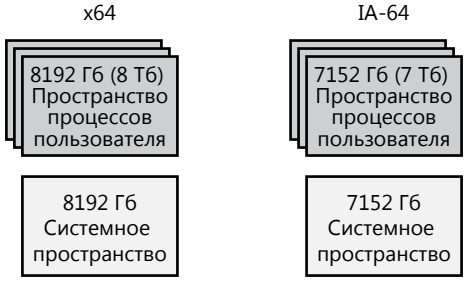

Рис. 1.5. Схемы адресных пространств для 64-разрядной версии Windows

Сравнение режима ядра и пользовательского режима

Чтобы защитить жизненно важные системные данные от доступа и (или) внесения изменений со стороны пользовательских приложений, в Windows используются два процессорных режима доступа (даже если процессор, на котором работает Windows, поддерживает более двух режимов): пользовательский режим и режим ядра. Код пользовательского приложения запускается в пользовательском режиме, а код операционной системы (например, системные службы и драйверы устройств) запускается в режиме ядра. Режим ядра — такой режим работы процессора, в котором предоставляется доступ ко всей системной памяти и ко всем инструкциям центрального процессора. Предоставляя программному обеспечению операционной системы более высокий уровень привилегий, нежели прикладному

программному обеспечению, процессор гарантирует, что приложения с неправильным поведением не смогут в целом нарушить стабильность работы системы.

ПРИМЕЧАНИЕ

В архитектурах процессоров x86 и x64 определены четыре уровня привилегий (или четыре кольца) для защиты системного кода и данных от непреднамеренной или злонамеренной перезаписи в результате выполнения кода, имеющего более низкий уровень привилегий. Windows использует уровень привилегий 0 (или кольцо 0) для режима ядра, и уровень привилегий 3 (или кольцо 3) для пользовательского режима. Причина, по которой в Windows используются только два уровня, заключается в том, что в некоторых аппаратных архитектурах, поддерживаемых в прошлом (например, Compaq Alpha и Silicon Graphics MIPS), были реализованы только два уровня привилегий.

Хотя у каждого Windows-процесса есть свое собственное закрытое адресное пространство, код операционной системы и код драйвера устройства, используют одно и то же общее виртуальное адресное пространство. Каждая страница в виртуальной памяти имеет пометку, показывающую, в каком режиме доступа должен быть процессор для чтения и (или) записи страницы. Доступ к страницам в системном пространстве может быть осуществлен только из режима ядра, тогда как доступ ко всем страницам в пользовательском адресном пространстве может быть осуществлен из пользовательского режима. Станицы, предназначенные только для чтения (например, те страницы, которые содержат статические данные), недоступны для записи из любого режима. Кроме того, при работе на процессорах, поддерживающих защиту той памяти, которая не содержит исполняемого кода (no-execute memory protection), Windows помечает страницы, содержащие данные, как неисполняемые, предотвращая тем самым неумышленное или злонамеренное выполнение кода из областей данных.

32-разрядные версии Windows не защищают закрытую системную память чтения-записи, используемую компонентами операционной системы, запущенными в режиме ядра. Иными словами, в режиме ядра код операционной системы и драйвера устройства имеют полный доступ к системному пространству памяти и могут обойти систему защиты Windows, получив доступ к объектам. Поскольку основная часть кода операционной системы Windows работает в режиме ядра, очень важно, чтобы компоненты, работающие в этом режиме, были тщательно проработаны и протестированы, чтобы не нарушать безопасность системы или не становиться причиной нестабильной работы системы.

Отсутствие защиты также подчеркивает необходимость проявлять особую осторожность при загрузке драйвера устройства стороннего производителя, потому что программное обеспечение, работающее в режиме ядра, имеет полный доступ ко всем данным операционной системы. Этот недостаток стал одной из причин введения в Windows механизма подписи драйверов, который выводит предупреждение пользователю при попытке добавления автоматически настраиваемого (Plug and Play) драйвера, не имеющего подписи (или, при определенной настройке, блокирует добавление такого драйвера). Помимо этого верификатор драйверов — Driver Verifier — помогает создателям драйверов выискивать просчеты (например, переполнение буферов или допущение утечек памяти), способные повлиять на безопасность или стабильность работы системы.

В 64-разрядных версиях Windows политика подписи кода в режиме ядра — Kernel Mode Code Signing (KMCS) — требует, чтобы все 64-разрядные драйверы устройств (не только автоматически настраиваемые) были подписаны криптографическим ключом, присвоенным одним из основных центров сертификации кода. Пользователь не может напрямую заставить систему установить неподписанный драйвер, даже имея права администратора, за единственным исключением: эти ограничения могут быть отключены вручную во время загрузки системы путем нажатия клавиши F8 и выбора дополнительного параметра загрузки Disable Driver Signature Enforcement (Выключить принуждение к подписыванию драйверов). При этом выключаются водяной знак на обоях для рабочего стола и определенные функции системы управления правами на цифровые материалы — digital rights management (DRM).

В главе 2 «Архитектура системы» будет показано, что пользовательские приложения осуществляют переключение из пользовательского режима в режим ядра при осуществлении вызова системной службы. Например, Windows-функции ReadFile, в конечном счете, необходим вызов внутренней стандартной программы Windows, управляющей чтением данных из файла. Поскольку эта стандартная программа обращается к структурам внутренних системных данных, она должна работать в режиме ядра. Переход из режима пользователя в режим ядра осуществляется за счет использования специальной инструкции процессора, которая заставляет процессор переключиться в режим ядра и войти в код диспетчеризации системных служб, вызывающий соответствующую внутреннюю функцию в Ntoskrnl.exe или в Win32k.sys. Перед тем как вернуть управление пользовательскому потоку, процессор переключается в прежний, пользовательский режим работы. Таким образом, операционная система защищает саму себя и свои данные от прочтения и модификации со стороны пользовательских процессов.

ПРИМЕЧАНИЕ

Переход из пользовательского режима в режим ядра (и назад) не влияет на планирование работы потоков как таковое — переход из режима в режим не является переключением контекста. Более подробно диспетчеризация системных служб рассматривается в главе 3.

Таким образом, пользовательский поток вполне может выполняться часть времени в пользовательском режиме, а другую часть времени — в режиме ядра. Фактически, из-за того что основная масса графики и оконная система также работают в режиме ядра, приложения, интенсивно использующие графику, проводят большую часть своего времени в режиме ядра, нежели в пользовательском режиме. Это легко проверить, если запустить приложение, интенсивно использующее графику, например Microsoft Paint или Microsoft Chess Titans, и посмотреть, как распределяется время между пользовательским режимом и режимом ядра, используя для этого один из счетчиков производительности, перечисленных в табл. 1.2. Более сложные приложения могут использовать такие новые технологии, как Direct2D и создание составных изображений (compositing), которые проводят основной объем вычислений в пользовательском режиме и отправляют ядру только исходные данные поверхностей, сокращая время, затрачиваемое на переходы между пользовательскими режимами и режимами ядра.

Таблица 1.2. Счетчики производительности, связанные с режимами работы процессора

Объект: Счетчик	Функция
Процессор: % работы в привилегированном режиме (Processor: % Privileged Time)	Процентный показатель работы отдельного центрального процессора (или всех центральных процессоров) в режиме ядра в течение определенного интервала времени
Процессор: % работы в пользовательском режиме (Processor: % User Time)	Процентный показатель отдельного центрального процессора (или всех центральных процессоров) в пользовательском режиме в течение определенного интервала времени
Процесс: % работы в привилегированном режиме (Process: % Privileged Time)	Процентный показатель работы потоков процесса в режиме ядра в течение определенного интервала времени
Процесс: % работы в пользовательском режиме (Process: % User Time)	Процентный показатель работы потоков процесса в пользовательском режиме в течение определенного интервала времени
Поток: % работы в привилегированном режиме (Thread: % Privileged Time)	Процентный показатель работы потока в режиме ядра в течение определенного интервала времени
Поток: % работы в пользовательском режиме (Thread: % User Time)	Процентный показатель работы потока в пользовательском режиме в течение определенного интервала времени

ЭКСПЕРИМЕНТ: СРАВНЕНИЕ ВРЕМЕНИ РАБОТЫ В РЕЖИМА ЯДРА И В ПОЛЬЗОВАТЕЛЬСКОМ РЕЖИМЕ

Чтобы посмотреть, сколько времени ваша система работает в режиме ядра по сравнению с работой в пользовательском режиме, можно воспользоваться Системным монитором (Performance Monitor). Выполните следующие действия:

1. Запустите Системный монитор (Performance Monitor), открыв меню Пуск (Start) и выбрав пункты Панель управления ▸ Администрирование ▸ Системный монитор (All Programs ▸ Administrative Tools ▸ Performance Monitor). На расположенном слева древовидном раскрывающемся списке инструментов Производительность (Performance) выберите пункты Средства наблюдения (Monitoring Tools)▸ Системный монитор (Performance Monitor).
2. Щелкните на кнопке добавления (+), которая находится на панели инструментов.
3. Раскройте раздел счетчиков Процессор (Processor), щелкните на пункте % работы в привилегированном режиме (% Privileged Time counter) и, удерживая в нажатом состоянии клавишу Ctrl, щелкните на пункте % работы в пользовательском режиме (% User Time).
4. Щелкните на кнопке Добавить (Add), а затем на кнопке OK.
5. Откройте окно командной строки и проведите непосредственное сканирование своего диска C по сети, набрав команду dir \\%computername%\c$ /s.

6. По окончании работы закройте окно инструментального средства.

Такую же картину можно быстро просмотреть с помощью Диспетчера задач (Task Manager). Щелкните на вкладке Производительность (Performance), а затем выберите в меню Вид (View) пункт Вывод времени ядра (Show Kernel Times). На графике загруженности центрального процессора зеленым цветом будет показана его общая загруженность, а красным — загруженность в режиме ядра.

Чтобы увидеть, сколько времени в режиме ядра и в пользовательском режиме использует сам Системный монитор (Performance Monitor), запустите его еще раз, но при этом добавьте отдельные счетчики процесса % работы в пользовательском режиме (% User Time) и % работы в привилегированном режиме (% Privileged Time) для каждого процесса в системе:

1. Если Системный монитор (Performance Monitor) не запущен, запустите его снова. (Если он уже запущен, начните работу с пустого отображения, щелкнув в области графиков правой кнопкой мыши и выбрав пункт Удалить все счетчики (Remove All Counters.)
2. Щелкните на кнопке добавления (+), которая находится на панели инструментов.
3. В доступной области счетчиков раскройте раздел Процесс (Process).
4. Выберите счетчики % работы в пользовательском режиме (% User Time) и % работы в привилегированном режиме (% Privileged Time).
5. Выберите несколько процессов в области Экземпляры выбранного объекта (Instance) (например, mmc, csrss и Idle).
6. Щелкните на кнопке Добавить (Add), а затем на кнопке OK.

7. Интенсивно подвигайте мышью в разные стороны.
8. Выберите на панели инструментов пункт Выделить (Highlight) или нажмите сочетание клавиш Ctrl+H, чтобы включить режим выделения. Текущий выбранный счетчик будет выделен черным цветом.
9. Прокрутите список счетчиков вниз для определения процессов, чьи потоки были запущены при перемещении указателя мыши, и обратите внимание на то, в каком режиме они были запущены, в пользовательском или в режиме ядра.

Вы должны увидеть (найдя в столбце Экземпляр (Instance) процесс mmc), что график времени выполнения процесса, принадлежащего Системному монитору, в режиме ядра и в пользовательском режиме при перемещении мыши пошел вверх, поскольку в нем выполняется прикладной код в пользовательском режиме, и вызываются Windows-функции, запускаемые в режиме ядра. Обратите также внимание на активность потока, принадлежащего процессу csrss и выполняемого в режиме ядра при перемещении мыши. Эта активность возникает благодаря тому, что этому процессу принадлежит исходный поток ввода той подсистемы Windows, выполняемой в режиме ядра, которая обрабатывает ввод с клавиатуры и с мыши. (Более подробная информация о системных потоках дается в главе 2.) И наконец, процесс Idle, который, как можно заметить, тратит почти 100 % своего времени на работу в режиме ядра, на самом деле процессом не является, это ложный процесс, используемый для подсчета холостых циклов центрального процессора. Судя по режиму, в котором запускаются потоки процесса Idle, когда Windows нечего делать, процесс ожидания происходит в режиме ядра. ■

Службы терминалов и множественные сеансы работы

Службы терминалов (Terminal Services)[1] относятся к поддержке в Windows нескольких интерактивных сеансов работы пользователей на одной и той же системе. С помощью этих служб удаленный пользователь может установить сеанс работы на другой машине, войти в систему и запустить приложения на сервере. Сервер передает клиенту графический интерфейс пользователя (а также другие настраиваемые ресурсы, такие как управление звуковой подсистемой и буфером обмена), а клиент передает обратно на сервер пользовательский ввод. (Как и X Window System, Windows разрешает запуск отдельных приложений с удаленным отображением на стороне клиента, вместо удаленного взаимодействия со всем рабочим столом.)

Первый сеанс считается сеансом служб, или сеансом нуль (session zero), и содержит процессы, в которых реализуются системные службы (более подробно этот сеанс рассматривается в главе 4 «Механизмы управления»). Первый сеанс входа в систему на физической консоли машины является сеансом номер один, а дополнительные сеансы могут создаваться с помощью программы подключения к удаленному рабочему столу (Mstsc.exe) или с использованием быстрого переключения пользователей, которое мы рассмотрим чуть позже.

Клиентские версии Windows допускают подключение к машине одного удаленного пользователя, но если кто-нибудь зарегистрируется в консоли, рабочая станция блокируется (то есть системой можно пользоваться либо в локальном, либо

[1] В последних версиях Windows — Службы удаленных рабочих столов. — *Примеч. перев.*

в удаленном режиме, но одновременно в обоих режимах ею пользоваться нельзя). Версии Windows, включающие Windows Media Center, допускают проведение одного интерактивного сеанса и до четырех сеансов Windows Media Center Extender.

Серверные системы Windows поддерживают два одновременных удаленных подключения (для содействия удаленному управлению, например для использования средств управления, требующих регистрации на управляемой машине) и более двух удаленных сеансов, если серверные системы соответствующим образом лицензированы и настроены в качестве терминального сервера.

Все клиентские версии поддерживают несколько созданных локально сеансов, которые могут использоваться поочередно с помощью функции, называемой быстрым переключением пользователей. Когда пользователь выбирает вместо выхода из сеанса отключение своего сеанса (например, щелчком на кнопке Пуск (Start) с последующим выбором пункта Сменить пользователя (Switch User) из подменю Завершение работы (Shutdown) или удержанием в нажатом состоянии клавиши Windows с последующим нажатием клавиши L и щелчком на кнопке Сменить пользователя (Switch User button)), текущий сеанс (процесс, запущенный в этом сеансе, и все, относящиеся к этому процессу структуры данных, дающих описание сеанса) остаются активными в системе, и система возвращается к основному экрану входа в систему. Если в системе регистрируется новый пользователь, создается новый сеанс.

Для приложений, которым нужно знать об их запуске в сеансе терминального сервера, есть набор интерфейсных функций Windows API для определения этого факта программным путем, а также для управления различными аспектами служб терминалов. (Подробности можно найти в документации по Windows SDK и по Remote Desktop Services API.)

В главе 2 дается краткое описание порядка создания сеансов и приводится ряд экспериментов, показывающих, как просматривать информацию о сеансах с помощью различных инструментальных средств, включая отладчик ядра. В главе 3 в разделе «Диспетчер объектов» рассматривается вопрос создания экземпляров системного пространства имен для объектов на сеансовой основе и вопрос определения приложениями, если это им необходимо, наличия других своих экземпляров на той же самой системе.

Объекты и дескрипторы

В операционной системе Windows объект ядра является единственным экземпляром типа объекта времени выполнения, определенного в статическом режиме. Тип объекта состоит из определяемого системой типа данных, функций (методов), работающих с экземплярами типа объекта и набора свойств объекта. При написании Windows-приложений могут встретиться объекты процессов, потоков, файлов и событий (и это лишь несколько примеров). Эти объекты основаны на низкоуровневых объектах, создаваемых и управляемых операционной системой Windows, в которой процесс является экземпляром типа объекта process, файл является экземпляром типа объекта file и т. д.

Свойство объекта является находящимся в объекте полем данных, которое в той или иной степени определяет состояние объекта. К примеру, у объекта типа process будут свойства, включающие идентификатор процесса (process ID),

основной приоритет, учитываемый при планировании запуска процесса, и указатель на объект маркера доступа. Методы объекта, то есть средства манипуляции объектами, обычно считывают или изменяют свойства объекта. Например, методу open объекта process в качестве входных данных будет передаваться идентификатор процесса, а на выходе он будет возвращать указатель на объект.

ПРИМЕЧАНИЕ

Хотя при создании объекта с использованием API диспетчера объектов ядра вызывающий код предоставляет аргумент ObjectAttributes (свойства объекта), этот аргумент не нужно путать с используемым в данной книге более общим значением термина «свойства объекта».

Наиболее существенным отличием объекта от обычной структуры данных является недоступность внутренней структуры объекта за его пределами. Для извлечения данных из объекта или для помещения данных в объект нужно вызвать службу объекта. Напрямую прочитать или изменить данные внутри объекта просто невозможно. Это отличие отделяет базовую реализацию объекта от того кода, который его просто использует. Благодаря такой технологии реализацию объектов впоследствии будет нетрудно изменить.

Объекты с помощью компонента ядра под названием «диспетчер объектов» предоставляют удобные средства для выполнения следующих четырех важных задач операционной системы:

- Предоставление легких для человеческого восприятия имен системных ресурсов.
- Распределение ресурсов и данных среди процессов.
- Защита ресурсов от неавторизованного доступа.
- Отслеживание ссылок, позволяющее системе узнать, когда объект больше не используется, чтобы можно было автоматически освободить выделенные под него ресурсы.

Но в операционной системе Windows объектами являются не все структуры данных. В объекты помещаются только те данные, которые нужно использовать совместно, защитить, снабдить именами или сделать видимыми (через системные службы) для программ, выполняемых в пользовательском режиме. Структуры, используемые только одним компонентом операционной системы для реализации внутренних функций, объектами не являются. Объекты и дескрипторы (ссылки на экземпляр объекта) рассматриваются в главе 3.

Безопасность

С самого начала Windows разрабатывалась с прицелом на безопасность и на соответствие различным официальным правительственным и промышленным требованиям к безопасности, таким как спецификация под названием «Общие критерии оценки безопасности информационных систем» — Common Criteria for Information Technology Security Evaluation (CCITSE). Достижение уровней безопасности, утвержденных правительством, позволяет операционной системе быть конкурентоспособной в сфере правительственных закупок. Разумеется,

многие из соответствующих этим требованиям возможностей являются предпочтительными свойствами для любой многопользовательской системы.

Основные возможности в сфере обеспечения безопасности Windows включают в себя:

- защиту, предоставляемую на усмотрение (узкого круга лиц), а также обязательную защиту целостности для всех общих системных объектов (файлов, каталогов, процессов, потоков и т. д.);
- контроль безопасности (для возможности идентификации субъектов, или пользователей, и инициированных ими действий);
- аутентификацию пользователей при входе в систему и предотвращение доступа одного пользователя к неинициализированным ресурсам (таким как свободное пространство памяти или дисковое пространство), освобожденным другим пользователем.

В Windows имеются три формы управления доступом к объектам. Первая форма — управление избирательным доступом (discretionary access control) — представляет собой защитный механизм, который многими и воспринимается в качестве системы безопасности операционной системы. Этот метод позволяет владельцам объектов (таких как файлы и принтеры) предоставлять доступ или отказывать в доступе всем остальным пользователям. При входе пользователей в систему им дается набор прав доступа, или контекст безопасности. При попытке доступа к объектам их контекст безопасности сравнивается со списком управления доступом (access control list) объекта, к которому осуществляется попытка доступа, чтобы определить, есть ли у того или иного пользователя права на осуществление запрошенной операции.

Когда управления избирательным доступом недостаточно, необходимо управление привилегированным доступом (privileged access control). Оно представляет собой метод, гарантирующий чей-либо доступ к защищенным объектам в отсутствие их владельца. Например, если работник уходит из компании, администратору нужен способ получения доступа к файлам, которые могут быть доступны только этому работнику. В случае когда работа ведется под управлением Windows, администратор может стать владельцем файла и, если это необходимо, распорядиться правами доступа к этому файлу.

И наконец, когда требуется дополнительный уровень безопасности для защиты тех объектов, которые доступны в пределах прав одной и той же учетной записи пользователя, нужен обязательный контроль целостности. Он используется как для изоляции браузера Internet Explorer, работающего в защищенном режиме, от пользовательской конфигурации, так и для защиты объектов, созданных при работе с правами учетной записи администратора с расширенными привилегиями, от доступа к ним со стороны пользователей, работающих с правами учетной записи администратора, не имеющего расширенных привилегий. (Более подробная информация об управлении учетными записями пользователей — User Account Control, UAC, дана в главе 6.)

Безопасность проникает и в интерфейс Windows API. Безопасность на основе объектов реализуется в подсистемах Windows точно так же, как это делается в самой операционной системе. Подсистемы Windows защищают общие Windows-объекты от неавторизованного доступа путем размещения в них дескрипторов безопасности.

При первой попытке приложения получить доступ к общему объекту подсистема Windows проверяет наличие у приложения прав на подобное действие. Если проверка безопасности проходит успешно, подсистема Windows позволяет приложению продолжить свое действие.

Более полное описание системы безопасности Windows дано в главе 6.

Реестр

Если вам приходилось работать с операционными системами Windows, вы, наверное, уже слышали о реестре или даже видели его. Нельзя говорить о внутреннем устройстве Windows, не упоминая при этом реестр, поскольку он представляет собой системную базу данных. В реестре содержится информация, необходимая для загрузки и настройки системы, общесистемных программных настроек, управляющих работой Windows, настроек базы данных безопасности и настроек конфигурации конкретного пользователя (например, какую заставку использовать).

Кроме того, реестр является своеобразным окном в запомненные непостоянные данные — например, данные, касающиеся текущего состояния оборудования системы (какие драйверы устройств загружены, каковы используемые ими ресурсы и т. д.), а также в счетчики производительности Windows. Эти счетчики, которые на самом деле находятся не в реестре, доступны через функции реестра. Более подробно о том, как из реестра можно получить доступ к информации счетчика производительности, рассказывается в главе 4.

Хотя многим пользователям и администраторам Windows заглядывать непосредственно в реестр может никогда и не понадобиться (просматривать или вносить изменения в большинство настроек конфигурации можно с помощью стандартных программ администрирования.), он все равно является полезным источником внутренней информации Windows, потому что в нем содержатся многие настройки, влияющие на производительность и поведение системы. Решив напрямую внести изменения в параметры реестра, нужно проявлять предельную осторожность, любые изменения могут плохо отразиться на производительности системы или, что еще хуже, помешать успешной загрузке системы. В данной книге будут встречаться ссылки на отдельные разделы реестра, относящиеся к рассматриваемым компонентам. Большинство разделов, на которые даются ссылки в данной книге, относятся к общесистемной конфигурации, находящейся в разделе HKEY_LOCAL_MACHINE, для которого будет использоваться аббревиатура HKLM.

Дополнительная информация о реестре и его внутренней структуре дается в главе 4.

Unicode

Windows отличается от других операционных систем тем, что большинство внутренних текстовых строк в ней хранится и обрабатывается в виде расширенных 16-разрядных символьных кодов Unicode. По своей сути Unicode является стандартом международных наборов символов, определяющим 16-разрядные значения для наиболее известных во всем мире наборов символов.

Поскольку многие приложения работают со строками, состоящими из 8-разрядных (однобайтовых) ANSI-символов, многие Windows-функции, которым передаются строковые параметры, имеют две точки входа: в версии Unicode

(расширенной, 16-разрядной) и в версии ANSI (узкой, 8-разрядной). При вызове узкой версии Windows-функции происходит небольшое снижение производительности, поскольку входящие строковые аргументы перед обработкой преобразуются в Unicode, а после обработки, при возвращении приложению, проходят обратное преобразование из Unicode в ANSI. Поэтому если у вас есть старая служба или старый фрагмент кода, который нужно запустить под управлением Windows, но этот фрагмент создан и с использованием текстовых строк, состоящих из ANSI-символов, Windows для своего собственного использования преобразует ANSI-символы в Unicode. Но Windows никогда не станет конвертировать данные внутри файлов: решение о том, в какой кодировки хранить данные: в Unicode или в ANSI, принимается самим приложением.

Независимо от языка, во всех версиях Windows содержатся одни и те же функции. Вместо использования отдельных версий для каждого языка, в Windows используется единый универсальный двоичный код, поэтому отдельно взятая установка может поддерживать несколько языков (путем добавления различных языковых пакетов). Приложения могут также воспользоваться Windows-функциями, предоставляющими возможность во всем мире использовать одни и те же исполняемые файлы приложений, поддерживающие сразу несколько языков.

Дополнительную информацию о Unicode можно найти на сайте www.unicode.org, а также в документации по программированию в библиотеке MSDN.

Подробное исследование внутреннего устройства Windows

Хотя основная масса информации в этой книге основана на чтении исходного кода Windows и на разговорах с разработчиками, вы не должны принимать все на веру. Многие подробности внутреннего устройства Windows могут быть обнаружены и продемонстрированы с помощью использования множества доступных инструментальных средств, поставляемых с Windows или со средствами отладки, работающими под управлением Windows. В данном разделе мы кратко рассмотрим эти инструментальные пакеты.

Чтобы пробудить в вас интерес к исследованию внутреннего устройства Windows, по всей книге разбросаны врезки «Эксперимент», в которых описываются действия, которые нужно выполнить для изучения того или иного аспекта внутреннего поведения Windows. Мы рекомендуем вам провести эти эксперименты, чтобы посмотреть в действии многие вопросы внутреннего устройства Windows, рассматриваемые в данной книге.

В табл. 1.3 приведен перечень основных инструментальных средств, используемых в данной книге, и указаны источники их поступления.

Таблица 1.3. Инструментальные средства для просмотра внутреннего устройства Windows

Инструментальное средство	Имя образа	Источник поступления
Startup Programs Viewer (Средство для просмотра программ, автоматически запускаемых при загрузке системы)	AUTORUNS	Sysinternals[1]

[1] http://technet.microsoft.com/ru-RU/sysinternals. — *Примеч. перев.*

Инструментальное средство	Имя образа	Источник поступления
Access Check (Средство для проверки прав доступа)	ACCESSCHK	Sysinternals
Dependency Walker (Средство для обхода зависимостей)	DEPENDS	www.dependencywalker.com
Global Flags (Средство для работы с глобальными флагами)	GFLAGS	Средства отладки
Handle Viewer (Средство для вывода сведений об открытых дескрипторах для любого процесса в системе)	HANDLE	Sysinternals
Kernel debuggers (Средство отладки ядра)	WINDBG, KD	Средства отладки, Windows SDK
Object Viewer (Средство для просмотра объектов)	WINOBJ	Sysinternals
Performance Monitor (Системный монитор)	PERFMON.MSC	Встроенное средство Windows
Pool Monitor (Монитор пула памяти)	POOLMON	Windows Driver Kit
Process Explorer (Средство для исследования процессов)	PROCEXP	Sysinternals
Process Monitor (Средство для отслеживания процессов)	PROCMON	Sysinternals
Task (Process) List (Средство для вывода перечня задач (процессов))	TLIST	Средства отладки
Task Manager (Диспетчер задач)	TASKMGR	Встроенное средство Windows

Системный монитор

Ссылки на Системный монитор (Performance Monitor), доступный через Панель управления (Control Panel), будут встречаться по всей книге. Особое внимание будет уделяться Системному монитору (Performance Monitor) и Монитору ресурсов (Resource Monitor). Системный монитор выполняет три функции: мониторинг системы, просмотр журналов счетчиков производительности и настройка оповещений (путем использования настроек сборщика данных, который также содержит журналы и трассировку счетчиков производительности и настроечные данные).

Системный монитор предоставляет больше информации о работе вашей системы, чем любое другое отдельно взятое средство. Он включает в себя сотни основных и расширенных счетчиков для различных объектов. Для каждой основной темы, рассматриваемой в данной книге, включается таблица самых важных счетчиков производительности Windows.

В Системном мониторе содержится краткое описание каждого счетчика. Чтобы увидеть описания, нужно в окне Добавить счетчики (Add Counters) установить флажок Отображать описание (Show Description).

Хотя весь низкоуровневый системный мониторинг, рассматриваемый в данной книге, может проводиться с помощью Системного монитора, Windows

также включает служебную программу Монитор ресурсов (запускается из меню Пуск или из вкладки Быстродействие (Performance) Диспетчера задач (Task Manager)), которая показывает четыре основных ресурса: центральный процессор, диск, сеть и память. В своих основных состояниях эти ресурсы показываются с тем же уровнем информации, который можно найти в Диспетчере задач. Но к этому добавляются области, которые могут быть развернуты для получения дополнительной информации.

При раскрытии вкладки ЦП (CPU) показывается информация об использовании центрального процессора для каждого процесса, точно так же, как в Диспетчере задач. Но в этой вкладке добавлен столбец для среднего показателя использования центрального процессора, который может дать более наглядное представление о том, какой из процессоров наиболее активен. Во вкладку ЦП (CPU) также включается отдельное отображение служб, используемого ими центрального процессора и среднего показателя использования этого процессора. Каждый процесс, в рамках которого выполняется служба (хост-процесс), идентифицируется группой той службы, которая на нем выполняется. Как и при использовании Process Explorer, выбор процесса (путем установки соответствующего флажка) приведет к отображению списка поименованных дескрипторов, открытых процессом, а также списка модулей (например, DLL-библиотек), загруженных в адресное пространство процесса. Поле Поиск дескрипторов (Search Handles) может также использоваться для поиска тех процессов, которые открыли дескриптор для заданного поименованного ресурса.

В разделе Память (Memory) отображается почти такая же информация, которую можно получить с помощью Диспетчера задач, но она упорядочена в отношении всей системы. Гистограмма физической памяти отображает текущую организацию этой памяти, разбивая весь объем памяти на зарезервированную, используемую, измененную, находящуюся в режиме ожидания и свободную.

В разделе Диск (Disk), в отличие от остальных разделов, пофайловая информация для ввода-вывода отображается таким образом, чтобы было проще определить наиболее востребованные по записи или чтению файлы системы. Эти результаты могут подвергаться дальнейшей фильтрации по процессам.

В разделе Сеть (Networking) отображаются активные сетевые подключения и владеющие ими процессы, а также количество данных, прошедшее через эти подключения. Эта информация дает возможность увидеть фоновую сетевую активность, которую другим способом может быть трудно обнаружить. Кроме этого показываются имеющиеся в системе активные TCP-подключения, упорядоченные по процессам, с демонстрацией таких данных, как удаленный и локальный порт и адрес, и задержка пакета. И наконец, отображается список прослушиваемых процессом портов, позволяющий администратору увидеть, какие службы (или приложения) в данный момент ожидают подключения к тому или иному порту. Также показываются протокол и политика брандмауэра для каждого порта и процесса.

Следует заметить, что все счетчики производительности Windows являются программно доступными. В разделе «HKEY_PERFORMANCE_DATA» главы 4 дается краткое описание компонентов, используемых для извлечения показаний счетчиков производительности с помощью Windows API.

Отладка ядра

Термин «отладка ядра» означает изучение внутренней структуры данных ядра и (или) пошаговую трассировку функций в ядре. Эта отладка является весьма полезным способом исследования внутреннего устройства Windows, поскольку она позволяет получить отображения внутренней системной информации, недоступной при использовании каких-либо других средств, и дает четкое представление о ходе выполнения кода в ядре.

Прежде чем рассматривать различные способы отладки ядра, давайте исследуем набор файлов, который понадобится для осуществления любого вида такой отладки.

Символы для отладки ядра

Файлы символов содержат имена функций и переменных, а также схему и формат структур данных. Они генерируются программой-компоновщиком (linker) и используются отладчиками для ссылок на эти имена и для их отображения во время сеанса отладки. Эта информация обычно не хранится в двоичном коде, поскольку при выполнении кода она не нужна. Это означает, что без нее двоичный код становится меньше по размеру и выполняется быстрее. Но это также означает, что при отладке нужно обеспечить отладчику доступ к файлам символов, связанных с двоичными образами, на которые идут ссылки во время сеанса отладки.

Для использования любого средства отладки в режиме ядра с целью исследования внутреннего устройства структуры данных ядра Windows (списка процессов, блоков потоков, списка загруженных драйверов, информации об использовании памяти и т. д.) вам нужны правильные файлы символов и, как минимум, файл символов для двоичного образа ядра — Ntoskrnl.exe. (Более подробно этот файл рассматривается в разделе «Краткий обзор архитектуры» главы 2.) Файлы таблицы символов должны соответствовать версии того двоичного образа, из которого они были извлечены. Например, если установлен пакет Windows Service Pack или какое-нибудь исправление, обновляющее ядро, нужно получить соответствующим образом обновленные файлы символов.

Загрузить и установить символы для различных версий Windows нетрудно, а вот обновить символы для исправлений удается не всегда. Проще всего получить нужную версию символов для отладки путем обращения к специально предназначенному для этого серверу символов Microsoft, воспользовавшись для этого специальным синтаксисом для пути к символам, указываемом в отладчике. Например, следующий путь к символам заставляет средства отладки загрузить символы с интернет-сервера символов и сохранить локальную копию в папке c:\symbols:

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

Подробные инструкции по использованию символьного сервера можно найти в файле справки средств отладки или в Интернете на веб-странице http://msdn.microsoft.com/en-us/windows/hardware/gg462988.aspx.

Средства отладки для Windows

В пакет средств отладки для Windows входят современные отладочные инструменты, с помощью которых в данной книге предлагается исследовать внутреннее

устройство Windows. В самую последнюю версию в качестве составной части включен набор для разработки программного обеспечения — Windows Software Development Kit (SDK). Средства из этого набора могут использоваться для отладки как процессов пользовательского режима, так и процессов ядра. (См. соответствующую врезку.)

ПРИМЕЧАНИЕ

Средства отладки Debugging Tools for Windows довольно часто обновляются и выпускаются независимо от версий операционной системы Windows, поэтому почаще проверяйте наличие новых версий.

СОВЕТ. ОТЛАДКА В ПОЛЬЗОВАТЕЛЬСКОМ РЕЖИМЕ

Средства отладки могут также использоваться для подключения к процессу пользовательского режима и для изучения и (или) изменения состояния памяти процесса. При подключении к процессу есть два варианта:

- Навязчивый (Invasive). Если при подключении к запущенному процессу не даны специальные указания, для подключения отладчика к отлаживаемому коду используется Windows-функция DebugActiveProcess. Тем самым создаются условия для исследования и (или) изменения памяти процесса, установки контрольных точек и выполнения других отладочных функций. Windows позволяет остановить отладку без прерывания целевого процесса, если отладчик отключается без прерывания своей работы.
- Ненавязчивый (Noninvasive). При таком варианте отладчик просто открывает процесс с помощью функции OpenProcess. Этот процесс не подключается к другому процессу в качестве отладчика. Это позволяет исследовать и (или) изменять память целевого процесса, но вы не можете устанавливать контрольные точки.

Со средствами отладки можно также открывать файлы дампа процесса пользовательского режима (см. главу 3, раздел, посвященный диспетчеризации исключений).

Для отладки ядра могут использоваться два отладчика: работающий в окне командной строки (Kd.exe) и имеющий графический пользовательский интерфейс, GUI (Windbg.exe). Оба отладчика предоставляют одинаковый набор команд, поэтому выбор всецело зависит от личных предпочтений. Эти средства позволяют выполнять три типа отладки ядра:

- Открыть аварийный дамп-файл, созданный в результате аварийного завершения работы системы.
- Подключиться к действующей, работающей системе и исследовать состояние системы (или установить контрольные точки, если ведется отладка кода драйвера устройства). Для этой операции нужны два компьютера — целевой и ведущий. Целевой компьютер содержит отлаживаемую систему, а ведущий — систему, на которой запущен отладчик. Целевая система может быть подключена к ведущей через нуль-модемный кабель, кабель IEEE 1394 или отладочный кабель USB 2.0. Целевая система должна быть загружена в ре-

жиме отладки (либо путем нажатия F8 в процессе загрузки и выбора пункта Режим отладки (Debugging Mode), либо путем настройки системы на запуск в режиме отладки, используя Bcdedit или Msconfig.exe). Можно также подключиться через поименованный канал, применяемый при отладке через виртуальную машину (созданную такими средствами, как Hyper-V, Virtual PC или VMWare), путем выставления гостевой операционной системой последовательного порта в качестве поименованного канального устройства.

- Системы Windows позволяют также подключиться к локальной системе и исследовать ее состояние. Это называется «локальной отладкой ядра». Чтобы приступить к локальной отладке ядра с помощью отладчика WinDbg, откройте меню File (Файл), выберите пункт Kernel Debug (Отладка ядра), щелкните на вкладке Local (Локальная), а затем щелкните на кнопке OK. Целевая система должна быть загружена в отладочном режиме. Пример появляющегося при этом экрана показан на рис. 1.6. В режиме локальной отладки ядра не работают некоторые команды отладчика ядра (например, команда .dump, предназначенная для создания дампа памяти, хотя такой дамп можно создать с помощью рассматриваемого далее средства LiveKd).

Рис. 1.6. Локальная отладка ядра

Для отображения содержимого внутренней структуры данных, включающей сведения о потоках, процессах, пакетах запросов на ввод-вывод данных и информации об управлении памятью, после подключения к режиму отладки ядра можно воспользоваться одной из множества команд расширения отладчика (команд, начинающихся с символа «!»). По каждой рассматриваемой теме в материал данной книги будут включаться соответствующие команды отладки ядра и состояния экрана отладчика. Отличным подсобным справочным материалом может послужить файл Debugger.chm, содержащийся в установочной папке отладчика WinDbg. В нем приводится документация по всем функциональным возможностям и расширениям отладчика ядра. В дополнение к этому команда dt (display type — отобразить тип) может отформатировать свыше 1000 структур ядра, поскольку в файлах символов ядра для Windows содержится информация о типах, которые отладчик может использовать для форматирования структур.

ЭКСПЕРИМЕНТ: ОТОБРАЖЕНИЕ ИНФОРМАЦИИ О ТИПАХ ДЛЯ СТРУКТУР ЯДРА

Чтобы вывести список структур ядра, чей тип информации включен в символы ядра, наберите в отладчике ядра команду dt nt!_*. Частичный образец вывода имеет следующий вид:

```
lkd> dt nt!_*
          nt!_LIST_ENTRY
          nt!_LIST_ENTRY
          nt!_IMAGE_NT_HEADERS
          nt!_IMAGE_FILE_HEADER
          nt!_IMAGE_OPTIONAL_HEADER
          nt!_IMAGE_NT_HEADERS
          nt!_LARGE_INTEGER
```

Командой dt можно также воспользоваться для поиска определенных структур, используя заложенную в эту команду возможность применения символов-заместителей. Например, если ведется поиск имени структуры для объекта interrupt, нужно набрать команду dt nt!_*interrupt*:

```
lkd> dt nt!_*interrupt*
          nt!_KINTERRUPT
          nt!_KINTERRUPT_MODE
          nt!_KINTERRUPT_POLARITY
          nt!_UNEXPECTED_INTERRUPT
```

Затем, как показано в следующем примере, команду dt можно использовать для форматирования определенной структуры:

```
lkd> dt nt!_kinterrupt
nt!_KINTERRUPT
   +0x000 Type                   : Int2B
   +0x002 Size                   : Int2B
   +0x008 InterruptListEntry     : _LIST_ENTRY
   +0x018 ServiceRoutine         : Ptr64     unsigned char
   +0x020 MessageServiceRoutine : Ptr64     unsigned char
   +0x028 MessageIndex           : Uint4B
   +0x030 ServiceContext         : Ptr64 Void
   +0x038 SpinLock               : Uint8B
   +0x040 TickCount              : Uint4B
   +0x048 ActualLock             : Ptr64 Uint8B
   +0x050 DispatchAddress        : Ptr64     void
   +0x058 Vector                 : Uint4B
   +0x05c Irql                   : UChar
   +0x05d SynchronizeIrql        : UChar
   +0x05e FloatingSave           : UChar
   +0x05f Connected              : UChar
   +0x060 Number                 : Uint4B
   +0x064 ShareVector            : UChar
   +0x065 Pad                    : [3] Char
   +0x068 Mode                   : _KINTERRUPT_MODE
```

```
   +0x06c Polarity         : _KINTERRUPT_POLARITY
   +0x070 ServiceCount     : Uint4B
   +0x074 DispatchCount    : Uint4B
   +0x078 Rsvd1            : Uint8B
   +0x080 TrapFrame        : Ptr64 _KTRAP_FRAME
   +0x088 Reserved         : Ptr64 Void
   +0x090 DispatchCode     : [4] Uint4B
```

Следует заметить, что при выполнении команды dt подструктуры (структуры внутри структур) по умолчанию не показываются. Для выполнения рекурсии подструктур нужно воспользоваться ключом –r. Например, воспользоваться этим ключом для вывода объекта прерывания ядра с показом формата структуры _LIST_ENTRY, хранящейся в поле InterruptListEntry:

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

В файле справки Debugging Tools for Windows также объясняется, как настраиваются и используются отладчики ядра. Дополнительные подробности использования отладчиков ядра, предназначенные непосредственно для создателей драйверов устройств, могут быть найдены в документации по набору Windows Driver Kit.

Инструментальное средство LiveKd

LiveKd является свободно распространяемым средством, предлагаемым на сайте Sysinternals. Это средство позволяет использовать только что рассмотренные стандартные отладчики ядра компании Microsoft для исследования запущенной системы без загрузки этой системы в режиме отладки. Этот подход может пригодиться, когда причины возникновения проблем нужно установить на машине, не запущенной в режиме отладки, — иногда встречаются такие проблемы, которые очень трудно воспроизвести, а перезагрузка с включенным режимом отладки может не показать наличия какой-либо ошибки.

Средство LiveKd запускается точно так же, как WinDbg или Kd, и передает выбранному вами отладчику любые ключи, указанные в командной строке. По умолчанию LiveKd запускает отладчик ядра, работающий в окне командной строки — Kd. Чтобы это средство запустило WinDbg, нужно указать ключ –w. Для вывода файлов справки, касающихся ключей LiveKd, нужно указать ключ –?.

LiveKd представляет отладчику смоделированный файл аварийного дампа, поэтому в LiveKd можно выполнять любые операции, поддерживаемые в отношении аварийного дампа. Поскольку при моделировании аварийного дампа

средство LiveKd полагается на физическую память, отладчик ядра может попасть в такую ситуацию, при которой структуры данных находятся в центре области, изменяемой системой, и теряют свою согласованность. При каждом запуске отладчика он начинает со свежего представления состояния системы. Если нужно обновить снимок состояния (snapshot), выйдите из отладчика (с помощью команды q), и LiveKd выведет запрос на его повторный запуск. Если отладчик входит в цикл вывода на печать, нажмите сочетание клавиш Ctrl+C для прерывания вывода и выхода из отладчика. Если отладчик завис, нажмите сочетание клавиш Ctrl+Break, чтобы остановить процесс отладки. LiveKd выведет запрос на повторный запуск отладчика.

Windows Software Development Kit

Набор инструментальных средств Windows Software Development Kit (SDK) доступен в виде части программы подписки MSDN или же может быть свободно загружен с сайта msdn.microsoft.com. Кроме средств отладки Debugging Tools, он содержит документацию, заголовочные файлы языка C и библиотеки, необходимые для компиляции и компоновки Windows-приложений. (Хотя Microsoft Visual C++ поставляется с копией этих заголовочных файлов, версии, содержащиеся в Windows SDK, всегда соответствуют самым свежим версиям операционных систем Windows, а версии, поставляемые с Visual C++, могут быть более старыми, актуальными на время выхода Visual C++.) С точки зрения внутреннего устройства, Windows SDK интересны заголовочные файлы Windows API (\Program Files\Microsoft SDKs\Windows\v7.0A\Include). Некоторые из инструментальных средств этого набора также поставляются в виде экземпляров исходного кода, как в Windows SDK, так и в библиотеке MSDN Library.

Windows Driver Kit

Набор инструментальных средств Windows Driver Kit (WDK) также доступен по программе подписки MSDN и точно так же, как и Windows SDK, его можно свободно скачать с сайта. Документация по Windows Driver Kit включена в библиотеку MSDN.

Хотя набор WDK предназначен для разработчиков драйверов устройств, он является обширным источником информации о внутреннем устройстве Windows. А в документации по WDK содержится полное описание всех Windows-функций поддержки ядра и механизмов, используемых драйверами устройств как в виде учебного пособия, так и в виде справочника.

В WDK, помимо документации, содержатся заголовочные файлы (в частности, ntddk.h, ntifs.h и wdm.h), определяющие ключевую внутреннюю структуру данных и константы, а также интерфейсы ко многим внутренним системным подпрограммам. Эти файлы пригодятся при исследовании внутренних структур данных Windows, используемых при отладке ядра, поскольку, несмотря на то что общая схема и содержимое этих структур показаны в данной книге, подробные описания вплоть до каждого поля (например, размера и типа данных) в ней не даны. А в WDK дается полное описание таких структур (например, заголовков для диспетчера объектов, блоков ожидания, событий, мутантов, семафоров и т. д.).

При желании углубиться в изучение системы ввода-вывода и модели драйверов, выходя за рамки предлагаемого в данной книге, читайте WDK-документацию (особенно Руководство по устройству архитектуры драйверов, работающих в режиме ядра — Kernel-Mode Driver Architecture Design Guide, и Справочные руководства). Также могут пригодиться книги Уолтера Они (Walter Oney) «Использование Microsoft Windows Driver Model», второе издание (Питер, 2007) и Пенни Орвика (Penny Orwick) и Гая Смита (Guy Smith) «Windows Driver Foundation. Разработка драйверов» (BHV, 2008).

Заключение

В данной главе были представлены ключевые технические понятия и термины Windows, используемые во всем остальном тексте книги. Вы также получили начальное представление о многих полезных инструментальных средствах, доступных для проникновения в глубь внутреннего устройства Windows. Теперь мы готовы приступить к исследованиям внутреннего устройства системы, начиная с общего вида системной архитектуры и ее ключевых компонентов.

Глава 2. Архитектура системы

После изучения терминов, понятий и инструментальных средств можно приступить к исследованию внутренних задач конструирования и структуры операционной системы Microsoft Windows. В данной главе рассматривается общая архитектура системы — основные компоненты, порядок их взаимодействия и контекст, в котором они работают. Чтобы заложить основы понимания внутреннего устройства Windows, сначала рассмотрим требования и цели, определяющие очертания исходной конструкции и спецификации системы.

Требования и цели разработки

В далеком 1989 году спецификация Windows NT определялась с учетом следующих требований:

- Создать по-настоящему передовую, 32-разрядную операционную систему, работающую с виртуальной памятью и допускающую повторный вход.
- Обеспечить возможность работы на разных аппаратных архитектурах и платформах.
- Обеспечить возможность работы и масштабирования на симметричных мультипроцессорных системах.
- Обеспечить возможность работы в качестве распределенной вычислительной платформы, как в роли сетевого клиента, так и в роли сервера.
- Обеспечить возможность запуска большинства существующих 16-разрядных приложений MS-DOS и Microsoft Windows 3.1.
- Обеспечить выполнение правительственных требований о совместимости со стандартом POSIX 1003.1.
- Обеспечить выполнение требований правительства и промышленности, касающихся безопасности операционных систем.
- Обеспечить адаптируемость к всемирному рынку за счет поддержки Unicode.

Чтобы заложить основы для принятия тысяч решений по созданию системы, отвечающей вышеперечисленным требованиям, команда разработчиков в самом начале работы над проектом решила реализовать следующие замыслы:

- **Расширяемость.** Код должен создаваться с учетом удобства его наращивания и изменения в соответствии с изменениями требований рынка.
- **Переносимость.** Система должна работать на разных аппаратных архитектурах, и, в соответствии с требованиями рынка, должна относительно легко переноситься на их новые образцы.
- **Надежность и отказоустойчивость.** Система должна защищать саму себя как от внешних угроз, так и от внутренних сбоев. Приложения не должны иметь возможность нанесения вреда операционной системе или другим приложениям.
- **Совместимость.** Хотя Windows NT должна была стать новым шагом по сравнению с существовавшей в то время технологией, ее пользовательский интерфейс и API должны быть совместимы с прежними версиями Windows

и с MS-DOS. У нее также должна быть возможность взаимодействия с другими системами, такими как UNIX, OS/2 и NetWare.

- **Производительность.** С учетом ограничений, накладываемых другими проектировочными замыслами, система должна проявлять максимально возможное быстродействие и реакцию на каждой аппаратной платформе.

По мере исследования подробностей внутренней структуры и работы Windows, вы увидите, как эти исходные замыслы проектирования и рыночные требования были успешно сплетены в конструкцию системы. Но прежде чем приступить к исследованиям, нужно изучить общую модель проектирования Windows и сравнить ее с моделями других современных операционных систем.

Модель операционной системы

В большинстве многопользовательских операционных систем приложения отделены от самой операционной системы. Код ядра операционной системы запускается в привилегированном режиме работы процессора (в данной книге он называется режимом ядра), имея доступ к системным данным и к оборудованию, а код приложения запускается в непривилегированном режиме работы процессора (пользовательский режим), с ограниченным набором доступных интерфейсов, ограниченным доступом к системным данным и без прямого доступа к оборудованию. Когда программа, запущенная в пользовательском режиме, вызывает системную службу, в процессоре выполняется специальная инструкция, переключающая вызывающий поток в режим ядра. Когда системная служба завершит свою работу, операционная система переключит контекст потока назад, в пользовательский режим, и позволит вызывающей программе продолжить свое выполнение. Windows похожа на большинство UNIX-систем тем, что она является монолитной операционной системой — в том смысле, что основная часть кода операционной системы и драйверов устройств совместно используют одно и то же защищенное пространство памяти, используемое в режиме ядра. Это означает, что любой компонент операционной системы или драйвер устройства потенциально может разрушить данные, используемые другими компонентами операционной системы. Но в Windows реализованы такие механизмы защиты ядра, как PatchGuard и Kernel Mode Code Signing (см. главу 3), помогающие ликвидировать и предотвратить проблемы, связанные с общим адресным пространством, используемым при работе в режиме ядра.

Разумеется, все эти компоненты операционной системы полностью защищены от неправильно работающих приложений, поскольку у этих приложений нет прямого доступа к коду и данным, находящимся в привилегированной части операционной системы (хотя они могут вызвать другие службы ядра). Эта защита является одной из причин наличия у Windows репутации отказоустойчивой и стабильной операционной системы, как в качестве сервера приложений, так и в качестве платформы для рабочей станции. Наряду с этим быстродействием Windows обладает такими службами ядра операционной системы, как управление виртуальной памятью, файловый ввод-вывод, сеть и совместное использование файлов и принтеров.

Компоненты Windows, работающие в режиме ядра, являются также воплощением принципов объектно-ориентированного проектирования. К примеру, они

вообще не проникают в чужие структуры данных для доступа к информации, поддерживаемой отдельными компонентами. Вместо этого для передачи параметров и доступа к структурам данных и (или) для их изменения используются формальные интерфейсы.

Несмотря на повсеместное использование объектов для представления общих системных ресурсов, Windows, в строгом смысле слова, объектно-ориентированной системой не является. Основная часть кода операционной системы написана на языке C для решения задач переносимости. Этот язык программирования не поддерживает напрямую такие объектно-ориентированные конструкции, как динамическое связывание типов данных, полиморфные функции или наследования классов. Поэтому имеющаяся в Windows реализация объектов на основе языка C заимствует свойства, присущие объектно-ориентированным языкам, но не зависит от них.

Краткий обзор архитектуры

После краткого обзора исходных замыслов проектирования и компоновки Windows рассмотрим основные компоненты системы, составляющие ее архитектуру. Упрощенная версия архитектуры показана на рис. 2.1. Нужно иметь в виду, что эта схема носит общий характер и не отображает все компоненты. (Например, на ней не показаны сетевые компоненты и иерархия различных типов драйверов устройств.)

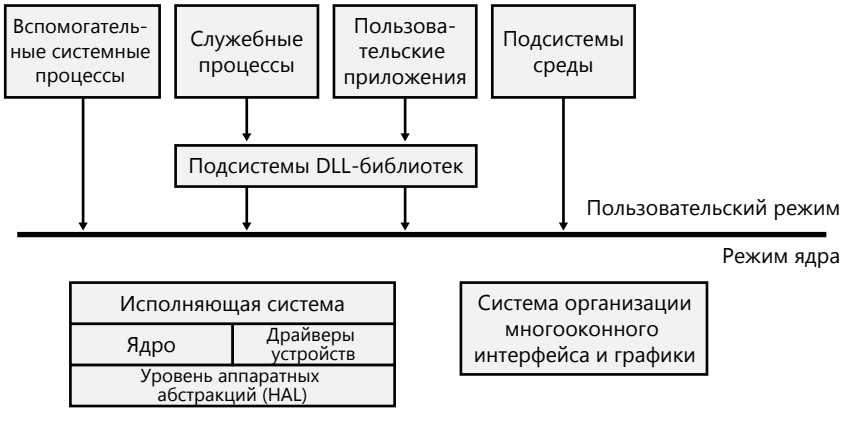

Рис. 2.1. Упрощенное представление архитектуры Windows

В первую очередь на рис. 2.1 нужно обратить внимание на прямую линию, разделяющую части операционной системы Windows, работающие в пользовательском режиме и в режиме ядра. Прямоугольники, находящиеся выше линии, представляют процессы, идущие в пользовательском режиме, а компоненты, показанные ниже линии, представляют системные службы, работающие в режиме ядра. Как уже говорилось в главе 1, потоки пользовательского режима выполняются в защищенном адресном пространстве процесса (когда они выполняются в режиме ядра, у них есть доступ к системному пространству). Таким образом, у вспомогательных системных процессов, у процессов служб, у пользовательских

приложений и у подсистем среды окружения, — у всех есть свое собственное закрытое адресное пространство.

Четырем основным процессам пользовательского режима можно дать следующие описания:

- Фиксированные (или реализованные на аппаратном уровне) *вспомогательные системные процессы*, такие как процесс входа в систему и администратор сеансов — Session Manager, которые не входят в службы Windows (они не запускаются диспетчером управления службами).
- *Служебные процессы*, реализующие такие службы Windows, как Диспетчер задач (Task Scheduler) и спулер печати (Print Spooler). Как правило, от служб требуется, чтобы они работали независимо от входов пользователей в систему. Многие серверные приложения Windows, такие как Microsoft SQL Server и Microsoft Exchange Server, также включают компоненты, работающие как службы.
- *Пользовательские приложения*, которые могут относиться к одному из следующих типов: для 32- или 64-разрядной версии Windows, для 16-разрядной версии Windows 3.1, для 16-разрядной версии MS-DOS или для 32- или 64-разрядной версии POSIX. Следует учесть, что 16-разрядные приложения могут запускаться только на 32-разрядной версии Windows.
- *Серверные процессы подсистемы окружения*, которые реализуют часть поддержки среды операционной системы или специализированную часть, представляемую пользователю и программисту. Изначально Windows NT поставляется тремя подсистемами среды: Windows, POSIX и OS/2. Но подсистемы POSIX и OS/2 последний раз поставлялись с Windows 2000. Выпуски клиентской версии Windows Ultimate и Enterprise, а также все серверные версии включают поддержку для усовершенствованной подсистемы POSIX, которая называется подсистемой для приложений на основе Unix (Unix-based Applications, SUA).

Обратите внимание на прямоугольник «Подсистемы DLL-библиотек», который на рис. 2.1 находится под прямоугольниками «Служебные процессы» и «Пользовательские приложения». При выполнении под управлением Windows пользовательские приложения не вызывают имеющиеся в операционной системе Windows службы напрямую, а проходят через одну или несколько подсистем динамически подключаемых библиотек (dynamic-link libraries, DLL). Подсистемы DLL-библиотек предназначены для перевода документированной функции в соответствующий внутренний (и зачастую недокументированный) вызов системной службы. Этот перевод может включать в себя (или не включать) отправку сообщения процессу подсистемы среды, обслуживающему пользовательское приложение.

В Windows входят следующие компоненты, работающие в режиме ядра:

- *Исполняющая система* Windows содержит основные службы операционной системы, такие как управление памятью, управление процессами и потоками, безопасность, ввод-вывод, сеть и связь между процессами.
- *Ядро* Windows состоит из низкоуровневых функций операционной системы, таких как диспетчеризация потоков, диспетчеризация прерываний и исключений и мультипроцессорная синхронизация. Оно также предоставляет набор подпрограмм и базовых объектов, используемых остальной исполняющей системой для реализации высокоуровневых конструктивных элементов.

- К *драйверам устройств* относятся как аппаратные драйверы устройств, которые переводят вызовы функций ввода-вывода в запросы ввода-вывода конкретного аппаратного устройства, так и неаппаратные драйверы устройств, такие как драйверы файловой системы и сети.
- *Уровень аппаратных абстракций* (hardware abstraction layer, HAL), являющийся уровнем кода, который изолирует ядро, драйверы устройств и остальную исполняющую систему Windows от аппаратных различий конкретных платформ (таких как различия между материнскими платами).
- *Система организации многооконного интерфейса и графики*, реализующая функции графического пользовательского интерфейса (graphical user interface, GUI), более известные как имеющиеся в Windows USER- и GDI-функции, предназначенные для работы с окнами, элементами управления пользовательского интерфейса и графикой.

В табл. 2.1 перечислены имена файлов основных компонентов операционной системы Windows. (Вам нужно знать эти имена файлов, потому что на некоторые системные файлы мы будем ссылаться по именам.) Каждый из этих компонентов будет более подробно рассмотрен либо в этой, либо в одной из следующих глав.

Таблица 2.1. Основные системные файлы Windows

Имя файла	Компоненты
Ntoskrnl.exe	Исполняющая система и ядро
Ntkrnlpa.exe (только в 32-разрядных системах)	Исполняющая система и ядро с поддержкой расширения физического адреса — Physical Address Extension (PAE), позволяющего 32-разрядным системам осуществлять адресацию вплоть до 64 Гб физической памяти и помечать память как не содержащую исполняемый код
Hal.dll	Уровень аппаратных абстракций
Win32k.sys	Часть подсистемы Windows, работающей в режиме ядра
Ntdll.dll	Внутренние вспомогательные функции и заглушки диспетчера системных служб к исполняющим функциям
Kernel32.dll, Advapi32.dll, User32.dll, Gdi32.dll	Основные Windows-подсистемы DLL-библиотек

Прежде чем приступить к подробному изучению этих компонентов системы, давайте рассмотрим некоторые основы конструкции ядра Windows. Начинем с того, как в Windows осуществляется переносимость, позволяющая этой операционной системе работать на нескольких аппаратных архитектурах.

Переносимость

Windows разрабатывалась для работы на разных аппаратных архитектурах. Исходный выпуск Windows NT поддерживал архитектуры x86 и MIPS. Вскоре после этого была добавлена поддержка процессора Alpha AXP компании Digital Equipment Corporation (она была куплена компанией Compaq, слившейся в последствии с компанией Hewlett-Packard). (Хотя процессор Alpha AXP был

64-разрядным, Windows NT запускалась в 32-разрядном режиме. В процессе разработки Windows 2000 на Alpha AXP была запущена собственная 64-разрядная версия, но она так и не была выпущена.) Поддержка четвертой процессорной архитектуры, Motorola PowerPC, была добавлена в Windows NT 3.51. Но из-за изменений рыночных потребностей к началу разработки Windows 2000 поддержка архитектур MIPS и PowerPC была прекращена. Чуть позже компания Compaq отозвала поддержку архитектуры Alpha AXP, и Windows 2000 стала поддерживать только архитектуру x86. В Windows XP и Windows Server 2003 была добавлена поддержка трех 64-разрядных процессорных семейств: Intel Itanium IA-64, AMD64 и 64-разрядных версий Intel Extension Technology (EM64T) для x86. (Они были совместимы с архитектурой AMD64, хотя и обладали некоторыми отличиями в поддерживаемых командах.) Последние два процессорных семейства получили название «64-разрядных расширенных систем», и в этой книге они будут рассматриваться как x64. Как Windows запускает 32-разрядные приложения на 64-разрядных версиях, будет показано в главе 3.

Переносимость Windows между аппаратными архитектурами и платформами достигается двумя основными способами:

- Windows имеет многоуровневую конструкцию, в которой нижний уровень системы предназначается для архитектуры конкретного процессора или конкретной платформы, и он выделен в отдельные модули, чтобы верхние уровни могли быть защищены от различий между архитектурами и между аппаратными платформами. Ключевыми компонентами, обеспечивающими переносимость операционной системы, являются ядро (которое содержится в файле Ntoskrnl.exe) и уровень аппаратных абстракций (или HAL, который содержится в библиотеке Hal.dll). Более подробно оба этих компонента еще будут рассмотрены в данной главе. Функции, зависящие от конкретной архитектуры (такие, как переключение контекста потока и диспетчеризация системных прерываний), реализованы в ядре. Функции, которые могут отличаться между системами в рамках одной архитектуры (например, при использовании разных материнских плат), реализованы в HAL. Единственным компонентом, имеющим существенный объем кода, предназначенного для конкретной архитектуры, является диспетчер памяти, но по сравнению со всей системой в целом этот объем незначителен.
- В подавляющем большинстве Windows написана на языке C, но встречаются и фрагменты, написанные на C++. Ассемблер используется только для тех частей операционной системы, которые должны напрямую взаимодействовать с системным оборудованием (например, обработчик системных прерываний) или для тех частей, работа которых сильно влияет на производительность системы (например, для переключения контекста). Код на языке ассемблера присутствует не только в ядре и на HAL-уровне, но также и в некоторых других основных местах операционной системы (например, в подпрограммах, реализующих инструкции взаимной блокировки, а также в модуле из вызова локальных процедур), в части подсистемы Windows, которая выполняется в режиме ядра, и даже в некоторых библиотеках пользовательского режима, например в коде запуска процесса в Ntdll.dll (в системной библиотеке, рассматриваемой далее в этой главе).

Симметричная мультипроцессорная обработка

Многозадачность представляет собой технологию операционной системы для совместного использования одного процессора несколькими потоками выполнения. Но когда у компьютера более одного процессора, он может выполнять несколько потоков одновременно. Таким образом, если многозадачная операционная система только кажется исполняющей одновременно несколько потоков, мультипроцессорная операционная система делает это на самом деле, выполняя по одному потоку на каждом своем процессоре.

Как говорилось в начале главы, одним из основных конструкторских замыслов для Windows была ее обязательная работа на мультипроцессорных компьютерных системах. Windows является *симметричной мультипроцессорной* (symmetric multiprocessing, SMP) операционной системой. В ней нет главного процессора — как операционная система, так и пользовательские потоки могут быть спланированы для работы на любом процессоре. Кроме этого, все процессоры совместно используют одно и то же адресное пространство. Эта модель отличается от *асимметричной мультипроцессорной обработки* (asymmetric multiprocessing, ASMP), при которой каждой операционной системе обычно выделяется один процессор для выполнения кода ядра операционной системы, а на других процессорах выполняется только пользовательский код. Различия между двумя мультипроцессорными моделями показаны на рис. 2.2.

Windows также поддерживает три современных типа мультипроцессорных систем: многоядерные, гиперпотоковость (Hyper-Threading) и с технологией доступа к неоднородной памяти — NUMA (non-uniform memory architecture). Они еще будут кратко упомянуты в следующих разделах. (Полное и подробное описание поддержки диспетчеризации этих систем будет дано в разделе, посвященном диспетчеризации потоков в главе 5 «Процессы, потоки и задания».)

Гиперпотоковость (Hyper-Threading) является технологией, представленной компанией Intel и предоставляющей два логических процессора для каждого физического ядра. У каждого логического процессора есть свое собственное состояние центрального процессора, но механизм исполнения команд и процессорная кэш-память у них общие. Это позволяет одному логическому центральному процессору успешно работать, в то время как другой логический центральный процессор остановлен (например, после промаха при обращении к кэш-памяти или после неправильно спрогнозированного условного перехода). Алгоритмы диспетчеризации усовершенствованы с учетом оптимального использования машин, допускающих гиперпотоковость, например путем диспетчеризации потоков на простаивающий физический процессор, а не на простаивающий логический процессор на физическом процессоре, где другой логический процессор занят работой. Более подробно диспетчеризация потоков рассмотрена в главе 5.

На NUMA-системах процессоры сгруппированы в небольшие блоки, называемые *узлами*. У каждого узла есть свои собственные процессоры и память, и он подключен к более крупной системе через кэш-когерентную объединяющую шину. Тем не менее Windows на NUMA-системе работает как SMP-система, в которой все процессоры имеют доступ ко всей памяти — при том, что обращение к локальной памяти узла осуществляется быстрее, чем к памяти, подключенной к другим узлам. Система пытается повысить производительность путем диспетчеризации потоков на процессах того же узла, на котором находится используемая

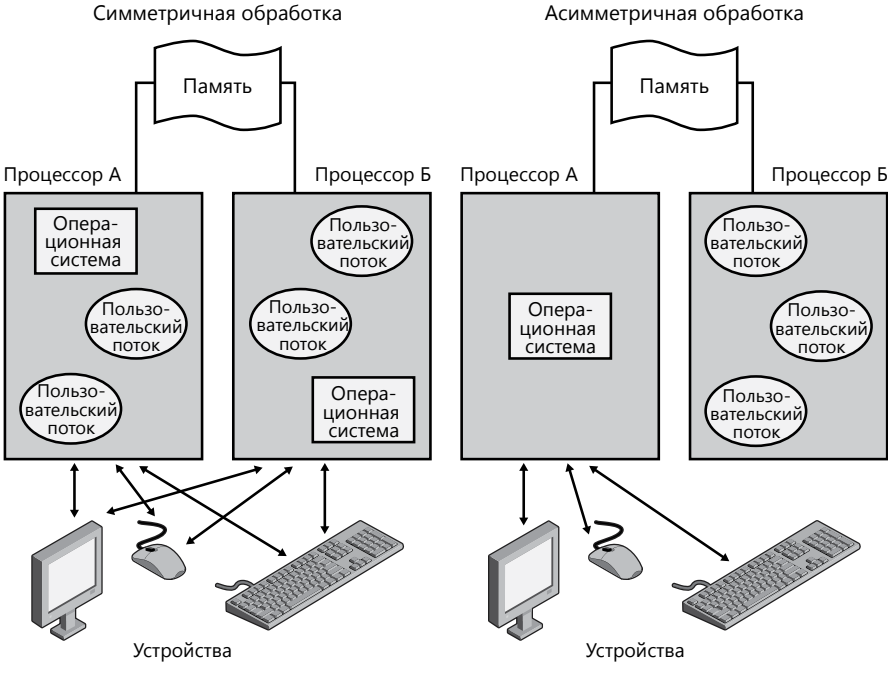

Рис. 2.2. Сравнение симметричной и асимметричной мультипроцессорной обработки

ими память. Она пытается удовлетворить запросы на выделение памяти в рамках узла, но при необходимости выделит память, подключенную и к другим узлам.

И конечно же, Windows также изначально поддерживает многоядерные системы, поскольку у этих систем есть настоящие физические ядра (они находятся на одном кристалле), исходный SMP-код в Windows считает их отдельными процессорами, за исключением некоторых задач учета использования ресурсов и идентификации (например, лицензирования), делающих различия между ядрами на одном и том же процессоре и ядрами на разных сокетах.

Изначально в Windows не предусматривался какой-либо лимит на количество процессоров, если не считать политик лицензирования, устанавливающих различия между разными версиями Windows. Но в целях удобства и эффективности Windows занимается отслеживанием процессоров (их общего числа, простоев, занятости и других таких же подробностей) в битовой маске (маской родственности, affinity mask), в которой количество бит соответствует исходной размерности данных машины (32- или 64-разрядной), что позволяет процессору манипулировать битами непосредственно в регистре. Из-за этого количество процессоров в системе Windows изначально ограничивалось исходной размерностью слова, поскольку маска родственности не могла быть увеличена произвольным образом. Чтобы обеспечить совместимость, а также поддержку более крупных процессорных систем, в Windows реализована конструкция более высокого порядка, называемая группой процессоров. Такая группа является набором процессоров, где все процессоры группы могут быть определены единой битовой маской родственности, и ядро операционной системы, как и приложения, могут

выбрать группу, к которой они обращаются при обновлении маски родственности. Совместимые с данной конструкцией приложения могут запросить количество поддерживаемых групп (ограничиваемое в настоящее время числом 4) а затем подсчитать битовую маску для каждой группы. При этом устаревшие приложения продолжают работать, видя только текущую группу. Информация о том, как именно Windows назначает процессоры для групп (что также относится и к NUMA), дается в главе 5.

Как уже упоминалось, фактическое число поддерживаемых лицензированных процессоров зависит от используемой версии Windows. (См. далее табл. 2.2.) Это количество хранится в файле лицензионной политики системы (\Windows\ServiceProfiles\NetworkService\AppData\Roaming\Microsoft\Software-ProtectionPlatform\tokens.dat) в виде значения политики под названием «Kernel-RegisteredProcessors». Следует иметь в виду, что фальсификация данных является нарушением лицензии на программное обеспечение, и изменение лицензионной политики с целью использования большего количества процессоров влечет за собой более серьезные последствия, чем простое изменение этого значения.

Масштабируемость

Одним из ключевых вопросов мультипроцессорных систем является масштабируемость. Для корректной работы на SMP-системе код операционной системы должен следовать строгим принципам и правилам. Борьба за ресурсы и решение других проблем, влияющих на производительность системы, в мультипроцессорных системах происходит сложнее, чем в однопроцессорных. Это должно быть учтено в конструкциях мультипроцессорных систем. В Windows имеется ряд функций, имеющих решающее значение для достижения ее успешной работы в качестве мультипроцессорной операционной системы:

- Возможность запуска кода операционной системы на любом доступном процессоре и одновременно на нескольких процессорах.
- Несколько потоков выполнения в одном процессе, каждый из которых может одновременно выполняться на разных процессорах.
- Тонкая синхронизация внутри ядра (с помощью спин-блокировок, спин-блокировок с очередью и пуш-блокировок, рассматриваемых в главе 3) наряду с драйверами устройств и служебными процессами, позволяющая большему количеству компонентов запускаться параллельно на нескольких процессорах.
- Программирование таких механизмов, как порты завершения ввода-вывода, способствующих эффективной реализации многопоточных серверных процессов, хорошо масштабируемых на мультипроцессорных системах.

Со временем масштабируемость ядра Windows улучшилась. Например, в Windows Server 2003 были представлены очереди планирования для каждого процессора, позволяющие принимать решения по диспетчеризации потоков параллельно на нескольких процессорах. В Windows 7 и Windows Server 2008 R2 была убрана глобальная блокировка в отношении диспетчеризации баз данных. Такому поэтапному улучшению детализации блокировки подверглись и другие области, например диспетчер памяти. Дополнительную информацию о мультипроцессорной синхронизации можно найти в главе 3.

Различия между клиентскими и серверными версиями

Windows поставляется как в клиентских, так и в серверных версиях. На момент написания данной книги существовало шесть клиентских версий Windows 7: Windows 7 Home Basic, Windows 7 Home Premium, Windows 7 Professional, Windows 7 Ultimate, Windows 7 Enterprise и Windows 7 Starter.

Существует семь различных серверных версий Windows Server 2008 R2: Windows Server 2008 R2 Foundation, Windows Server 2008 R2 Standard, Windows Server 2008 R2 Enterprise, Windows Server 2008 R2 Datacenter, Windows Web Server 2008 R2, Windows HPC Server 2008 R2 и Windows Server 2008 R2 for Itanium-Based Systems (выпуском Windows для процессора Intel Itanium).

Кроме этого существуют клиентские «N»-версии, не включающие в себя Windows Media Player. И наконец, версии Windows Server 2008 R2 Standard, Enterprise и Datacenter также включают выпуски «с Hyper-V», в которых присутствует Hyper-V. (Виртуализация Hyper-V рассматривается в главе 3.)

Все эти версии отличаются друг от друга следующими показателями:

- числом поддерживаемых процессоров (в понятиях сокетов, а не ядер или потоков);
- объемом поддерживаемой физической памяти (фактически, самый большой физический адрес, доступный для оперативной памяти);
- количеством поддерживаемых параллельных сетевых подключений (Например, в клиентской версии к файловым и принтерным службам допускается максимально 10 параллельных подключений.);
- поддержкой Media Center;
- поддержкой Multi-Touch, Aero и Диспетчера рабочего стола (Desktop Compositing);
- поддержкой таких свойств, как BitLocker, VHD Booting, AppLocker, Windows XP Compatibility Mode и более ста других значений настраиваемой политики лицензирования;
- многоуровневыми службами, поставляемыми с версиями Windows Server и не поставляемыми с клиентскими версиями (например, службами каталогов и кластеризации).

Различия в поддержке памяти и процессоров для Windows 7 и Windows Server 2008 R2 показаны в табл. 2.2. Подробная сравнительная таблица различных версий Windows Server 2008 R2 представлена на веб-сайте www.microsoft.com/windowsserver2008/en/us/r2-compare-specs.aspx.

Несмотря на то что операционная система Windows распространяется в виде нескольких клиентских и серверных пакетов поставки, все они используют один и тот же набор основных системных файлов, включая образ ядра, Ntoskrnl.exe (а в PAE-версии Ntkrnlpa.exe), HAL-библиотеки, драйверы устройств и базовые системные утилиты и DLL-библиотеки. Эти файлы идентичны для всех версий Windows 7 и Windows Server 2008 R2.

Откуда, при наличии такого разнообразия версий Windows с одинаковым образом ядра, система знает, какую именно версию загружать? Для этого делается запрос значений реестра ProductType и ProductSuite, находящихся в разделе HKLM\SYSTEM\CurrentControlSet\Control\ProductOptions. Значение ProductType используется для того, чтобы отличить клиентскую систему от серверной (любой разновидности). Эти значения загружаются в реестр на основе

Таблица 2.2. Различия между Windows 7 и Windows Server 2008 R2

	Количество поддерживаемых сокетов (32-разр. версия)	Объем поддерживаемой физической памяти (32-разр. версия), Гбайт	Количество поддерживаемых сокетов (64-разр. версия)	Объем поддерживаемой физической памяти (Itanium-версии), Гбайт	Объем поддерживаемой физической памяти (x64-версии), Гбайт
Windows 7 Starter 1	1	2	Нет	Нет	2
Windows 7 Home Basic	1	4	1	Нет	8
Windows 7 Home Premium	1	4	1	Нет	16
Windows 7 Professional	2	4	2	Нет	192
Windows 7 Enterprise	2	4	2	Нет	192
Windows 7 Ultimate	2	4	2	Нет	192
Windows Server 2008 R2 Foundation	Нет	Нет	1	Нет	8
Windows Web Server 2008 R2	Нет	Нет	4	Нет	32
Windows Server 2008 R2 Standard	Нет	Нет	4	Нет	32
Windows HPC Server 2008 R2	Нет	Нет	4	Нет	128
Windows Server 2008 R2 Enterprise	Нет	Нет	8	Нет	2048
Windows Server 2008 R2 Datacenter	Нет	Нет	64	Нет	2048
Windows Server 2008 R2 for Itanium-Based Systems	Нет	Нет	64	2048	Нет

рассмотренного ранее файла политики лицензирования. Допустимые значения перечислены в табл. 2.3. Это значение может быть запрошено из функции пользовательского режима `GetVersionEx` или из драйвера устройства с помощью вспомогательной функции режима ядра `RtlGetVersion`.

Таблица 2.3. Значения параметра ProductType, имеющегося в реестре

Версия Windows	Значение ProductType
Windows client	WinNT
Windows server (контроллер домена)	LanmanNT
Windows server (только сервер)	ServerNT

Другое значение реестра, ProductPolicy, содержит кэшированную копию данных, находящихся в файле tokens.dat, который устанавливает различия между версиями Windows и допускаемыми в них функциями.

Если пользовательским программам нужно определить, под какой версией Windows они работают, они могут вызвать Windows-функцию `VerifyVersionInfo` (см. документацию по SDK). Драйверы устройств могут вызвать функцию режима ядра `RtlVerifyVersionInfo` (см. документацию по WDK).

Но если основные файлы, по сути, одинаковы для клиентской и серверной версий, чем системы отличаются в работе? Вкратце, серверные системы по умолчанию оптимизированы под системную пропускную способность, позволяющую им выступать в роли высокопроизводительных серверов приложений, а клиентская версия (при наличии серверных возможностей) оптимизирована по времени отклика для интерактивного использования в качестве рабочего стола. Например, на основе типа продукта по-другому принимается ряд решений по распределению ресурсов в процессе загрузки системы. В частности, это касается размеров и количества областей памяти, выделяемых программе для динамически размещаемых структур данных (или пулов), количества внутренних рабочих потоков системы и размера кэш-памяти системных данных. Также серверная и клиентская версии отличаются друг от друга решениями политики времени выполнения, способом учета диспетчером памяти потребностей в системной памяти и в памяти процессов. Отличия между двумя семействами прослеживаются даже в некоторых деталях диспетчеризации потоков, составляющих их поведение по умолчанию (см. главу 5). Все существенные функциональные различия между двумя продуктами выделены в соответствующих главах данной книги. Если не сделано специальных оговорок, то все, описанное в данной книге, относится как к клиентским, так и к серверным версиям.

ЭКСПЕРИМЕНТ: ОПРЕДЕЛЕНИЕ ВОЗМОЖНОСТЕЙ, РАЗРЕШЕННЫХ ПОЛИТИКОЙ ЛИЦЕНЗИРОВАНИЯ

Как уже ранее упоминалось, Windows поддерживает более ста различных функций, которые могут быть разрешены посредством механизма лицензирования программного обеспечения. Соответствующие настройки политики определяют различия не только между клиентской и серверной установками, но также и отличие каждой версии (или идентификатора товарной позиции — stock-keeping unit, SKU) операционной системы, в частности это касается поддержки такого средства, как BitLocker (доступного на серверных версиях Windows, а также на клиентских версиях Windows Ultimate и Enterprise). Для отображения значений политики, определенной для вашей машины, можно

воспользоваться средством SlPolicy, доступным на веб-сайте Winsider Seminars & Solutions (www.winsiderss.com/tools/slpolicy.htm).

Настройки политики организованы по объектам, представляющим владельца модуля, к которому применяется политика. Запустив программу Slpolicy.exe с ключом –f, можно вывести список всех объектов, имеющихся в вашей системе:

```
C:\>SlPolicy.exe -f
SlPolicy v1.05 - Show Software Licensing Policies
Copyright (C) 2008-2011 Winsider Seminars & Solutions Inc.
www.winsiderss.com
Software Licensing Facilities:
Kernel
Licensing and Activation
Core
DWM
SMB
IIS
.
.
.
```

Чтобы вывести значение политики в отношении любого объекта, можно после ключа добавить его имя. Например, чтобы просмотреть ограничения, касающиеся центральных процессоров, доступной памяти нужно указать объект ядра — Kernel. Для машины с запущенной системой Windows 7 Ultimate можно ожидать следующий вывод:

```
C:\>SlPolicy.exe -f Kernel
SlPolicy v1.05 - Show Software Licensing Policies
Copyright (C) 2008-2011 Winsider Seminars & Solutions Inc.
www.winsiderss.com
Kernel
------
Processor Limit: 2
Maximum Memory Allowed (x86): 4096
Maximum Memory Allowed (x64): 196608
Maximum Memory Allowed (IA64): 196608
Maximum Physical Page: 4096
Addition of Physical Memory Allowed: No
Addition of Physical Memory Allowed, if virtualized: Yes
Product Information: 1
Dynamic Partitioning Supported: No
Virtual Dynamic Partitioning Supported: No
Memory Mirroring Supported: No
Native VHD Boot Supported: Yes
Bad Memory List Persistance Supported: No
Number of MUI Languages Allowed: 1000
List of Allowed Languages: EMPTY
List of Disallowed Languages: EMPTY
MUI Language SKU:
Expiration Date: 0
```

Отладочная сборка

Существует версия Windows, называемая отладочной сборкой (checked build), которая доступна только подписчикам MSDN Operating Systems. Это перекомпилированный исходный код Windows с выставленным флажком времени компиляции DBG (включает условный код отладки и трассировки). Кроме того, чтобы было проще понять машинный код, не производится последующая обработка двоичных кодов Windows, оптимизирующая расположение кода для быстрого выполнения. (См. раздел «Debugging Performance-Optimized Code» в файле справки Debugging Tools for Windows.)

В первую очередь отладочная сборка предназначена для помощи разработчикам драйверов устройств, потому что в ней выполняются более строгие проверки на наличие ошибок в функциях режима ядра, вызываемых драйверами устройств или другим системным кодом. Например, если драйвер (или какой-нибудь другой фрагмент кода, выполняемого в режиме ядра) осуществляет неверный вызов системной функции, которая ведет проверку аргументов (например, запрос спинблокировки на неправильном уровне прерывания), система при обнаружении проблемы останавливает выполнение, не допуская разрушения какой-нибудь структуры данных и возможной в последующем аварии системы.

ЭКСПЕРИМЕНТ: ОПРЕДЕЛЕНИЕ ФАКТА ЗАПУСКА ОТЛАДОЧНОЙ СБОРКИ

Чтобы вывести на экран информацию о том, какая именно сборка запущена, отладочная или поступающая в продажу (которая называется свободной), встроенного средства не существует. Но эта информация доступна через свойство «Debug» класса `Win32_OperatingSystem` инструментария управления Windows Management Instrumentation (WMI). Значение этого свойства выводится с помощью следующего примера сценария Microsoft Visual Basic:

```
strComputer = "."
Set objWMIService = GetObject("winmgmts:" _
 & "{impersonationLevel=impersonate}!\\" & strComputer & "\root\cimv2")
Set colOperatingSystems = objWMIService.ExecQuery _
 ("SELECT * FROM Win32_OperatingSystem")
For Each objOperatingSystem in colOperatingSystems
 Wscript.Echo "Заголовок: " & objOperatingSystem.Caption
 Wscript.Echo "Отладка: " & objOperatingSystem.Debug
 Wscript.Echo "Версия: " & objOperatingSystem.Version
Next
```

Чтобы увидеть его в работе, наберите предыдущий код и сохраните его в файле.

```
C:\>cscript osversion.vbs
```

При запуске сценария на экран будет выведена следующая информация:

```
Сервер сценариев Windows (Microsoft R) версия 5.8
© Корпорация Майкрософт (Microsoft Corp.), 1996-2001. Все права защищены.

Заголовок: Microsoft Windows 7 Ultimate
Отладка: Ложь
Версия: 6.1.7600
```

Эта система запущена не из отладочной сборки, поскольку показанный здесь флажок отладки не установлен, то есть имеет значение `False` (Ложь). ■

Основная часть дополнительного кода в исполняемых файлах отладочной версии является результатом использования макроса ASSERT и (или) макроса NT_ASSERT, которые определены в заголовочном файле WDK Wdm.h и описаны в WDK-документации. Макрос проверяет условие (например, приемлемость структуры данных или параметра), и если выражение вычисляется в FALSE, макрос вызывает функцию RtlAssert, работающую в режиме ядра, которая вызывает функцию DbgPrintEx для отправки текста отладочного сообщения в предназначенный для этого сообщения буфер. Если подключен отладчик ядра, это сообщение выводится автоматически вместе с вопросом пользователю о том, что делать в связи с сообщением об отказе (установить контрольную точку, проигнорировать, завершить процесс или завершить поток). Если система не была запущена с отладчиком ядра (с использованием параметра отладки в базе данных конфигурации загрузки — Boot Configuration Database, BCD) и отладчик ядра не подключен, неудачное выполнение теста утверждения — ASSERT приведет к ошибке проверки системы. Перечень проверок ASSERT, проводимых некоторыми подпрограммами поддержки ядра, приводится в разделе «Checked Build ASSERTs» WDK-документации.

Отладочная сборка может также пригодиться системным администраторам своей дополнительной подробной информационной трассировкой, которая может быть включена для некоторых компонентов. (Подробные инструкции даны в статье базы знаний Microsoft «HOWTO: Enable Verbose Debug Tracing in Various Drivers and Subsystems».) Этот информационный вывод отправляется во внутренний буфер отладочных сообщений с использованием ранее упомянутой функции DbgPrintEx. Для просмотра отладочных сообщений можно либо подключить к целевой системе отладчик ядра (что требует загрузки целевой системы в отладочном режиме), воспользовавшись после этого командой !dbgprint при осуществлении отладки локального ядра, либо воспользоваться средством Dbgview.exe из набора Sysinternals (www.microsoft.com/technet/sysinternals).

Чтобы воспользоваться отладочной версией операционной системы необязательно устанавливать всю отладочную сборку. Достаточно в обычную поставляемую установку скопировать отладочную версию образа ядра (Ntoskrnl.exe) и соответствующую HAL-библиотеку (Hal.dll). Преимущество такого подхода заключается в том, что драйверы устройств и другие фрагменты кода ядра получают строгий контроль, присущий отладочной сборке без необходимости запуска работающих медленнее отладочных версий всех компонентов системы. Подробные инструкции приведены в разделе «Installing Just the Checked Operating System and HAL» WDK-документации.

И наконец, отладочная сборка может также пригодиться для тестирования кода, выполняемого в пользовательском режиме, что обусловлено только лишь разницей в синхронизации системы. (Причина в том, что в ядре проводятся дополнительные проверки и компоненты скомпилированы без оптимизации.) Зачастую ошибки многопоточной синхронизации связаны с особыми условиями ее организации. При запуске тестов на системе, где работает отладочная сборка (или, как минимум, отладочная версия ядра и соответствующая HAL-библиотека), сам факт другой синхронизации всей системы может выявить скрытые ошибки синхронизации, которые не проявляют себя на системах их обычных комплектов поставки.

Основные компоненты системы

После знакомства с высокоуровневой архитектурой Windows давайте углубимся в ее внутреннюю структуру и посмотрим, какую роль играет каждый основной компонент операционной системы. На рис. 2.3 показана более подробная

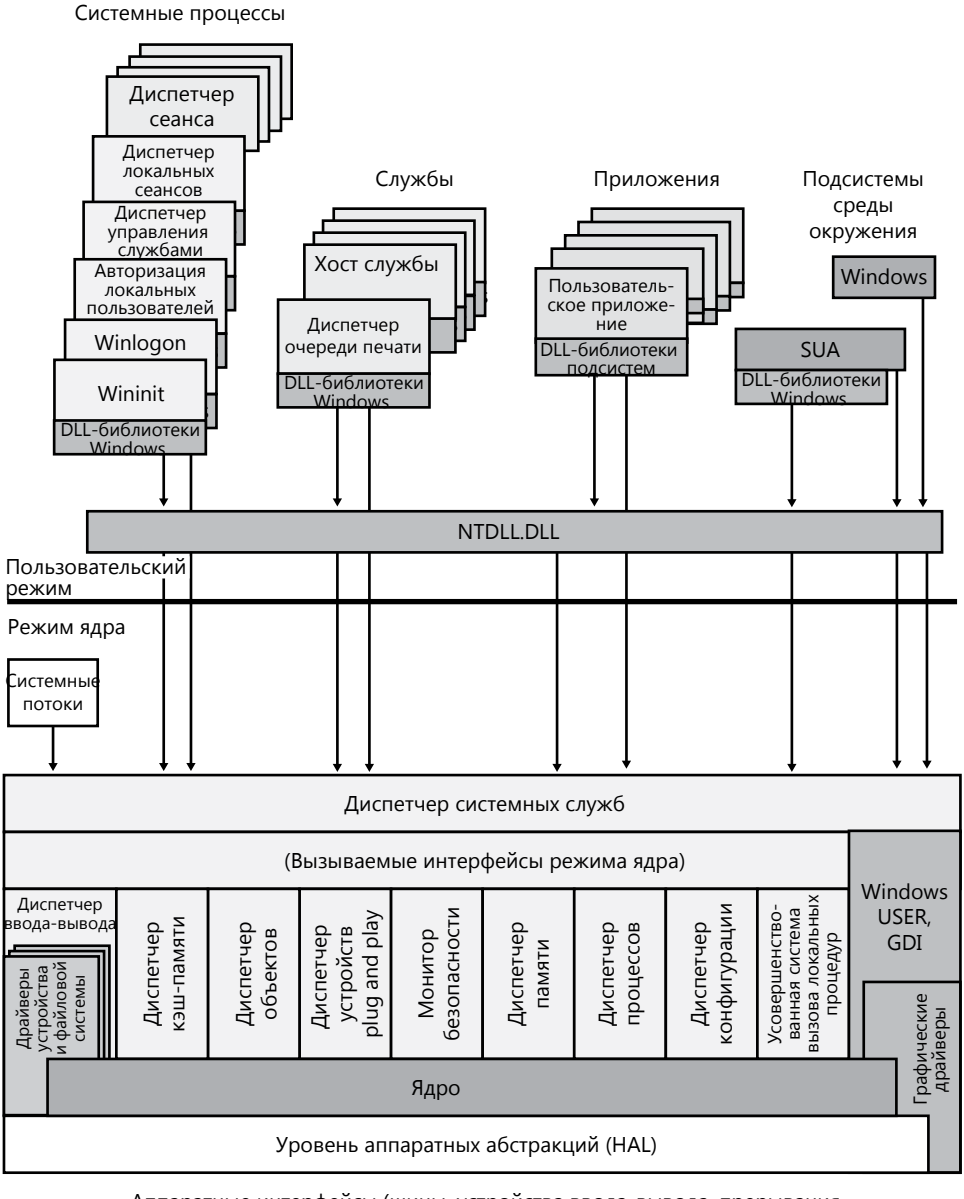

Рис. 2.3. Архитектура Windows

и полная схема архитектуры ядра и компонентов Windows, показанных ранее в данной главе на рис. 2.1. Следует учесть, что на этой схеме показаны еще не все компоненты (в частности, отсутствуют сетевые компоненты, рассматриваемые в главе 7 «Сеть»).

Каждый основной элемент данной схемы подробно рассмотрен в следующих разделах. В главе 3 объясняются первичные механизмы управления, используемые системой (например, диспетчер объектов, прерывания и т. д.). В главе 4 рассматриваются подробности таких механизмов управления, как реестр, служебные процессы и инструментарий управления Windows Management Instrumentation. В остальных главах еще более подробно исследуется внутренняя структура и работа таких ключевых областей, как процессы и потоки, диспетчер памяти, система безопасности, диспетчер ввода-вывода, управление хранением данных, диспетчер кэш-памяти, файловая система Windows (NTFS) и сеть.

Подсистемы среды окружения и DLL-библиотеки подсистем

Роль подсистемы среды окружения заключается в предоставлении прикладным программам того или иного поднабора базовых служб исполняющей системы Windows. Каждая подсистема может предоставить доступ к различным поднаборам служб, присущих Windows. Это означает, что из приложений, построенных на использовании одной подсистемы, могут выполняться некие действия, которые не могут выполняться приложением, построенным на использовании другой подсистемы. Например, приложение Windows не может использовать SUA-функцию `fork`.

Каждый исполняемый образ (.exe) привязан к одной и только к одной подсистеме. При запуске образа код создания процесса исследует код типа подсистемы в заголовке образа, чтобы уведомить соответствующую подсистему о новом процессе. В Microsoft Visual C++ этот код типа указывается с помощью спецификатора `/SUBSYSTEM` команды `link`.

Как уже ранее упоминалось, пользовательские приложения не вызывают напрямую системные службы Windows. Вместо этого ими используется одна или несколько DLL-библиотек подсистемы. Эти библиотеки экспортируют документированный интерфейс, который может быть использован программами, связанными с данной подсистемой. Например, API-функции Windows реализованы в DLL-библиотеках подсистемы Windows, таких, как Kernel32.dll, Advapi32.dll, User32.dll и Gdi32.dll. А API-функции SUA реализованы в DLL-библиотеке подсистемы SUA (Psxdll.dll).

ЭКСПЕРИМЕНТ: ПРОСМОТР ТИПА ПОДСИСТЕМЫ ОБРАЗА

Тип подсистемы образа можно просмотреть с помощью средства Dependency Walker (Depends.exe) (доступно на сайте www.dependencywalker.com). Например, обратите внимание на типы образа для двух различных Windows-образов, Notepad.exe (простой текстовый редактор) и Cmd.exe (командная строка Windows):

Здесь показано, что Notepad (Блокнот) является GUI-программой, а Cmd является консольной (console), или текстовой программой. И хотя это означает, что существует две разные подсистемы для GUI-программ и текстовых программ, на самом деле есть только одна подсистема Windows, и GUI-программы могут иметь консоли, а консольные программы могут отображать GUI-элементы. ■

Когда приложение вызывает функцию, находящуюся в DLL-библиотеке подсистемы, может реализоваться одно из трех обстоятельств:

- Функция полностью реализована в режиме пользователя в DLL-библиотеке подсистемы. Иными словами, процессу подсистемы среды никакое сообщение не отправляется и никакие службы исполняющей системы Windows не вызываются. Функция выполняется в пользовательском режиме, а результаты возвращаются вызвавшему ее коду. Примерами таких функций могут послужить:
  - `GetCurrentProcess` (всегда возвращающая `-1`, значение, определяемое во всех, связанных с процессами функциях для ссылки на текущий процесс);
  - `GetCurrentProcessId` (ID процесса для запущенного процесса не изменяется, поэтому этот ID извлекается из ячейки кэш-памяти, исключая тем самым необходимость обращения к ядру).
- Функция требует один или несколько вызовов исполняющей системы Windows. Например, Windows-функции `ReadFile` и `WriteFile` включают соответственно вызовы базовых внутренних (и недокументированных) системных служб ввода-вывода Windows `NtReadFile` и `NtWriteFile`.
- Функция требует проведения в процессе подсистемы среды окружения некоторой работы. (Процессы подсистем среды окружения, запущенные в пользовательском режиме, отвечают за обслуживание состояния клиентских приложений, находящихся под их управлением.) В этом случае к подсистеме среды окружения делается клиент-серверный запрос посредством отправки подсистеме

сообщения для выполнения той или иной операции. Затем DLL-библиотека подсистемы, прежде чем вернуть управление вызывающему коду, ждет ответа.

Некоторые функции могут представлять собой комбинацию из только что перечисленных второго и третьего вариантов. В качестве примеров можно привести Windows-функции `CreateProcess` и `CreateThread`.

Запуск подсистем

Подсистемы запускаются процессом Диспетчера сеанса — Session Manager (Smss.exe). Информация, касающаяся запуска подсистем, хранится в разделе реестра HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\SubSystems. Значения, хранящиеся в этом разделе, показаны на рис. 2.4.

Рис. 2.4. Редактор реестра показывает информацию, касающуюся запуска подсистем

В значении `Required` перечисляются подсистемы, загружаемые при загрузке системы. Значение содержит две строки: `Windows` и `Debug`. Значение `Windows` содержит спецификацию файла подсистемы Windows, Csrss.exe (от Client/Server Run-Time Subsystem, клиент-серверная подсистема времени выполнения). Значение `Debug` оставлено пустым (поскольку оно используется для внутреннего тестирования) и поэтому не вызывает никаких действий. Значение `Optional` показывает, что по запросу будет запущена подсистема SUA. Значение реестра `Kmode` содержит имя файла той части подсистемы Windows, которая работает в режиме ядра, Win32k.sys (об этом мы поговорим чуть позже).

Давайте более пристально присмотримся к каждой из подсистем среды окружения.

Подсистема Windows

Хотя Windows была разработана для поддержки нескольких независимых подсистем среды окружения, с практической точки зрения наличие в каждой подсистеме всего кода для организации многооконного интерфейса и отображения ввода-вывода может привести к большой степени дублированности системных функций, что, в конечном счете, негативно скажется как на размере, так и на

производительности системы. Поскольку Windows была первичной подсистемой, разработчики Windows решили поместить эти базовые функции в ней и заставить другие подсистемы для отображения ввода-вывода вызывать подсистему Windows. Таким образом, подсистема SUA для отображения ввода-вывода вызывает службы в подсистеме Windows.

В результате такого конструкторского решения подсистема Windows является необходимым компонентом для любой Windows-системы, даже на серверных системах без зарегистрированных интерактивных пользователей. По этой причине процесс помечен как необходимый (стало быть, если по каким-то причинам происходит выход из этого процесса, система дает сбой).

Подсистема Windows состоит из следующих основных компонентов:

- Для каждого сеанса экземпляр процесса подсистемы среды (Csrss.exe) загружает три DLL-библиотеки (Basesrv.dll, Winsrv.dll и Csrsrv.dll), содержащие поддержку:
  - создания и удаления процессов и потоков;
  - частей, поддерживающих процессы 16-разрядной виртуальной DOS-машины (VDM) (только для 32-разрядной версии Windows);
  - Side-by-Side (SxS) сборок (Fusion) и манифестов;
  - других разнообразных функций, например `GetTempFile`, `DefineDosDevice`, `ExitWindowsEx` и нескольких естественных функций поддержки языка.
- Драйвер устройства режима ядра (Win32k.sys), включающий в себя:
  - диспетчер окон, который управляет выводом окон, осуществляет экранный вывод, получает ввод с клавиатуры, мыши и других устройств и передает приложениям пользовательские сообщения;
  - интерфейс графических устройств — Graphics Device Interface (GDI), представляющий собой библиотеку функций для устройств графического вывода. Он включает функции для рисования прямых линий, текста и фигур и для манипулирования графическими объектами;
  - оболочки для поддержки набора DirectX, реализуемого в другом драйвере ядра (Dxgkrnl.sys).
- Хост-процесс консоли — console host process (Conhost.exe), — предоставляющий поддержку для консольных (символьных) приложений.
- DLL-библиотеки подсистем (например, Kernel32.dll, Advapi32.dll, User32.dll и Gdi32.dll), которые превращают документированные функции Windows API в соответствующие и большей частью недокументированные вызовы системных служб режима ядра в Ntoskrnl.exe и Win32k.sys.
- Драйверы графических устройств для аппаратно-зависимых драйверов графических дисплеев, драйверов принтеров и драйверов видеомини-портов.

Для создания на дисплее таких элементов управления пользовательского интерфейса, как окна и кнопки, приложения вызывают стандартные USER-функции. Диспетчер окон передает соответствующие требования интерфейсу графических устройств GDI, а тот передает их драйверам графических устройств, где они имеют формат, соответствующий устройству отображения. Для завершения поддержки видеодисплея драйвер дисплея работает в паре с драйвером видеомини-порта.

ПРИМЕЧАНИЕ

В той части оптимизации работ, которая в Windows-архитектуре называется MinWin, DLL-библиотеки подсистем в настоящее время, как правило, состоят из специфических библиотек, реализующих API-наборы, которые затем компонуются вместе в DLL-библиотеку подсистемы и разрешаются с помощью специальной схемы перенаправления. Более подробная информация об этой оптимизации дана в главе 3, в разделе «Загрузчик образов».

GDI предоставляет набор стандартных двумерных функций, позволяющих приложениям обмениваться данными с графическими устройствами, ничего не зная об этих устройствах. GDI-функции являются посредниками между приложениями и такими графическими устройствами, как драйверы дисплеев и принтеров. GDI-интерфейс переводит запросы приложения на графический вывод и отправляет запросы драйверам графического дисплея. Он также предоставляет стандартный интерфейс для приложений для использования различных устройств графического вывода. Этот интерфейс позволяет коду приложения быть независимым от аппаратных устройств и их драйверов. GDI приспосабливает свои сообщения под возможности устройства, зачастую разделяя запрос на управляемые части. Например, некоторые устройства могут понимать направления рисования эллипса, в то время как другие требуют от GDI интерпретировать команду в виде серии пикселов, размещаемых по соответствующим координатам. (Дополнительная информация, касающаяся графики и архитектуры видеодрайверов, дана в Windows Driver Kit, в разделе «Design Guide» главы «Display (Adapters and Monitors)».)

Поскольку основная часть подсистемы, в частности функции дисплейного ввода-вывода, работает в режиме ядра, сообщения процессу подсистемы Windows отправляются всего лишь несколькими Windows-функциями: создания и завершения процесса и потока, отображения буквы сетевого диска и создания временных файлов. В общем, работающее Windows-приложение не будет вызывать многочисленных переключений к процессу подсистемы Windows (или вообще не будет вызывать таких переключений).

ХОСТ ОКНА КОНСОЛИ

В исходной конструкции подсистемы Windows процесс подсистемы (Csrss.exe) отвечал за управление из окон консоли и каждого консольного приложения (например, Cmd.exe, окна командной строки), осуществлявшего обмен данными с Csrss. Теперь Windows использует для каждого окна консоли, имеющего в системе отдельный процесс — хост окна консоли, — console window host (Conhost.exe). Одно и то же окно консоли может совместно использоваться несколькими консольными приложениями, как при запуске одной командной строки из другой командной строки. По умолчанию вторая командная строка совместно использует окно консоли первой командной строки.

Всякий раз, когда консольное приложение регистрируется с помощью экземпляра подсистемы Csrss, запущенной в текущем сеансе, Csrss создает новый экземпляр Conhost, используя вместо системного маркера Csrss маркер доступа клиентского процесса. Затем подсистема отображает на хосты окна консоли используемый ею общий раздел памяти, чтобы позволить всем хостам Conhost совместно использовать часть их памяти с Csrss

для эффективной обработки буфера (их потоки уже не существуют внутри Csrss), и создает поименованный порт асинхронного вызова локальной процедуры (Asynchronous Local Procedure Call, ALPC) в каталоге объектов управления \RPC (см. главу 3.) Имя порту дается в формате *console-PID-lpc-handle*, где PID является идентификатором Conhost-процесса. Затем происходит регистрация этого PID со структурой процесса ядра, связанной с пользовательским приложением, которое затем может запрашивать эту информацию для открытия только что созданного ALPC-порта. Этот процесс также задает соответствие объекта общего раздела памяти между приложением командной строки и его Conhost, чтобы они могли обмениваться данными. И наконец, в каталоге BaseNamedObjects сеанса 0 создается событие ожидания (которое называется ConsoleEvent-PID), чтобы приложение командной строки и Conhost смогли уведомить друг друга о новых данных буфера. На следующем рисунке показывается процесс Conhost с дескрипторами, открытыми для его ALPC-порта и события.

Поскольку Conhost работает с полномочиями пользователя (под которыми также подразумевается уровень прав пользователя), так же как в процессе, связанном с самим приложением консоли, процессы консоли защищаются механизмом безопасности, который называется «изоляцией привилегий пользовательского интерфейса» (User Interface Privilege Isolation). (UIPI-механизм описан в главе 6 «Безопасность».) Кроме этого консольные приложения, ограниченные возможностями центрального процессора, могут быть идентифицированы тем хост-процессом консоли, который их поддерживает (и который пользователь при необходимости может завершить). Поскольку теперь Conhost-процессы запускаются за пределами специального анклава подсистемы Csrss, возникает побочный эффект, благодаря которому консольные приложения (чьими окнами фактически владеет Conhost) могут быть полностью настроены на тему, загружать DLL-библиотеки сторонних производителей и запускаться с полноценными оконными возможностями.

Подсистема для приложений на Unix-основе

Подсистема для приложений на основе Unix — Subsystem for UNIX-based Applications (SUA) — позволяет компилировать и запускать пользовательские

приложения на UNIX-основе на компьютере под управлением серверной операционной системы Windows или клиентской операционной системы Windows версий Enterprise или Ultimate. SUA предоставляет около 2000 UNIX-функций и 300 UNIX-подобных средств и утилит. (Дополнительную информацию, касающуюся SUA, можно найти на сайте http://technet.microsoft.com/en-us/library/cc771470.aspx.) Дополнительная информация о том, как Windows обрабатывает запущенные SUA-приложения, дана в главе 5, в разделе «Порядок работы функции CreateProcess».

ИСХОДНАЯ ПОДСИСТЕМА POSIX

POSIX — широко известный акроним, означающий «a portable operating system interface based on UNIX» (интерфейс переносимой операционной системы на основе UNIX), ссылающийся на коллекцию международных стандартов для интерфейсов операционных систем UNIX-стиля. Стандарты POSIX подстегивают поставщиков, реализующих интерфейсы UNIX-стиля, сделать их совместимыми, чтобы программисты могли легко перемещать свои приложения из одной системы в другую.

Изначально в Windows был реализован только один из многих POSIX-стандартов, POSIX.1, официально известный как стандарт ISO/IEC 9945-1:1990 или IEEE POSIX standard 1003.1-1990. Этот стандарт был включен главным образом для того, чтобы соответствовать набору требований о государственных закупках США во второй половине 1980-х годов. От POSIX.1 требовалось соответствовать федеральному стандарту обработки информации — Federal Information Processing Standard (FIPS) 151-2, разработанному национальным институтом стандартов и технологии — National Institute of Standards and Technology. Windows NT 3.5, 3.51 и 4 были официально протестированы и сертифицированы как соответствующие стандарту FIPS 151-2.

Поскольку соответствие POSIX.1 было для Windows обязательной задачей, операционная система была разработана с гарантированным присутствием требуемой базовой системной поддержки, позволяющей реализовать подсистему POSIX.1 (это касается функции `fork`, реализованной в исполняющей системе Windows, и поддержки жестких ссылок на файлы в файловой системе Windows).

Ntdll.dll

Ntdll.dll является специальной библиотекой системной поддержки, предназначенной, главным образом, для использования DLL-библиотек подсистем. В ней содержатся функции двух типов:

- функции-заглушки, обеспечивающие переходы от диспетчера системных служб к системным службам исполняющей системы Windows;
- вспомогательные внутренние функции, используемые подсистемами, DLL-библиотеками подсистем и другими исходными образами.

Первая группа функций предоставляет интерфейс к службам исполняющей системы Windows, которые могут быть вызваны из пользовательского режима. К этой группе относятся более чем 400 функций, среди которых `NtCreateFile`, `NtSetEvent` и т. д. Как уже отмечалось, основная часть возможностей, присущих

данным функциям, доступна через Windows API. Но некоторые возможности недоступны и предназначены для использования только внутри операционной системы.

Для каждой из этих функций в Ntdll содержится точка входа с именем, совпадающим с именем функции. Код внутри функции содержит зависящую от конкретной архитектуры инструкцию, осуществляющую переход в режим ядра для вызова диспетчера системных служб (более подробно этот вопрос рассмотрен в главе 3), который после проверки ряда параметров вызывает настоящую системную службу режима ядра, реальный код которой содержится в файле Ntoskrnl.exe.

Ntdll также содержит множество вспомогательных функций, таких как загрузчики образов (префикс `Ldr`), диспетчер динамической области памяти и функции обмена данными между процессами подсистемы Windows (префикс `Csr`). Ntdll включает также общие подпрограммы библиотеки времени выполнения (префикс `Rtl`), поддержку отладки в пользовательском режиме (префикс `DbgUi`) и отслеживания событий для Windows (Event Tracing for Windows) (префикс `Etw`), а также диспетчер асинхронных вызовов процедур и исключений пользовательского режима (APC). (APC-вызовы и исключения рассматриваются в главе 3.) И наконец, в Ntdll находится небольшой поднабор подпрограмм времени выполнения языка C (CRT), ограниченный подпрограммами, являющимися частью строковых и стандартных библиотек (в качестве примера можно привести подпрограммы `memcpy`, `strcpy`, `itoa` и т. д.).

Исполняющая система

Исполняющая система Windows находится на верхнем уровне файла Ntoskrnl.exe. (Ядро составляет его нижний уровень.) Она включает функции следующих типов:

- Функции, экспортируемые и вызываемые из пользовательского режима. Эти функции называются системными службами и экспортируются посредством Ntdll. Большинство служб доступно через Windows API или через API-интерфейсы других подсистем среды окружения. Но часть служб не доступна ни через какие документированные функции подсистем. (В качестве примера можно привести ALPC и различные функции запросов, например `NtQueryInformationProcess`, специализированные функции, такие как `NtCreatePagingFile` и т. д.).
- Функции драйверов устройств, вызываемые с помощью функции `DeviceIoControl`, которая предоставляет общий интерфейс из пользовательского режима к режиму ядра для вызова тех функций в драйверах устройств, которые не связаны с чтением или записью.
- Функции, которые могут быть вызваны только из режима ядра, экспортируемые из WDK и документированные в этом инструментальном наборе.
- Функции, экспортируемые и вызываемые из режима ядра, но недокументированные в WDK (например, функции, вызываемые загрузочным видеодрайвером, чьи имена начинаются с префикса `Inbv`).
- Функции, определенные в качестве глобальных символов, но при этом не подлежащие экспорту. К их числу относятся внутренние вспомогательные функции, вызываемые внутри Ntoskrnl, например такие функции, чьи имена начинаются

с префикса `Iop` (внутренние функции поддержки диспетчера ввода-вывода) или с префикса `Mi` (внутренние функции поддержки диспетчера памяти).

- Функции, являющиеся внутренними по отношению к модулю, но не определенные в качестве глобальных символов.

Исполняющая система содержит следующие основные компоненты, каждый из которых подробно рассмотрен в следующих главах данной книги:

- *Диспетчер конфигурации* (см. главу 4), который отвечает за реализацию и управление системным реестром.
- *Диспетчер процессов* (см. главу 5) создает процессы и потоки и завершает их работу. Исходная поддержка процессов и потоков реализована в ядре Windows; исполняющая система добавляет к этим низкоуровневым объектам дополнительную семантику и функции.
- *Монитор безопасности* (security reference monitor, SRM) (см. главу 6) обеспечивает соблюдение политики безопасности на локальном компьютере. Он охраняет ресурсы операционной системы, выполняя защиту и проверку объектов времени выполнения.
- *Диспетчер ввода-вывода* реализует аппаратно-независимый ввод-вывод и отвечает за направление на соответствующие драйверы устройств для дальнейшей обработки.
- *Диспетчер устройств plug and play* (PnP) определяет, какие драйверы требуются для поддержки конкретного устройства, и загружает эти драйверы. В процессе переписи устройств для каждого из них извлекаются требования к аппаратным ресурсам. На основе требований к ресурсам каждого устройства диспетчер PnP назначает соответствующие аппаратные ресурсы, такие как порты ввода-вывода, линии запроса на прерывание (IRQ), DMA-каналы и адреса памяти. Он также отвечает за отправку соответствующих уведомлений о событиях при изменениях в устройствах (добавлении или удалении устройства) в системе.
- *Диспетчер электропитания* согласовывает события электропитания и генерирует уведомления ввода-вывода, касающиеся управления электропитания, посылаемые драйверам устройств. При простое системы диспетчер электропитания может быть настроен на снижение расхода электроэнергии путем перевода центрального процессора в спящий режим. Изменения в энергопотреблении отдельными устройствами управляются драйверами устройств, но согласовываются диспетчером электропитания.
- *Подпрограммы инструментария управления Windows для модели драйверов — Windows Driver Model Windows Management Instrumentation routines* (см. главу 4) позволяют драйверам устройств публиковать информацию о производительности и конфигурации, а также получать команды от WMI-службы пользовательского режима. Потребители WMI-информации могут быть на локальной машине или на удаленной, имеющей сетевой доступ
- *Диспетчер кэша* повышает производительность файлового ввода-вывода, размещая данные, полученные с диска, к которым недавно было обращение, в основной памяти для ускорения доступа к этим данным (и задерживая записи на диск путем кратковременного хранения обновлений в памяти перед их отправкой на диск). Вы увидите, что это делается путем поддержки диспетчером памяти отображаемых файлов.

- *Диспетчер памяти* реализует виртуальную память, схему управления памятью, предоставляющую каждому процессу большое, закрытое адресное пространство, которое может превышать по объему доступную физическую память. Диспетчер памяти также предоставляет исходную поддержку диспетчеру кэша.
- *Логическая предвыборка* и *Superfetch* ускоряют работу системы и запуск процесса путем оптимизации загрузки данных, на которые есть ссылка во время запуска системы или процесса.

Помимо этого исполняющая система содержит четыре основные группы вспомогательных функций, используемых только что перечисленными исполняющими компонентами. Около трети этих вспомогательных функций документированы в WDK, поскольку они также используются драйверами устройств. Есть четыре категории вспомогательных функций:

- *Диспетчер объектов*, который работает с исполняющими объектами Windows и абстрактными типами данных, используемыми для представления таких ресурсов операционной системы, как процессы, потоки и различные объекты синхронизации, создает эти объекты, управляет ими и удаляет их. Диспетчер объектов рассматривается в главе 3.
- *Усовершенствованная система вызова локальных процедур* (ALPC, рассматриваемая в главе 3) передает сообщения между клиентским и серверным процессами на одном и том же компьютере. Кроме всего прочего, ALPC используется в качестве локального средства сообщения для вызова удаленных процедур (remote procedure call, RPC), стандартного средства связи для клиентских и серверных процессов по сети.
- Широкий набор *библиотечных функций времени выполнения* общего назначения, среди которых функции обработки строк, арифметических операций, преобразований типов данных и работы со структурой безопасности.
- Вспомогательные подпрограммы исполняющей системы, к которым относятся подпрограммы распределения системной памяти (выгружаемого и невыгружаемого пула), доступа к памяти с взаимной блокировкой, а также три специальных типа объектов синхронизации: ресурсы (resources), быстрые мьютексы (fast mutexes) и пуш-блокировки (pushlocks).

Исполняющая система также содержит различные инфраструктурные подпрограммы (некоторые из них будут кратко упомянуты в данной книге):

- Библиотеку *отладчика ядра*, позволяющую вести отладку ядра из поддерживающего отладку KD, переносимого протокола, поддерживаемого через различные средства транспортировки данных (такие как USB и IEEE 1394) и реализованного в утилитах WinDbg и Kd.exe.
- *Среду отладки в режиме пользователя*, которая отвечает за отправку событий API-интерфейсу отладки в режиме пользователя и позволяющую расставлять контрольные точки и проводить пошаговое выполнение кода для его отработки, а также для изменения контекста запущенных потоков.
- *Диспетчер транзакций ядра*, предоставляющий диспетчерам ресурсов простой, двухфазный механизм выделения ресурсов, такой как транзакционный реестр —(TxR) и транзакционная NTFS (TxF).

- *Библиотеку гипервизора*, часть стека Hyper-V в Windows Server 2008, предоставляющая на уровне ядра поддержку среды виртуальных машин и оптимизирующая определенные части кода, когда системе известно, что она работает в клиентском разделе (виртуальной среде).
- *Диспетчер исправлений*, предоставляющий пути обхода для нестандартных или несовместимых аппаратных устройств.
- *Средство проверки драйверов* — Driver Verifier, реализующее дополнительные проверки целостности кода и драйверов режима ядра.
- *Средство трассировки событий* — Event Tracing for Windows, предоставляющее вспомогательные подпрограммы для трассировки общесистемных событий для компонентов режима ядра и пользовательского режима.
- *Инфраструктуру диагностики Windows*, допускающую интеллектуальное отслеживание деятельности системы на основе сценариев диагностики.
- *Подпрограммы поддержки архитектуры аппаратных ошибок Windows*, предоставляющие общую среду для отчетов об ошибках оборудования.
- *Библиотеку времени выполнения файловой системы*, предоставляющую подпрограммы общей поддержки для драйверов файловой системы.

Ядро

Ядро состоит из набора функций, находящихся в файле Ntoskrnl.exe. Этот набор предоставляет основные механизмы (службы диспетчеризации потоков и синхронизации), используемые компонентами исполняющей системы, а также поддержкой архитектурно-зависимого оборудования низкого уровня (например, диспетчеризацией прерываний и исключений), которое имеет отличия в архитектуре каждого процессора. Код ядра написан главным образом на C, с ассемблерным кодом, предназначенным для задач, требующих доступа к специальным инструкциям и регистрам процессора, доступ к которым из кода на языке C затруднен.

Подобно различным вспомогательным функциям, упомянутым в предыдущем разделе, ряд функций ядра документированы в WDK (и их описание может быть найдено при поиске функций, начинающихся с префикса `Ke`), поскольку они нужны для реализации драйверов устройств.

Объекты ядра

Ядро предоставляет низкоуровневую базу из четко определенных, предсказуемых примитивов и механизмов операционной системы, позволяющую высокоуровневым компонентам исполняющей системы выполнять свои функции. Само ядро отделено от остальной исполняющей системы путем реализации механизмов операционной системы и уклонения от выработки политики. Оно оставляет почти все политические решения, за исключением планирования и диспетчеризации потоков, реализуемых ядром, за исполняющей системой.

За пределами ядра исполняющая система представляет потоки и другие ресурсы совместного использования в виде объектов. Эти объекты требуют некоторых издержек, например на дескрипторы для управления ими, на проверки безопасности для их защиты и на ресурсные квоты, выделяемые при их создании.

В ядре, реализующем набор менее сложных объектов, называемых «объектами ядра», подобные издержки исключены, что помогает ядру управлять основной обработкой и поддерживать создание исполняющих объектов. Большинство объектов уровня исполнения инкапсулируют один или несколько объектов ядра, принимая их, определенные в ядре свойства.

Один набор объектов ядра, называемых «управляющими объектами», определяет семантику управления различными функциями операционной системы. В этот набор включены объекты асинхронного вызова процедур — APC, отложенного вызова процедур — deferred procedure call (DPC), и несколько объектов, используемых диспетчером ввода-вывода, например объект прерывания.

Еще один набор объектов ядра, известных как «объекты-диспетчеры», включает возможности синхронизации, изменяющие или влияющие на планирование потоков. Объекты-диспетчеры включают поток ядра, мьютекс (называемый среди специалистов «мутантом»), событие, пару событий ядра, семафор, таймер и таймер ожидания. Исполняющая система использует функции ядра для создания экземпляров объектов ядра, работы с ними и создания более сложных объектов, предоставляемых в пользовательском режиме. Более подробно объекты рассматриваются в главе 3, а процессы и потоки рассматриваются в главе 5.

Область ядра, относящаяся к управлению процессором, и блок управления (KPCR и KPRCB)

Для хранения специфических для процессора данных ядром используется структура данных, называемая областью, относящейся к управлению процессором, или KPCR (Kernel Processor Control Region). KPCR содержит основную информацию, такую как процессорная таблица диспетчеризации прерываний (interrupt dispatch table, IDT), сегмент состояния задачи (task-state segment, TSS) и таблица глобальных дескрипторов (global descriptor table, GDT). Она также включает состояние контроллера прерываний, которое используется вместе с другими модулями, такими как ACPI-драйвер и HAL. Для обеспечения простого доступа к KPCR ядро хранит указатель на эту область в регистре `fs` на 32-разрядной системе Windows и в регистре `gs` на Windows-системе x64. На системах IA64 KPCR всегда находится по адресу `0xe0000000ffff0000`.

KPCR также содержит вложенную структуру данных, которая называется блоком управления процессором (kernel processor control block, KPRCB). В отличие от области KPCR, которая документирована для драйверов сторонних производителей и для других внутренних компонентов ядра Windows, KPRCB является закрытой структурой, используемой только кодом ядра, который находится в файле Ntoskrnl.exe. В этом блоке содержится:

- информация о планировании (такая как текущий, следующий и приостановленный потоки, предназначенные для выполнения на процессоре);
- предназначенная для процессора база данных диспетчера (включающая готовые очереди для каждого приоритетного уровня);
- DPC-очередь;
- информация о производителе центрального процессора и идентификационная информация (модель, степпинг, скорость, биты особенностей);

- информация о топологии центрального процессора и о технологии доступа к неоднородной памяти — NUMA (информация об узле, о количестве логических процессоров в каждом ядре и т. д.);
- информация о размерах кэш-памяти;
- информация об учете времени (такая как время DPC и обработки прерывания) и многое другое.

В KPRCB также содержится вся статистика процессора, такая как статистика ввода-вывода, статистика диспетчера кэша, статистика DPC и статистика диспетчера памяти. И наконец, KPRCB иногда используется для хранения структур выравнивания границ кэша для каждого процессора, необходимых для оптимизации доступа к памяти, особенно на NUMA-системах. Например, система невыгружаемого и выгружаемого пула со стороны выглядит как списки, хранящиеся в KPRCB.

ЭКСПЕРИМЕНТ: ПРОСМОТР KPCR И KPRCB

Содержимое KPCR и KPRCB можно просмотреть, используя команды отладки ядра `!pcr` и `!prcb`. Без пометок отладчик по умолчанию покажет информацию для центрального процессора 0; но вы можете указать центральный процессор, добавив после команды его номер (например, `!pcr 2`). В следующем примере показано, как выглядит вывод команд `!pcr` и `!prcb`. Если в системе имелись задержанные DPC-вызовы, эта информация также будет отображена на экране.

```
lkd> !pcr
KPCR for Processor 0 at 81d09800:
    Major 1 Minor 1
    NtTib.ExceptionList: 9b31ca3c
        NtTib.StackBase: 00000000
       NtTib.StackLimit: 00000000
     NtTib.SubSystemTib: 80150000
          NtTib.Version: 1c47209e
      NtTib.UserPointer: 00000001
          NtTib.SelfTib: 7ffde000
                SelfPcr: 81d09800
                   Prcb: 81d09920
                   Irql: 00000002
                    IRR: 00000000
                    IDR: ffffffff
          InterruptMode: 00000000
                    IDT: 82fb8400
                    GDT: 82fb8000
                    TSS: 80150000
          CurrentThread: 86d317e8
             NextThread: 00000000
             IdleThread: 81d0d640
              DpcQueue:
lkd> !prcb
PRCB for Processor 0 at 81d09920:
Current IRQL -- 0
```

```
Threads--  Current 86d317e8 Next 00000000 Idle 81d0d640
Number 0 SetMember 1
Interrupt Count -- 294ccce0
Times -- Dpc    0002a87f Interrupt 00010b87
         Kernel 026270a1 User      00140e5e
```

Для непосредственного вывода дампа структур данных _KPCR и _KPRCB можно воспользоваться командой dt, поскольку обе команды отладки дают вам адреса структур (которые для наглядности выделены в предыдущем выводе жирным шрифтом). Например, если нужно определить скорость процессора, можно с помощью следующей команды посмотреть на поле MHz:

```
lkd> dt nt!_KPRCB 81d09920 MHz
   +0x3c4 MHz : 0xbb4
lkd> ? bb4
Evaluate expression: 2996 = 00000bb4
```

На данной машине процессор был запущен на частоте около 3 ГГц.

Поддержка оборудования

Другой основной задачей ядра является абстрагирование или изоляция исполняющей системы и драйверов устройств от различий аппаратных архитектур, поддерживаемых Windows. Эта задача включает обработку вариантов в таких функциях, как обработка прерываний, диспетчеризация исключений и мультипроцессорная синхронизация.

Даже для этих, зависящих от аппаратных особенностей, функций конструкция ядра пытается максимизировать объем общего кода. Ядро поддерживает набор переносимых и семантически идентичных для разных архитектур интерфейсов. Основная часть кода, реализующего эти переносимые интерфейсы, также идентична для разных архитектур.

Часть этих интерфейсов для разных архитектур реализована по-разному или частично реализована с использованием кода, специфичного для конкретной архитектуры. Такие, независимые от архитектуры, интерфейсы могут быть вызваны на любой машине, и семантические характеристики интерфейса будут одинаковыми, независимо от различий в коде, связанных с архитектурой. Некоторые интерфейсы ядра (см. подпрограммы спин-блокировки в главе 3) в действительности реализованы на уровне аппаратных абстракций — HAL, поскольку их реализация может варьироваться для систем внутри одного и того же архитектурного семейства.

Ядро также содержит небольшой объем кода с интерфейсами для x86-систем, необходимыми для поддержки старых программ MS-DOS. Эти x86-интерфейсы не обладают переносимостью, в том смысле, что они не могут вызываться на машинах, основанных на каких-нибудь других архитектурах; их там просто не будет. Этот код, предназначенный для x86-систем, к примеру, поддерживает вызовы для обращения с таблицами глобальных дескрипторов (GDT) и таблицами локальных дескрипторов (LDT), являющихся аппаратными особенностями x86.

Другие примеры кода ядра, зависящего от конкретной архитектуры, включают интерфейсы для обеспечения поддержки буфера трансляции и кэш-памяти

центрального процессора. Из-за способов реализации кэш-памяти эта поддержка требует применения разного кода для разных архитектур.

Другим примером может послужить переключение контекста. Хотя на высоком уровне для выбора потока и переключения контекста используется один и тот же алгоритм (контекст предыдущего потока сохраняется, загружается контекст нового потока и запускается новый поток), в реализациях на разных процессорах существуют архитектурные различия. Поскольку контекст описывается состоянием процессора (его регистров и т. д.), сохраняемый и загружаемый контекст варьируется в зависимости от архитектуры.

Уровень аппаратных абстракций

Как уже упоминалось в начале главы, одним из наиболее важных элементов конструкции Windows является ее переносимость между разнообразными аппаратными платформами. Уровень аппаратных абстракций — hardware abstraction layer (HAL) является ключевой частью, обеспечивающей возможность такой переносимости. HAL является загружаемым модулем режима ядра (Hal.dll), обеспечивающим низкоуровневый интерфейс с аппаратной платформой, на которой запущена Windows. Он скрывает подробности, зависящие от аппаратуры, такие как интерфейсы ввода-вывода, контроллеры прерываний и механизмы взаимодействия процессоров, — любые функции, имеющие как архитектурные, так и машинные зависимости.

Поэтому вместо непосредственного доступа к оборудованию, внутренние компоненты Windows, а также написанные пользователями драйверы устройств, при необходимости получения информации, зависящей от платформы, поддерживают переносимость путем вызова HAL-подпрограмм. По этой причине HAL-подпрограммы документированы в WDK. Для получения дополнительной информации о HAL и его использовании драйверами устройств нужно обратиться к WDK.

Хотя в операционную систему включено несколько HAL-модулей (см. табл. 2.4), у Windows есть возможность определить во время загрузки, какой HAL-модуль должен использоваться, исключая проблемы, существовавшие в ранее выпущенных версиях Windows при попытке загрузки установки Windows на разных типах систем.

Таблица 2.4. Перечень HAL-модулей для x86

Имя HAL-файла	Поддерживаемые системы
Halacpi.dll	Персональные компьютеры с усовершенствованным интерфейсом управления конфигурированием и энергопотреблением — Advanced Configuration and Power Interface (ACPI). Предназначается только для однопроцессорной машины без поддержки усовершенствованного программируемого контроллера прерываний — APIC (наличие любого из таких контроллеров заставит систему использовать вместо этого HAL-модуль, показанный ниже)
Halmacpi.dll	Персональные компьютеры с усовершенствованным программируемым контроллером прерываний — Advanced Programmable Interrupt Controller (APIC), имеющие ACPI. Наличие APIC подразумевает поддержку симметричной мультипроцессорной обработки — SMP

ПРИМЕЧАНИЕ

На x64-машинах имеется только один HAL-образ по имени Hal.dll. Это обусловлено наличием у всех x64-машин материнских плат одинаковой конфигурации, поскольку процессы требуют поддержки ACPI и APIC. Следовательно, поддержка машин без ACPI или со стандартного программируемого контроллера прерываний — PIC, не требуется.

ЭКСПЕРИМЕНТ: ОПРЕДЕЛЕНИЕ ЗАПУЩЕННОГО HAL-МОДУЛЯ

Определить, какая версия HAL-модуля запущена, можно с помощью WinDbg и открытия сеанса локальной отладки ядра. Обеспечьте путем ввода команды `.reload` загрузку символов, а затем наберите команду `lm vm hal`. Например, следующий вывод получен на системе, запустившей ACPI HAL:

```
lkd> lm vm hal
start    end        module name
fffff800'0181b000 fffff800'01864000   hal        (deferred)
    Loaded symbol image file: halmacpi.dll
    Image path: halmacpi.dll
    Image name: halmacpi.dll
    Timestamp:        Mon Jul 13 21:27:36 2009 (4A5BDF08)
    CheckSum:         0004BD36
    ImageSize:        00049000
    File version:     6.1.7600.16385
    Product version:  6.1.7600.16385
    File flags:       0 (Mask 3F)
    File OS:          40004 NT Win32
    File type:        2.0 Dll
    File date:        00000000.00000000
    Translations:     0409.04b0
    CompanyName:      Microsoft Corporation
    ProductName:      Microsoft® Windows® Operating System
    InternalName:     halmacpi.dll
    OriginalFilename: halmacpi.dll
    ProductVersion:   6.1.7600.16385
    FileVersion:      6.1.7600.16385 (win7_rtm.090713-1255)
    FileDescription:  Hardware Abstraction Layer DLL
    LegalCopyright:   © Microsoft Corporation. All rights reserved.
```

■

ЭКСПЕРИМЕНТ: ПРОСМОТР ЗАВИСИМОСТЕЙ NTOSKRNL И HAL

Взаимоотношения между ядром и HAL-образами можно просмотреть путем изучения их таблиц экспорта и импорта с помощью средства Dependency Walker (Depends.exe). Для изучения образа в Dependency Walker выберите пункт Open (Открыть) в меню File (Файл), чтобы открыть требуемый файл образа.

Пример вывода, который можно увидеть путем просмотра зависимостей Ntoskrnl с использованием этого средства может иметь следующий вид.

Обратите внимание на то, что Ntoskrnl связан с HAL, который, в свою очередь, связан с Ntoskrnl. (Они оба используют функции друг друга.) Ntoskrnl также связан со следующими исполняемыми файлами:

- Pshed.dll, драйвер ошибок оборудования — Platform-Specific Hardware Error Driver (PSHED), зависящий от используемой платформы. Этот драйвер предоставляет абстракцию средства выдачи отчетов об ошибках оборудования исходной платформы, скрывая подробности присущих платформе механизмов обработки ошибок от операционной системы и открывая операционной системе Windows однообразный интерфейс.
- Bootvid.dll (только на 32-разрядных операционных системах), видеодрайвер загрузки. Bootvid предоставляет поддержку команд VGA, требуемых для отображения при запуске загрузочного текста и загрузочного логотипа. На системах x64 эта библиотека во избежание конфликтов со средством защиты ядра от исправлений — Kernel Patch Protection (KPP) встроена в ядро. (Более подробно информация о KPP и PatchGuard дана в главе 3.)
- Kdcom.dll, библиотека протокола коммуникаций отладчика ядра — Kernel Debugger Protocol (KD) Communications Library.
- Ci.dll, библиотека целостности кода. (Дополнительная информация о целостности кода дана в главе 3.)
- Clfs.sys, драйвер общей системы файлов журнала, используемых, кроме всего прочего, диспетчером транзакций ядра — Transaction Manager (KTM). (Дополнительная информация о KTM дана в главе 3.)

Подробное описание информации, показываемой этим средством, дана в справочном файле по Dependency Walker (Depends.hlp). ■

Драйверы устройств

В этом разделе предоставляется краткий обзор типов драйверов и объясняется, как получить список драйверов, установленных и загруженных на вашей системе.

Драйверы устройств являются загружаемыми модулями режима ядра (имена файлов которых обычно имеют расширение .sys), обеспечивающими интерфейс между диспетчером ввода-вывода и соответствующим оборудованием. Они запускаются в режиме ядра в одном их трех контекстов:

- в контексте пользовательского потока, инициировавшего функцию ввода-вывода;
- в контексте системного потока режима ядра;
- в результате прерывания (и поэтому вне контекста любого конкретного процесса или потока — какой бы процесс или поток ни был текущим при возникновении прерывания).

Как утверждалось в предыдущем разделе, драйверы устройств в Windows не работают с оборудованием напрямую, а вызывают для получения интерфейса с оборудованием функции в HAL. Драйверы обычно пишутся на C (иногда на C++) и поэтому, при правильном использовании подпрограмм HAL могут переноситься в виде исходного кода между архитектурами центральных процессоров, поддерживаемых Windows, а в виде исполняемого файла могут переноситься внутри архитектурного семейства.

Существует несколько типов драйверов устройств:

- *Драйверы аппаратных устройств*, управляющие (с помощью HAL) записью на физическое устройство или в сеть или чтением оттуда. Существует множество типов драйверов аппаратных устройств — драйверы шин, драйверы интерфейса с пользователем, драйверы запоминающих устройств большой емкости и т. д.
- *Драйверы файловой системы*, являющиеся драйверами Windows, которые принимают запросы на файловый ввод-вывод и транслируют их в запросы ввода-вывода, направляемые конкретному устройству.
- *Драйверы фильтра файловой системы*, обеспечивающие зеркалирование и шифрование дисков, перехваты ввода-вывода и некоторую дополнительную обработку перед передачей ввода-вывода на следующий уровень.
- *Сетевые редиректоры и серверы*, являющиеся драйверами файловой системы, передающими запросы ввода-вывода файловой системы машине, находящейся в сети, и получающими от нее аналогичные запросы.
- *Драйверы протоколов*, реализующие такие сетевые протоколы, как TCP/IP, NetBEUI и IPX/SPX.
- *Драйверы потоковых фильтров ядра*, собранные в цепочку для обработки сигналов потока данных, например, при записи или воспроизведении аудио- и видеопотоков.

Поскольку единственным способом добавления к системе написанного пользователям кода режима ядра является установка драйвера устройства, некоторые программисты пишут драйверы устройств просто для получения способа

доступа к внутренним функциям или структурам данных операционной системы, недоступным из пользовательского режима (но документированным и поддерживаемым в WDK). Например, многие утилиты из Sysinternals сочетают в себе приложение Windows GUI и драйвер устройства, используемый для сбора внутреннего состояния системы и вызова функций, доступных только в режиме ядра и недоступных из Windows API в режиме пользователя.

Модель драйверов Windows (WDM)

В Windows 2000 была добавлена поддержка технологии Plug and Play, настроек электропитания и расширение модели драйверов Windows NT, названной моделью драйверов Windows (WDM). Windows 2000 и более поздние версии могут запускать драйверы, унаследованные у Windows NT 4, но, поскольку они не поддерживают технологию Plug and Play настройки электропитания, системы, запускающие эти драйверы, будут вынуждены ограничивать возможности в этих двух областях.

С точки зрения WDM, существуют драйверы трех типов:

- *Драйвер шины*, обслуживающий контроллер шины, адаптер, мост или любое устройство, имеющее дочерние устройства. Драйверы шины нуждаются в драйверах, и Microsoft, как правило, их предоставляет; каждый тип шины (такой как PCI, PCMCIA и USB), имеющийся в системе, имеет один драйвер шины. Сторонние производители могут создавать драйверы шины для предоставления поддержки новых шин, таких как VMEbus, Multibus и Futurebus.
- *Функциональный драйвер*, являющийся основным драйвером устройства и предоставляющий для него управляющий интерфейс. Драйвер нужен в том случае, если устройство не используется напрямую (в варианте реализации, при которой ввод-вывод осуществляется драйвером шины и любыми драйверами фильтра шины, в качестве примера можно привести SCSI PassThru). Функциональный драйвер по определению является драйвером, который знает о конкретном устройстве практически все, и обычно он является единственным драйвером, обращающимся к специфическим регистрам устройства.
- *Драйвер фильтра*, использующийся для добавления функциональности к устройству (или к существующему драйверу) или для изменения запросов ввода-вывода или ответов от других драйверов (для настройки оборудования, предоставляющего неверную информацию о требованиях к аппаратным ресурсам). Драйверы фильтра являются дополнительными и могут присутствовать в любом количестве, размещаясь выше или ниже функционального драйвера и выше драйвера шины. Обычно драйверы фильтра поставляются OEM-производителями или независимыми поставщиками оборудования (IHV).

В среде окружения WDM все аспекты устройства контролируются не одним драйвером: драйвер шины занимается отправкой диспетчеру PnP отчетов об устройствах, подключенных к его шине, а функциональный драйвер управляет самим устройством.

В большинстве случаев драйверы фильтра, находящиеся на нижнем уровне, изменяют поведение устройства. Например, если устройство сообщает своему драйверу шины, что ему нужно 4 порта ввода-вывода, в то время как ему факти-

чески нужно 16 портов ввода-вывода, функциональный драйвер фильтра для данного конкретного устройства может перехватить перечень аппаратных ресурсов, о котором драйвер шины сообщает диспетчеру PnP, и исправить количество портов ввода-вывода.

Драйверы фильтра, находящиеся на верхнем уровне, обычно предоставляют устройству какие-нибудь дополнительные свойства. Например, драйвер фильтра такого устройства, как клавиатура, находящийся на верхнем уровне, может навязывать дополнительные проверки безопасности.

Обработка прерывания рассматривается в главе 3.

Windows Driver Foundation

Набор инструментальных средств Windows Driver Foundation (WDF) упрощает разработку драйверов Windows, предоставляя для этого две среды: Kernel-Mode Driver Framework (KMDF) и User-Mode Driver Framework (UMDF). Разработчики могут использовать KMDF для написания драйверов для Windows 2000 SP4 и более поздних версий, в то время как UMDF поддерживает Windows XP и более поздние версии.

KMDF предоставляет простой интерфейс к WDM и скрывает все ее сложности от создателя драйвера, не изменяя исходной модели «шина-функция-фильтр». Драйверы KMDF отвечают за события, которые могут быть ими зарегистрированы, и осуществляют вызовы к библиотеке KMDF для выполнения работы, неспецифичной для управляемого ими оборудования, например для общего управления электропитанием или для синхронизации. (Ранее это должен был самостоятельно реализовывать каждый драйвер.) В некоторых случаях более чем 200 строк кода WDM можно заменить одним лишь вызовом функции KMDF.

UMDF позволяет создавать драйверы определенных классов (главным образом основанные на USB или других шинах, использующих протокол с большими задержками), например драйверы для видеокамер, MP3-плееров, мобильных телефонов, наладонников и принтеров, которые могут быть реализованы как драйверы пользовательского режима. По сути UMDF запускает каждый драйвер пользовательского режима в виде службы пользовательского режима и использует ALPC для связи с драйвером-упаковщиком режима ядра, предоставляющего фактический доступ к оборудованию. Если UMDF-драйвер попадает в аварийную ситуацию, процесс завершается и, как правило, перезапускается, поэтому система сохраняет стабильность — устройство просто становится недоступным, пока перезапускается служба, являющаяся хозяином драйвера. И наконец, UMDF-драйверы пишутся на C++ с использованием COM-подобных классов и семантики, тем самым еще больше понижая планку для программистов, пишущих драйверы устройств.

ЭКСПЕРИМЕНТ: ПРОСМОТР УСТАНОВЛЕННЫХ ДРАЙВЕРОВ УСТРОЙСТВ

Получить перечень установленных драйверов можно путем запуска Msinfo32. (Для запуска этого средства щелкните на кнопке Пуск (Start), наберите `Msinfo32` и нажмите клавишу Ввод.) В разделе Сведения о системе (System Summary) раскройте пункт Программная среда (Software Environment) и откройте окно Системные драйверы (System Drivers). Пример вывода перечня установленных драйверов имеет следующий вид.

В этом окне выведен перечень драйверов устройств, обнаруженных в реестре с указанием их типов и состояния — Running (работает) или Stopped (остановлен). И драйверы устройств, и процессы служб Windows определены в одном и том же месте: HKLM\SYSTEM\CurrentControlSet\Services. Но они различаются по типу кода, например тип 1 относится к драйверу устройства режима ядра. (Полный перечень информации, сохраненной в реестре для драйверов устройств, дан в табл. 4.7 в главе 4.)

Вместо этого перечень текущих загруженных драйверов устройств можно получить, выбрав System process в Process Explorer и открыв просмотр DLL. ■

ПРИСМАТРИВАЯСЬ К НЕДОКУМЕНТИРОВАННЫМ ИНТЕРФЕЙСАМ

Прояснить ситуацию можно путем изучения имен экспортированных или глобальных символов в ключевых системных образах (таких как Ntoskrnl.exe, Hal.dll или Ntdll.dll). Вы можете получить представление о тех вещах, которые может сделать Windows, а не только о том, что уже документировано и поддержано на данный момент. Разумеется, одно только знание имен этих функций еще не означает, что вы можете или должны их вызвать, — интерфейсы не документированы и могут подвергаться изменениям. Мы предлагаем вам только присмотреться к этим функциям для получения более полного представления о видах внутренних функций, выполняемых Windows, а не для обхода поддерживаемых интерфейсов.

Например, просмотр списка функций в Ntdll.dll даст вам возможность сравнить список всех системных служб, предоставляемых Windows DLL-библиотекам подсистем пользовательского режима, с подмножеством, показываемым каждой подсистемой. Хотя многие из этих функций отображаются на документированные и поддерживаемые Windows-функции, некоторые из них недоступны из Windows API. (См. статью «Inside the Native API» на сайте Sysinternals.)

С другой стороны, также интересно изучить то, что импортируют DLL-библиотеки подсистемы Windows (такие как Kernel32.dll или Advapi32.dll) и какие функции они вызывают в Ntdll.

Также интересно посмотреть на дамп файла Ntoskrnl.exe. Хотя многие экспортируемые подпрограммы, используемые драйверами устройств режима ядра, документированы в Windows Driver Kit, довольно многие из них остались недокументированными. Может также вызвать интерес просмотр таблицы импорта для Ntoskrnl и HAL; в этой таблице показан список функций в HAL, которые используются в Ntoskrnl, и наоборот.

В табл. 2.5 показан список большинства широко используемых префиксов имен функций компонентов исполняющей системы. Каждый из этих основных компонентов исполнительной системы также использует слегка измененные префиксы для обозначения внутренних функций — либо за первой буквой префикса следует буква *i* (означающая *internal*, внутренняя), либо за всем префиксом следует буква *p* (означающая *private*, закрытая). Например, *Ki* представляет внутренние функции ядра, а *Psp* ссылается на внутренние функции поддержки процесса.

Упростить расшифровку имен этих экспортируемых функций поможет понимание соглашения об именах системных подпрограмм Windows. Общий формат имеет следующий вид:

```
<Префикс><Операция><Объект>
```

В этом формате *Префикс* представляет внутренний компонент, экспортирующий подпрограмму, *Операция* сообщает о том, что будет сделано с объектом или ресурсом, а Объект идентифицирует то, над чем будет проводиться операция.

Например, `ExAllocatePoolWithTag` является вспомогательной процедурой исполняющей системы для выделения из выгружаемого или невыгружаемого пула. `KeInitializeThread` является процедурой, которая назначает и создает объект ядра «поток».

Таблица 2.5. Часто используемые префиксы

Префикс	Компонент
Alpc	Расширенное локальное межпроцессное взаимодействие
Cc	Общая кэш-память
Cm	Диспетчер конфигурации
Dbgk	Среда отладки для пользовательского режима
Em	Диспетчер исправлений
Etw	Отслеживание событий для Windows
Ex	Подпрограммы поддержки исполняющей системы
FsRtl	Библиотека времени выполнения драйвера файловой системы
Hvl	Библиотека гипервизора
Io	Диспетчер ввода-вывода
Kd	Отладчик ядра
Ke	Ядро
Lsa	Авторизация локальных пользователей
Mm	Диспетчер памяти

Таблица 2.5 *(продолжение)*

Префикс	Компонент
Nt	Системные службы NT (большинство из которых экспортируется в качестве Windows-функций)
Ob	Диспетчер объектов
Pf	Prefetcher
Po	Диспетчер электропитания
Pp	Диспетчер PnP
Ps	Поддержка процессов
Rtl	Библиотека времени выполнения
Se	Безопасность
Sm	Диспетчер запоминающего устройства
Tm	Диспетчер транзакций
Vf	Верификатор
Wdi	Инфраструктура диагностики Windows
Whea	Архитектура ошибок оборудования Windows
Wmi	Инструментарий управления Windows
Zw	Зеркальная точка входа для системных служб (имена которых начинаются с Nt), которая устанавливает предыдущий режим доступа к ядру, что исключает проверку параметров, поскольку системные службы Nt проверяют параметры только в том случае, если предыдущий режим доступа был пользовательским

Системные процессы

В каждой системе Windows появляются следующие системные процессы (Idle и System не являются полноценными процессами, поскольку в них не запускается какой-нибудь исполняемый код пользовательского режима):

- процесс Idle (содержащий по одному потоку на каждый центральный процессор для подсчета времени его простоя);
- процесс System (содержащий основную часть системных потоков режима ядра);
- диспетчер сеанса (Smss.exe);
- диспетчер локальных сеансов (Lsm.exe);
- подсистема Windows (Csrss.exe);
- инициализация сеанса 0 (Wininit.exe);
- процесс входа в систему (Winlogon.exe);
- диспетчер управления службами (Services.exe) и создаваемые им процессы дочерних служб (например, поддерживаемый системой универсальный сервис-хост процесс, Svchost.exe);
- сервер проверки подлинности локальной системы безопасности (Lsass.exe).

Чтобы понять взаимосвязь этих процессов, полезно будет просмотреть «дерево» процессов, то есть связь между родительскими и дочерними процессами. Эта связь поможет понять, откуда появляется тот или иной процесс. На рис. 2.5 показана копия экрана дерева процессов, просматриваемого после загрузочной трассировки, предпринятой средством Process Monitor. Использование средства Process Monitor позволяет видеть процессы, выход из которых на тот момент уже был осуществлен (помеченные блеклыми значками).

Рис. 2.5. Исходное дерево системных процессов

Ключевые системные процессы, показанные на рис. 2.5, рассматриваются в следующих разделах. Также будет кратко показан порядок запуска процессов.

Процесс простоя системы

Первым в списке на рис. 2.5 показан процесс простоя системы — Idle. Как будет объяснено в главе 5, процессы идентифицируются по именам их образов. Но этот процесс (и процесс System) не запускает какой-нибудь настоящий образ пользовательского режима (поэтому в каталоге \Windows не существует «System Idle Process.exe»). Кроме того, разными утилитами имя для этого процесса показывается по-разному (из-за особенностей реализации). В табл. 2.6 перечислен ряд имен, даваемых процессу Idle (с идентификатором процесса 0). Процесс Idle более подробно рассматривается в главе 5.

Таблица 2.6. Имена для процесса с идентификатором 0 в разных утилитах

Утилита	Имя для процесса с идентификатором 0
Task Manager	System Idle Process
Process Status (Pstat.exe)	Idle Process
Process Explorer (Procexp.exe)	System Idle Process
Task List (Tasklist.exe)	System Idle Process
Tlist (Tlist.exe)	System Process

Теперь давайте посмотрим на системные потоки и на предназначение каждого из системных процессов, в которых запущены реальные образы.

Процесс System и системные потоки

Процесс System (с идентификатором 4) дает начало потоку особого рода, запускаемому только в режиме ядра: *системному потоку режима ядра*. У системных потоков есть все атрибуты и контекстные составляющие, присущие обычным потокам пользовательского режима (например, контекст оборудования, приоритет и т. д.), но их отличие состоит в том, что они запускаются только в исполняемом коде в режиме ядра, который загружен в системное пространство, будь то код Ntoskrnl.exe или код любого другого загруженного драйвера устройства. Кроме того, у системных потоков нет адресного пространства пользовательского процесса и поэтому им нужно выделять любую динамическую память из динамически распределяемой памяти операционной системы, например выгружаемый или невыгружаемый пул.

Системные потоки создаются функцией `PsCreateSystemThread` (см. WDK), которая может быть вызвана только из режима ядра. Windows, а также различные драйверы устройств создают системные потоки при инициализации системы для выполнения операций, которым требуется контекст потока: например, для выдачи запросов на ввод-вывод и ожидания ответной реакции, или для ожидания реакции других объектов, или для опроса какого-нибудь устройства. Например, диспетчер памяти использует системные потоки для реализации таких функций, как запись измененных страниц в страничные файлы или в отображенные файлы, свопинг процессов в память и из памяти и т. д. Ядро создает системный поток под названием *диспетчер настройки баланса* (balance set manager), который активизируется каждую секунду для возможной инициации событий, связанных с планированием и управлением памятью. Системные потоки используются также диспетчером кэша для реализации опережающего чтения и отложенной записи при операциях ввода вывода. Драйвер файлового сервера (Srv2.sys) использует системные потоки для ответа на сетевые запросы ввода-вывода применительно к файловым данным, находящимся на общих для сети разделах диска. Для опроса устройства системные потоки есть даже у драйвера дисковода гибких дисков. (Опрос в данном случае более эффективен, поскольку привод гибких дисков, управляемый с помощью прерываний, расходует больше системных ресурсов.) Дополнительная информация о конкретных системных потоках включена в главы, в которых описывается тот или иной компонент.

По умолчанию системные потоки принадлежат процессу System, но драйвер устройства может создать системный поток в любом процессе. Например, драйвер устройства подсистемы Windows (Win32k.sys) создает системный поток внутри драйвера дисплея Canonical Display Driver (Cdd.dll), части процесса подсистемы Windows (Csrss.exe), чтобы он мог иметь упрощенный доступ к данным этого процесса, находящимся в адресном пространстве пользовательского режима.

При поиске неисправностей или анализе системы полезно сопоставить выполнение отдельных системных потоков с драйвером или даже с подпрограммой, содержащей код. Например, на сильно загруженном файловом сервере процесс System будет, скорее всего, потреблять существенную долю времени центрального процессора. Но для определения, какой именно драйвер устройства или компонент операционной системы запущен, будет недостаточно знания о том, что при запуске процесса System запускается «некий системный поток».

Поэтому, если потоки запущены в процессе System, сначала нужно определить, какой из них запущен (например, с помощью монитора производительности — Performance Monitor). При обнаружении запущенного потока (или потоков) посмотрите, в каком драйвере начал выполняться системный поток (что, по крайней мере, подскажет вам, какой именно драйвер, скорее всего, создал поток), или изучите стек вызовов (или, как минимум, текущий адрес) данного потока, что покажет, где поток выполняется в данное время.

Оба этих приема показаны в следующих экспериментах.

ЭКСПЕРИМЕНТ: ОТОБРАЖЕНИЕ СИСТЕМНОГО ПОТОКА НА ДРАЙВЕР УСТРОЙСТВА

В данном эксперименте будет показано, как установить соответствие активности центрального процессора с процессом System и с его системным потоком, который вызвал эту активность (и с драйвером, в который попадает этот поток). Это важно, потому что, когда запущен процесс System, нужно разобраться с его потоками, чтобы понять, что происходит. Для данного эксперимента мы сгенерируем активность системного потока путем генерации активности файлового сервера на вашей машине. (Драйвер файлового сервера, Srv2.sys, создает системные потоки для обработки входящих запросов файлового ввода-вывода. Дополнительная информация, касающаяся данного компонента, дана в главе 7.)

1. Откройте окно командной строки.
2. Выведите список каталогов всего вашего диска C, используя для доступа к нему сетевой путь. Например, если имя вашего компьютера `COMPUTER1`, наберите команду `dir \\computer1\c$ /s` (ключ `/s` приведет к выводу всех подкаталогов).
3. Запустите средство Process Explorer и дважды щелкните на строке процесса System.
4. Щелкните на вкладке Threads (Потоки).
5. Отсортируйте таблицу по столбцу CSwitch Delta (изменения в переключениях контекста). Вы должны увидеть один или несколько потоков, запущенных в Srv2.sys, таких как, например, следующие.

Если вы видите запущенный системный поток и не уверены, к какому драйверу он относится, щелкните на кнопке Module (Модуль), которая позволит извлечь свойства файла. Щелчок на кнопке Module (Модуль) при выделенном как на предыдущем рисунке потоком в Srv2.sys, приведет к отображению следующей информации.

Диспетчер сеанса (Smss)

Диспетчер сеанса (%SystemRoot%\System32\Smss.exe) является первым процессом пользовательского режима, созданным в системе. Этот процесс создается системным потоком режима ядра, выполняющим финальную фазу инициализации исполняющей системы и ядра.

При запуске Smss осуществляется проверка наличия его первого экземпляра (ведущего Smss) или своего собственного экземпляра, запущенного ведущим Smss для создания сеанса. (При наличии аргументов командной строки это будет последний экземпляр.) Путем создания нескольких экземпляров себя самого во время загрузки и создания сеанса Служб терминалов Smss может одновременно создать несколько сеансов (максимум четыре текущих сеанса, четыре параллельных сеанса, плюс еще один для каждого дополнительного центрального процессора, кроме первого). Эта возможность повышает производительность при входе в систему на системах Служб терминалов при одновременном подключении сразу нескольких пользователей. Когда завершится инициализация сеанса, копия Smss завершает свою работу. В результате остается активным только исходный процесс Smss.exe.

Ведущий Smss выполняет следующие единовременные этапы инициализации:

1. Помечает процесс и исходный поток как критический. (Если процесс или поток, помеченный как критический, по каким-то причинам завершается, происходит аварийное завершение работы Windows. Дополнительная информация дана в главе 5.)
2. Повышает базовый приоритет процесса до 11.
3. Если система поддерживает горячее добавление процессора, включает автоматическое обновление процессорного сходства, чтобы при добавлении нового процессора новые сеансы могли воспользоваться новыми процессорами. (Дополнительная информация о динамическом добавлении процессоров дана в главе 5.)
4. Создает поименованные каналы (pipes) и почтовые слоты (mailslots), используемые для связи между Smss, Csrss и Lsm (рассматриваемом далее).
5. Создает ALPC-порт для получения команд.
6. Создает общесистемные переменные среды окружения, определяемые в разделе HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Environment.
7. Создает символические ссылки для устройств, определенных в разделе HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\DOS Devices в каталоге \Global?? в пространстве имен диспетчера объектов.
8. Создает в пространстве имен диспетчера объектов корневой каталог \Sessions.
9. Запускает программы, указанные в разделе HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\BootExecute. (По умолчанию там указывается программа Autochk.exe, осуществляющая проверку диска.)
10. Проводит отложенный перенос файлов, указанный в разделе HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\PendingFileRenameOperations.

11. Инициализирует файл (файлы) подкачки.
12. Инициализирует всю остальную часть реестра (разделы HKLM Software, SAM и Security).
13. Запускает программы, указанные в разделе HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\SetupExecute.
14. Открывает известные DLL-библиотеки (HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\KnownDLLs) и отображает их как постоянные разделы (отображаемые файлы).
15. Создает поток, отвечающий за обработку запросов на создание сеанса.
16. Создает Smss для инициализации сеанса 0 (неинтерактивного сеанса).
17. Создает Smss для инициализации сеанса 1 (интерактивного сеанса).

Когда эти этапы будут выполнены, Smss переходит в режим постоянного ожидания дескриптора экземпляра Csrss.exe с нулевым сеансом. Поскольку Csrss имеет метку критического процесса, если происходит выход из Csrss, это ожидание из-за аварийного завершения работы системы никогда не заканчивается.

Экземпляр Smss, запускающий сеанс, выполняет следующие действия:

1. Вызывает функцию `NtSetSystemInformation` с запросом на создание структуры данных сеанса режима ядра. Та, в свою очередь, вызывает внутреннюю функцию диспетчера памяти `MmSessionCreate`, настраивающую виртуальное адресное пространство сеанса, которое будет содержать нерезидентный пул сеанса и структуру данных данного сеанса, выделяемые той частью подсистемы Windows, которая работает в режиме ядра (Win32k.sys), и другими драйверами устройств, относящимися к пространству сеанса.
2. Создает для сеанса процесс (процессы) подсистемы (по умолчанию Csrss.exe подсистемы Windows).
3. Создает экземпляр Winlogon (интерактивные сеансы) или Wininit (для сеанса 0). Дополнительная информация об этих двух процессах будет дана далее.

Затем происходит выход из этого промежуточного процесса Smss (после чего остаются процессы подсистемы, и в качестве процесса, не имеющего своего родительского процесса, остается процесс Winlogon или Wininit).

Процесс инициализации Windows (Wininit.exe)

Процесс Wininit.exe выполняет следующие функции инициализации системы:

- помечает сам себя как критический, чтобы при постоянных выходах из него и загрузке системы в режиме отладки он выходил в отладчик (если этого не случится, произойдет аварийное завершение работы системы);
- инициализирует инфраструктуру планирования пользовательского режима;
- создает папку %windir%\temp;
- создает станцию окна (Winsta0) и два рабочих стола (Winlogon и Default) для процессов, запускаемых в сеансе 0;
- создает поток Services.exe (Диспетчер управления службами — Service Control Manager или SCM);

- запускает Lsass.exe (Local Security Authentication Subsystem Server — Сервер проверки подлинности локальной системы безопасности);
- запускает Lsm.exe (Диспетчер локальных сеансов);
- входит в режим бесконечного ожидания завершения работы системы.

Диспетчер управления службами (SCM)

Вспомним, что ранее в данной главе под «службой» в Windows понимался либо серверный процесс, либо драйвер устройства. В этом разделе службы рассматриваются в качестве процессов пользовательского режима. Службы похожи на «процессы-демоны» в UNIX или на «обособленные процессы» VMS тем, что они могут быть настроены на автоматический запуск при загрузке системы, не требуя при этом интерактивного входа в систему. Они также могут быть запущены вручную (например, путем запуска средства администрирования Службы или путем вызова Windows-функции `StartService`). Обычно службы не взаимодействуют с пользователями, вошедшими в систему, хотя есть особые условия, открывающие такую возможность (см. главу 4).

Диспетчер управления службами является специальным системным процессом, запустившим образ %SystemRoot%\System32\Services.exe, отвечающим за запуск и остановку процессов служб, а также за взаимодействие с ними. Программы служб фактически являются Windows-образами, которые вызывают специальные Windows-функции для взаимодействия с диспетчером управления службами, чтобы выполнить такие действия, как регистрация успешного запуска службы, ответы на запросы о ее состоянии, или приостановки или полной остановки службы. Службы определены в реестре в разделе HKLM\SYSTEM\CurrentControlSet\Services.

Следует иметь в виду, что у служб три имени: имя процесса, который виден запущенным в системе, внутреннее имя в реестре и имя, показываемое в средстве администрирования Службы. Отображаемое имя есть не у всех служб. Если у службы нет отображаемого имени, то показывается ее внутреннее имя. В Windows службы могут также иметь поле описания, дающее более глубокое представление о том, чем занимается та или иная служба.

Чтобы сопоставить процесс службы со службой, содержащейся в этом процессе, используется команда `tlist /s` или команда `tasklist /svc`. Следует заметить, что между процессами служб и запущенными службами однозначное соответствие бывает не всегда, потому что некоторые службы используют процесс совместно с другими службами. Код типа, имеющийся в реестре, показывает, запущена ли служба в своем собственный процессе или делит процесс с остальными службами образа.

Некоторые компоненты Windows реализованы в виде служб. К ним относятся Диспетчер печати, Журнал событий, Планировщик заданий и различные сетевые компоненты. Более подробные сведения о службах даны в главе 4.

ЭКСПЕРИМЕНТ: ВЫВОД СПИСКА УСТАНОВЛЕННЫХ СЛУЖБ

Для вывода списка установленных служб выберите в окне Панель управления (Control Panel) пункт Администрирование (Administrative Tools), а затем выберите пункт Службы (Services). В результате должна быть выведена информация, похожая на следующую:

Чтобы увидеть детализированные свойства службы, щелкните на имени службы правой кнопкой мыши и выберите пункт Свойства (Properties). Например, на следующем рисунке показаны свойства службы под названием Диспетчер печати (чье имя выделено на предыдущем рисунке).

Обратите внимание на то, что в поле Исполняемый файл показана программа, содержащая данную службу. Следует помнить, что некоторые службы используют процесс совместно с другими службами, поэтому однозначное сопоставление службы и процесса получается не всегда. ■

Диспетчер локальных сеансов (Lsm.exe)

Диспетчер локальных сеансов (Lsm.exe) управляет на локальной машине состоянием сеансов службы терминалов. Он отправляет через ALPC-порт SmSsWinStationApiPort запросы к Smss на запуск новых сеансов (например, на создание процессов Csrss и Winlogon), как при выборе пользователем в Explorer пункта Сменить пользователя (Switch User). Lsm также поддерживает связь с Winlogon и Csrss (используя RPC локальной системы). Он информирует Csrss о таких событиях, как подключение, отключение, завершение и рассылка системного сообщения. Он получает уведомление от Winlogon о следующих событиях:

- Вход и выход.
- Запуск и остановка оболочки.
- Подключение к сеансу.
- Отключение от сеанса.
- Установка или снятие блокировки рабочего стола.

ЭКСПЕРИМЕНТ: ПРОСМОТР ПОДРОБНОСТЕЙ СЛУЖБЫ В ЕЕ ПРОЦЕССАХ

Средство Process Explorer выделяет хост-процессы одной или нескольких служб. Свойство выделения можно настроить, выбрав в меню Options (Настройки) пункт Configure Colors (Цветовые настройки). Если дважды щелкнуть на имени хост-процесса одной или нескольких служб, во вкладке Services (Службы) можно увидеть перечень служб в процессе, имя параметра реестра, в котором определена служба, отображаемое имя, которое видит администратор, текст описания службы (если таковой имеется) и для служб Svchost путь к DLL-библиотеке, реализующей службу. Например, перечень служб в процессе Svchost.exe, запущенном под учетной записью System, выглядит следующим образом.

Winlogon, LogonUI и Userinit

Процесс входа в Windows (Winlogon, %SystemRoot%\System32\Winlogon.exe) обрабатывает интерактивные пользовательские входы в систему и выходы из нее. Winlogon получает уведомление о запросе пользователя на вход в систему, когда с помощью определенной комбинации клавиш вводится безопасная последовательность привлечения внимания — secure attention sequence (SAS). В качестве SAS в Windows используется комбинация Ctrl+Alt+Delete. Причиной применения SAS является защита пользователей от программ перехвата паролей, имитирующих процесс входа в систему (эта клавиатурная последовательность не может быть перехвачена приложением, работающем в пользовательском режиме).

Аспекты идентификации и аутентификации процесса входа в систему реализованы через DLL-библиотеки, называемые поставщиками учетных данных (credential providers). Стандартные поставщики учетных данных Windows реализуют интерфейсы аутентификации, используемые в Windows по умолчанию: паролей и смарт-карт. Но разработчики могут предоставить своих собственных поставщиков учетных данных для реализации вместо стандартного для Windows метода имя пользователя-пароль других механизмов идентификации и аутентификации (например, на основе распознавания голоса или биометрического устройства вроде сканера отпечатков пальцев). Поскольку Winlogon является критическим системным процессом, от которого зависит работа системы, поставщики учетных данных и пользовательский интерфейс для вывода диалогового окна входа в систему запускаются внутри дочернего процесса Winlogon под названием LogonUI. Когда Winlogon обнаруживает SAS, он запускает этот процесс, который инициализирует поставщиков учетных данных. Как только пользователь вводит свои учетные данные или отказывается от интерфейса входа в систему, процесс LogonUI завершается.

Кроме того, Winlogon может загрузить дополнительные DLL-библиотеки сетевых поставщиков (network providers), необходимые для дополнительной аутентификации. Эта возможность позволяет нескольким сетевым поставщикам за один раз во время обычного входа в систему собрать информацию, касающуюся идентификации и аутентификации.

Как только будут зарегистрированы имя пользователя и пароль, они будут отправлены для аутентификации процессу сервера проверки подлинности локальной системы безопасности (%SystemRoot%\System32\Lsass.exe, см. главу 6) to be authenticated. LSASS вызывает соответствующий пакет аутентификации (реализованный в виде DLL-библиотеки) для выполнения фактической проверки — например, для проверки соответствия пароля тому, что сохранено в Active Directory или в SAM (части реестра, в которой содержатся определения локальных пользователей и групп).

После успешной аутентификации LSASS вызывает функцию в мониторе безопасности (например, `NtCreateToken`) для генерации объекта маркера доступа, содержащего профиль безопасности пользователя. Если используется управление учетными записями пользователей (User Account Control, UAC) и входящий в систему пользователь относится к группе администраторов или имеет права администратора, LSASS создаст вторую, ограниченную версию маркера. Затем этот маркер доступа используется Winlogon для создания исходного процесса

(процессов) в пользовательском сеансе. Исходный процесс (процессы) хранится в параметре реестра Userinit, который находится в разделе HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon. (По умолчанию это Userinit.exe, но в списке может быть более одного образа.)

Userinit выполняет действия по инициализации пользовательской среды окружения (например, запускает сценарий входа в систему и применяет групповую политику), а затем смотрит в реестр на значение параметра Shell (в разделе Winlogon) и создает процесс для запуска определяемой системой оболочки (по умолчанию Explorer.exe). Затем Userinit завершается. Именно поэтому для процесса Explorer.exe родительский процесс не показывается — его родительский процесс завершился, и, как объяснялось в главе 1, средство tlist выравнивает имена процессов, чьи родительские процессы не работают по левому краю. (По другому это выглядит так, что процесс Explorer является внуком процесса Winlogon.)

Winlogon является активным не только во время входа пользователя в систему и выхода из нее, но также и при перехвате SAS с клавиатуры. Например, если войдя в систему нажать комбинацию клавиш Ctrl+Alt+Delete, появится экран безопасности Windows, предоставляющий возможности для выхода из системы, запуска диспетчера задач, блокировки компьютера и т. д. Этим взаимодействием занимаются процессы Winlogon и LogonUI.

Подробности проверки подлинности рассмотрены в главе 6. Подробности вызываемых функций, устанавливающих связь с LSASS (функций, чьи имена начинаются с префикса Lsa), даны в документации по Windows SDK.

Заключение

В данной главе мы ознакомились с общей системной архитектурой Windows. Мы изучили основные компоненты Windows и их взаимосвязи. В следующей главе более подробно будут рассмотрены основные системные механизмы, на которых построены эти компоненты, такие как диспетчер объектов и синхронизация.

Глава 3. Системные механизмы

Операционная система Windows предоставляет ряд основных механизмов, используемых такими компонентами режима ядра, как исполняющая система, ядро и драйверы устройств. В данной главе рассматриваются следующие системные механизмы и описывается порядок их использования:

- диспетчеризация системных прерываний, включая прерывания, отложенные вызовы процедур (DPC), асинхронные вызовы процедур (APC), диспетчеризация исключений и диспетчеризация системных служб;
- диспетчер объектов исполняющей системы;
- синхронизация, включая спин-блокировки, объекты диспетчера ядра, порядок реализации ожиданий, а также примитивы синхронизации, относящиеся к пользовательскому режиму и избегающие переходов в режим ядра (в отличие от обычных объектов диспетчера);
- системные рабочие потоки;
- различные механизмы, например глобальные флаги Windows;
- усовершенствованные вызовы локальных процедур (ALPC);
- трассировка событий ядра (Kernel event tracing);
- Wow64;
- отладка в пользовательском режиме;
- загрузчик образов (image loader);
- гипервизор (Hyper-V);
- диспетчер транзакций ядра (Kernel Transaction Manager, KTM);
- защита ядра от исправлений (Kernel Patch Protection, KPP);
- обеспечение целостности кода.

Диспетчеризация системных прерываний

Прерывания и исключения являются условиями операционной системы, отвлекающими процессор на выполнение кода, находящегося за пределами нормального потока управления. Они могут быть обнаружены либо аппаратными, либо программными средствами. Термин *системное прерывание* относится к механизму процессора, предназначенному для захвата выполняемого потока при возникновении исключения или прерывания и для передачи управления в определенное место в операционной системе. В Windows процессор передает управление *обработчику системного прерывания*, который является функцией, характерной для того или иного прерывания или исключения. На рис. 3.1 проиллюстрированы некоторые условия активации обработчиков системных прерываний.

Ядро различает прерывания и исключения следующим образом. *Прерывание* является асинхронным событием (которое может произойти в любое время), не связанным с текущей задачей процессора. Прерывания генерируются, главным образом, устройствами ввода-вывода, процессорными часами или таймерами, и они могут быть разрешены (включены) или запрещены (выключены). *Исключение*, напротив, является синхронным условием, которое обычно возникает в резуль-

тате выполнения конкретной инструкции. (Аварийные завершения работы, например, из-за машинного сбоя, обычно не связаны с выполнением инструкции.) Повторение исключений может быть вызвано повторным запуском программы с теми же данными и при тех же условиях. В качестве примеров исключений можно привести нарушения доступа к памяти, определенные инструкции отладки и ошибки деления на нуль. Ядро также считает исключениями вызовы системных служб (хотя технически они являются системными прерываниями).

Рис. 3.1. Диспетчеризация системных прерываний

Исключения и прерывания могут генерироваться как оборудованием, так и программами. Например, причиной исключения, связанного с ошибкой шины, является оборудование, а исключение, связанное с делением на нуль, является результатом программной ошибки. Точно так же прерывание может генерироваться устройством ввода-вывода, или же программное прерывание может быть выдано самим ядром (прерывания APC или DPC будут рассмотрены в этой главе).

При генерации аппаратного исключения или прерывания процессор записывает довольно большой объем информации о состоянии машины в стек ядра того потока, который был прерван, для возвращения к нужной точке потока управления и продолжения выполнения, как будто ничего не случилось. Если поток выполнялся в пользовательском режиме, Windows переключается на стек потоков режима ядра. Затем Windows создает в стеке ядра *фрейм системного прерывания* прерванного потока, в котором она сохраняет состояние выполнения потока. Фрейм системного прерывания является подмножеством полного контекста потока, и его определение можно увидеть, набрав в отладчике ядра команду dt nt!_ktrap_frame (см. главу 5). Ядро обрабатывает программные прерывания либо как часть обработки аппаратных прерываний, либо одновременно с ними, когда поток вызывает функции ядра, относящиеся к программному прерыванию.

В большинстве случаев ядро устанавливает внешние функции обработки системных прерываний, которые выполняют общие задачи их обработки до и после передачи управления другим функциям, выставившим системное прерывание. Например, если ситуация была вызвана прерыванием от какого-нибудь устройства, находящийся в ядре обработчик аппаратных системных прерываний передает управление *процедуре обработки прерывания* (Interrupt service routine, ISR), которую драйвер устройства предоставил для устройства, вызвавшего прерывание. Если ситуация создалась из-за вызова системной службы, общий обработчик системных прерываний, связанных с системными службами, передает управление конкретной функции системной службы в исполняющей системе. Ядро также устанавливает обработчики системных прерываний для неожиданных или необрабатываемых им прерываний. Эти обработчики системных прерываний обычно выполняют системную функцию KeBugCheckEx, останавливающую компьютер, когда ядро обнаруживает проблемное или некорректное поведение, которое, если его не предотвратить, может привести к повреждению данных. Более подробно прерывания, исключения и диспетчеризация системных служб рассматриваются в следующих разделах.

Диспетчеризация прерываний

Аппаратно генерируемые прерывания обычно исходят от устройств ввода-вывода, которые должны уведомить процессор о том, что они нуждаются в обслуживании. Устройства, управляемые прерываниями, позволяют операционной системе использовать процессор максимально эффективно, совмещая основную обработку данных с операциями ввода-вывода. Поток запускает передачу ввода-вывода в адрес устройства или от него, а затем может выполнять другую полезную работу, пока устройство не завершит передачу. Когда устройство завершит работу, оно прерывает процессор на свое обслуживание. Как правило, прерываниями управляются устройства указания координат, принтеры, клавиатуры, дисковые приводы и сетевые карты.

Прерывания могут также генерироваться системными программами. Например, ядро может выдать программное прерывание для инициирования диспетчера потока и для асинхронного проникновения в выполнение потока.

Ядро может также запретить прерывания, но делается это крайне редко, только в критических ситуациях, например во время программирования контроллера прерываний или диспетчеризации исключений. Для ответа на прерывания, исходящие от устройств, ядро устанавливает обработчики системных прерываний. Эти обработчики передают управление либо внешней процедуре (ISR), обрабатывающей прерывание, либо внутренней процедуре ядра, реагирующей на прерывание. Для обслуживания прерываний от устройств драйверы устройств предоставляют ISR-процедуры, а ядро предоставляет процедуры, обрабатывающие другие типы прерываний.

В следующих разделах вы узнаете, как оборудование уведомляет процессор о прерываниях, исходящих от устройств, о типах прерываний, поддерживаемых ядром, о том, как драйверы устройств взаимодействуют с ядром (в том, что касается обработки прерываний), о том, какие программные прерывания распознаются ядром, а также об объектах ядра, используемых для их реализации.