ВАЛЕРИЯ КОМИССАРОВА

bhv®

ПРОГРАММИРОВАНИЕ ДРАЙВЕРОВ для Windows

Архитектуры WDM и WDF

Драйверы для всей линейки операционных систем Windows NT, включая Windows Vista

Драйверы для многопроцессорных систем

Драйверы для видеокарты, USB-камеры и др.

PRO

ПРОФЕССИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ

Валерия Комиссарова

ПРОГРАММИРОВАНИЕ ДРАЙВЕРОВ для Windows

Санкт-Петербург
«БХВ-Петербург»
2007

УДК 681.3.06
ББК 32.973.26-018.1
К63

Комиссарова В.

К63 Программирование драйверов для Windows. — СПб.: БХВ-Петербург, 2007. — 256 с.: ил. — (Профессиональное программирование)

ISBN 978-5-9775-0023-4

Книга представляет собой практическое руководство по программированию драйверов для всей линейки операционных систем Windows NT, включая новую ОС Windows Vista. Разбираются важнейшие драйверные архитектуры — традиционная WDM и новая WDF. Излагаются основы теории программирования драйверов для многопроцессорных систем. Показано, как создать простейший драйвер, а также приведены практические примеры написания сложных драйверов для принтера, монитора, видеокарты и USB-камеры.

Для программистов

УДК 681.3.06
ББК 32.973.26-018.1

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Игорь Шишигин*
Зав. редакцией	*Григорий Добин*
Редактор	*Анна Кузьмина*
Компьютерная верстка	*Натальи Смирновой*
Корректор	*Наталия Першакова*
Дизайн серии	*Инны Тачиной*
Оформление обложки	*Елены Беляевой*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 05.03.07.
Формат 70×100$^{1}/_{16}$. Печать офсетная. Усл. печ. л. 20,64.
Тираж 2500 экз. Заказ №
"БХВ-Петербург", 194354, Санкт-Петербург, ул. Есенина, 5Б.

Отпечатано с готовых диапозитивов
в ГУП "Типография "Наука"
199034, Санкт-Петербург, 9 линия, 12

ISBN 978-5-9775-0023-6

Оглавление

Очень, очень хорошему человеку — О. Ф. Литвину

Предисловие

Драйверная концепция — неотъемлемая часть современных операционных систем. Эта концепция — основа взаимодействия системы (пользователя) с какими бы то ни было устройствами (системными/периферийными, реальными/виртуальными и т. д.). К сожалению, даже системные программисты (не говоря уже о прикладных программистах или, вовсе, о рядовых пользователях) далеко не всегда имеют какое-либо представление об этой концепции, о принципах ее работы, о программировании с использованием этой концепции. А, как известно, системное программирование — ключ к пониманию основ IT. Тем более, такой его раздел, как написание драйверов. Поэтому необходимость качественного изучения его — очевидна.

Написание драйверов — достаточно сложная, но, тем не менее, очень интересная и, что немаловажно, актуальная отрасль программирования. Знание особенностей технологий написания драйверов открывает огромное количество возможностей — написание драйверов для устройств, уже не поддерживаемых производителем, для устройств, драйверы к которым еще не написаны, исправление ошибок в драйверах, написание драйверов к различным промышленным устройствам... Список можно продолжать долго.

Большую помощь в деле освоения какого бы то ни было раздела программирования оказывает соответствующая литература. Но, к сожалению, на российском рынке компьютерной литературы остро ощущается нехватка изданий по написанию драйверов для различных операционных систем и платформ (особенно в последнее время). Кроме трех-четырех книг (причем далеко не все из них хотя бы приемлемого качества) — больше ничего нет. В результате этого наблюдения и возникла идея написания подобной книги.

Тема написания драйверов настолько огромна, что в такой маленькой по объему книге невозможно рассказать и половины того, что хотелось бы, и многие темы описаны лишь обзорно. Во-первых, не стоит забывать об операционных системах, отличных от семейства операционных систем Windows, использующихся повсеместно (самая популярная из которых, пожалуй, Linux), написание драйверов для которых осталось за рамками книги. Во-вторых, следует помнить о большом количестве устройств, написание драйверов для которых — очень трудное дело, со своей спецификой, требующее нетривиальных знаний и т. д. Все эти темы мы оставим в надежде на появление в будущем хорошей литературы по вышеуказанным и многим

другим предметам, входящим в обширнейшую область знаний под названием "программирование драйверов".

Актуальность темы программирования драйверов не уменьшается в течение уже долгого времени. Меняются лишь какие-либо драйверные модели (как, например, в случае с WDM на WDF, описанном в этой книге), но смены концепции драйверов как таковой не предвидится еще очень долго. Этому есть свои причины. Концепция драйверов до сих пор жива далеко не только потому, что IT-индустрия просто привыкла к ней. Драйверная концепция обладает рядом неоспоримых преимуществ, которые позволяют ей оставаться "на плаву". Изменение драйверной концепции "тянет" за собой изменение большого количества компонентов (таких как архитектура существующих операционных систем, например), тесно с ней взаимосвязанных (и наоборот). Пока что, повторяю, это не предвидится.

Структура книги

- **Глава 1. Основные понятия разработки драйверов.**

  В данной, вводной главе разъясняются основные понятия и концепции, с которыми сталкивается любой программист драйверов. Прочтение этой главы даст вам возможность уже без особых трудностей воспринимать последующий материал, изобилующий специфическими терминами и понятиями. Также в этой главе дается обзор самых главных и популярных (что важно, заслуженно) инструментов для написания драйверов.

- **Глава 2. Архитектура Windows.**

  В этой главе дается краткий обзор архитектуры операционной системы Windows. Хорошо знать и понимать архитектуру операционной системы, код для которой пишет системный программист, ему совершенно необходимо.

- **Глава 3. Архитектура WDM.**

  Здесь приводится подробное описание драйверной архитектуры WDM, которая долгое время являлась главной технологией и концепцией написания драйверов для ОС Windows. Вам необходимо разбираться в этой технологии, чтобы писать хорошие драйверы, грамотно использующие концепцию, с помощью которой они создаются, для любых операционных систем Windows, за исключением самых последних, использующих новую драйверную модель WDF.

- **Глава 4. Программирование в режиме ядра.**

  Процесс программирования в режиме ядра имеет очень существенные отличия от такового в пользовательском режиме. Фактически вы должны заново научиться программировать. Это не громкая фраза — в режиме

ядра свои "законы" программирования, свое API и т. д. Написание драйверов режима ядра невозможно без знания и понимания этих различий — главные из которых и описываются в этой главе.

- **Глава 5. Структура драйвера.**

  В этой главе показана и объяснена общая структура кода любого драйвера — как простого, так и сложного. Знание и понимание этой структуры помогает писать хорошо оформленный, удобный для чтения, исправления и сопровождения, эффективный код драйвера.

- **Глава 6. Простейший драйвер для Windows.**

  В этой главе подробно описывается весь процесс создания простейшего драйвера с минимальными функциями для ОС Windows NT — от написания самого кода, его компиляция и инсталляции до отладки драйвера. Глава даст вам (ну или, во всяком случае, попытается это сделать) все необходимые знания для осуществления этого процесса и немного больше того.

- **Глава 7. Сложные драйверы для Windows.**

  В этой главе подробно описан процесс написания настоящих сложных драйверов с большим количеством функций для определенных устройств — принтера, монитора и видеокарты, фильтр-драйвера камеры. Написанию драйверов для каждого из вышеперечисленных типов устройств можно посвятить отдельную книгу (пусть и не очень большую). В противовес этому — всего одна глава. Ее задача — объяснить главные принципы, концепции написания "полноценных" драйверов, выполняющих сложную работу с устройством, рассказать о характерных приемах, используемых при этом, дать представление о спектре знаний, необходимых для успешной работы в этой области, и т. д. и т. п.

- **Глава 8. Мультипроцессорная парадигма программирования.**

  Так как в этой книге, помимо всего прочего, рассказывается и о написании многопоточных драйверов, то я считаю необходимым, кроме того, рассказать и объяснить, что такое многопроцессорные системы вообще, какова их архитектура и особенности (как высоко-, так и низкоуровневые), в чем суть новой многопоточной парадигмы программирования и т. д. Эта вводная информация абсолютно необходима для полноценного понимания основ и принципов многопроцессорных систем и написания профессиональных драйверов для них.

- **Глава 9. Написание 64-битных драйверов и драйверов для многопроцессорных систем.**

  В этой главе рассказывается о написании многопоточных и 64-битных драйверов.

❒ **Глава 10. Новая операционная система Microsoft — Windows Vista.**

Прежде чем приступать к рассказу о написании драйверов под новейшую ОС Windows Vista от компании Microsoft, необходимо опять-таки тщательно разобраться в самой системе — в ее новых возможностях, особенностях и т. д. В этом вам поможет материал данной главы. Мы рассмотрим сначала общую перспективу ОС Vista, затем перейдем к изучению более низкоуровневых ее особенностей.

❒ **Глава 11. Windows Driver Foundation (WDF).**

Вместе с новой ОС Vista компания Microsoft, соответственно, выпустила и новую драйверную модель WDF. В этой главе подробно рассказывается об этой новой технологии, без знания которой нельзя писать качественные драйверы под Vista. Вы получите все необходимые для написания драйверов с использованием этой модели знания.

❒ **Глава 12. Написание драйверов в Vista — KMDF.**

В этой главе рассказывается о написании драйверов режима ядра в Vista с использованием KMDF — Kernel-Mode Driver Framework (среда для написания драйверов режима ядра). Концепция "от простого — к сложному", важные детали, примеры кода — все, что нужно для того, чтобы свободно чувствовать себя в новой ОС при написании драйверов режима ядра, уметь легко разбираться в новых знаниях и получать их.

Задача этой и следующей глав — не "изобрести велосипед", а попытаться как-то "скрасить" недостатки имеющейся документации по WDF, которая, на мой взгляд, в настоящий момент является чрезвычайно трудно понятной начинающему программисту. Сделана попытка объяснить неясные моменты в имеющейся документации и исходных кодах и более-менее систематизировать имеющиеся данные.

Отмечу, что задача эта чрезвычайно трудная — документация постоянно развивается, поэтому описывать какие-то мелкие и сложные детали реализации пока, к сожалению, представляется мало возможным — поэтому сделана такая выборка информации, которая является основной; ее изменения достаточно маловероятны.

❒ **Глава 13. Написание драйверов в Vista — UMDF.**

В этой главе рассказывается о написании драйверов пользовательского режима в Vista с помощью UMDF — User-Mode Driver Framework (среда для написания драйверов пользовательского режима). То же, что в предыдущей главе — но для драйверов пользовательского режима. Те же замечания, что и к предыдущей главе.

❒ Приложения.

- **Приложение 1. Краткий словарь терминов.**
- **Приложение 2. Полезные исходные коды из DDK.**

- **Приложение 3. Полезные исходные коды из KMDF.**
- **Приложение 4. Полезные исходные коды из UMDF.**

В приложениях к книге размещены: во-первых, справочная информация (об архитектуре WDM Streaming — знать ее нужно для успешного и осознанного написания драйверов для устройств, работающих с потоками, как, например, камеры) и краткий словарь самых необходимых терминов; а во-вторых, специально подобранные и наиболее полезные (в рамках охвата тем данной книги) исходные коды из DDK, UMDF и KMDF.

❒ **Список полезной литературы.**

Здесь приведен перечень полезной литературы.

Кому адресована эта книга

Данная книга предназначена для всех, кто хочет ознакомиться с азами написания драйверов. В ней освещаются вопросы написания драйверов как под Windows серии NT, так и под новейшую версию Windows — Vista. Текст книги построен по принципу "от простого — к сложному", поэтому вероятность возникновения проблем при написании более сложных драйверов была сведена к минимуму. В конце книги размещена самая необходимая справочная информация, теоретические главы (рассказывающие, например, об архитектуре WDM) чередуются с практическими примерами, — закрепляющими теорию. Все это, вместе взятое, способствует улучшению восприятия материала книги, а также делает ее интересной как для начинающих, так и для искушенных в деле написания драйверов читателей. Тем не менее, минимальные требования к читателю все же есть — знание языка C и хотя бы минимальные знания в области системного программирования (т. е. "начинающий" — предполагается только в деле написания драйверов, а не в программировании вообще).

Отмечу, что книга в наибольшей степени имеет практический характер; поэтому во всех главах предпочтение отдается практическим навыкам (пусть даже пока минимальным), а не теоретическим обоснованиям.

Об авторе

Комиссарова Валерия — обладатель сертификатов MCP, MCSD .NET. Имеет публикации в журналах "Хакер" и "IT-Спец" (бывший "Хакер-Спец"). Автор статей на сайтах **www.xakep.ru** и **www.securitylab.ru**.

Глава 1

Основные понятия разработки драйверов

В этой главе читатель получит минимум информации, необходимой для успешного понимания и изучения дальнейших глав этой книги.

Здесь мы рассмотрим основные понятия и термины, используемые в программировании драйверов, и инструменты, которые чаще всего применяются для написания драйверов.

1.1. Общие понятия

Изучение программирования драйверов — так же, как и изучение чего бы то ни было — нужно начинать с изучения теоретических основ. Так и поступим.

Прежде всего — базовые понятия. Итак, что такое драйвер? *Драйвер* — это часть кода операционной системы, отвечающая за взаимодействие с аппаратурой. В данном контексте слово "аппаратура" имеет самый широкий смысл. Под этим словом можно подразумевать как реальные физические устройства, так и виртуальные или логические. Но это уже подводит нас к вопросу о том, какие вообще бывают драйверы (и, соответственно, для каких устройств), а об этом мы поговорим позднее.

С момента своего появления до сегодняшнего дня драйвер беспрерывно эволюционировал, и процесс этот до сих пор не закончился. Один из моментов эволюции драйвера — это эволюция концепции драйвера, как легко заменяемой части операционной системы. Как отдельный и довольно независимый модуль, драйвер сформировался не сразу. Да и сейчас многие драйверы практически неотделимы от операционной системы. Во многих случаях это приводит к необходимости переустановки системы (ОС

Windows) или пересборки ее (ядра) (в UNIX-системах). Такое же различие есть и между ветками операционной системы Windows: Windows 9*x* и Windows NT. В первом случае процесс работы с драйверами происходит (практически всегда) как с отдельными "кирпичиками", а во втором дела обстоят намного хуже (множество (если не большинство) драйверов "вшито" в ядро).

Список основных общих концепций драйверов в Windows- и UNIX-системах выглядит так:

- способ работы с драйверами как файлами;
- драйвер, как легко заменяемая часть ОС (учитывая сказанное выше);
- существование режима ядра.

Объясню подробнее первый пункт. Способ работы с драйверами как файлами означает, что функции, используемые при взаимодействии с файлами, практически идентичны таковым при взаимодействии с драйверами (имеется в виду лексически): `open`, `close`, `read` и т. д. О режиме ядра я расскажу позднее.

И напоследок стоит отметить (добавить к нашему списку) идентичность механизма IOCTL (Input/Output Control Code, код управления вводом/выводом) — запросов.

Теперь рассмотрим классификацию типов драйверов (замечу, довольно условную) для ОС Windows NT:

- драйверы пользовательского режима (User-Mode Drivers):
  - драйверы виртуальных устройств (Virtual Device Drivers, VDD) — используются для поддержки программ MS-DOS;
  - драйверы принтеров (Printer Drivers);
- драйверы режима ядра (Kernel-Mode Drivers):
  - драйверы файловой системы (File System Drivers) — осуществляют ввод/вывод на локальные и сетевые диски;
  - унаследованные драйверы (Legacy Drivers) — написаны для предыдущих версий Windows NT;
  - драйверы видеоадаптеров (Video Drivers) — реализуют графические операции;
  - драйверы потоковых устройств (Streaming Drivers) — осуществляют ввод/вывод потокового видео и звука;
  - WDM-драйверы (Windows Driver Model, WDM) — поддерживают технологию Plag and Play и управления электропитанием.

Замечу, что в эту классификацию я намеренно не включила драйверы новой драйверной модели WDF, т. к. считаю, что это будет уместнее сделать, когда уже оформятся окончательные версии как модели WDF, так и сопутствующих продуктов.

Конечно, рассмотреть все эти типы драйверов в одной книге мы не сможем. Главное — это дать направление и теоретическую и практическую подготовку, достаточные для дальнейшего самостоятельного освоения темы.

Далее стоит отметить, что драйверы бывают одно- и многоуровневыми. Если драйвер является многоуровневым, то обработка запросов ввода/вывода распределяется между несколькими драйверами, каждый из которых выполняет свою часть работы. Между этими драйверами можно "поставить" любое количество фильтр-драйверов (filter-drivers). Также сейчас необходимо запомнить два термина — вышестоящие (higher-level) и нижестоящие (lower-level) драйверы. При обработке запроса данные идут от вышестоящих драйверов к нижестоящим, а при возврате результатов — наоборот. Ну и, понятно, одноуровневый (monolithic) драйвер просто является противоположностью многоуровневому.

Для технологии Plug and Play существуют три уровня-типа драйверов:

- ❒ шинные драйверы;
- ❒ фильтр-драйверы;
- ❒ функциональные драйверы.

На низшей ступени находится шинный драйвер, выше него — функциональный драйвер. Между и над ними находится определенное количество фильтр-драйверов. Если точнее, то:

1. Над шинным драйвером — фильтр-драйвер шины; эти два драйвера, очевидно, шинные.
2. Нижестоящие фильтр-драйвер устройства и классовый фильтр-драйвер.
3. Затем — собственно функциональный драйвер.
4. И, наконец, вышестоящие фильтр-драйвер устройства и классовый фильтр-драйвер; все драйверы со 2 по настоящий пункт относятся к драйверам устройства.

Напоминаю, что любой неясный вам термин вы можете посмотреть в словаре терминов, находящемся в *приложении 1.*

Упомянем о таком базисном понятии, как уровни запросов прерываний (IRQL).

Как известно, прерывания обрабатываются в соответствии с их приоритетом. В Windows NT используется особая схема прерываний, называемая

уровнями запросов прерываний. Всего уровней IRQL 32, самый низкий — 0 (passive), самый высокий — 31 (high). Прерывания с уровня 0 по 2 (DPC\dispatch) являются программными, а с 3 по 31 — аппаратными. Существуют специальные функции ядра, позволяющие узнать текущий уровень IRQL, а также сменить (понизить или повысить) его. Это довольно непростое, однако, дело, в котором есть множество своих нюансов (с каких уровней какие операции можно производить и т. д.) Но об этом подробнее не сейчас.

После того как мы более или менее разобрались с общими понятиями, мы уже можем приступить к обсуждению каких-то более сложных технологий. В частности, о технологии Plug and Play, которую я упоминала несколькими абзацами выше.

Технология Plug and Play (в условном переводе — "подключи и работай") — это технология, состоящая как из программной, так и из аппаратной поддержки механизма, позволяющего подключать/отключать, настраивать и т. д. применительно к системе все устройства, подключаемые к ней (конечно же, при условии, что подключаемые устройства поддерживают Plug and Play-технологию). В идеале весь этот процесс осуществляет только механизм Plug and Play, и какие-то действия со стороны пользователя вообще не требуются. Для каких-то устройств это так и происходит, для других — проблем, к сожалению, может быть гораздо больше. Кроме того, для успешной работы Plug and Play необходима не только поддержка этой технологии со стороны устройств, но также, конечно, со стороны драйверов и системного ПО.

Какие возможности предоставляет системное ПО (вместе с драйверами), поддерживающее технологию Plug and Play?

- автоматическое распознание подключенных к системе устройств;
- распределение и перераспределение ресурсов (таких как, например, порты ввода/вывода и участки памяти) между запросившими их устройствами;
- загрузка необходимых драйверов;
- предоставление драйверам необходимого интерфейса для взаимодействия с технологией Plug and Play;
- реализация механизма, позволяющего драйверам и приложениям получать информацию касаемо изменений в наборе устройств, подключенных к системе устройств, и совершить необходимые действия.

Главное перечислили. А теперь перейдем к рассмотрению структуры механизма Plug and Play.

Система Plug and Play состоит из двух компонентов, находящихся соответственно в пользовательском режиме и режиме ядра — менеджера Plug and Play пользовательского режима и менеджера Plug and Play "ядерного" режима.

Менеджер Plug and Play режима ядра работает с ОС и драйверами для конфигурирования, управления и обслуживания устройств. Менеджер Plug and Play пользовательского режима же взаимодействует с установочными компонентами пользовательского режима для конфигурирования и установки устройств. Также, при необходимости, менеджер Plug and Play взаимодействует с приложениями.

PnP (сокращенное обозначение Plug and Play) может успешно работать со следующими типами устройств:

- физические устройства;
- виртуальные устройства;
- логические устройства.

Об управлении питанием мы поговорим в *главе 3*, посвященной драйверной архитектуре WDM.

Какие условия драйвер должен выполнить для осуществления полной поддержки Plug and Play?

- наличие функции `DriverEntry`;
- наличие функции `AddDevice`;
- наличие функции `DispatchPnp`;
- наличие функции `DispatchPower`;
- наличие функции `Unload`;
- наличие cat-файла (файла каталога), содержащего сигнатуру WHQL;
- наличие inf-файла для установки драйвера.

Подробнее о технологии Plug and Play, об обработке PnP-запросов, о функциях, перечисленных в этом списке, об inf-файлах и т. д. я расскажу в *главах 3*, *5* и *6*, об архитектуре WDM, о структуре драйвера и, собственно, написании драйверов.

А сейчас сделаем небольшой обзор наиболее распространенных и полезных инструментов, используемых при написании драйверов.

1.2. Инструментарий

Описать и/или упомянуть обо всех утилитах, могущих понадобиться при разработке драйверов, — невозможно. Расскажу только об общих направлениях.

Без чего нельзя обойтись ни в коем случае — это Microsoft DDK (Driver Development Kit). К этому грандиозному пакету прилагается и обширная документация. Ее ценность — вопрос спорный. Но в любом случае хотя бы ознакомиться с первоисточником информации по написанию драйверов для Windows — обязательно. В принципе, можно компилировать драйверы и в Visual Studio, но для этого необходимо трудно и долго исправлять sln- и vcproj-файлы проектов для того, чтобы код вашего драйвера нормально компилировался. В любом случае исходные коды придется писать в Visual Studio, т. к. в DDK не входит полноценная интегрированная среда разработки (Integrated Development Environment, IDE). Есть пакеты разработки драйверов и от третьих фирм: WinDriver или NuMega Driver Studio, например. Но у них есть отличия базиса функций Microsoft (порой довольно большие) и масса других мелких неудобств. Так что DDK — лучший вариант. Для написания драйверов с использованием новейших технологий и нововведений Microsoft — априори KMDF и UMDF. Если же вы хотите писать драйверы исключительно на ассемблере, вам подойдет KmdKit (KernelMode Driver DevelopmentKit) для MASM32. Правда, этот пакет только для Windows 2000/XP.

Теперь можно поговорить о сторонних утилитах. Некоторые уже включены в стандартную поставку Windows: например, редактор реестра. Но их в любом случае не хватит, и многие программы нужно будет инсталлировать отдельно. Огромное количество таких программ создали патриархи системного программирования под Windows: Марк Руссинович, Гарри Нэббет, Свен Шрайбер и т. д. Марк Руссинович создал много полезных утилит: RegMon (рис. 1.1), FileMon (рис. 1.2) (мониторы обращений к реестру и файлам соответственно), WinObj (рис. 1.3) (средство просмотра каталогов имен объектов), DebugView (рис. 1.4), DebugPrint (программы просмотра, сохранения и т. д. отладочных сообщений) и проч., и проч. Все эти утилиты и огромное количество других можно найти на знаменитом сайте Руссиновича **http://www.sysinternals.com/**.

На диске, прилагающемся к известной книге "Недокументированные возможности Windows 2000" Свена Шрайбера [4], есть замечательные утилиты w2k_svc, -_sym, -_mem, позволяющие просматривать установленные драйверы, приложения и службы, работающие в режиме ядра, делать дамп памяти и т. д. Все эти утилиты, а также другие программы с диска, прилагающегося к книге, можно скачать с **http://www.orgon.com/w2k_internals/cd.html**.

Напоследок нельзя не упомянуть такие хорошие программы, как PE Explorer (рис. 1.5), PE Browse Professional Interactive (рис. 1.6), OllyDbg (рис. 1.7, 1.8), и такие незаменимые, как дизассемблер IDA (рис. 1.9, 1.10) и лучший отладчик SoftICE (рис. 1.11).

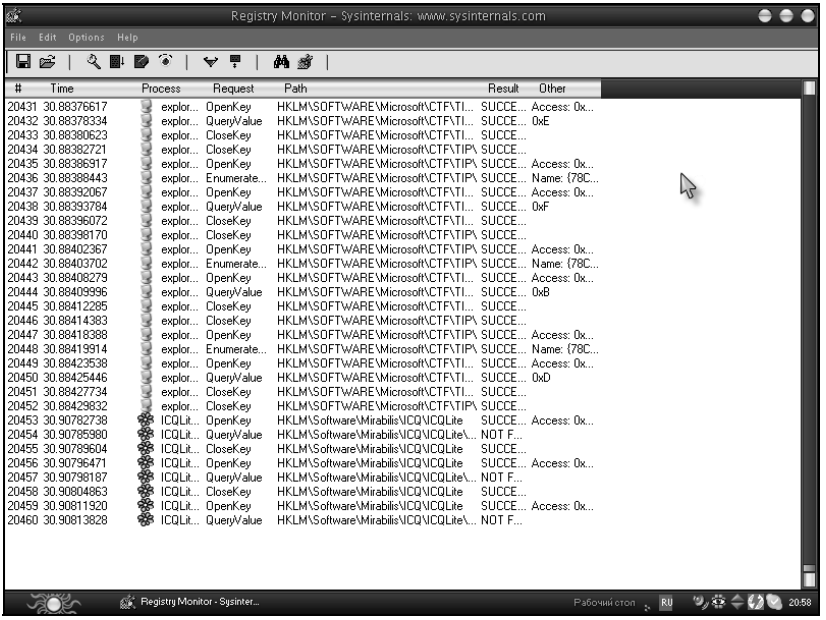

Рис. 1.1. Интерфейс программы RegMon

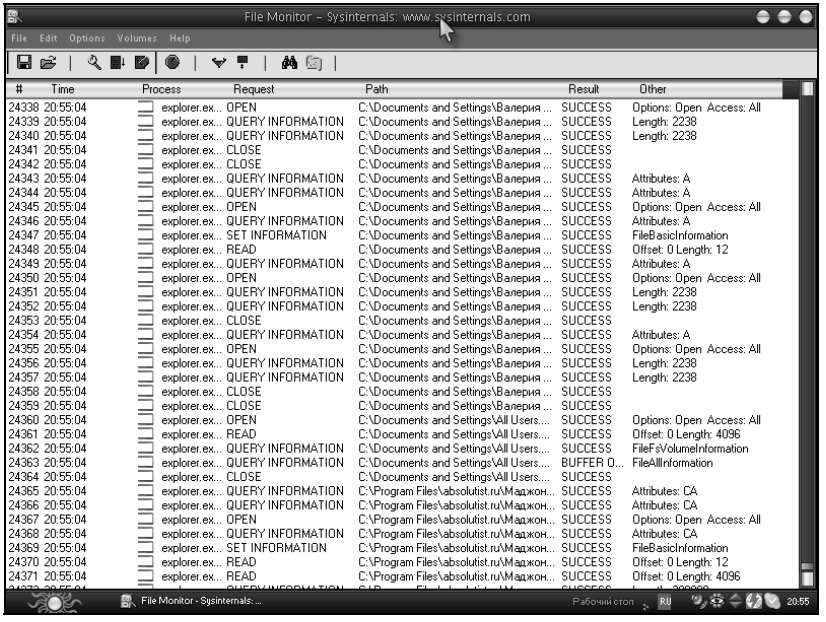

Рис. 1.2. Интерфейс программы FileMon

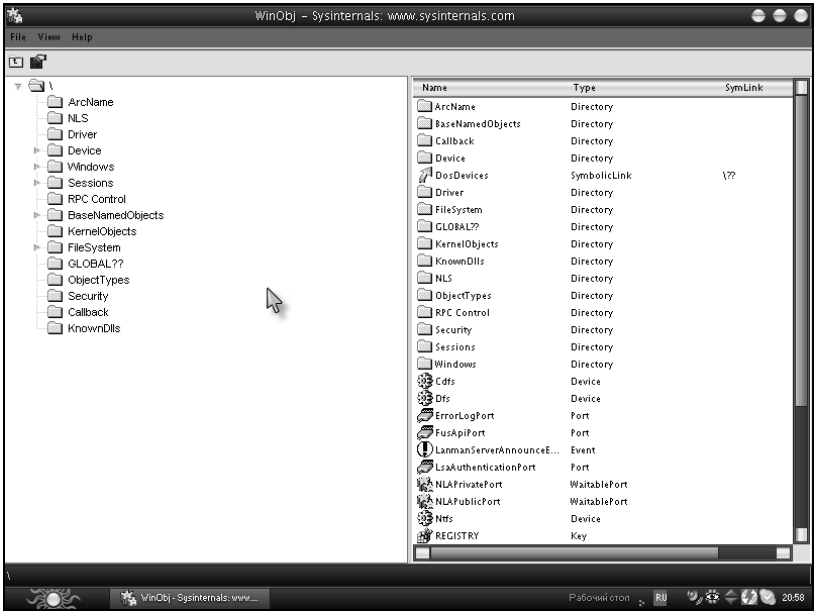

Рис. 1.3. Интерфейс программы WinObj

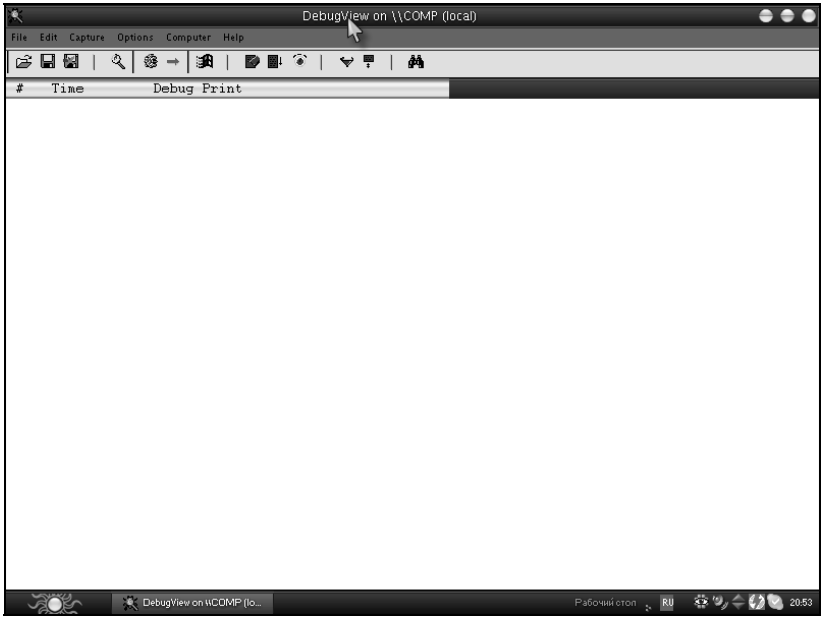

Рис. 1.4. Интерфейс программы DebugView

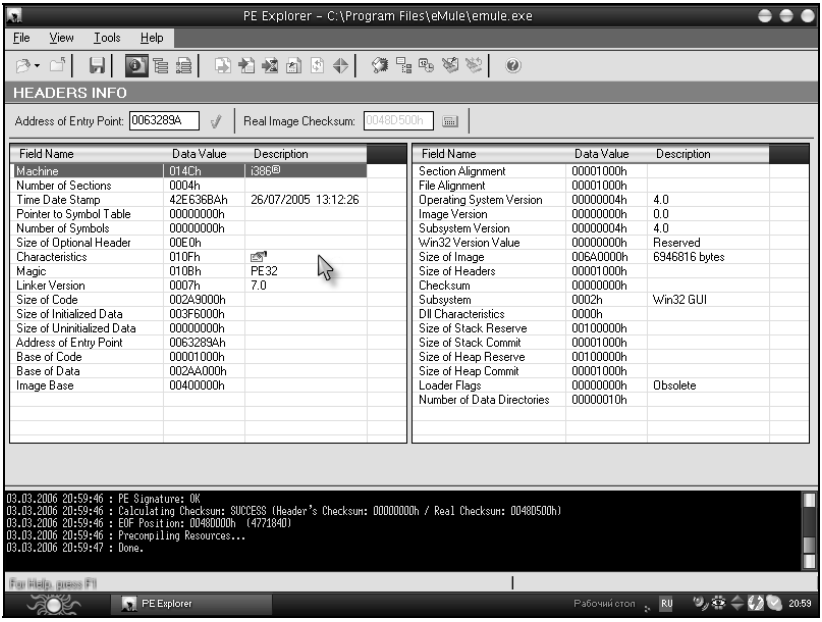

Рис. 1.5. Интерфейс программы PE Explorer

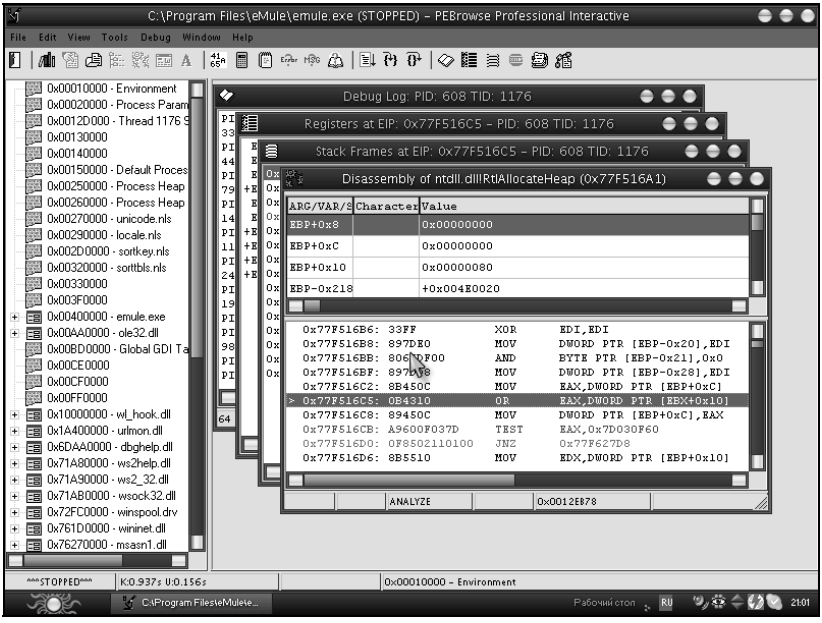

Рис. 1.6. Интерфейс PE Browse Professional Interactive

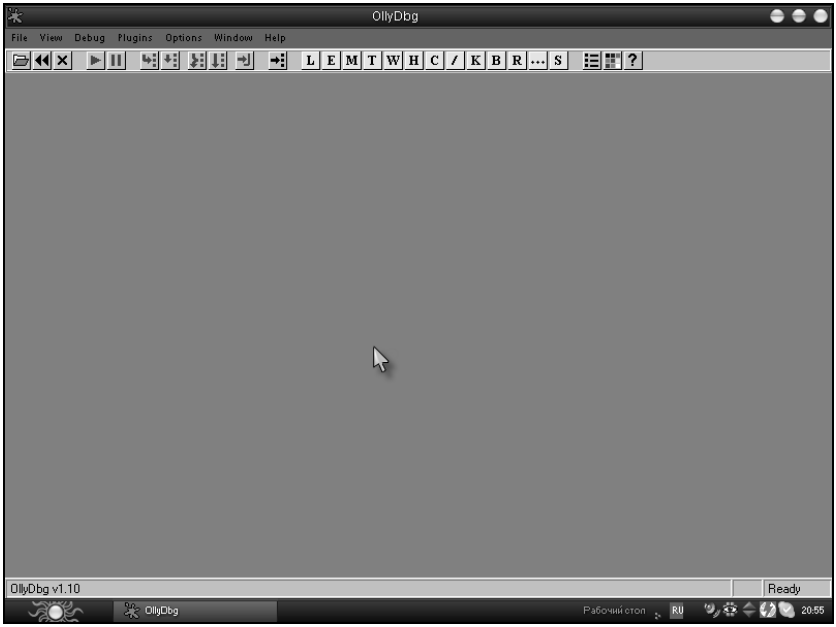

Рис. 1.7. Интерфейс программы OllyDbg при запуске

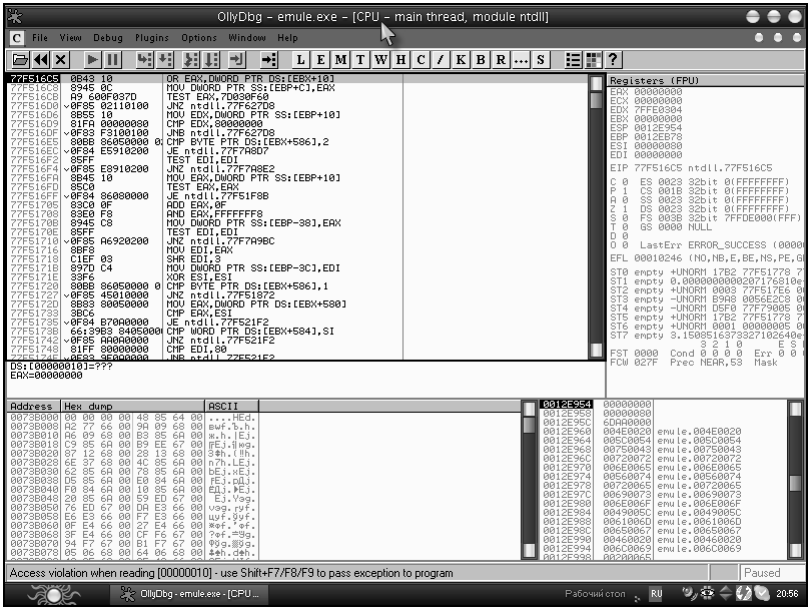

Рис. 1.8. Интерфейс программы OllyDbg с загруженным файлом

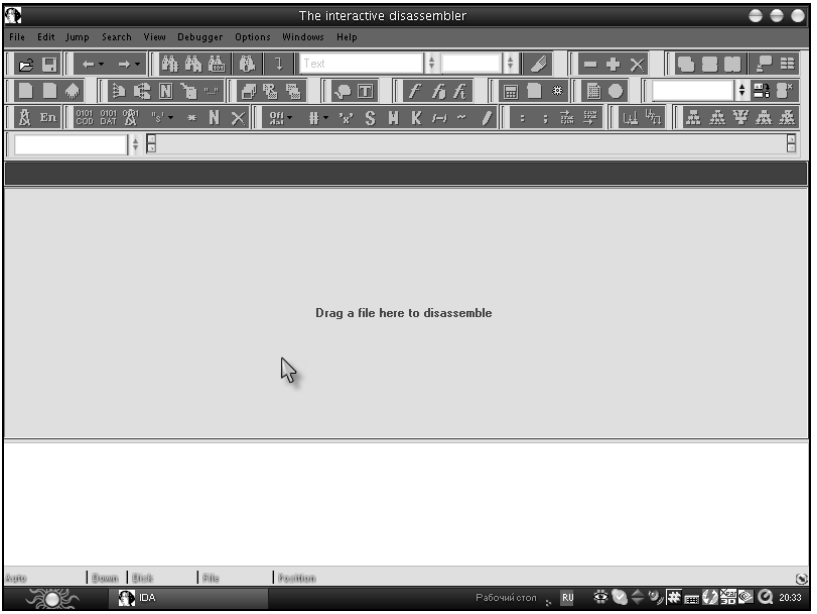

Рис. 1.9. Интерфейс программы IDA Professional при запуске

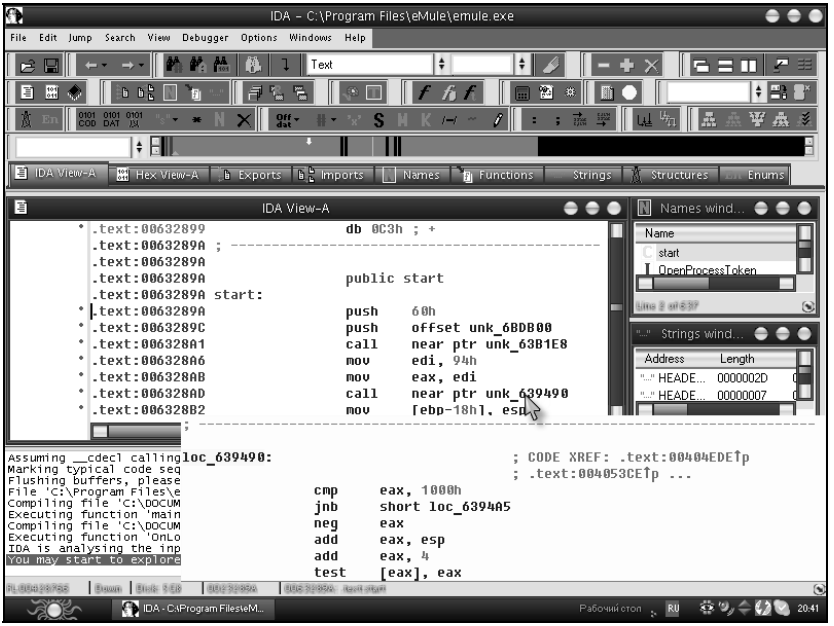

Рис. 1.10. Интерфейс программы IDA Professional с загруженным файлом

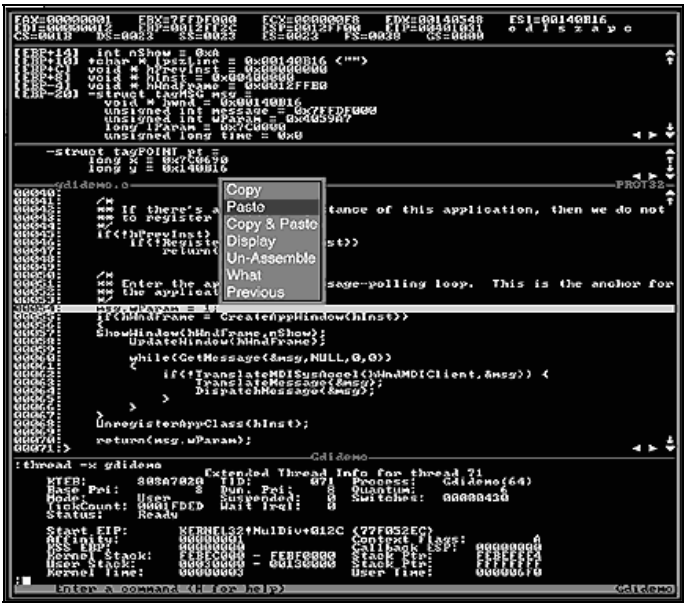

Рис. 1.11. Интерфейс программы SoftICE

* * *

Пожалуй, вводной информации пока достаточно. Перейдем к рассмотрению архитектуры Windows.

Глава 2

Архитектура Windows

В этой главе мы изучим наиболее важные и интересные особенности архитектуры ОС Windows.

Вот главные особенности операционной системы Windows семейства NT:

- ❒ модель измененного микроядра;
- ❒ возможность эмуляции нескольких ОС (наличие различных подсистем);
- ❒ многопоточность;
- ❒ интегрированная поддержка сети.

Наиболее важные из этих пунктов мы рассмотрим в этой главе (или же с отсылкой на другие).

Поговорим об уровнях разграничения привилегий, как об одном из важнейших моментов в архитектуре Windows NT. Я уже упоминала об User mode и Kernel mode. Эти два понятия тесно связаны с так называемыми кольцами. Их (колец) в Windows всего четыре: Ring 3, 2, 1 и 0. Ring 3 — наименее привилегированное кольцо, в котором есть множество ограничений по работе с устройствами, памятью и т. д. Например, в третьем кольце приложения не могут видеть адресное пространство других приложений без особого на то разрешения, выполнять привилегированные команды процессора, напрямую обращаться к оборудованию и т. д. В третьем кольце находится User mode. Kernel mode находится в нулевом кольце — наивысшем уровне привилегий. В этом кольце можно делать все: без всяких ограничений работать с системными данными и кодом, напрямую или через HAL обращаться к оборудованию... Вообще, в Kernel mode можно делать все, чего нельзя в User mode, и еще чуть-чуть. Процессоры Intel *x*86 поддерживают четыре уровня привилегий (четыре кольца), но Windows использует только два — 0 и 3. Понятно, что эти так называемые кольца определяются, прежде

всего, процессором (его аппаратными средствами). Поговорим чуть-чуть подробнее о режиме ядра.

Режим ядра (защищенный режим — по-другому) — это основной режим работы процессора (32-разрядного). Вот главные механизмы, реализуемые режимом ядра:

- механизм защиты памяти и ввода/вывода, состоящий из 4 уровней;
- механизм переключения задач;
- особая организация памяти. При этой организации памяти используются два различных способа ее преобразования: разбивка на страницы и сегментация;
- механизм защиты из четырех уровней — это уже упоминавшиеся выше 4 кольца.

Что такое переключение задач? Любая задача имеет состояние — иными словами (и с низкоуровневой точки зрения) состояние всех регистров процессора, с ней связанных (попросту — совокупность их значений). И состояние каждой задачи может быть сохранено. Где? Для этого есть специальные сегменты — сегменты состояния задач. Вот теперь и перейдем к разговору о двух механизмах преобразования памяти: сегментации и разбивке на страницы (страничная память, paging). Сначала о сегментации.

Что такое *сегмент*? Это отдельный блок общего пространства памяти. Максимальный размер сегмента — 4 Гбайт. Максимальное количество сегментов — 8192. Естественно, все эти цифры верны, только принимая во внимание использование 32-разрядной адресации.

Сегмент описывается особой структурой — *дескриптором*, размером в 8 байтов. В дескрипторе сегмента, в том числе, содержится информация о назначенных сегменту правах доступа (чтение, запись, чтение/запись) и назначенном уровне привилегий.

Сегментация обеспечивает неплохую защиту данных. Этому способствуют следующие ее особенности:

- исключается нарушение прав доступа;
- исключается обращение к сегменту без наличия нужного уровня привилегий;
- исключается обращение к элементам, находящимся за пределами сегмента (ошибочная адресация).

Ну, все, общее представление о сегментации получили. Перейдем к рассмотрению *страничного способа организации памяти*.

Главное то, что страничная организация памяти помогает использовать большее количество памяти, чем сегментация. Базируется она также на 32-

разрядной адресации, но в качестве базового объекта использует отдельный блок памяти размером 4 Кбайт.

Теперь вернемся к нашему самому первому списку и поговорим сначала о микроядре.

Итак, в чем заключается концепция микроядра? Есть программная база (очень маленькая), реализующая основные системные функции (примитивы). Это и есть *микроядро*, которое находится, конечно же, в привилегированном режиме. Все остальные компоненты ОС выполняются уже как отдельные системные процессы (не входящие в микроядро).

В чем плюсы и минусы использования такой технологии? Очевидные плюсы — легкость изменения и обновления всех компонентов ОС, выполненных в виде отдельных от микроядра системных процессов (т. к. эти изменения не затрагивают микроядро; также и наоборот). Главное, чтобы измененные компоненты при необходимости (если нет необходимости модифицировать и ядро тоже) экспортировали прежний интерфейс. Кроме того, это — в очень большой степени — залог устойчивости системы; если какие-либо компоненты ОС (отделенные от ядра) "упадут", то микроядро сможет сделать все возможное, чтобы без каких-либо сбоев в работе ОС перезагрузить эти компоненты.

При всех больших достоинствах использования архитектуры микроядра, конечно же, есть в этом и недостатки. Главный — низкая производительность архитектуры, использующей микроядро. Но для Windows NT в большой степени такой проблемы не существует, т. к. данная ОС использует измененный вариант этой архитектуры — архитектуру, использующую модифицированное микроядро.

Чем таким особенным отличается эта архитектура от обычной архитектуры микроядра? Тем, что теперь из "ядерного" режима в пользовательский перенесен целый набор подсистем, находящиеся в котором подсистемы (прошу прощения за повторение) делятся на два класса — подсистемы окружения и неотделимые (неделимые) подсистемы. Подробнее о подсистемах мы поговорим немного ниже.

Как работает при такой архитектуре прикладная программа? Прикладная программа работает с интерфейсом программирования, предоставляемым ей нужной подсистемой. Но, при необходимости, прикладная программа может использовать и свой интерфейс программирования.

Итак, как обстоят дела с пользовательским режимом при использовании этой архитектуры, мы более или менее разобрались. Что же находится в режиме ядра? В режиме ядра работает NT Executive (исполняющая система NT). Из чего она состоит? Из комплекта подсистем, микроядра и HAL. На-

бор подсистем и микроядро находятся в файле ntoskrnl.exe. HAL же находится (как можно интуитивно догадаться) в файле hal.dll.

А теперь рассмотрим архитектуру Windows NT (рис. 2.1).

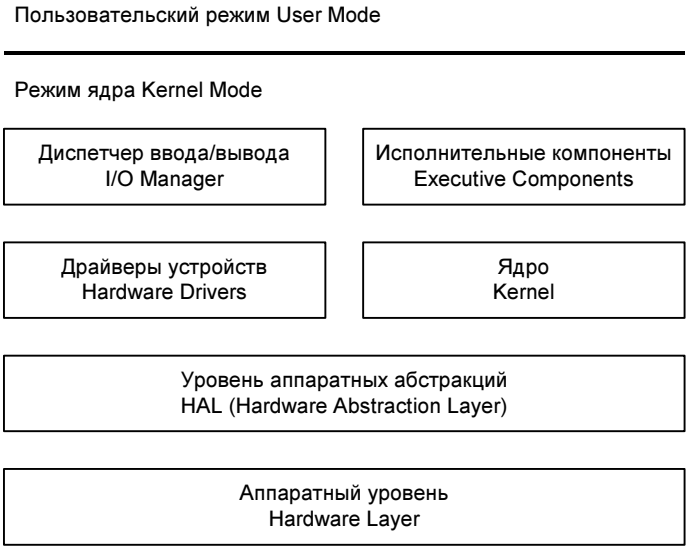

Рис. 2.1. Схема архитектуры Windows

Разберем эту схему поподробнее. С пользовательским режимом все понятно. В Kernel mode самый низкий уровень — аппаратный. Дальше идет HAL, выше — диспетчер ввода/вывода и драйверы устройств в одной связке, а также ядро вместе с исполнительными компонентами. Поподробнее поговорим об исполнительных компонентах (executive components). Что они дают? Прежде всего, они приносят пользу ядру. Как вы уже наверняка уяснили себе по схеме, ядро отделено от исполнительных компонентов. Возникает вопрос: почему? Просто на ядре оставили только одну задачу: простое управление потоками, а все остальные задачи (управление доступом, памятью для процессов и т. д.) берут на себя исполнительные компоненты. Они реализованы по модульной схеме, но несколько компонентов ее (схему) не поддерживают. Такая концепция имеет свои преимущества: таким образом, облегчается расширяемость системы.

Перечислю наиболее важные исполнительные компоненты:

- System Service Interface (Интерфейс системных служб);
- Configuration Manager (Менеджер конфигурирования);

- I/O Manager (Диспетчер ввода/вывода, ДВВ);
- Virtual Machine Manager, VMM (Менеджер виртуальных машин);
- Local Procedure Call, LPC (Локальный процедурный вызов);
- Process Manager (Диспетчер процессов);
- Object Manager (Менеджер объектов)

Вкратце расскажу о предназначении некоторых (наиболее важных/интересных) из них. System Service Interface дает приложениям пользовательского уровня возможность безопасно вызывать процедуры режима ядра. Local Procedure Call реализует механизм локальных вызовов между процессами на одном компьютере. Configuration Manager создает и хранит в единой базе данных (системном реестре) модель всего доступного аппаратного обеспечения и установленного программного обеспечения. Назначение диспетчеров процессов и ввода/вывода, а также менеджера объектов, я думаю, в пояснении не нуждается. Менеджер виртуальной памяти управляет выделением памяти в куче для кода режима ядра, выделением памяти для пользовательских приложений, виртуализацией запросов (создание иллюзии наличия свободной памяти путем выделения страниц на жестком диске (paging))... Одним словом, управляет памятью (от имени операционной системы).

Отложим пока в сторону наш главный список и отметим такое важное понятие, как в архитектуре ОС, так и в программировании вообще — об API.

API (Application Programming Interface) — это интерфейс прикладного программирования. Он позволяет обращаться прикладным программам к системным сервисам через их специальные абстракции.

API-интерфейсов несколько; таким образом, в Windows-системах присутствуют несколько подсистем.

- Подсистема Win32. Она отвечает за графический интерфейс пользователя, за обеспечение работоспособности Win32 API и за консольный ввод/вывод. Каждой реализуемой задаче соответствуют и свои функции: функции, отвечающие за графический интерфейс, за консольный ввод/вывод (GDI-функции), функции управления потоками, файлами и т. д.
- Подсистема VDM (Virtual DOS Machine, виртуальная DOS-машина). Задача подсистемы VDM (виртуальной DOS-машины) — эмулировать внутри ОС Windows NT для соответствующих приложений операционную систему MS-DOS.
- Подсистема POSIX (обеспечивает совместимость UNIX-программ). Подсистема POSIX делает то же самое, но только для POSIX-совместимых программ (только для них она, естественно, эмулирует не MS-DOS, а среду POSIX).

- Подсистема WOW (Windows on Windows). WOW 16 обеспечивает совместимость 32-разрядной системы с 16-битными приложениями. В 64-разрядных системах есть подсистема WOW 32, которая обеспечивает аналогичную поддержку 32-битных приложений.
- Подсистема OS/2. Обеспечивает совместимость с OS/2-приложениями.

Теперь, как я и обещала, поговорим подробнее о подсистемах (убежать от них уже просто некуда). Что вообще такое подсистема? *Подсистема* — это сервис, реализующий тот или иной комплект API, соответствующий той или иной операционной системе (поэтому есть подсистемы UNIX, DOS и т. д.). Главная подсистема — это, конечно, реализующая API-интерфейс самой ОС Windows — Win32. Подсистемы в NT основаны на клиент-серверной архитектуре.

Все остальные подсистемы (отличные от Win32), несмотря на то, что предоставляют свои собственные системы API, для работы с пользователем, конечно (такой, например, как отображение ему результатов), в любом случае используют подсистему Win32.

Как соотносятся ПО с подсистемами? Любая программа (так же, как и любой модуль) может работать только с одной из подсистем (как вариант — вообще ни с одной из них).

Естественно, в силу своей важности подсистема Win32 заслуживает того, чтобы поговорить о ней подробней. Так и поступим.

Подсистема Win32 состоит из двух кирпичиков — подсистемы среды и драйверов режима ядра. Подсистема среды отвечает за консольные окна, создание процессов, потоков и проч. Драйвер режима ядра поддерживает тоже множество вещей: и менеджер окон, и GDI, и т. д. Естественно, что все эти компоненты теснейшим образом связаны между собой.

И еще одна важная вещь — это NTDLL.DLL. Этот файл содержит особую систему, поддерживающую DLL-библиотеки. Поддерживает два типа функций: одна группа реализует интерфейс доступа к NT-службам, вторая группа — функции поддержки (APC, диспетчер исключений и т. д.).

* * *

Ну, все. Думаю, этого об архитектуре Windows достаточно. Перейдем к обсуждению архитектуры драйверной модели Windows — WDM.

Глава 3

Архитектура WDM

В предыдущих главах мы уже получили начальную теоретическую подготовку: разобрали основные термины и понятия, используемые в области программирования драйверов, поговорили об архитектуре Windows и о многих других вещах. Теперь настал момент, когда мы вплотную подошли к написанию драйверов. И начнем мы изучение этого процесса с WDM.

Что такое WDM? WDM (Windows Driver Model) — это драйверная модель от Microsoft для ОС Windows, пришедшая на смену предыдущей среде написания драйверов для ОС Windows — VxD (virtual device driver).

WDM в настоящий момент — одна из важнейших концепций в написании драйверов, разбираться в которой совершенно необходимо любому маломальски профессиональному разработчику драйверов под Windows. Поэтому не поговорить о ней и не разобраться в ней я считаю кощунством. Разберем архитектуру WDM, а в ее контексте изучим основные функции драйвера и их назначение.

Итак, мы уже разобрались, что WDM (Windows Driver Model) — это новая модель драйверов Windows. Ее главные особенности:

- ❒ совместимость на уровне двоичных кодов между драйверами для систем Windows 98 и NT;
- ❒ поддержка управления питанием (power management);
- ❒ поддержка Plug and Play;
- ❒ поддержка "продвинутого" шинного управления (advanced bus management).

Первый и последний пункты в пояснениях не нуждаются. Поговорим подробнее о втором.

Второй пункт — управление питанием. Технология управления питанием дает дополнительные возможности системе и драйверам устройств. Эти воз-