

Московский государственный университет  
приборостроения и информатики

Кафедра "Персональные ЭВМ"

**А. В. Рощин**

**ОРГАНИЗАЦИЯ ВВОДА–ВЫВОДА**

Часть 2

**ДРАЙВЕРЫ ДЛЯ WINDOWS NT**

Москва 2006

УДК 681.3

Организация ввода-вывода. Часть 2. Драйверы для WINDOWS NT.  
Учебное пособие/ А.В.Рощин. – М.: МГУПИ, 2006. – 112 с.: ил.

ISBN

Рекомендовано Ученым Советом МГАПИ в качестве учебного пособия для специальности 2201.

Рецензенты: профессор Зеленко Г.В.  
доцент Туманов М.П.

Предлагаемая работа может рассматриваться как пособие-справочник для студентов, осваивающих основы системного программирования под WINDOWS NT. В пособии рассмотрены особенности работы процессоров 386+ в защищенном режиме. Рассмотрена структура и организация работы операционной системы WINDOWS 2000. В пособии рассмотрены минимальные требования к драйверу устройства и дан пример драйвера виртуального диска, как простейшего примера такого драйвера.

Для успешной работы с этим пособием необходимо знание основ написания драйверов устройств под DOS или другой операционной системой. Необходимы также знания языка Си и ассемблера 386+.

Л ————— без объявл.

ISBN

© А.В.Рощин. 2006.

## СОДЕРЖАНИЕ

ВВЕДЕНИЕ	4
1 ОСОБЕННОСТИ ЗАЩИЩЕННОГО РЕЖИМА i386+	5
1.1 Организация памяти в защищенном режиме	5
1.2 Структуры данных защищенного режима	12
1.3 Привилегии	21
1.4 Защита	25
1.5 Переключение задач	28
1.6 Страничное преобразование	32
1.7 Резюме	49
1.8 Контрольные вопросы	54
2 СТРУКТУРА WINDOWS 2000	57
2.1 Основные компоненты системы	61
2.2 Подсистемы окружения	64
2.3 Исполнительная система	67
2.4 Ядро	71
2.5 Уровень аппаратных абстракций <i>HAL</i>	74
2.6 Основные компоненты подсистемы ввода-вывода	76
2.7 Диспетчер ввода-вывода	80
2.8 Драйверы устройств	82
2.9 Резюме	85
2.10 Контрольные вопросы	88
3 СОЗДАНИЕ ДРАЙВЕРОВ WINDOWS 2000	91
3.1 Стандартные процедуры	94
3.2 Дополнительные стандартные процедуры	95
3.3 Сервисные системные вызовы	99
3.4 Драйвер виртуального диска	100
Некоторые важные сокращения	110
Литература	111

## *ВВЕДЕНИЕ*

Программирование на уровне ядра операционной системы и, в частности, написание драйверов, требует глубокого знания принципов и механизмов устройства операционной системы.

При написании драйверов режима ядра приходится иметь дело с такими, вроде бы, далекими от программиста вещами, как страничная организация памяти, переключение задач, механизмы защиты и т. д. Так, многие драйверы требуют явного указания своих блоков, которые не должны выгружаться на диск ни при каких условиях.

Предлагаемое учебное пособие призвано дать студентам представление о возможностях и способах написания драйверов для операционной системы Windows 2000.

Для этого в пособии представлены три раздела, связанные с особенностями написания такого типа драйверов.

В первом разделе рассмотрены особенности работы процессоров семейства 386+ в защищенном режиме – механизмы работы с памятью, механизмы защиты и переключения задач.

Во втором разделе рассмотрены особенности организации операционной системы Windows 2000 – основные компоненты системы, подсистема ввода-вывода и модель драйверов.

В третьем разделе рассмотрены конкретные вопросы создания простого драйвера и дан пример драйвера виртуального диска с необходимыми комментариями.

## 1 ОСОБЕННОСТИ ЗАЩИЩЕННОГО РЕЖИМА i386+

### 1.1 Организация памяти в защищенном режиме

При работе процессора i386+ в защищенном режиме используется два метода организации памяти:

- сегментная,
- страничная.

Операционная система Windows NT, о которой будет идти речь в дальнейшем, использует оба этих механизма.

В защищенном режиме рассматриваемого семейства процессоров может быть определено до 8192 ( $2^{13}$ ) сегментов, причем, каждый из них может иметь размер до 4 Гбайт ( $2^{32}$  байтов). К сожалению, семейство операционных систем Windows минимальным образом использует возможности сегментного режима: это семейство использует плоскую 32-разрядную модель памяти с размером линейного адресного пространства 4 Гбайта.

Процесс формирования линейного адреса в защищенном режиме показан на рисунке 1.1. Формат селектора показан на рисунке 1.2.

Каждый сегмент в процессоре описывается 8-байтной структурой данных – *дескриптором* сегмента. Дескриптор определяет положение элемента в памяти, размер занимаемой им области (лимит), его назначение и характеристики защиты. Для указания конкретного сегмента внутри таблицы дескрипторов используется 16-разрядный селектор. Он определяет номер записи внутри таблицы дескрипторов. Младшие 2 бита селектора определяют уровень привилегий режима, который может воспользоваться данным дескриптором. Третий бит определяет тип таблицы дескрипторов – глобальная или локальная. Остальные 13

разрядов селектора и определяют максимально возможное количество сегментов, к которым можно обратиться с помощью селектора.

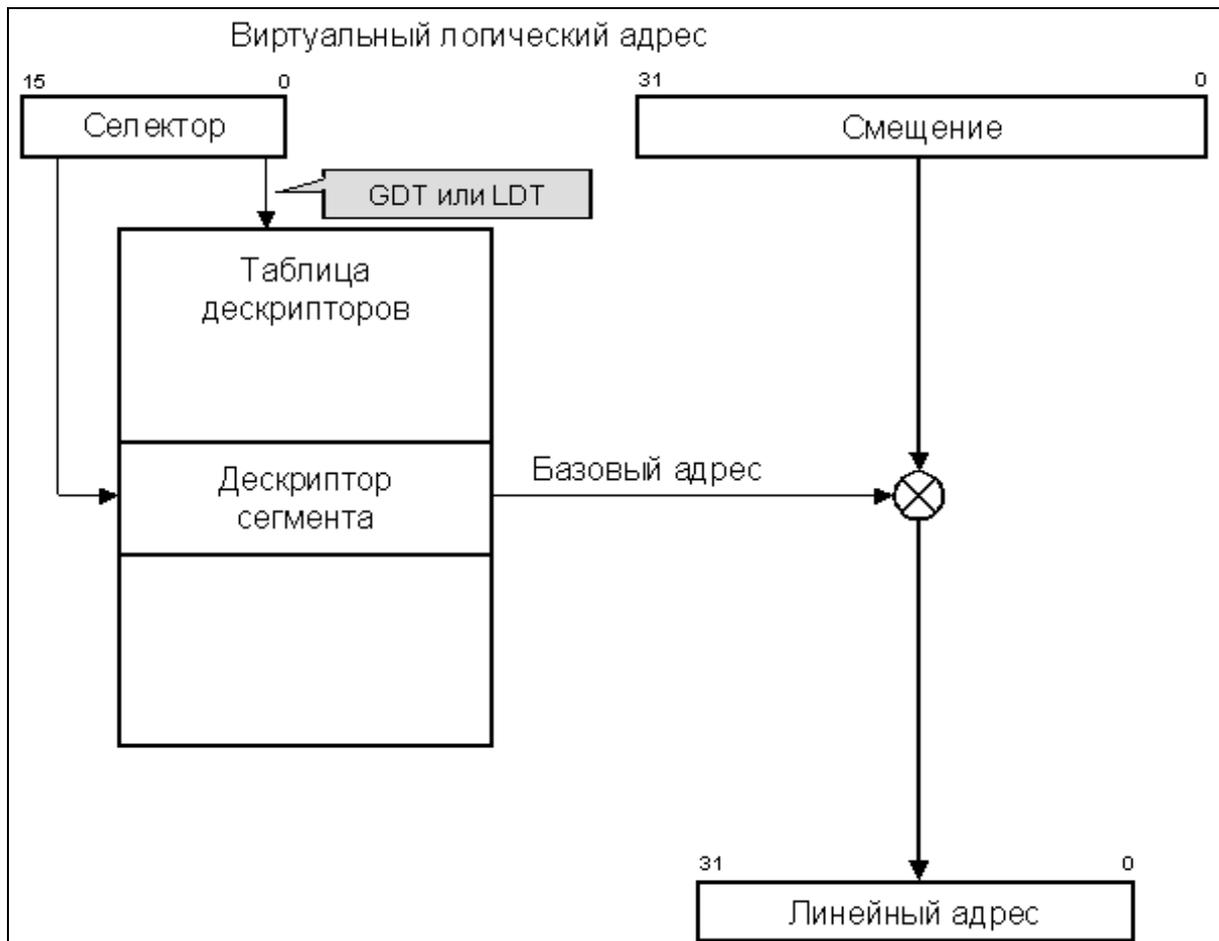


Рисунок 1.1 – Формирование линейного адреса в защищенном режиме

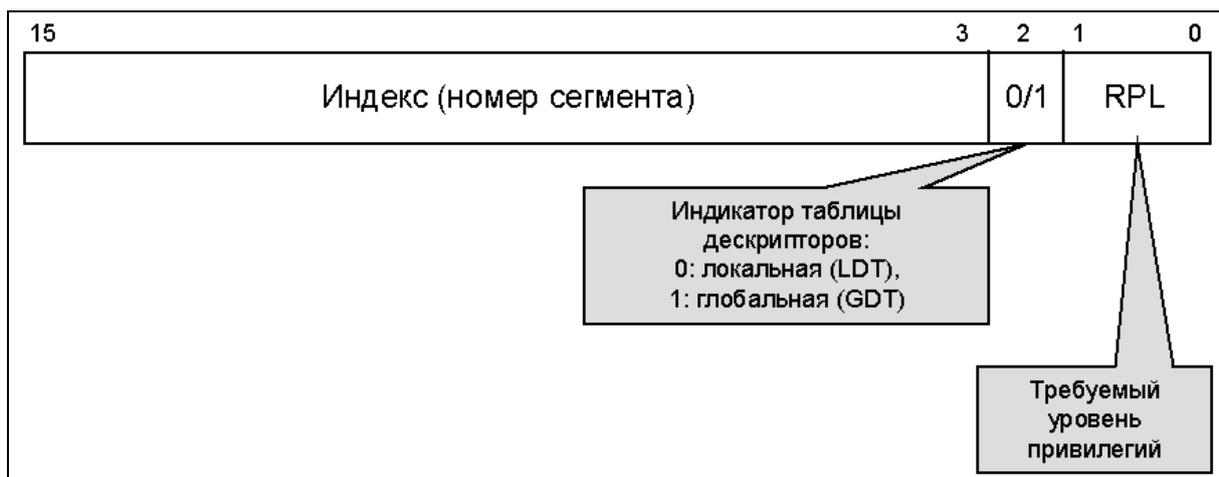


Рисунок 1.2 – Формат селектора сегмента

Дескрипторы расположены в специальной таблице дескрипторов *GDT* – Global Descriptor Table, расположенной в памяти. Механизм работы

с глобальной и локальной таблицами дескрипторов показан на рисунке 1.3.

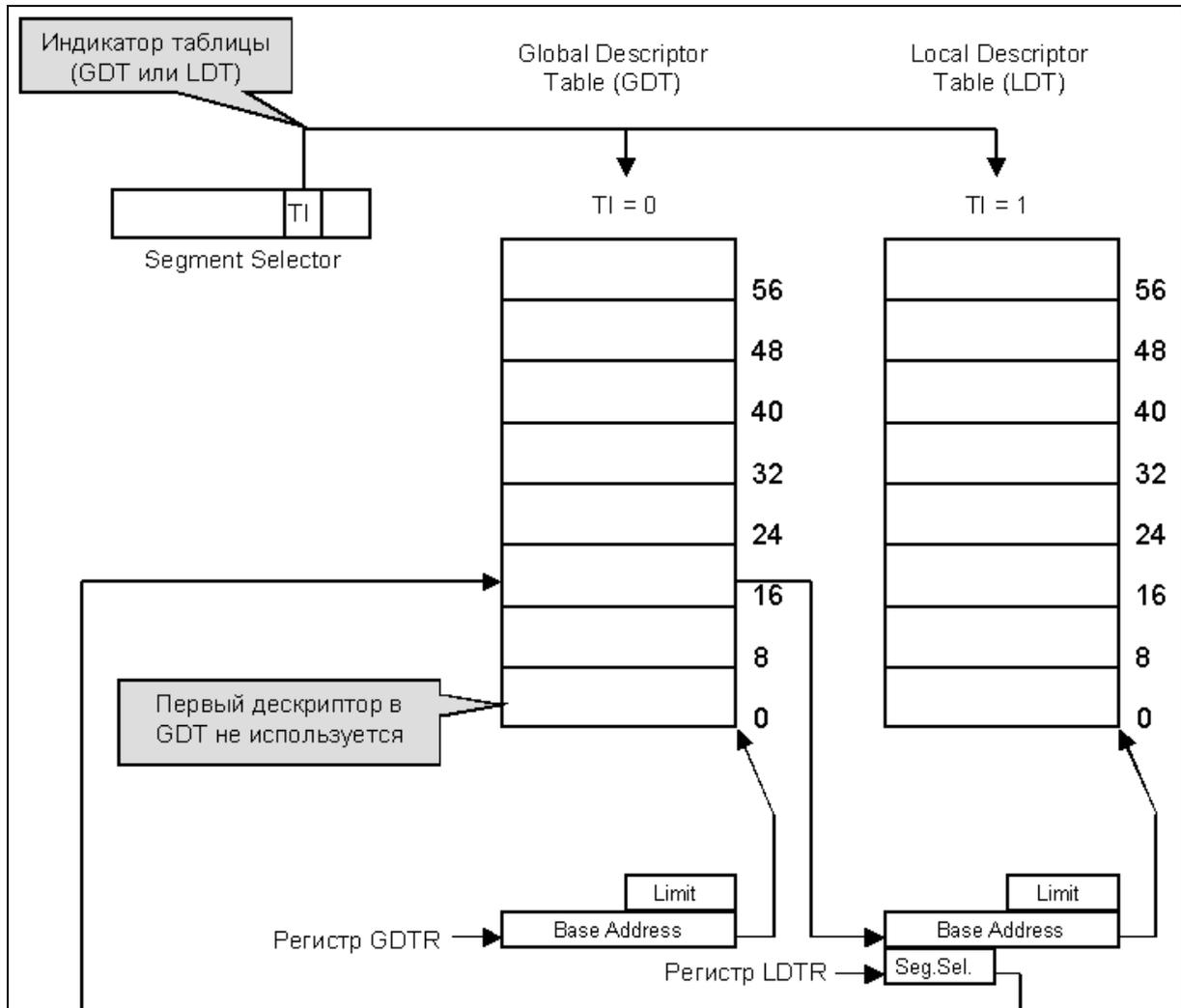


Рисунок 1.3 – Схема работы с глобальной и локальной таблицами дескрипторов

В Windows NT определено 11 селекторов, из которых наиболее важными являются первые четыре:

Таблица 1.1 – Первые 4 сегмента Windows NT

Селектор Hex (bin)	Назначение	База	Предел	DPL	Тип
08 (001000)	Code32	00000000	FFFFFFFF	0	RE
10 (010000)	Data32	00000000	FFFFFFFF	0	RW
1b (011011)	Code32	00000000	FFFFFFFF	3	RE
23 (100011)	Data32	00000000	FFFFFFFF	3	RW

Каждый из этих четырех селекторов позволяет адресовать все 4 Гбайта линейного адресного пространства, причем трансляция производится в одни и те же физические адреса.

Первые два селектора имеют требуемый уровень привилегий  $DPL = 0$  (наивысший приоритет,  $DPL$  – *Descriptor Privilege Level*, уровень привилегий дескриптора) и используются драйверами и системными компонентами для доступа к системному коду, данным и стеку. (Здесь следует отметить, что из четырех возможных уровней привилегий 0 – 4, предоставляемых данным типом процессоров, в операционных системах Windows используются лишь два уровня – 0 и 3.)

Вторые два селектора используются кодом пользовательского режима для доступа к коду, данным и стеку пользовательского же режима. Эти селекторы являются константами для Windows NT.

Преобразование пары *селектор:смещение* для указанных селекторов дает 32-битный линейный адрес, совпадающий со значением смещения виртуального адреса. То есть, в данном случае виртуальный и линейный адреса совпадают.

Наличие в дескрипторе поля, определяющего возможность чтения/записи/исполнения кода, может навести на мысль о том, что именно на этом уровне может быть выполнена защита памяти от неверного использования. Например, код прикладной программы в пользовательском режиме находится в сегменте с селектором 1b. Для этого селектора разрешены только операции чтения и исполнения. То есть, используя селектор 1b, программа не сможет модифицировать свой собственный код. Однако, та же программа, обращаясь к данным или стеку (селектор 23) найдет свой код по тому же смещению, что и в сегменте 1b (так как для всех указанных сегментов трансляция производится в одни и те же физические адреса), но режим доступа в

данном случае позволяет выполнять операции чтения и записи. Таким образом, базовый способ использования сегментации в операционных системах семейства Windows не обеспечивает надлежащей защиты кода.

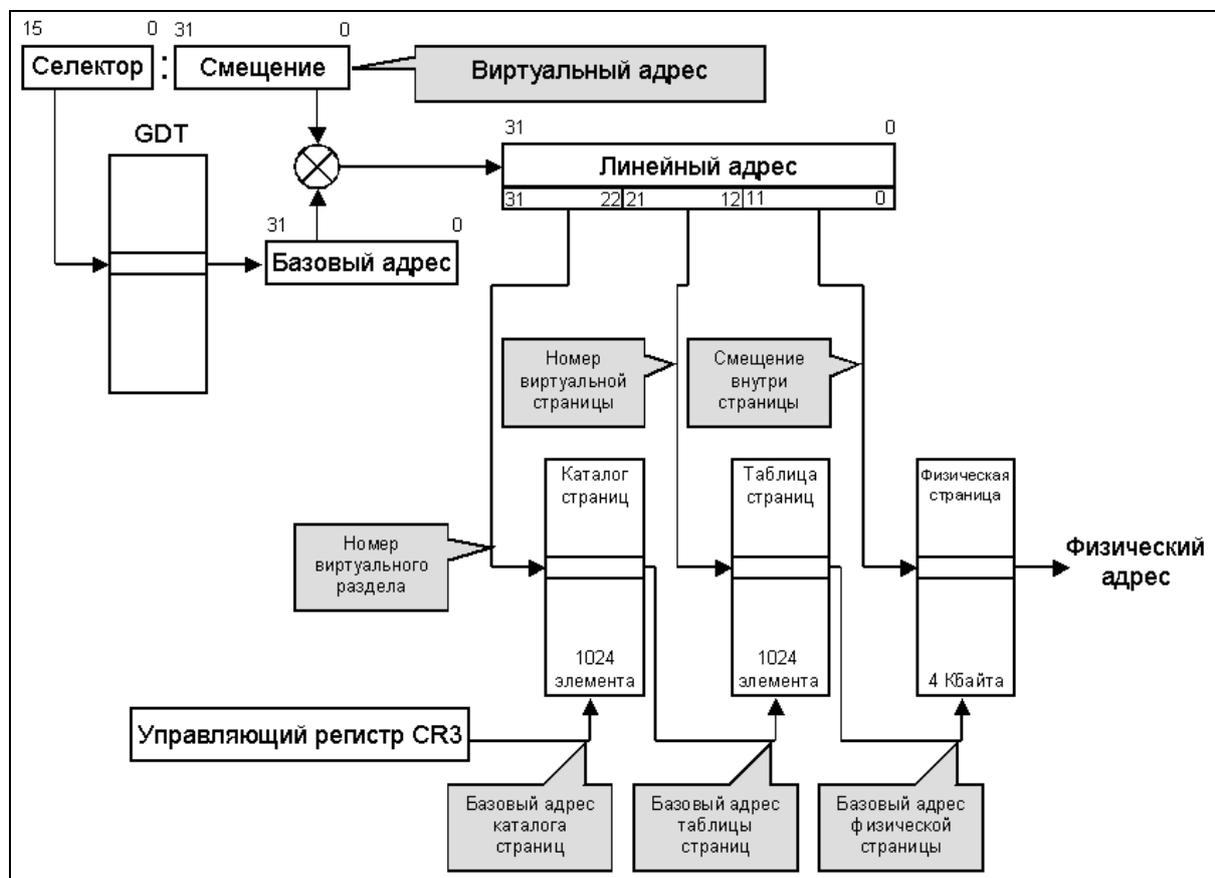


Рисунок 1.4 – Схема преобразования виртуального адреса в физический

Схема формирования линейного адреса и преобразования его в физический показана на рисунке 1.4.

После формирования линейного адреса задействуется механизм страничной организации памяти. Здесь следует отметить следующее:

- каждый контекст памяти (адресное пространство процесса) представлено собственной таблицей трансляции линейного адреса в физический (адрес соответствующей таблицы загружается в управляющий регистр *CR3*),

- каждый элемент таблицы страниц содержит бит, указывающий на возможность доступа к странице из пользовательского режима (все страницы доступны из режима ядра),
- каждый элемент таблицы страниц содержит бит, указывающий на возможность записи в соответствующую страницу памяти.

Не надо забывать, что каждый элемент таблицы страниц содержит также бит, указывающий, представлена ли данная страница в оперативной памяти.

Два бита разрешения доступа и записи формируют следующий набор правил работы со страницами:

- страница всегда может быть прочитана из режима ядра,
- на страницу может быть произведена запись из режима ядра, только если установлен бит разрешения записи,
- страница может быть прочитана из пользовательского режима, только если установлен бит доступа к странице из пользовательского режима,
- на страницу может быть произведена запись из режима пользователя, только если установлены оба бита (разрешения записи и доступ из пользовательского режима),
- если страница может быть прочитана, она может быть исполнена.

По умолчанию страницы памяти с исполняемым кодом не имеют разрешения на запись. Поэтому при попытке использования селектора данных для модификации кода будет сгенерировано исключение.

Для того, чтобы ядро операционной системы (Windows NT) – компоненты самой операционной системы и драйверы – всегда располагались по фиксированным виртуальным адресам, независимо от текущего контекста памяти, в NT осуществляется одинаковая трансляция для верхних двух гигабайт диапазона виртуальных адресов.

Для защиты кода операционной системы соответствующие элементы таблицы трансляции виртуальных адресов в физические помечены как недоступные из пользовательского режима.

Соответственно, диапазон виртуальных адресов 2 – 4 Гб называют *системным адресным пространством* (system address space), а диапазон 0 – 2 Гб – *пользовательским адресным пространством* (user address space).

Здесь следует избегать путаницы между терминами *адресное пространство процесса* – контекст памяти процесса – и *пользовательское адресное пространство*. Пользовательское адресное пространство – это виртуальное адресное пространство, в котором в принципе может находиться любой пользовательский процесс.

## 1.2 Структуры данных защищенного режима

Процессор аппаратно контролирует доступ программ к любому адресу в оперативной памяти. При работе в защищенном режиме адресное пространство процессора делится на

- глобальное – общее для всех задач,
- локальное – отдельное для каждой задачи.

Для получения доступа целевой адрес, к которому хочет получить доступ программа, должен быть описан для программы. Это означает, что участок физической памяти, содержащий нужный адрес, должен быть описан с помощью некоторого *дескриптора сегмента*, который помещается в одну из трех таблиц дескрипторов. Локализация этих таблиц осуществляется с использованием одного из системных регистров:

- *gdt* (Global Descriptor Table Register – регистр таблицы глобальных дескрипторов), имеет размер 48 бит (6 байтов) и содержит 32-битный (биты 16-47) базовый адрес *глобальной таблицы дескрипторов GDT* и 16-битное (биты 0-15) значение *предела*, представляющее собой размер в байтах *GDT*,
- *ldtr* (Local Descriptor Table Register – регистр таблицы локальных дескрипторов), имеет размер 16 бит и содержит *селектор дескриптора локальной таблицы дескрипторов LDT*, являющийся указателем на дескриптор в таблице глобальных дескрипторов *GDT*, который и описывает сегмент, содержащий локальную таблицу дескрипторов *LDT*,
- *idtr* (Interrupt Descriptor Table Register – регистр таблицы дескрипторов прерываний), имеет размер 48 бит (6 байтов) и содержит 32-битный (биты 16-47) базовый адрес *таблицы дескрипторов прерываний IDT* и 16-битное (биты 0-15) значение *предела*, представляющее собой размер в байтах *IDT*.

Кроме того, в процессоре имеется 16-битный регистр задачи *tr* (Task Register), который, как и *ldtr* содержит селектор, то есть, указатель на дескриптор в таблице *GDT*. Этот дескриптор описывает текущий *сегмент состояния задачи TSS* (Task State Segment). Такой сегмент создается для каждой задачи в системе, имеет жестко регламентированную структуру и содержит *контекст* (текущее состояние) задачи. Основное назначение *TSS* – сохранять текущее состояние задачи в момент переключения на другую задачу.

Программе, которая желает использовать данный участок памяти, должен быть сообщен указатель на соответствующий дескриптор в одной из таблиц дескрипторов *GDT* или *LDT*. Указатель на дескриптор сегмента в одной из таблиц *GDT* или *LDT*, в зависимости от функционального назначения описываемого дескриптором участка памяти (сегмента), помещается в один из шести сегментных регистров. Таким образом, в защищенном режиме меняется роль сегментных регистров. Они теперь содержат не адрес, а *селектор* или *индекс* в таблице дескрипторов сегментов. Само же назначение сегментных регистров остается неизменным – они по-прежнему указывают на сегменты команд, данных и стека, однако механизмы, используемые при этом, меняются.

Структура дескриптора сегмента показана на рисунке 1.5. Поля дескриптора, указанные на рисунке 1.5, подробно описаны в таблице 1.2.

При взгляде на рисунок 1.5, может возникнуть ряд вопросов. Первый из них – почему поля, определяющие размер сегмента, не расположены непрерывно, а разбиты на несколько непонятных фрагментов? Ответ прост. Все определяется необходимостью соблюдения совместимости с младшими моделями процессоров.

Впервые защищенный режим появился в процессоре i286. Он имел 24-разрядную адресную шину (мог адресовать до 16 Мбайт оперативной

памяти) и оперировал с сегментами, размером, не превышающим 64 Кбайт (для размера сегмента использовались 16 разрядов).

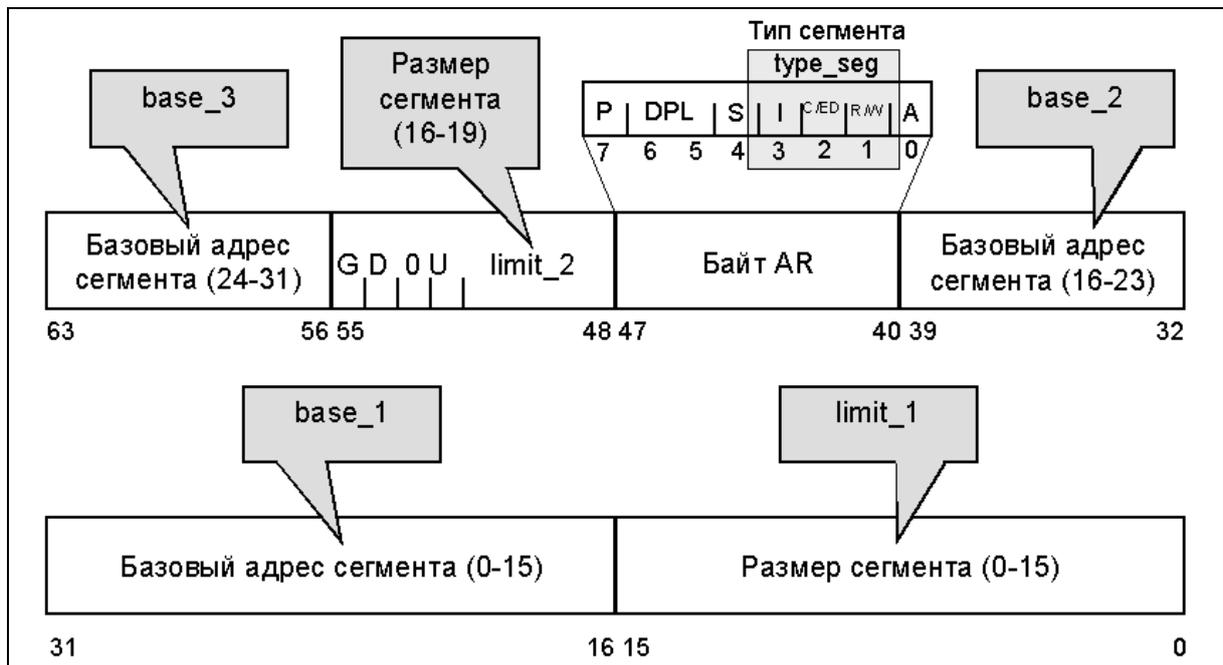


Рисунок 1.5 – Структура дескриптора сегмента

После появления полностью 32-разрядного процессора i386 появилась необходимость в увеличении размера соответствующих полей. Однако из соображений все той же совместимости формат существующего дескриптора не изменили, а просто добавили недостающие поля в дополнительных байтах. Внутри процессора эти поля непрерывны, и неудобства ему не доставляют. Программистам же приходится иметь дело с фрагментированными полями.

Второй вопрос такой: как при поле размера сегмента всего в 20 разрядов (адресуемое пространство – 1 Мбайт) размер описываемого сегмента может достигать 4 Гбайт? Все дело в поле *гранулярности G*. Если это битовое поле имеет значение «0», то размер сегмента в поле обозначается в байтах, и не может превышать 1 Мбайта. Если же  $G = 1$ , то размер сегмента выражается в страницах. Так как размер страницы составляет 4 Кбайта, то 1 М страниц по 4 Кбайта составляет в сумме 4 Гбайта.

Таблица 1.2 – Значение полей дескриптора сегмента

Номер байта	Количество битов в поле	Символическое обозначение	Значение поля
0...1	16	limit_1	Младшие биты (0...15) 20-разрядного поля размера сегмента, определяемого в единицах, соответствующих значению поля гранулярности G
2...4	24	base_1	Биты 0...23 32-разрядной базы сегмента. Она определяет значение линейного адреса начала сегмента в памяти
5	8	AR	Байт, поля которого определяют права доступа к сегменту (таблица 1.3)
6	0...3	limit_2	Старшие биты (16...19) 20-разрядного предела сегмента
6	1	U	Бит пользователя (User). Не имеет специального назначения, может использоваться по усмотрению программиста
6	1	–	0 – бит не используется
6	1	D	Бит разрядности операндов и адресов: 0 – в программе используются 16-разрядные операнды и режимы 16-разрядной адресации, 1 – в программе используются 32-разрядные операнды и режимы 32-разрядной адресации
6	1	G	Бит гранулярности: 0 – размер сегмента равен значению поля limit в байтах, 1 – размер сегмента равен значению поля limit в страницах
7	8	base_2	Биты 24-31 32-разрядной базы сегмента

Таблица 1.3 – Байт AR дескриптора сегмента

Номер бита в байте AR	Символическое обозначение	Значение поля
0	A	Бит доступа (Accessed) к сегменту. Устанавливается аппаратно при обращении к сегменту

Таблица 1.3 (продолжение)

Номер бита в байте AR	Символическое обозначение	Значение поля
1	R	Для сегментов кода – это бит доступа по чтению (Readable). определяет, возможно ли чтение из сегмента кода при осуществлении замены префикса сегмента: 0 – чтение из сегмента запрещено, 1 – чтение из сегмента разрешено
	W	Для сегмента данных – это бит записи: 0 – модификация данных в сегменте запрещена, 1 – модификация данных в сегменте разрешена
2	C	Для сегмента кода – это бит подчинения (Conforming): 0 – обычный сегмент кода, 1 – подчиненный сегмент кода
	ED	Для многозадачного режима определяет особенности смены значения текущего уровня привилегий. Для сегментов данных – это бит расширения вниз (Expand Down), служит для различения сегментов стека и данных, а также определяет трактовку поля limit: 0 – сегмент данных, 1 – сегмент стека
3	I	Бит предназначения (Intending): 0 – сегмент данных или стека, 1 – сегмент кода
4	S	Если $S = 1$ – это бит сегмента (Segment). Для любых сегментов в памяти равен 1. Назначение и порядок использования сегмента уточняется битами C и R. Если $S = 0$ – это бит системный (System). Такое состояние бита S говорит о том, что данный дескриптор описывает специальный системный объект, который может и не быть сегментом в памяти
5...6	DPL	Поле уровня привилегий сегмента (Descriptor Privilege Level). Содержит численное значение в диапазоне от 0 до 3 привилегированности сегмента, описываемого данным дескриптором
7	P	Бит присутствия (Present): 0 – сегмента нет в оперативной памяти в данный момент, 1 – сегмент находится в данный момент в оперативной памяти В дескрипторах сегментов, отсутствующих в оперативной памяти ( $P = 0$ ), для процессора действителен только байт управления доступом. Остальные байты могут использоваться операционной системой по своему усмотрению. (В частности, они могут использоваться для указания места сегмента на внешних носителях.)

В представленной выше таблице биты, входящие в поля типа, рассмотрены по отдельности. Часто для их совместной интерпретации рассматривают комбинации этих битов в целом. В таблице 1.4 показаны некоторые комбинации этих битов.

Таблица 1.4 – Типичные комбинации в поле типа сегмента

Комбинации битов в поле type_seg	Назначение сегмента
000	Сегмент данных только для чтения
001	Сегмент данных с разрешением чтения и записи
010	Не определено
011	Сегмент стека с разрешением чтения и записи
100	Сегмент кода с разрешением только выполнения
101	Сегмент кода с разрешением выполнения и чтения из него
110	Подчиненный сегмент кода с разрешением выполнения
111	Подчиненный сегмент кода с разрешением выполнения и чтения из него

Как видно из содержимого поля типа в байте *AR*, возможны два принципиально разных вида сегментов: данных и кода. Сегмент стека является разновидностью сегмента данных, но с особой трактовкой поля размера сегмента. Это объясняется спецификой использования стека (он растет в направлении от старших адресов памяти к младшим). Ясно, что поле типа ограничивает использование объявленных сегментов. В частности, программные сегменты не могут быть модифицированы без применения специальных мер. Доступ к сегменту данных также может быть ограничен только чтением.

*Системные сегменты* предназначены для хранения локальных таблиц дескрипторов *LDT* и состояния задач *TSS* (Task State Segment). Их дескрипторы определяют базовый адрес, лимит сегмента (1 – 64 Кбайт), права доступа (чтение, чтение/запись, только исполнение кода, исполнение/чтение) и присутствие сегмента в физической памяти (рисунок 1.6)

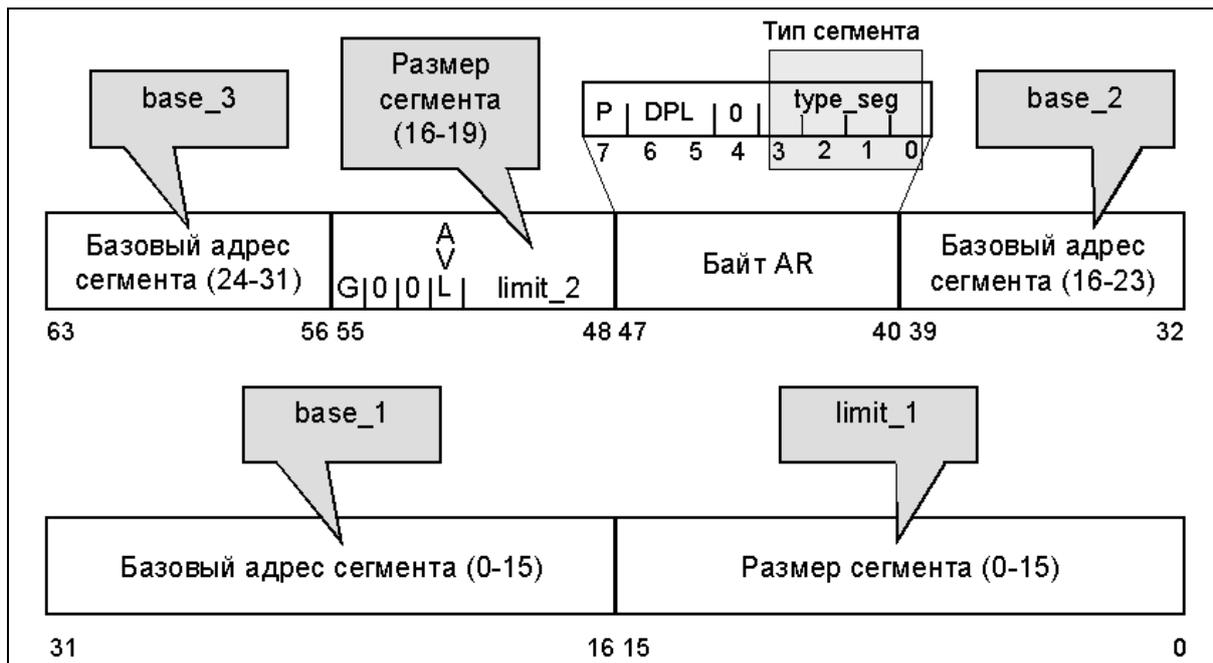


Рисунок 1.6 – Дескриптор системных сегментов

Бит *AVL* доступен для использования операционной системой. В байте управления доступом у этих дескрипторов бит *P* определяет действительность ( $P = 1$ ) или недействительность ( $P = 0$ ) содержимого сегмента. Поле *DPL* в системных сегментах используется только в дескрипторах состояния задач. В дескрипторах локальных таблиц это поле не используется, так как обращение к локальным дескрипторам возможно только по привилегированным командам. Поле *Type\_seg* (1 – 3, 9 – В) определяет тип сегмента:

- 0,8 – недопустимые значения,
- 1 – доступный сегмент состояния задачи 80286,
- 2 – таблица локальных дескрипторов *LDT*,
- 3 – занятый сегмент состояния задачи 80286,
- 9 – доступный сегмент состояния задачи 386+,
- А – не определено,
- В – занятый сегмент состояния задачи 386+.

Непосредственная *межсегментная передача управления* (командами *JMP*, *CALL*, *INT*, *RET*, *IRET*) возможна только к сегментам кода с тем же

уровнем привилегий, либо к подчиненным сегментам, уровень которых выше, *CPL* – Current Privilege Level (при этом *CPL* не изменяется). Для переходов с изменением уровня привилегий используются *вентили* (Gate), называемые также *иллюзами*. Для каждого способа косвенной межсегментной передачи управления имеются соответствующие вентили. Их использование позволяет процессору автоматически осуществлять контроль защиты. *Вентили вызова* (Call Gates) используются для вызова процедур со сменой уровня привилегий. *Вентили задач* (Task Gates) используются для переключения задач. *Вентили прерываний* (Interrupt Gates) и *ловушек* (Trap Gates) определяют процедуры обслуживания прерываний.

Вентили вызова позволяют автоматически копировать заданное число слов из старого стека в новый. Вентили прерываний отличаются от вентиля ловушек только тем, что они запрещают прерывания (сбрасывают флаг *IF*), а вентили ловушек – нет. Для каждого типа вентиля используются соответствующие *дескрипторы вентиля* (Gate Descriptors). Формат дескрипторов вентиля приведен на рисунке 1.7.

В байте управления доступом у этих дескрипторов бит *P* определяет действительность ( $P = 1$ ) или недействительность ( $P = 0$ ) содержимого сегмента. Поле *DPL* задает уровень привилегий. Поле *type\_seg* определяет тип вентиля:

- 4 – вентиль вызова 80286 (Call Gate),
- 5 – вентиль задачи 80286 (Task Gate),
- 6 – вентиль прерывания 80286 (Interrupt Gate),
- 7 – вентиль ловушки 80286 (Trap Gate),
- C – вентиль вызова 386+ (Call Gate),
- D – вентиль задачи 386+ (Task Gate),
- E – вентиль прерывания 386+ (Interrupt Gate),

- F – вентиль ловушки 386+ (Trap Gate).

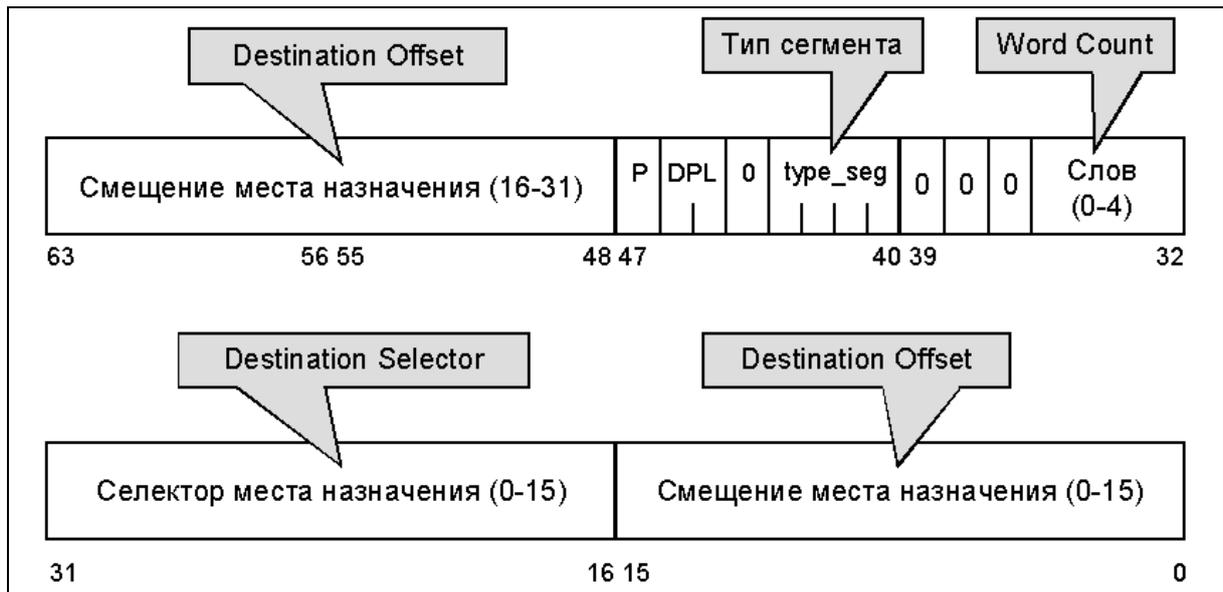


Рисунок 1.7 – Формат дескрипторов вентилей

Поле *Word Count* используется только в вентилях вызова и определяет количество слов из стека вызывающего процесса, автоматически копируемых в стек вызываемой процедуры. Для сегментов 80286 слова 16-битные, для 386+ – 32-битные.

Поле *Destination Selector* для вентилей вызова, прерываний и ловушек задает селектор целевого сегмента кода, а для вентиля задачи – селектор целевого *TSS*.

Поле *Destination Segment* задает смещение (адрес) точки входа в целевом сегменте.

При использовании вентилей может возникнуть исключение *#GP*, которое означает, что селектор указывает на некорректный тип дескриптора. При попытке использования недействительного вентиля ( $P = 0$ ) возникает исключение *#NP* (см. раздел 1.4).

### 1.3 Привилегии

С самого начала следует отметить, что понятие *привилегии* относится только к защищенному режиму процессора. В реальном режиме любая программа выполняется с наивысшим уровнем привилегий, и ей доступны любые области памяти и любые команды.

В защищенном режиме процессор имеет четырехуровневую систему привилегий, которая управляет использованием привилегированных команд и доступом к дескрипторам и соответствующим сегментам. Уровни привилегий нумеруются от 0 до 3, причем значение 0 соответствует высшему уровню привилегий. Привилегии обеспечивают защиту задач, изоляция которых друг от друга обеспечивается локальными таблицами дескрипторов. Каждая часть операционной системы – системные сервисы, обработчики прерываний и др. – работает на своем уровне привилегий. Задачи, дескрипторы и селекторы имеют свои уровни привилегий.

*Привилегии задач* (Task Privilege) оказывают влияние на выполнение инструкций и использование дескрипторов. Текущий уровень привилегий задачи *CPL* (Current Privilege Level) определяется двумя младшими битами регистра *CS* (в котором находится соответствующий селектор кодового сегмента). *CPL* задачи может изменяться только при передаче управления новому сегменту через дескриптор вентиля. Задача начинает выполняться с уровня *CPL*, указанного селектором кодового сегмента внутри *TSS*, когда задача инициируется с помощью операции переключения задач. Задача, выполняемая на нулевом уровне привилегий, имеет доступ ко всем сегментам, описанным в *GDT*, и является самой привилегированной. Задача, выполняемая на уровне привилегий 3, имеет самые ограниченные права доступа. Текущий уровень привилегий может изменяться только при передаче управления через вентили.

Привилегии дескриптора (Descriptor Privilege) задаются полем *DPL* байта управления доступом. *DPL* определяет наибольший номер уровня привилегий (наименьшие привилегии), с которыми возможен доступ к данному дескриптору. Самый защищенный дескриптор имеет  $DPL=0$ , к нему имеют доступ только задачи с  $CPL=0$ . Самый беззащитный дескриптор имеет  $DPL=3$ , его могут использовать задачи с  $CPL=0, 1, 2, 3$ . Это правило применимо ко всем дескрипторам, за исключением дескриптора *LDT*.

*Привилегии селектора* (Selector Privilege) задаются полем *RPL* (Requested Privilege Level) – двумя младшими битами селектора. С помощью *RPL* можно редуцировать *эффективный уровень привилегий EPL* (Effective Privilege Level), который определяется, как максимальное из значение *CPL* и *RPL*. Селектор с  $RPL=0$  не вводит дополнительных ограничений.

*Контроль доступа к сегментам данных* производится при выполнении команд, загружающих селекторы *SS, DS, ES, FS* и *GS*. Команды загрузки *DS, ES, FS* и *GS* должны ссылаться на дескрипторы сегментов данных или сегментов кодов, допускающих чтение. Для получения доступа эффективный уровень привилегий *EPL* должен быть равным или меньшим (по значению) уровня привилегий *DPL* дескриптора. Исключением из этого правила является читаемый подчиненный сегмент кода, который может быть прочитан с любым *CPL*. Если эффективный уровень привилегий не разрешает доступ, или ссылка производится на некорректный тип дескриптора (на дескриптор вентиля или на дескриптор только выполняемого сегмента), генерируется исключение *#GP*. При ссылке на несуществующий дескриптор вырабатывается исключение *#NP*.

*Контроль типов и привилегий при передаче управления* производится при загрузке селектора в регистр *CS*. Тип селектора, на

который ссылается данный селектор, должен соответствовать выполняемой инструкции. Нарушение типа (например, ссылка инструкции *JMP* на вентиль вызова) порождает исключение *#GP*. При передаче управления действуют следующие правила привилегий, нарушение которых также приводит к исключению *#GP*:

- команды *JMP* или *CALL* могут ссылаться либо на подчиненный сегмент кода с *DPL*, большим или равным *CPL*, либо на неподчиненный сегмент с *DPL* равным *CPL*,
- прерывания внутри задачи или вызовы, которые могут изменить уровень привилегий, могут передавать управление кодовому сегменту с уровнем привилегий, равным или большим привилегий *CPL*, только через вентили с тем же или меньшим уровнем привилегий, чем *CPL*,
- инструкции возврата, которые не переключают задачи, могут передать управление только кодовому сегменту с таким же или меньшим уровнем привилегий,
- переключение задач может выполняться с помощью вызова, перехода или прерывания, которые ссылаются на вентиль задач или сегмент состояния задачи (*TSS*) с тем же или меньшим уровнем привилегий.

Смена уровня привилегий, происходящая при передаче управления, автоматически вызывает переопределение стека. Начальное значение указателя стека *SS:SP* для уровня привилегий 0, 1, 2 содержится в *TSS*. При передаче управления по командам *JMP* или *CALL* в *SS:SP* загружается новое значение указателя стека, а старые значения помещаются в новый стек. При возврате на прежний уровень привилегий его стек восстанавливается (как часть инструкции *RET* или *IRET*). Для вызовов подпрограмм с передачей параметров через стек и сменой уровня

привилегий из предыдущего стека в новый копируется фиксированное число слов, заданное в вентиле. Команда межсегментного возврата *RET* с выравниванием указателя стека при возврате корректно восстановит значение предыдущего указателя.

*Привилегии и битовая карта разрешения ввода/вывода* контролируют возможность выполнения операций ввода/вывода и управления флагом разрешения прерываний *IF*. Уровень привилегий ввода/вывода определяется полем *IOPL* (Input/Output Privilege Level) регистра флагов. Значение *IOPL* можно изменять только при *CPL=0*.

При  $CPL \leq IOPL$  на операции ввода/вывода и управление флагом разрешения прерываний *IF* никаких ограничений не накладывается. При  $CPL > IOPL$  попытка ввода/вывода, выполненная с *TSS* класса 80286, вызывает исключение *#GP* (отказ). Если  $CPL > IOPL$ , а с задачей связан *TSS* класса 386+, инструкции ввода/вывода могут выполняться только по адресам портов, для которых установлены нулевые биты в карте разрешения ввода/вывода, имеющейся в *TSS*. Попытки обращения к портам, которым соответствуют единичные биты карты или которые не попали в карту (ее размер может усекаться), вызывают исключение *#GP*.

При  $CPL \leq IOPL$  попытка выполнения команд *CLI* или *STI* вызывает исключение *#GP*. Неявное управление флагом разрешения прерываний инструкциями загрузки или восстановления регистра флагов блокируется без генерации исключений.

## 1.4 Защита

Для обеспечения надежной работы многозадачных операционных систем необходимо защищать задачи друг от друга. Защита предназначена для предотвращения несанкционированного доступа к памяти и выполнения критических (опасных для системы) инструкций – команды *HLT*, которая останавливает процессор, команд ввода/вывода, управления флагом разрешения прерываний и команд, влияющих на сегменты кода и данных. Механизмы защиты вводят следующие ограничения:

- ограничение *использования* сегментов (например, запрет записи в только читаемые сегменты данных или попытка исполнения данных, как кода); для использования доступны только сегменты, дескрипторы которых описаны в *GDT* и *LDT*,
- ограничение *доступа* к сегментам через правила привилегий,
- ограничение набора выполняемых инструкций – выделение *привилегированных инструкций* или операций, которые можно выполнять только при определенных уровнях *CPL* и *IOPL*,
- ограничение возможности межсегментных вызовов и передачи управления.

В защищенном режиме при выполнении команд процессор выполняет *проверку условий*, порождающих исключения.

*Проверка при загрузке сегментных регистров:*

- превышение лимита таблицы дескрипторов – *#GP*,
- несуществующий дескриптор сегмента – *#NP* или *#SS*,
- нарушение привилегий – *#GP*,
- загрузка неверного дескриптора или типа сегмента – *#GP*:
  - загрузка в *SS* сегмента кода или сегмента данных только для чтения,

- загрузка управляющих дескрипторов в *DS*, *ES* или *SS*,
- загрузка только исполняемых сегментов в *DS*, *ES* или *SS*,
- загрузка сегмента данных в *CS*.

*Проверка ссылок операндов:*

- запись в сегмент кода или в сегмент данных только для чтения – *#GP*,
- чтение из только исполняемого сегмента кодов – *#GP*,
- превышение лимита сегмента – *#SS* или *#GP*.

*Проверка привилегий инструкций:*

- $CPL \neq 0$  при выполнении команд *LIDT*, *LLDT*, *LGDT*, *LTR*, *LMSW*, *CTS*, *HLT*, *INVD*, *INVLPG*, *WBINVD*, операции с регистрами *DRn*, *TRn*, *CRn* – *#GP*,
- $CPL > IOPL$  при выполнении команд *STI*, *CLI*,
- $CPL > IOPL$  при выполнении инструкций *IN*, *INS*, *OUT*, *OUTS* с портами, не разрешенными битовой картой ввода/вывода – *#GP*.

При выполнении команд *IRET* и *POPF* с недостаточным уровнем привилегий биты *IF* и *IOPL* в регистре флагов не изменяются, исключения не генерируются:

- *IF* не меняется при  $CPL > IOPL$ ,
- *IOPL* не меняется, если  $CPL > 0$ .

*Проверка при передаче управления* по инструкциям *JMP*, *CALL*, *RET*, *INT* *IRET* включают, как проверку ссылок по лимиту (в ближних формах *JMP*, *CALL*, *RET* выполняются только эти проверки), так и проверку правил привилегий при межсегментных передачах через вентили.

Для того, чтобы задачи не вызывали срабатывания защиты, в систему команд введены специальные *инструкции тестирования указателей*. Они позволяют быстро удостовериться в возможности использования селектора или сегмента без риска порождения исключения:

- *ARPL* – выравнивание *RPL*. При ее исполнении *RPL* селектора приравнивается максимальному значению из текущего *RPL* селектора и поля *RPL* в указанном регистре. Если при этом *RPL* изменился, устанавливается флаг нуля  $ZF = 1$ ,
- *VERR* – проверка возможности чтения: если сегмент, на который указывает селектор, допускает чтение, устанавливается  $ZF = 1$ ,
- *VERW* – проверка возможности записи: если сегмент, на который указывает селектор, допускает запись, устанавливается  $ZF = 1$ ,
- *LSL* – чтение лимита сегмента в регистр, если позволяют привилегии; при успешном выполнении устанавливается  $ZF = 1$ ,
- *LAR* – чтение байта доступа дескриптора в регистр, если позволяют привилегии; при успешном выполнении устанавливается  $ZF = 1$ .

Некоторые функции защиты выполняются и механизмом страничной переадресации, однако, в отличие от сегментной защиты, которая может быть обойдена только на нулевом уровне привилегий, страничную защиту можно обойти на уровне пользователя ( $CPL = 3$ ).

### 1.5 Переключение задач

Для многозадачных и многопользовательских операционных систем важна способность процессора к быстрому переключению выполняемых задач. Операция переключения задач процессора сохраняет состояние процессора и связь с предыдущей задачей, загружает состояние новой задачи и начинает ее выполнение. Состояние каждой задачи сохраняется в сегменте состояния задачи (*TSS*), который, как и любой другой сегмент, определяется дескриптором (см. рисунок 1.6).

Дескриптор *TSS* может быть расположен только в *GDT*. Ни в *LDT*, ни в *IDT* он не может быть расположен. При попытке обращения к *TSS* селектором сегмента с установленным флагом *TI* (который указывает на текущую *LDT*) генерируется исключение общей защиты *#GP*. Исключение общей защиты генерируется и в том случае, если производится попытка загрузки селектора *TSS* в сегментный регистр.

Флаг занятости (*B*) в поле типа говорит о том, занята задача, или нет. Занятой является выполняемая или приостановленная задача. Значение поля типа *1001b* индицирует неактивную задачу, а *1011b* – занятую. Задачи не являются рекурсивными.

Переключение задач выполняется по инструкции межсегментного перехода (*JMP*) или вызова (*CALL*), ссылающейся на сегмент состояния задачи (*TSS*) или дескриптор вентиля задачи в *GDT* или *LDT*. Переключение задач может происходить также по аппаратным и программным прерываниям и исключениям, если соответствующий элемент в *IDT* является дескриптором вентиля задачи. Дескриптор *TSS* указывает на сегмент, содержащий полное состояние процессора, а дескриптор вентиля задачи содержит селектор, указывающий на дескриптор *TSS*.

Каждая задача должна иметь связанный с ней *TSS*. 32-разрядные процессоры допускают и 16-битный формат *TSS*. Оба типа сегментов содержат образы регистров процессора, отдельные указатели стеков для уровней 0, 1 и 2, а также обратную ссылку на *TSS* вызвавшей задачи. Свободное поле *TSS* может использоваться по усмотрению операционной системы. *TSS* для процессоров 386+ содержит элементы, отсутствующие в *TSS* 80286: битовые карты разрешения ввода/вывода и перенаправления прерываний, а также бит отладочной ловушки *T* (при  $T=1$  переключение в данную задачу вызывает исключение отладки). Последним элементом *TSS* 386+ должен быть байт 0FFh. Значение поля лимита дескриптора для *TSS* 386+ должно превышать 0064h. Структура сегмента состояния задачи показана на рисунке 1.8.

*Карта разрешения ввода/вывода* (I/O Permission Bit Map), расположенная в конце *TSS* 386+, имеет по одному биту на каждый адрес портов ввода/вывода. Разрешению обращения соответствует нулевое значение бита. Максимальный размер таблицы (2000h), соответствующий всем 64 К адресам, может быть урезан лимитом *TSS*. Порты с адресами, не попавшими в усеченную таблицу, считаются недоступными.

Текущий *TSS* идентифицируется специальным регистром задачи *TR* (Task Register). Этот регистр содержит селектор, ссылающийся на дескриптор текущего *TSS*. Программно-невидимые регистры базового адреса и лимита, связанные *TR*, загружаются при загрузке в *TR* нового селектора.

Для возврата управления задаче, вызвавшей текущую задачу, или ей прерванной, используется инструкция *IRET*. В регистре флагов имеется флаг вложенной задачи *NT* (Nested Task), который управляет действием инструкции *IRET*. При  $NT = 0$  *IRET* работает обычным образом, оставаясь

в текущей задаче. При  $NT = 1$  (текущая задача – вложенная) *IRET* выполняет переключение в предыдущую задачу.

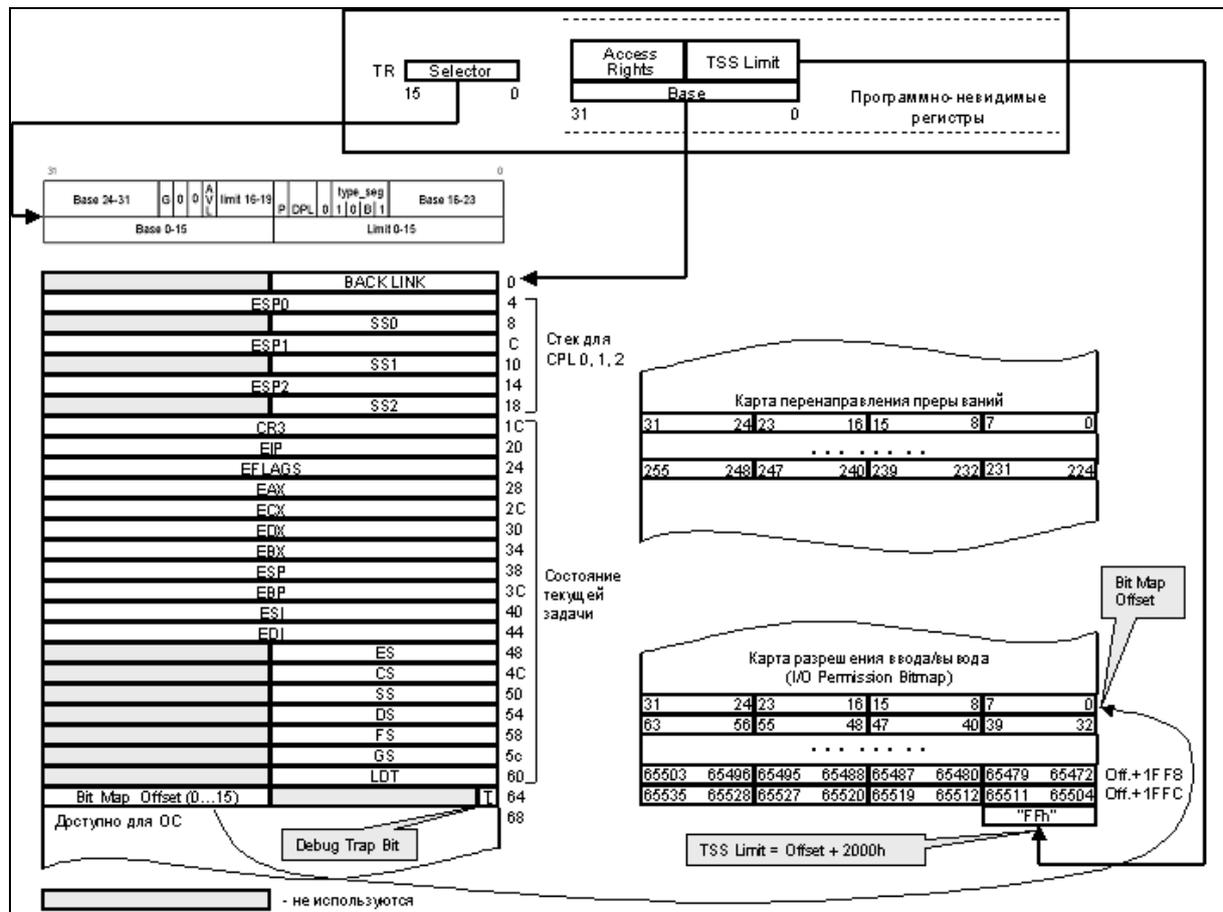


Рисунок 1.8 – Структура сегмента состояния задачи

Когда инструкции *CALL*, *JMP* или *INT* выполняют переключение задач, старый (кроме случая *JMP*) и новый *TSS* помечаются занятыми (меняется значение *Type* в их дескрипторах), и в поле обратной ссылки в новом *TSS* устанавливается значение селектора старого *TSS*. Инструкции *CALL* и *INT*, переключающие задачи, устанавливают в новой задаче бит *NT*. Прерывание, не вызывающее переключения задач, сбросит бит *NT*. Этот бит может устанавливаться и сбрасываться инструкциями *POPF* и *IRET*.

Смена контекста сопроцессора при переключении задач автоматически не производится, так как новой задаче сопроцессор может и не понадобиться. Однако, если процессор обнаруживает первое

использование сопроцессора после переключения задачи, он вырабатывает исключение *#NM*. Обработчик этого исключения сам определит, необходима ли смена контекста. Каждый раз при переключении задач процессор устанавливает бит *TS* (Task Switched) в *MSW*. Это указывает на то, что контекст процессора может относиться к другой задаче. При выполнении инструкций *ESC* или *WAIT*, если *TS* = 1 и *MP* = 1 (сoproцессор присутствует), вырабатывается исключение *#NM* (отсутствующий сопроцессор).

## 1.6 Страничное преобразование

Страничное управление памятью (Paging), как уже было показано в разделе 1.1, является средством организации виртуальной памяти с возможностью подкачки страниц по запросу (Demand-Paged Virtual Memory). В отличие от сегментации, которая организует программы и данные в модули различного размера, страничная организация работает со страницами одинакового размера. В момент обращения страница может присутствовать в физической оперативной памяти, а может в ней и отсутствовать (она может быть выгружена на внешнюю, например, дисковую, память). При обращении к отсутствующей в физической оперативной памяти странице процессор генерирует исключение *#PF* (Page Fault – отказ страницы), а программный обработчик этого исключения (обычно являющийся частью операционной системы) должен получить необходимую информацию для загрузки – «подкачки» страницы с внешнего носителя (обычно диска). Страницы не связаны напрямую с логической структурой данных или программ. В отличие от сегментов, которые являются модулями кодов и данных, а соответствующие селекторы этих сегментов – логическими идентификаторами этих модулей, страницы представляют одинаковые по размеру фрагменты этих модулей. Так как программам и данным присуще свойство локальности – каждая последующая ссылка на коды или данные статистически расположена вблизи к уже выбранным кодам и данным – в физической оперативной памяти одновременно обычно хранят небольшие фрагменты соответствующих сегментов, необходимые активным процессам и задачам. Страничная организация памяти и позволяет реализовать эту возможность. В процессоре i386, в котором впервые появилась возможность страничной организации памяти, могли использоваться страницы размером лишь в 4 Кбайта. Начиная с Pentium'a, появилась

возможность использования страниц размером 4 Мбайта, а, начиная с Pentium Pro, адресная шина процессора увеличилась до 36 разрядов (адресуемое пространство до 64 Гбайт), и появилась возможность использования страниц размеров в 2 Мбайта.

*Базовый механизм* страничного управления памятью использует двухуровневую табличную трансляцию линейного адреса в физический (рисунок 1.9).

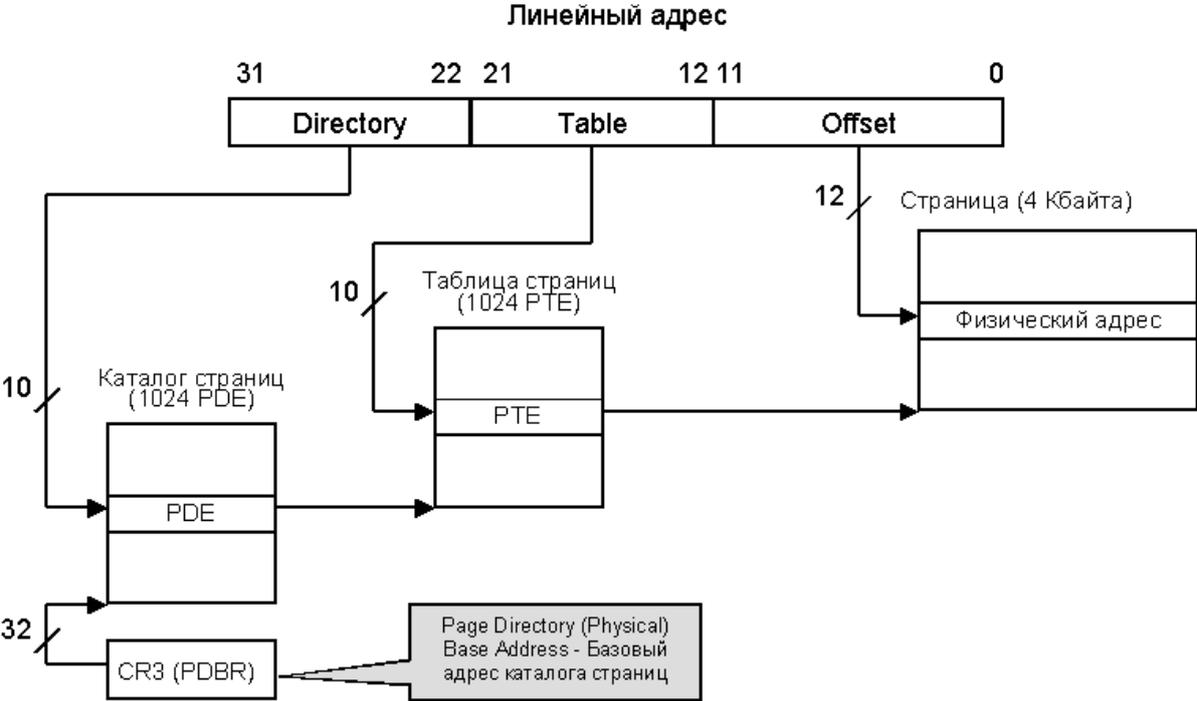


Рисунок 1.9 – Базовый механизм страничного преобразования

Механизм преобразования включает в себя три этапа: *каталог страниц* (Page Directory), *таблицы страниц* (Page Table) и *собственно страницы* (Page Frame). Механизм страничного преобразования включается установкой бита  $PG = 1$  в регистре  $CR0$ . В регистре  $CR2$  хранится *линейный адрес отказа* (Page Fault Linear Address) – адрес в памяти, по которому был обнаружен последний отказ страницы. В регистре  $CR3$  хранится *физический адрес каталога страниц* (Page Directory Physical Base Address). Младшие 12 бит этого регистра всегда нулевые, так как каталог всегда выравнивается по границе страницы.

Каталог страниц размером 4 Кбайта содержит 1024 32-битных записи *PDE* (Page Directory Entry). Структура записи *PDE* показана на рисунке 1.10а.

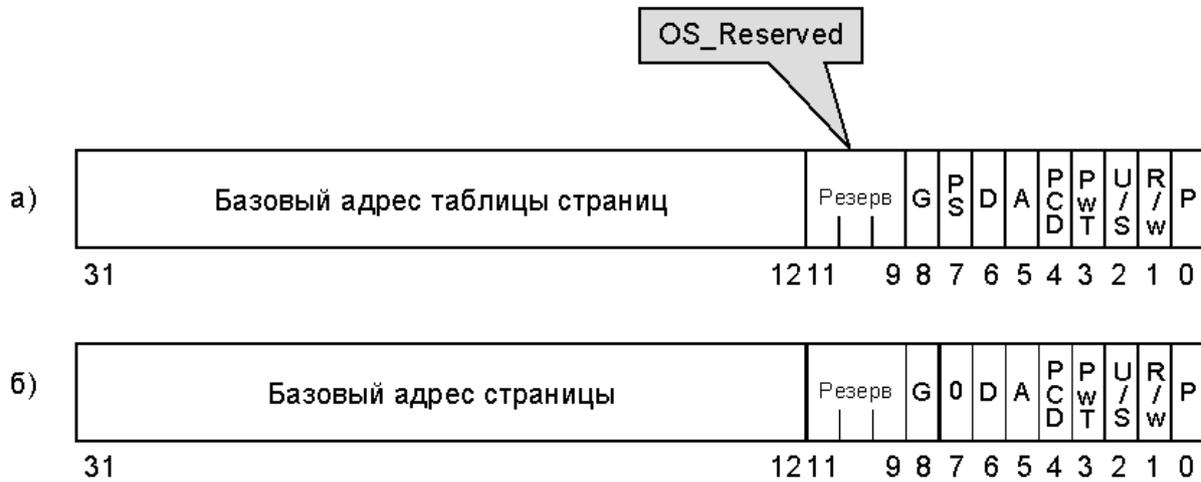


Рисунок 1.10 – Структура элементов страничного преобразования

а – строка каталога, б – строка таблицы страниц

Из рисунка видно, что в каждой строке содержится 20 старших разрядов адреса таблицы страниц (младшие биты адреса всегда равны нулю, так как адрес таблицы всегда выровнен по границе страницы) и набор атрибутов таблицы страниц. Значения атрибутов показаны в таблице 1.5. Индексом поиска в каталоге страниц являются 10 старших разрядов линейного адреса (22 – 31).

*Таблица страниц* также состоит из 1024 строк *PTE* (Page Table Entry). Формат записей в строках *PTE* (рисунок 1.10б) аналогичен предыдущему, но вместо базового адреса таблицы страниц он содержит базовый физический адрес страницы (Page Frame Address). Наборы атрибутов в записях практически совпадают (исключение – бит 7). Индексом поиска в таблице страниц являются следующие 10 разрядов линейного адреса (12 – 21). *Физический адрес операнда* получается из адреса страницы, взятого из таблицы страниц и младших 12 разрядов линейного адреса (смещения внутри страницы).

Таблица 1.5 – Значение атрибутов строк каталога и страниц

Бит	Обозначение	Назначение
0	<i>P</i> (Present)	Бит присутствия. <i>P</i> = 1 означает возможность использования данной строки для трансляции адреса. Для текущего исполняемого кода бит присутствия <i>PDE</i> должен быть установлен. Программный код не должен изменять его «на лету». Если <i>P</i> = 0, то все остальные биты доступны операционной системе. В частности, они могут использоваться для указания местонахождения данной страницы.
1	<i>R/W</i> (Read/Write) [ <i>W</i> (Writable)]	Это атрибут защиты от записи. Атрибут в строке каталога страниц относится ко всем страницам, на которые ссылается данная строка через таблицу второго уровня. Атрибут в строке таблицы страниц относится к соответствующей странице памяти. Права доступа, определяемые битами 1 и 2, приведены в таблице 1.6. Если атрибуты защиты в <i>PDE</i> и <i>PTE</i> различны, то результирующее значение атрибутов доступа определяются таблицей 1.7. Защита на уровне страниц включается установкой бита <i>WP</i> (Write Protect) в управляющем регистре <i>CR0</i> . При аппаратном сбросе этот бит обнуляется.
2	<i>U/S</i> (User/Supervisor) [ <i>U</i> (User)]	Этот атрибут определяет два уровня привилегий: пользователь (User) и супервайзер (Supervisor). Пользователю соответствует уровень привилегий 3, супервайзеру – уровни 0, 1 и 2. Подробности приведена в таблицах 1.6 и 1.7.
3	<i>PWT</i> (Page Write Through)	Этот атрибут определяет политику записи при кэшировании.
4	<i>PCD</i> (Page Cache Disable) <sup>*</sup> )	Этот атрибут запрещает кэширование памяти для обслуживаемых страниц или таблиц. (Атрибут действителен, начиная с процессоров i486).
5	<i>A</i> (Accessed) <sup>*</sup> )	Признак доступа. Этот бит устанавливается перед любым обращением (чтением или записью) по адресу, в преобразовании которого участвует данная строка.
6	<i>D</i> (Dirty) <sup>*</sup> )	Признак, которым помечается «грязная» страница (в которую была осуществлена запись). Этот бит устанавливается перед операцией записи, в преобразовании которой участвует данная строка.
7	<i>PS</i> (Page Size)	Этот атрибут задает размер страницы (только в <i>PDE</i> ). При <i>PS</i> = 0 страница имеет размер 4 Кбайта. Значение <i>PS</i> = 1 используется в расширениях <i>PAE</i> (Page Size Extension – расширение размере страницы) и <i>PSE</i> (Physical Address Extension – расширение физического адреса), которые будут описаны ниже.
	0	В строке таблицы страниц ( <i>PTE</i> ) этот атрибут не используется.

Таблица 1.5 – Значение атрибутов строк каталога и страниц (продолжение)

Бит	Обозначение	Назначение
8	<i>G (Global)</i>	Этот атрибут действителен для процессоров, начиная с Pentium Pro. Он определяет глобальность страницы – позволяет пометить страницы глобального использования (например, страницы ядра операционной системы). При установленном бите <i>PGE</i> в управляющем регистре <i>CR4</i> строки <i>PDE</i> и <i>PTE</i> с указателями на глобальные таблицы не будут аннулироваться в буфере ассоциативной трансляции <i>TLB</i> ( <i>Translation Lookaside Buffers</i> – аппаратный кэш, в котором хранятся последние использованные <i>PDE</i> и <i>PTE</i> ) при загрузке <i>CR3</i> или при переключении задач, что ускоряет обслуживание виртуальной памяти.
9-11	<i>OS_Reserved</i>	Это поле может использоваться операционной системой по своему усмотрению. Например, там может храниться информация о «возрасте» страницы, необходимая для реализации замещения страницы по алгоритму <i>LRU</i> ( <i>Least Recently Used</i> – наиболее долго не используемая страница замещается первой).

\*) Биты *P*, *A* и *D* модифицируются процессором аппаратно в заблокированных шинных циклах. При их программной модификации в многопроцессорных системах необходимо использовать префикс *LOCK*, гарантирующий сохранение целостности данных.

Таблица 1.6 – Защита на уровне страниц

<i>U/S (U)</i>	<i>R/W (W)</i>	Разрешено при <i>PL = 3</i>	Разрешено при <i>PL = 0, 1, 2</i>
0	0	Нет	Чтение/Запись
0	1	Нет	Чтение/Запись
1	0	Только чтение	Чтение/Запись
1	1	Чтение/Запись	Чтение/Запись

Механизм страничного управления при обращении к памяти может породить исключение *#PF*. Оно возникает при обращении к отсутствующей (не представленной) странице или при нарушении прав доступа, определяемых уровнем привилегий и битами *U* и *W*. Для идентификации причины отказа в стек помещается 16-битный код ошибки, формат которого приведен на рисунке 1.11. Здесь названия битов совпадают с атрибутами строк, но их назначение другое. Бит *U/S* указывает, при каком уровне привилегий произошел отказ (1 – уровень



Если при каждом обращении к памяти процессор обращался бы к двум таблицам, расположенным в оперативной памяти, это существенно снизило бы производительность системы. Для предотвращения такого замедления в процессоре имеется буфер ассоциативной трансляции *TLB* (Translation Lookaside Buffer) для хранения активно используемых строк таблиц страниц.

В процессорах *i386* и *i486* этот буфер представляет собой ассоциативный кэш на 32 строки таблиц трансляции. Такой размер кэша позволяет хранить информацию для трансляции 128 Кбайт памяти. Для большинства мультизадачных применений это дает 98% кэш-попаданий. В 2% случаев требуются дополнительные обращения к таблицам.

В процессорах *Pentium* имеются отдельные *TLB* для инструкций и данных. В *Pentium Pro* и позже эти буферы разделены и по размеру страниц (4 Кбайта и 2 Мбайта/4 Мбайта).

Работа механизма страничного преобразования выглядит следующим образом:

- если страничное управление разрешено (бит *PG* в регистре *CR0* установлен в 1), блок страничного преобразования получает 32-битный линейный адрес от блока сегментации,
- старшие 20 разрядов полученного 32-битного адреса сравниваются со значениями, хранящимися в *TLB*,
- в случае кэш-попадания по начальному адресу страницы, полученному из *TLB*, вычисляется физический адрес и выводится на шину адреса,
- если соответствующей строки в *TLB* нет, выполняется чтение строки из страничного каталога,

- если строка имеет бит  $P = 1$  (таблица присутствует в памяти), в ней устанавливается бит доступа  $A$  и выполняется чтение указанной ею строки из таблицы второго уровня,
- если и в этой строке бит  $P = 1$ , процессор обновляет в ней биты  $A$  и  $D$ , вычисляет физический адрес страницы и выполняет обращение по этому адресу,
- если на каком-либо из этапов встречается  $P = 0$ , вырабатывается исключение  $\#PF$ , обработчик которого должен загрузить востребованную страницу в оперативную память; так как это исключение имеет тип «отказ» (Fault), после его успешной обработки повторяется запрос доступа к той же ячейке памяти

После генерации процессором исключения  $\#PF$  операционная система должна выполнить следующие действия:

- если необходимо, скопировать страницу с дискового накопителя в физическую память,
- загрузить адрес страницы в таблицу страниц или в каталог страниц,
- установить в обновленном элементе таблицы бит присутствия ( $P$ ), а также биты доступа ( $A$ ) и «грязности» ( $D$ ),
- аннулировать текущий  $PTE$  (элемент таблицы страниц) в буфере ассоциативной трансляции,
- вернуться из обработчика отказа страницы и возобновить работу прерванной программы или задачи.

Буферы ассоциативной трансляции  $TLB$  программно невидимы для прикладных задач ( $CPL > 0$ ). С этими буферами может работать только операционная система с  $CPL = 0$ . Операционная система должна корректно сгенерировать начальные таблицы трансляции и обрабатывать исключения отказов. При изменении таблиц трансляции, а также при

изменении бита присутствия  $P$  в любых таблицах система должна аннулировать буферы ассоциативной трансляции  $TLB$  частично или полностью. Очистка всех ассоциативных буферов происходит при загрузке управляющего регистра  $CR3$ , выполняемой принудительно или при переключении задач. (Если установлен бит  $PGE$  – Page Global Enable в управляющем регистре  $CR4$ , строки в  $TLB$ , относящиеся к глобальным страницам, не аннулируются.)

Для изменения отображения одиночной страницы очистка  $TLB$  может осуществляться инструкцией  $INVLPG$  (Invalidate  $TLB$  Entry), которая, если возможно, очистит строку конкретной страницы в  $TLB$ , но может обновить и весь буфер, если такой возможности нет.

В процессорах, начиная с Pentium, кроме стандартных страниц в 4 Кбайта, могут использоваться страницы размером 4 Мбайта. Увеличение размера страницы связано с общим увеличением физического объема используемой оперативной памяти, и с увеличением накладных расходов на обслуживание маленьких страниц. Для включения *расширения размера страницы* ( $PSE$  – Page Size Extension) необходимо установить бит  $PSE$  в управляющем регистре  $CR4$ . При  $CR4.PSE = 0$  работает базовый вариант страничного преобразования (рисунок 1.9). При  $CR4.PSE = 1$  процессор анализирует бит 7 ( $PS$ , Page Size – размер страницы) строки каталога страниц  $PDE$ . Если  $PDE.PS = 0$ , эта строка ссылается на таблицу 4-килобайтных страниц, и преобразование выполняется по базовой схеме (рисунок 1.9). Если  $PDE.PS = 1$ , то разряды 22 – 31 этой строки представляют базовый физический адрес 4-мегабайтной страницы. (При 4-мегабайтных страницах этап с таблицами страниц исключен.)

Механизм страничного преобразования для 4-мегабайтных страниц приведен на рисунке 1.12, а формат строки каталога ( $PDE$ ) для этого случая показан на рисунке 1.13.



Рисунок 1.12 – Страничная переадресация в режиме *PSE* (страницы по 4 Мбайт)

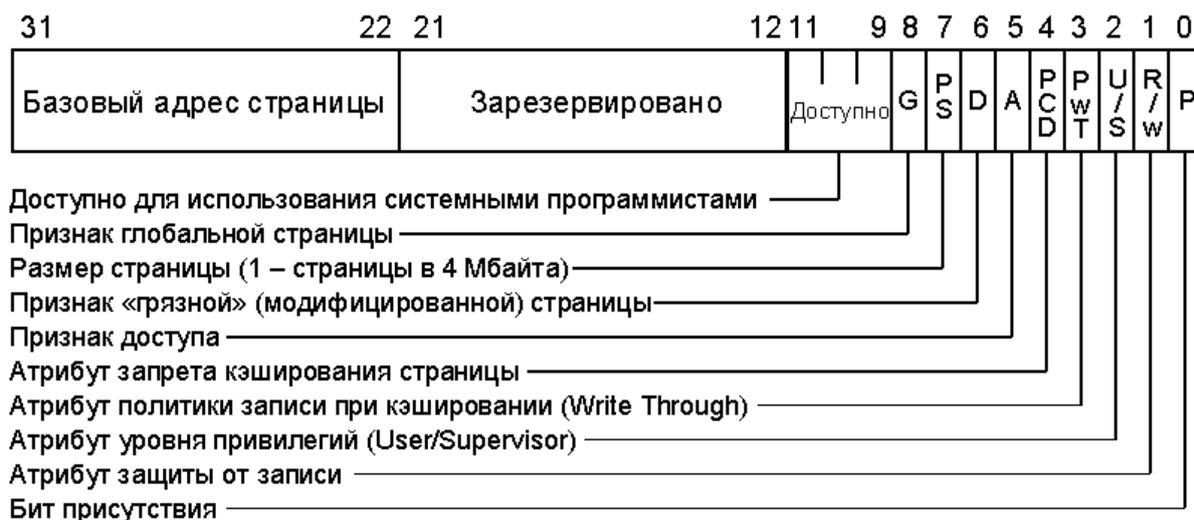


Рисунок 1.13 – Строка каталога для страницы размером 4 Мбайта

Начиная с процессоров Pentium Pro, поддерживается еще один режим страничного преобразования – *PAE* (*Physical Address Extension*) – расширение физического адреса с 32-разрядного до 36-разрядного. Это расширение включается установкой в «1» бита *PAE* в управляющем регистре *CR4* (при этом бит *PSE* игнорируется, и соответствующий режим становится недоступным). У процессоров, начиная с Pentium Pro, шина адреса 36-разрядная, однако, дополнительные 4 разряда адреса доступны

лишь в режиме *PAE* при разрешении страничного преобразования (то есть, одновременно должны быть установлены биты *CR0.PG* и *CR4.PAE*).

Если разрешен режим расширения физического адреса *PAE*, процессор поддерживает страницы нескольких размеров: 4 Кбайта, 2 Мбайта и 4 Мбайта. Как и при 32-разрядной адресации, страницы этих размеров могут адресоваться некоторым набором таблиц страничной трансляции (таблица каталогов может указывать на страницы размером 2 или 4 Мбайта, а таблица страниц – на 4 килобайтные страницы). Для поддержки 36-разрядной физической адресации в структуры данных страничного преобразования внесены следующие изменения:

- Разрядность элементов страничной переадресации увеличена до 64 бит, чтобы в него мог поместиться 36-разрядный базовый физический адрес. Каждый 4-килобайтный каталог страниц и таблица страниц может содержать до 512 записей (входов).
- В иерархию трансляции линейного адреса добавлена новая таблица, называемая таблицей указателей на таблицы каталогов, и расположенная в первых 4 Гбайтах памяти. Эта таблица содержит 4 64-разрядных строки, и она находится в иерархии выше таблиц каталогов. В режиме *PAE* процессор поддерживает до 4 таблиц каталогов страниц.
- В управляющем регистре *CR3* вместо 20-разрядного поля базового адреса таблицы каталогов страниц используется 27-разрядное поле базового адреса таблицы указателя на таблицы каталогов страниц (см. рисунок 1.14). В этом случае регистр *CR3* называется *PDPTR* (Page Directory Pointer Table Register). Это поле представляет старшие 27 разрядов физического адреса первого байта таблицы указателей каталогов страниц, который

выровнен по 32-байтной границе (младшие 5 двоичных разрядов равны нулю).

- Трансляция линейного адреса изменена таким образом, чтобы позволить разместить 32-битные линейные адреса в большем физическом адресном пространстве.

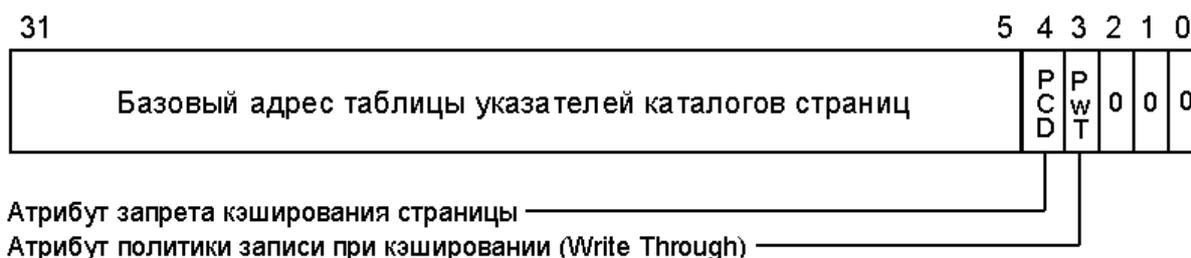


Рисунок 1.14 – Формат управляющего регистра CR3 в режиме расширения физического адреса *PAE*

В режиме *PAE* процессор может работать со страницами размером 4 Кбайта и 2 Мбайта. На рисунке 1.15 показан механизм страничного преобразования в режиме расширения физического адреса *PAE* при размере страниц в 4 Кбайта, а на рисунке 1.16 – для страниц в 2 Мбайта. Из рисунка видно, что в 32-битном управляющем регистре *CR3* хранится базовый адрес таблицы указателей каталогов страниц. Эта таблица состоит всего из четырех 64-разрядных записей. Два старших бита (30 – 31) линейного адреса выбирают из этой таблицы указатель на одну из четырех таблиц каталогов. Следующие 9 разрядов (21 – 29) линейного адреса выбирают элемент из этой таблицы, который может быть ссылкой на таблицу страниц или базовым адресом страницы памяти. Что именно определяет элемент этой таблицы, определяется значением атрибута *PS* записи. При *PS* = 0 запись определяет ссылку на таблицу страниц. Тогда разряды 12 – 20 линейного адреса определяют страницу, размером в 4 Кбайта, в таблице страниц, а разряды 0 – 11 являются смещением в этой странице. При *PS* = 1 запись определяет базовый адрес страницы,

размером в 2 Мбайта. Тогда разряды 0 – 20 линейного адреса являются смещением внутри выбранной страницы с этим базовым адресом.

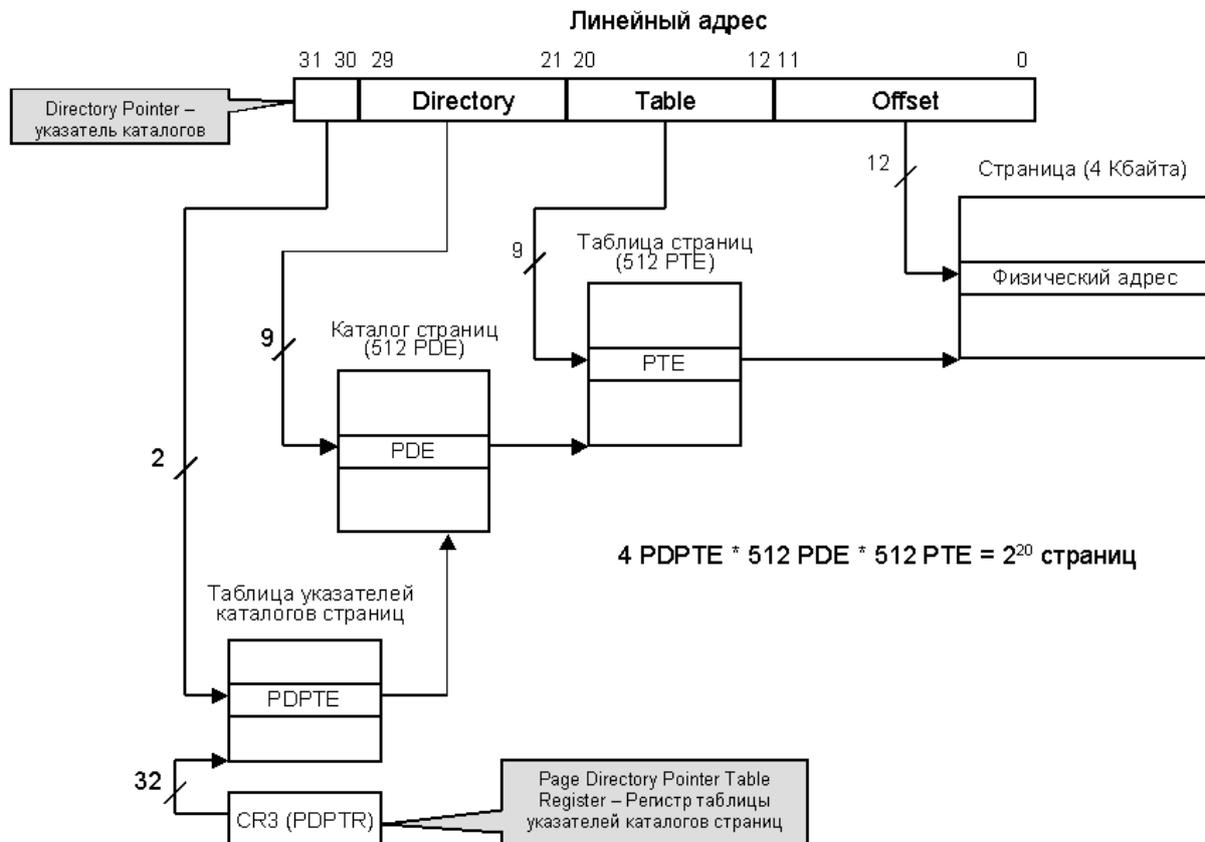


Рисунок 1.15 – Страничное преобразование в режиме расширения физического адреса PAE для страниц размером 4 Кбайта

На рисунке 1.17 – 1.20 показана структура 64-битных элементов страничного преобразования для различного размере страниц. На рисунке 1.17 показана строка таблицы указателей на каталоги (напомним, что эта таблица может состоять не более, чем из четырех элементов). На рисунке 1.18 показана строка каталога страниц, а на рисунке 1.19 – строка таблицы страниц для страниц размером в 4 Кбайта. На рисунке 1.20 показана строка каталога страниц для страниц размером в 2 Мбайта.

Все показанные на рисунках атрибуты имеют те же названия и то же назначение, что и на рисунке 1.13 для строки каталога страниц для страниц размером в 4 Мбайта.

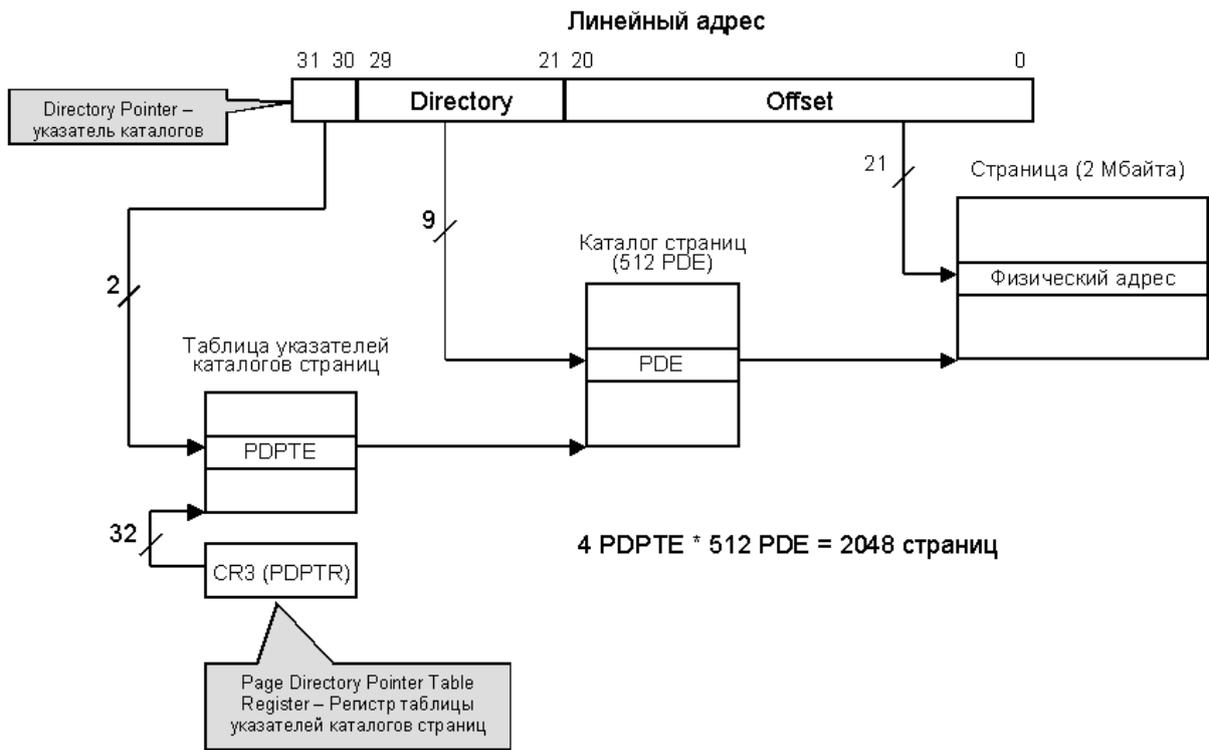


Рисунок 1.16 – Страничное преобразование в режиме расширения физического адреса *PAE* для страниц размером 2 Мбайта

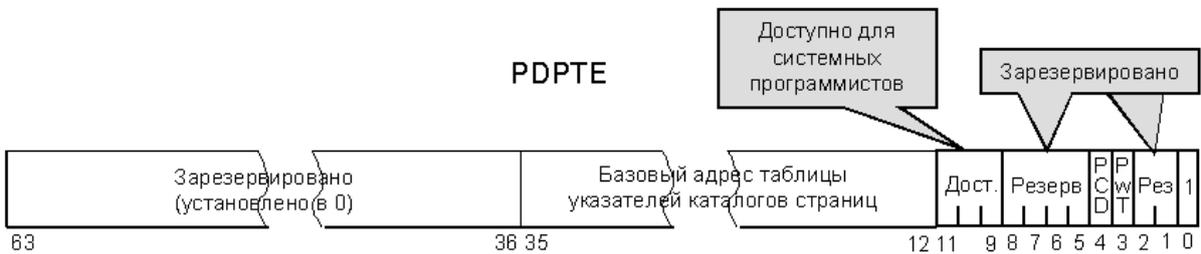


Рисунок 1.17 – Структура 64-битного элемента таблицы указателей на каталоги для режима *PAE*



Рисунок 1.18 – Структура 64-битного элемента таблицы каталога страниц в режиме *PAE* для страниц размером в 4 Кбайта



Рисунок 1.19 – Структура 64-битного элемента таблицы страниц в режиме PAE для страниц размером в 4 Кбайта



Рисунок 1.20 – Структура 64-битного элемента таблицы каталога страниц в режиме PAE для страниц размером в 2 Мбайта

В зависимости от типа записей, изображенных на рисунках 1.17 – 1.20, указанный в них базовый физический адрес обозначает следующее:

- в элементе таблицы указателей на каталоги страниц (*PDPTE*) – физический адрес первого байта каталога 4 Кбайтных страниц (рисунок 1.17),
- в элементе таблицы каталога страниц (*PDE*) – физический адрес первого байта таблицы 4 Кбайтных страниц (рисунок 1.18) или физический адрес первого байта 2 Мбайтной страницы (рисунок 1.20),
- в элементе таблицы страниц (*PTE*) – физический адрес первого байта 4 Кбайтной страницы (рисунок 1.19).

Для всех рассмотренных записей (за исключением записей в таблице каталогов 2 Мбайтных страниц) биты базового адреса интерпретируются как 24 старших разряда 36-разрядного физического адреса, что требует

выравнивания таблиц страниц и самих страниц по 4 Кбайтным границам. Если запись в таблице каталога страниц указывает на 2 Мбайтную страницу, то базовый адрес в этой записи интерпретируется как 15 старших разрядов 36-разрядного физического адреса, что требует выравнивания страниц по 2-мегабайтным границам.

Если разрешен режим расширения физического адреса *PAE*, во всех записях таблицы указателей каталогов страниц (*PDPTE*) должен быть установлен флаг присутствия *P* (бит 0 в записи). То есть, флаг присутствия *P* должен быть установлен, если взведены флаги *PAE* (бит 5 в управляющем регистре *CR4*) и *PG* – Paging, страничное преобразование (бит 31 в управляющем регистре *CR0*). Если при разрешенном режиме расширения физического адреса *PAE* флаг присутствия не установлен во всех четырех записях таблицы указателей каталогов страниц (*PDPTE*), генерируется исключение общей защиты (*#GP*).

Флаг размера страницы (*PS* – бит 7 в записи таблицы каталогов страниц *PDE*) определяет, указывает ли данная запись на таблицу 4 Кбайтных страниц или на 2 Мбайтную страницу. Если флаг сброшен, запись указывает на таблицу 4 Кбайтных страниц, если же он установлен, запись указывает на 2 Мбайтную страницу. Этот флаг позволяет в пределах одного и того же набора таблиц преобразования использовать одновременно 4 Кбайтные и 2 Мбайтные страницы.

Флаги доступа *A* и «грязности» *D* (биты 5 и 6 в записях таблиц) используются в тех элементах таблиц, которые указывают на страницы.

Биты 9 – 11 во всех элементах таблиц в режиме расширения физического адреса *PAE* доступны для программного использования. (Если сброшен флаг присутствия *P*, то для программного использования доступны биты с 1 по 63.)

Все биты на изображениях элементов таблиц страничного преобразования, помеченные, как «зарезервированные» или, как «0», должны быть установлены в «0», и недоступны для программного использования. Когда установлены флаги *PSE* и/или *PAE* в управляющем регистре CR4, процессор генерирует исключение «отказ страницы» (*#PF*), если не установлены в нуль зарезервированные биты элементов таблиц каталогов (*PDE*) и таблиц страниц (*PTE*), или исключение общей защиты (*#GP*), если не установлены в нуль зарезервированные биты элементов таблицы указателей каталогов страниц (*PDPTE*).

Для того чтобы проверить, поддерживает ли конкретный процессор расширения размеров страниц и физического адреса, достаточно попытаться выполнить команду *CPUID*, которая появилась в наборе команд процессора одновременно с указанными расширениями. Если процессор не поддерживает команду *CPUID*, то он не поддерживает и указанные расширения.

## 1.7 Резюме

При работе процессора i386+ в защищенном режиме используется два метода организации памяти – сегментная и страничная. Операционная система Windows NT, о которой будет идти речь в дальнейшем, использует оба этих механизма.

Каждый сегмент в процессоре описывается 8-байтным *дескриптором* сегмента, который определяет положение элемента в памяти, размер занимаемой им области, его назначение и характеристики защиты. Для указания сегмента в таблице дескрипторов используется селектор.

*Системные сегменты* предназначены для хранения локальных таблиц дескрипторов *LDT* и состояния задач *TSS*. Их дескрипторы определяют базовый адрес, лимит сегмента, права доступа и присутствие сегмента в физической памяти.

Защищенный режим имеет четырехуровневую систему привилегий, которая управляет использованием привилегированных команд и доступом к дескрипторам и сегментам. Уровни привилегий нумеруются от 0 до 3, причем значение 0 соответствует высшему уровню привилегий. Привилегии обеспечивают защиту задач друг от друга посредством локальных таблиц дескрипторов. Каждая часть операционной системы – системные сервисы, обработчики прерываний и др. – работает на своем уровне привилегий. Задачи, дескрипторы и селекторы имеют свои уровни привилегий.

Для надежной работы операционной системы необходимо защищать задачи друг от друга. Защита предназначена для предотвращения несанкционированного доступа к памяти и выполнения критических инструкций – команды *HLT*, которая останавливает процессор, команд

ввода/вывода, управления флагом разрешения прерываний и команд, влияющих на сегменты кода и данных.

В защищенном режиме при выполнении команд процессор выполняет *проверку условий*, порождающих исключения.

Для многозадачных операционных систем важна способность процессора к быстрому переключению задач. Операция переключения задач процессора сохраняет состояние процессора и связь с предыдущей задачей, загружает состояние новой задачи и начинает ее выполнение. Состояние каждой задачи сохраняется в сегменте состояния задачи *TSS*, который, как и любой другой сегмент, определяется дескриптором. Дескриптор *TSS* может быть расположен только в *GDT*.

Страничное управление памятью (*Paging*) является одним из важнейших средств организации виртуальной памяти с возможностью подкачки страниц. В отличие от сегментации, которая организует программы и данные в модули различного размера, страничная организация работает со страницами одинакового размера. В момент обращения страница может присутствовать в физической оперативной памяти, а может в ней и отсутствовать (она может быть выгружена на внешнюю, например, дисковую, память). При обращении к отсутствующей в физической оперативной памяти странице процессор генерирует исключение *#PF* (*Page Fault* – отказ страницы), а программный обработчик этого исключения (обычно являющийся частью операционной системы) должен получить необходимую информацию для загрузки – «подкачки» страницы с внешнего носителя (обычно диска). Страницы не связаны напрямую с логической структурой данных или программ. В отличие от сегментов, которые являются модулями кодов и данных, а соответствующие селекторы этих сегментов – логическими

идентификаторами этих модулей, страницы представляют одинаковые по размеру фрагменты этих модулей.

Базовый механизм страничного управления памятью использует двухуровневую табличную трансляцию линейного адреса в физический и страницы, размером 4 Кбайт. Элементы таблиц иерархии страничного преобразования содержат не только базовый физический следующей по иерархии таблицы или самой страницы, но и целый набор атрибутов, одни из которых определяют способы защиты на уровне страниц, а другие используются в реализации дисциплин обслуживания страничного механизма преобразования.

Механизм страничного преобразования при обращении к памяти может породить исключение *#PF*. Оно возникает при обращении к отсутствующей (не представленной) странице или при нарушении прав доступа, определяемых уровнем привилегий и битами *U* и *W*. Для идентификации причины отказа в стек помещается 16-битный код ошибки.

Для предотвращения замедления, связанного с обращением к двум таблицам в оперативной памяти при каждом обращении к памяти в процессоре имеется буфер ассоциативной трансляции *TLB* (Translation Lookaside Buffer) для хранения активно используемых строк таблиц страниц.

В процессорах *i386* и *i486* этот буфер представляет собой ассоциативный кэш на 32 строки таблиц трансляции. Такой размер кэша позволяет хранить информацию для трансляции 128 Кбайт памяти. Для большинства мультизадачных применений это дает 98% кэш-попаданий. В 2% случаев требуются дополнительные обращения к таблицам.

В процессорах, начиная с *Pentium*, кроме стандартных страниц в 4 Кбайта, могут использоваться страницы размером 4 Мбайта. Увеличение

размера страницы связано с общим увеличением физического объема используемой оперативной памяти, и с увеличением накладных расходов на обслуживание маленьких страниц. Для включения *расширения размера страницы* (*PSE* – Page Size Extension) необходимо установить бит *PSE* в управляющем регистре *CR4*. При  $CR4.PSE = 0$  работает базовый вариант страничного преобразования (рисунок 1.9). При  $CR4.PSE = 1$  процессор анализирует бит 7 (*PS*, Page Size – размер страницы) строки каталога страниц *PDE*. Если  $PDE.PS = 0$ , эта строка ссылается на таблицу 4-килобайтных страниц, и преобразование выполняется по базовой схеме. Если  $PDE.PS = 1$ , то разряды 22 – 31 этой строки представляют базовый физический адрес 4-мегабайтной страницы. (При 4-мегабайтных страницах этап с таблицами страниц исключен.)

Начиная с процессоров Pentium Pro, поддерживается еще один режим страничного преобразования – *PAE* (Physical Address Extension) – *расширение физического адреса* с 32-разрядного до 36-разрядного. Это расширение включается установкой в «1» бита *PAE* в управляющем регистре *CR4* (при этом бит *PSE* игнорируется, и соответствующий режим становится недоступным). У процессоров, начиная с Pentium Pro, шина адреса 36-разрядная, однако, дополнительные 4 разряда адреса доступны лишь в режиме *PAE* при разрешении страничного преобразования (то есть, одновременно должны быть установлены биты *CR0.PG* и *CR4.PAE*).

Если разрешен режим расширения физического адреса *PAE*, процессор поддерживает страницы нескольких размеров: 4 Кбайта, 2 Мбайта и 4 Мбайта. Как и при 32-разрядной адресации, страницы этих размеров могут адресоваться некоторым набором таблиц страничной трансляции (таблица каталогов может указывать на страницы размером 2 или 4 Мбайта, а таблица страниц – на 4 килобайтные страницы).

Для поддержки 36-разрядной физической адресации в структуры данных страничного преобразования внесены следующие изменения:

- Разрядность элементов страничной переадресации увеличена до 64 бит, чтобы в него мог поместиться 36-разрядный базовый физический адрес. Каждый 4-килобайтный каталог страниц и таблица страниц может содержать до 512 записей.
- В иерархию трансляции линейного адреса добавлена новая таблица, называемая таблицей указателей на таблицы каталогов, и расположенная в первых 4 Гбайтах памяти. Эта таблица содержит 4 64-разрядных строки, и она находится в иерархии выше таблиц каталогов. В режиме *PAE* процессор поддерживает до 4 таблиц каталогов страниц.
- В управляющем регистре *CR3* вместо 20-разрядного поля базового адреса таблицы каталогов страниц используется 27-разрядное поле базового адреса таблицы указателя на таблицы каталогов страниц (см. рисунок 1.14). В этом случае регистр *CR3* называется *PDPTR* (Page Directory Pointer Table Register). Это поле представляет старшие 27 разрядов физического адреса первого байта таблицы указателей каталогов страниц, который выровнен по 32-байтной границе (младшие 5 двоичных разрядов равны нулю).
- Трансляция линейного адреса изменена таким образом, чтобы позволить разместить 32-битные линейные адреса в большем физическом адресном пространстве.

## *1.8 Контрольные вопросы*

1. Какие методы организации памяти реализованы в защищенном режиме i386+?
2. Какие методы управления памятью используются в Windows NT?
3. Каков максимальный размер сегмента в процессоре i386+?
4. Какая модель памяти используется в Windows NT?
5. Как формируется линейный адрес из виртуального в защищенном режиме?
6. Что такое дескриптор сегмента?
7. Что такое селектор сегмента?
8. Чем отличается глобальная таблица дескрипторов от локальной?
9. Укажите первые 4 сегмента Windows NT
10. Что такое требуемый уровень привилегий?
11. Что такое уровень привилегий дескриптора?
12. Что такое текущий уровень привилегий?
13. Как формируется физический адрес в защищенном режиме?
14. Как задается адрес таблица трансляции линейного адреса в физический?
15. Как и когда загружается управляющий регистр CR3?
16. Как процессор узнает о возможности доступа к странице из пользовательского режима?
17. Как процессор узнает о возможности записи в страницу?
18. Что такое бит присутствия страницы?
19. Что происходит, если происходит обращение к непредставленной странице?
20. Что произойдет при попытке использования селектора данных для модификации кода?

21. Доступны ли элементы таблицы трансляции виртуальных адресов в физические из пользовательского режима?
22. Каковы системное и пользовательское адресные пространства Windows NT?
23. Что такое глобальная таблица дескрипторов?
24. Что такое локальная таблица дескрипторов?
25. Что такое таблица дескрипторов прерываний?
26. Что такое регистр задачи, и что в нем хранится?
27. Что такое бит гранулярности?
28. Байт прав доступа дескриптора
29. Что такое бит разрядности операндов в дескрипторе?
30. Что такое бит доступа к сегменту?
31. Что такое бит доступа по чтению?
32. Что такое бит доступа по записи?
33. Что такое бит предназначения (Intending) и для чего он нужен?
34. Что такое бит сегмента?
35. Что такое системные сегменты?
36. Что такое бит AVL?
37. Для чего используется поле DPL в системных сегментах?
38. Для чего используется поле DPL в дескрипторах локальных таблиц?
39. Когда возможна непосредственная межсегментная передача управления?
40. Для чего используются вентили и шлюзы?
41. Что используется для вызова процедур со сменой уровня привилегий?
42. Для чего используются вентили задач?
43. Какие механизмы позволяют копировать массив слов из старого стека в новый?

44. Какое исключение возникает при указании на некорректный тип дескриптора?
45. Какие исключения возникают при указании на недействительный тип дескриптора?
46. Что такое привилегии?
47. На каком уровне привилегий работает программа в реальном режиме?
48. Для чего нужны привилегии?
49. С какого уровня привилегий начинает выполняться задача?
50. Какой уровень привилегий имеет самый защищенный дескриптор?
51. Какой уровень привилегий имеет самый незащищенный дескриптор?
52. Что такое эффективный уровень привилегий?
53. Когда производится контроль доступа к сегментам данных?
54. В каком случае возможен доступ к сегменту данных?
55. С каким текущим уровнем привилегий может быть прочитан подчиненный сегмент кода?
56. В каком случае производится контроль типов и привилегий при передаче управления?
57. Правила привилегий для команд JMP или CALL
58. Правила привилегий при прерываниях внутри задачи
59. Правила привилегий для инструкций возврата
60. Правила привилегий при переключении задач

## 2 СТРУКТУРА WINDOWS 2000

Подсистема ввода-вывода операционной системы Windows 2000 состоит из ряда компонентов исполнительной системы ОС, которые управляют аппаратными средствами и предоставляют интерфейсы для обращения к ним системе и приложениям пользователя. Основным средством работы прикладного программного обеспечения с аппаратным обеспечением являются драйверы. В ОС Windows 2000 используются различные типы драйверов, которые по-разному взаимодействуют с системными компонентами и с программами пользователя.

Прежде, чем рассматривать устройство драйверов, необходимо достаточно подробно исследовать структуру операционной системы, место в ней подсистемы ввода-вывода, а также взаимодействие компонентов подсистемы между собой и с системными компонентами.

Упрощенная схема архитектуры операционной системы Windows 2000 показана на рисунке 2.1 [1].

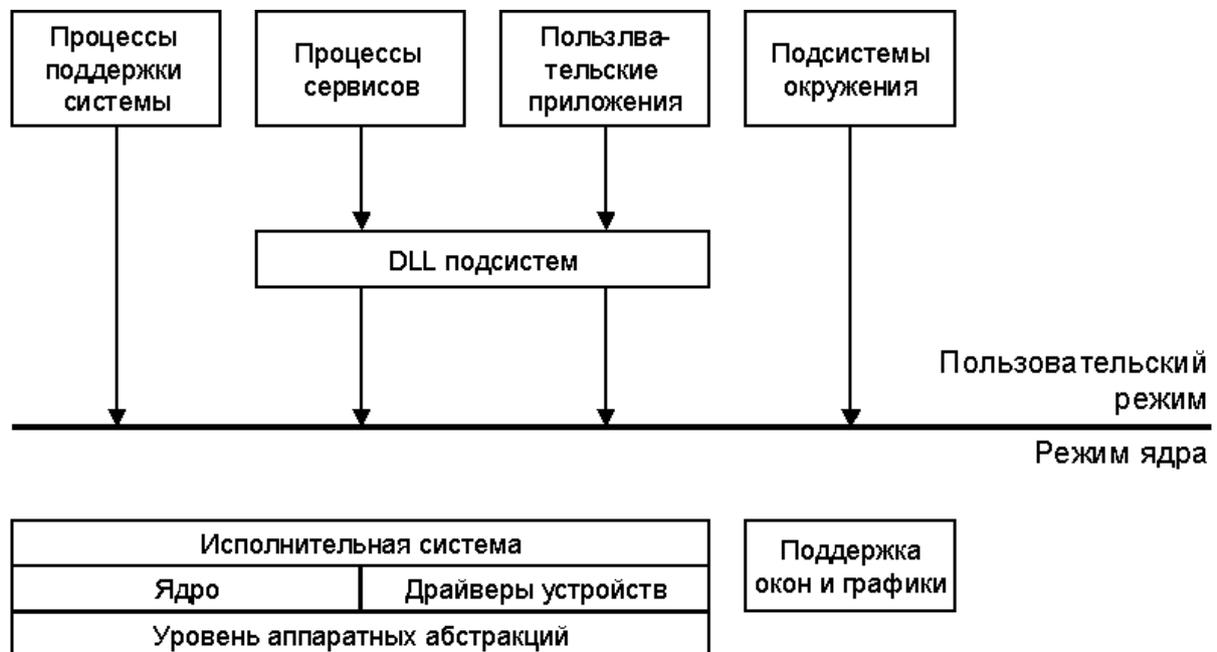


Рисунок 2.1 – Упрощенная архитектура Windows 2000

Понятно, что представленная на рисунке схема дает лишь приближенное представление об элементах системы и об их взаимодействии. Однако уже на этой упрощенной схеме можно заметить, что элементы операционной системы разделяются на два класса – одни выполняются в пользовательском режиме, другие – в режиме ядра. Потоки процессов пользовательского режима выполняются в защищенных адресных пространствах процессов (хотя при выполнении в режиме ядра они получают доступ к системному пространству). Таким образом, процессы поддержки системы, сервисов, приложений и подсистем окружения выполняются в своих адресных пространствах.

Существует четыре типа пользовательских процессов [1]:

- фиксированные *процессы поддержки системы* (System Support Processes) – например, процесс обработки входа в систему и диспетчер сеансов, не являющиеся сервисами Windows 2000 (то есть, они не запускаются диспетчером управления сервисами),
- *процессы сервисов* (Service processes) – носители Win32-сервисов, вроде *Task Scheduler* (планировщик задач) и *Spooler* (спулер печати); многие серверные приложения Windows 2000, например, Microsoft SQL Server и Microsoft Exchange Server, также включают в себя компоненты, выполняемые, как сервисы,
- *пользовательские приложения* (User Applications) – бывают пяти типов: *Win32*, *Windows 3.1*, *MS-DOS*, *POSIX* и *OS/2 1.2*,
- подсистемы окружения (Environment Subsystems) – предоставляют пользовательским приложениям сервисы, встроенные в операционную систему, через набор вызываемых функций, образуя таким образом *окружение операционной среды*.

Следует обратить особое внимание на элемент «*DLL* подсистем». Его присутствие связано с тем, что в Windows 2000 пользовательские

приложения не могут напрямую вызывать встроенные сервисы операционной системы – они работают через одну или несколько *DLL подсистем* (Subsystem *DLL*). Они предназначены для трансляции документированных функций в соответствующие недокументированные внутренние вызовы системных сервисов Windows 2000.

Windows 2000 включает следующие компоненты режима ядра [1]:

- *исполнительная система* (executive), содержащая базовые сервисы операционной системы (обеспечивающие управление памятью, процессами и потоками, защиту, ввод-вывод и взаимодействие между процессами),
- *ядро* (kernel), содержащее низкоуровневые функции операционной системы (поддерживающие планирование потоков, диспетчеризацию прерываний и исключений, синхронизацию процессов и т. д.); ядро предоставляет также набор процедур и базовых объектов, используемых исполнительной системой для реализации структур высшего уровня,
- *драйверы устройств* (Device Drivers) – драйверы аппаратных устройств, транслирующие пользовательские вызовы функций ввода-вывода в специализированные запросы для конкретных устройств, сетевые драйверы и драйверы файловых систем,
- *уровень аппаратных абстракций* (Hardware Abstraction Layer, *HAL*), изолирующий ядро, драйверы и исполнительную систему Windows 2000 от специфики внешних устройств и аппаратной платформы,
- *подсистема поддержки окон и графики* (Windowing and Graphics System), реализующая функции графического пользовательского интерфейса (*GUI*), более известные как *Win32*-функции модулей *User* и *GDI*; эти функции обеспечивают поддержку окон,

элементов управления интерфейса пользователя и прорисовку графики.

В таблице 2.1 приведены основные файлы системных компонентов Windows 2000.

Таблица 2.1 – Основные файлы системных компонентов Windows 2000

Имя файла	Компоненты системы
Ntoskrnl.exe	Исполнительная система и ядро
Ntkrnlpa.exe	Исполнительная система и ядро с поддержкой механизма Physical Address Extension (PAE), позволяющего использовать 36-разрядную адресацию (адресное пространство в 64 Гбайта)
Hal.dll	Уровень аппаратных абстракций
Win32k.sys	Часть подсистемы Win32, работающая в режиме ядра
Ntdll.dll	Внутренние функции поддержки и интерфейсы диспетчера системных сервисов с функциями исполнительной системы
Kernel32.dll, Advapi32.dll, User32.dll, Gdi32.dll	Основные DLL подсистемы Win32

## 2.1 Основные компоненты системы

Рассмотрим теперь более подробно компоненты операционной системы Windows 2000, примерная архитектура которой была изображена на рисунке 2.1. Более подробно архитектура Windows 2000 представлена на рисунке 2.2 [1].

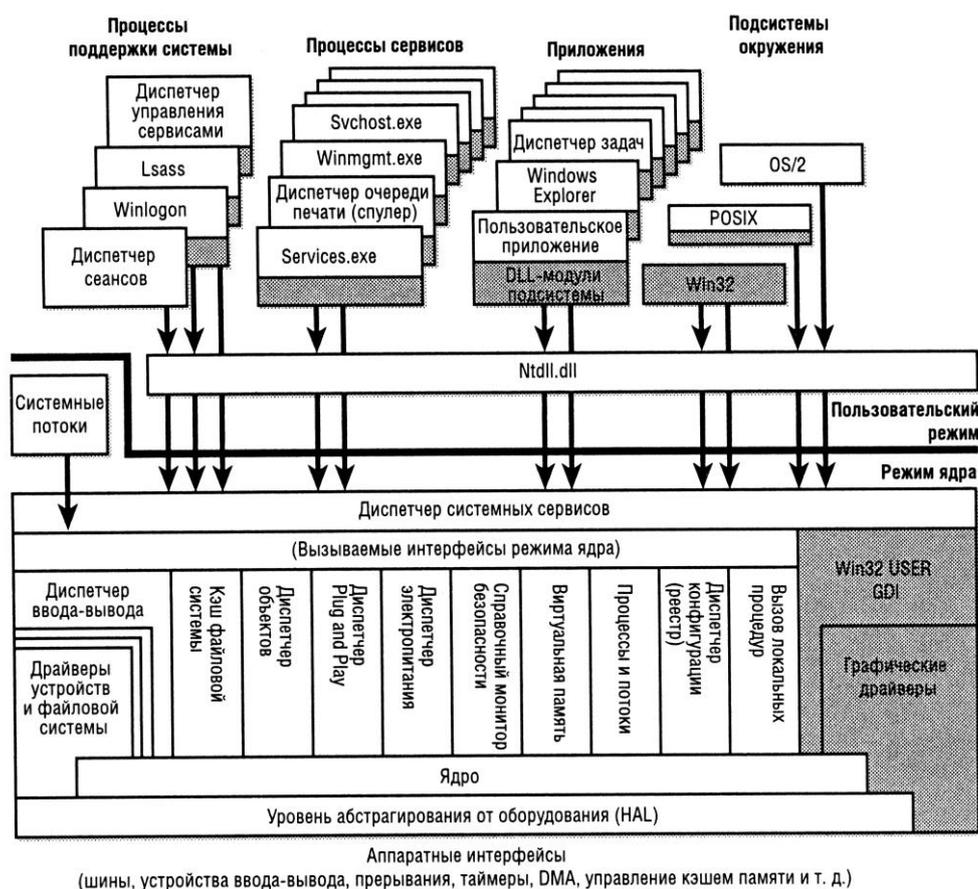


Рисунок 2.2 – Архитектура Windows 2000

Из рисунка видно, что в Windows 2000 имеется три подсистемы окружения: *Win32*, *POSIX* и *OS/2*. Подсистема *Win32* здесь стоит на первом месте не случайно – без нее Windows 2000 работать не может. Эта подсистема обрабатывает все, что связано с клавиатурой, мышью и экраном. Она нужна даже на тех серверах, у которых нет интерактивных пользователей. Подсистема *Win32* работает всегда, а остальные две подсистемы запускаются только по требованию.

Какие подсистемы будут загружены при старте Windows 2000, определяется значением параметра *Required* в разделе реестра *HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Subsystem*. В этом параметре просто указан список подсистем, загружаемых при запуске.

Каждая подсистема окружения предоставляет прикладным программам свое подмножество базовых сервисов исполнительной системы Windows 2000. Это значит, что приложение, созданное для одной подсистемы, может выполнять только операции этой подсистемы, и не может другой. Так Win32-приложения не могут использовать POSIX-функцию *fork*.

Каждый исполняемый файл (.EXE) принадлежит одной, и только одной подсистеме. При запуске его образа код, отвечающий за создание процесса, получает тип подсистемы, указанный в заголовке образа, и уведомляет соответствующую подсистему о новом процессе. В Microsoft Visual C++ тип указывается спецификатором */SUBSYSTEM* в команде *link*.

Одновременные вызовы функций различных подсистем невозможны. То есть, приложения *POSIX* могут вызывать только сервисы, экспортируемые этой подсистемой, а приложения *Win32* – только сервисы, экспортируемые подсистемой *Win32*.

Как уже сказано выше, приложения пользователя не могут напрямую вызывать системные сервисы Windows 2000. Они обращаются к *DLL* подсистем. Эти *DLL* предоставляют документированный интерфейс между программой и вызываемой ей подсистемой. *DLL* подсистемы *Win32* (*Kernel32.dll*, *Advapi32.dll*, *User32.dll*, *Gdi32.dll*) реализуют функции *Win32 API*, а *DLL* подсистемы *POSIX* реализует функции *POSIX 1003.1 API*.

При вызове приложением одной из функций *DLL* подсистемы может возникнуть один из трех случаев:

- Функция полностью реализована в пользовательском режиме внутри *DLL* подсистемы. То есть вызова исполнительной системы Windows 2000 не происходит, и после выполнения функции в пользовательском режиме ее результат возвращается вызвавшей функцию программе.
- Функция требует одного или более вызовов исполнительной системы Windows 2000. Например, *Win32*-функции *ReadFile* и *WriteFile* обращаются к внутренним недокументированным сервисам ввода-вывода – к *NtReadFile* и *NtWriteFile*, соответственно.
- Функция требует выполнения каких-либо операций в процессе подсистемы окружения. В этом случае подсистеме окружения передается сообщение с клиент-серверным запросом выполнения какой-либо операции. *DLL* подсистемы в этом случае возвращает управление вызвавшей программе только после получения соответствующего ответа.

Некоторые функции, вроде *CreateProcess* или *CreateThread* могут требовать выполнение и второго, и третьего пунктов.

Так как *Win32* является главной подсистемой окружения Windows 2000, код для обработки окон и отображения ввода-вывода помещен именно в эту подсистему. Другие подсистемы окружения для выполнения базовых функций ввода-вывода вызывают соответствующие сервисы *Win32*. Далее мы рассмотрим более подробно компоненты Windows 2000, изображенные на рисунке 2.2.

## 2.2 Подсистемы окружения

Как уже было показано, в состав Windows 2000 входит три подсистемы окружения *Win32*, *POSIX* и *OS/2*.

Подсистема *Win32* состоит из следующих основных элементов:

- процесса подсистемы окружения (*Csrss.exe*), предоставляющего:
  - поддержку консольных (текстовых) окон,
  - поддержку создания и удаления процессов и потоков,
  - другие функции типа *GetTempFile*, *DefineDosDevice*, *TxitWindowsEx*, а также некоторые функции поддержки естественных языков,
- драйвера режима ядра (*Win32k.sys*), включающего:
  - диспетчер окон, управляющий прорисовкой и выводом окон на экран, принимающий ввод с клавиатуры, мыши и других устройств, а также передающий пользовательские сообщения приложениям,
- *DLL*-модулей подсистем (*Kernel32.dll*, *Advapi32.dll*, *User32.dll*, *Gdi32.dll*), транслирующих вызовы документированных функций *Win32 API* в вызовы соответствующих недокументированных сервисов режима ядра из *Ntoskrnl.exe* и *Win32.sys*,
- драйверов графических устройств, представляющих собой специфические для конкретного оборудования драйверы дисплея, принтера и минипорт-драйверы видеокарт.

Подсистема *POSIX* (Portable Operating System Interface Based on *UNIX* – переносимый интерфейс операционной системе на основе *UNIX*) – это совокупность международных стандартов на интерфейсы операционных систем типа *UNIX*. Набор функций, доступный приложениям *POSIX* по умолчанию, строго ограничен сервисами, определяемыми стандартом *POSIX.1*. Эти ограничения заключаются в том,

что приложение *POSIX* не может создать поток или окно в Windows 2000, а также не может использовать *RPC* (Remote Procedure Call – стандарт сетевого программирования, позволяющий создавать приложения, состоящие из произвольного числа процедур, часть из которых выполняется локально, а часть – на удаленных компьютерах через сеть) и сокеты (конечная точка коммуникационного соединения).

Подсистема *OS/2*, как и подсистема *POSIX*, обладает ограниченной функциональностью и поддерживает лишь 16-разрядные приложения *OS/2* версии 1.2 с символьным или графическим вводом-выводом. Как и подсистема *POSIX*, подсистема *OS/2* автоматически запускается при первой активизации *OS/2*-совместимого приложения и продолжает работать до перезагрузки системы.

Модуль *Ntdll.dll* – специальная библиотека системной поддержки, необходимая в основном при использовании *DLL*-подсистем. Она содержит функции двух типов:

- интерфейсы диспетчера системных сервисов (System Service Dispatch Stubs) к сервисам исполнительной системы Windows 2000,
- внутренние функции поддержки, используемые подсистемами, *DLL* подсистем и другими компонентами операционной системы.

Первая группа функций предоставляет интерфейс к сервисам исполнительной системы Windows 2000, которые можно вызывать из пользовательского режима. Таких функций более 200, например, *NtCreateFile*, *NtSetEvent* и т. д. Большинство из этих функций доступно через *Win32 API*, но некоторые из них доступны только для внутреннего применения. Для каждой из функций в *Ntdll* существует точка входа с именем функции. В коде функции содержится специфическая для конкретной аппаратуры команда перехода в режим ядра для вызова

системных сервисов, которая после проверки некоторых параметров вызывает уже настоящий сервис режима ядра из *Ntoskrnl.exe*.

*Ntdll* включает также множество функций поддержки, например:

- загрузчик образов (функции, имена которых начинаются с *Ldr*),
- диспетчер куч,
- функции для взаимодействия с процессом подсистемы *Win32* (функции, имена которых начинаются с *Csr*),
- универсальные процедуры библиотек времени выполнения (функции, имена которых начинаются с *Rtl*),
- диспетчер *APC* (Asynchronous Procedure Call) пользовательского режима и
- диспетчер исключений.

### 2.3 Исполнительная система

Исполнительная система (*Executive*) находится на верхнем уровне *Ntoskrnl.exe* (ядро располагается на более низком уровне). В ее состав входят следующие функции:

- экспортируемые функции, доступные для вызова из пользовательского режима, называемые *системными сервисами* и экспортирующиеся через *Ntdll*; большинство сервисов доступно через *Win32 API* или *API* других подсистем окружения, однако, некоторые из них недоступны через документированные функции (например, *LPC – Local Procedure Call*, локальный вызов процедуры, функции запросов типа *NtQueryInformationxxx*, специализированные функции типа *NtCreatePagingFile* и т. д.),
- экспортируемые функции, доступные для вызова только из режима ядра и описанные в *Windows 2000 DDK (Driver Development Kit)* или в *Windows 2000 IFS (Installable File System) Kit*,
- экспортируемые функции доступные для вызова только из режима ядра, но не описанные в *Windows 2000 DDK* или в *Windows 2000 IFS Kit* (например, функции, используемые видеодрайвером, работающим на этапе загрузки, чьи имена начинаются с *Inbv*),
- функции, определенные, как глобальные, но не экспортируемые символы (например, внутренние функции поддержки, вызываемые в *Ntoskrnl*, чьи имена начинаются с *Iop* – функции поддержки диспетчера ввода-вывода – или с *Mi* – функции поддержки управления памятью),
- внутренние функции в каком-либо модуле, не определенные как глобальные символы.

Исполнительная система состоит из следующих основных компонентов:

- *диспетчер конфигурации*, отвечающий за управление системным реестром,
- *диспетчер процессов и потоков*, создающий и завершающий процессы и потоки,
- *справочный монитор безопасности*, реализующий политики безопасности на локальном компьютере (он охраняет ресурсы операционной системы и контролирует объекты во время выполнения),
- *диспетчер ввода-вывода*, реализующий аппаратно-независимый ввод-вывод и отвечающий за пересылку ввода-вывода нужным драйверам устройств для дальнейшей обработки.
- *диспетчер Plug and Play*, определяющий, какие драйверы нужны для поддержки конкретного устройства, и загружающий их; требования каждого устройства к аппаратным ресурсам определяются в процессе *перечисления* устройств (в зависимости от требований устройств диспетчер *PnP* распределяет такие ресурсы, как порты ввода-вывода, *IRQ*, каналы *DMA* и области памяти, а также отвечает за посылку соответствующих уведомлений об изменениях в аппаратном обеспечении системы при добавлении или удалении устройств),
- *диспетчер электропитания*, который координирует события, связанные с электропитанием и генерирует уведомления системы управления электропитанием для драйверов (изменение энергопотребления отдельных устройств возлагается на их драйверы, но координируется диспетчером электропитания),

- *подпрограммы WMI* (Windows Management Instrumentation – инструментарий управления Windows), позволяющие драйверам публиковать информацию о своих рабочих характеристиках и конфигурации, а также получать команды от службы *WMI* пользовательского режима (потребители информации *WMI* могут находиться как на локальной машине, так и на любом компьютере в сети),
- *диспетчер кэша*, повышающий производительность файлового ввода-вывода за счет сохранения в основной памяти дисковых данных, к которым недавно было обращение (это уменьшает также общее число обращений к диску для записи, так как модифицированные данные предварительно накапливаются в памяти в течение определенного периода),
- *диспетчер виртуальной памяти*, реализующий виртуальную память – схему управления памятью, позволяющую выделять каждому процессу большое закрытое адресное пространство, объем которого может превышать доступную физическую память.

В состав исполнительной системы входит также четыре основные группы функций поддержки, которые используются уже перечисленными компонентами. К ним относятся:

- *диспетчер объектов*, который создает, управляет и удаляет объекты и абстрактные типы данных исполнительной системы (они используются для представления таких ресурсов операционной системы, как процессы, потоки и различные синхронизирующие объекты),
- *механизм LPC* (Local Procedure Call), который передает сообщения между клиентским и серверным процессами на одном компьютере (*LPC* является оптимизированной версией *RPC* –

- Remote Procedure Call) и представляет собой стандартный механизм взаимодействия между клиентскими и серверными процессами через сеть,
- набор стандартных библиотечных функций для обработки строк, арифметических операций, преобразования типов данных и обработки структур безопасности,
  - *подпрограммы поддержки исполнительной системы*, например, для выделения системной памяти, доступа к памяти с взаимоблокировкой, а также два специальных типа синхронизирующих объектов: ресурс (*Recourse*) и быстродействующий мьютекс (*Fast Mutex*).

## 2.4 Ядро

Ядро состоит из набора фундаментальных функций [1] (в том числе, планирование потоков и синхронизация), которые расположены в файле *Ntoskrnl.exe*, и используются компонентами исполнительной системы и низкоуровневыми (аппаратно-зависимыми) средствами – диспетчерами прерываний и исключений, которые непосредственно «привязаны» к конкретной аппаратной платформе.

Ядро состоит из низкоуровневых четко определенных и хорошо предсказуемых примитивов и механизмов операционной системы, поддерживающих функции компонентов исполнительной системы более высокого уровня. Ядро отделено от исполнительной системы – оно реализует системные механизмы, но не участвует в принятии решений, связанных с системной политикой. Все такие решения, кроме планирования и диспетчеризации потоков, принимаются исполнительной системой.

Вне ядра исполнительная система представляет потоки и другие разделяемые ресурсы в виде объектов. Для каждого объекта необходим дескриптор, позволяющий манипулировать им, средства защиты и квоты, резервируемые при их создании, поэтому обслуживание объектов требует немалых ресурсов. Ядро реализует набор более простых объектов, называемых *объектами ядра (Kernel Objects)*, что позволяет снизить издержки на их обслуживание. Объекты ядра позволяют ядру контролировать обработку данных процессором и поддерживают объекты исполнительной системы. Большинство объектов исполнительной системы инкапсулирует один или более объектов ядра, включая в себя их атрибуты, определенные ядром.

Одна из групп объектов ядра, называемых *управляющими (Control Objects)*, определяет семантику управления различными функциями

операционной системы. В эту группу входят объекты *APC (Asynchronous Procedure Call* – асинхронный вызов процедуры), *DPC (Deferred Procedure Call* – отсроченный вызов процедуры) и некоторые объекты, используемые диспетчером ввода-вывода (например, объект прерывания).

Другая группа объектов называется *объекты диспетчера (Dispatcher Objects)* реализует средства синхронизации, позволяющие планировать потоки. В эту группу входят поток ядра (*Kernel Thread*), мьютекс (*Mutex*), событие (*Event*), семафор (*Semaphore*), таймер (*Timer*), таймер ожидания (*Waitable Timer*), и некоторые другие. С помощью функций ядра исполнительная система создает объекты ядра, манипулирует ими и конструирует более сложные объекты, предоставляемые в пользовательском режиме.

Не менее важной задачей ядра является абстрагирование или изоляция исполнительной системы и драйверов устройств от особенностей конкретной аппаратной реализации вычислительной системы. К особенностям конкретной аппаратной реализации относятся различия в обработке прерываний и исключений, синхронизация нескольких процессоров и т. п.

Архитектура ядра нацелена на максимальную унификацию кода, даже для аппаратно-зависимых функций. Интерфейсы, поддерживаемые ядром, унифицированы и могут использоваться в различных аппаратных платформах и конфигурациях.

Однако при всем стремлении разработчиков сделать интерфейсы аппаратно-независимыми, некоторая часть кода должна работать непосредственно с конкретной аппаратурой. Одним из примеров аппаратно-специфического кода может служить интерфейс, предоставляющий поддержку буфера трансляции процессорного кэша.

Различие кода здесь объясняется различной реализацией кэша в разных архитектурах.

Вторым примером может служить переключение контекста. Хотя алгоритм переключения контекста один и тот же (сохранить контекст предыдущего потока, загрузить контекст нового потока, запустить новый поток), однако, сам контекст зависит от аппаратной архитектуры (состояние процессора и его регистров), реализация этой функции зависит от аппаратуры.

Третьим примером может служить набор x86-специфичных интерфейсов для поддержки старых программ *MS-DOS*. Эти интерфейсы принципиально не являются переносимыми, так как в другой аппаратной архитектуре они просто отсутствуют.

Для того, чтобы операционная система Windows 2000 все-таки была переносимой между различными аппаратными архитектурами, в состав системы входит специальный компонент, который изолирует основные функции ядра системы от аппаратной конфигурации – уровень аппаратных абстракций.

## 2.5 Уровень аппаратных абстракций HAL

Уровень аппаратных абстракций (*HAL – Hardware Abstraction Layer*) является ключевым компонентом, обеспечивающим переносимость Windows 2000 между различными аппаратными архитектурами. *HAL* – это загружаемый модуль режима ядра (*Hal.dll*), предоставляющий низкоуровневый интерфейс с аппаратной платформой, на которой выполняется Windows 2000. Он скрывает от операционной системы специфику конкретной аппаратной платформы (интерфейсов ввода-вывода, контроллеров прерываний, механизмов взаимодействия между процессорами и т. д.), то есть, все функции, которые зависят от аппаратной архитектуры и конкретной ЭВМ.

Любая платформенно-зависимая информация, нужная внутренним компонентам Windows 2000 и драйверам устройств, получается от подпрограмм *HAL*, что и обеспечивает переносимость операционной системы. Именно поэтому подпрограммы *HAL* подробно описаны в Windows 2000 *DDK*.

В составе дистрибутива Windows 2000 имеется несколько модулей *HAL* (таблица 2.2), однако, при установке системы на жесткий диск компьютера копируется только один из модулей.

Таблица 2.2 – Список модулей *HAL*

Имя файла <i>HAL</i>	Поддерживаемые системы
Hal.dll	Стандартные персональные компьютеры (ПК)
Halacpi.dll	ПК с <i>ACPI</i> (Advanced Configuration and Power Interface – Расширенный интерфейс управления питанием и конфигурациями)
Halapic.dll	ПК с <i>APIC</i> (Advanced Programmable Interrupt Controller – Расширенный программируемый контроллер прерываний)
Halaacpi.dll	ПК с <i>ACPI</i> и <i>APIC</i>
Halmps.dll	Многопроцессорные ПК
Halmacpi.dll	Многопроцессорные ПК с <i>ACPI</i>
Halborg.dll	Рабочие станции Silicon Graphics (в настоящее время не выпускаются)
Halsp.dll	Compaq System Pro

Достаточно просто можно определить, какой именно модуль *HAL* используется в конкретном компьютере. Это можно сделать тремя способами:

1. Надо открыть файл `\Winnt\repair\setup.log`, найти в нем строку «...*hal.dll* =», и посмотреть имя, стоящее после знака равенства (например, "*halaacpi.dll*" – расшифровку см. в таблице 2.2).
2. Надо нажать правую кнопку мыши на значке *My Computer* (*Мой компьютер*) на рабочем столе, выбрать из контекстного меню пункт *Properties* (*Свойства*), открыть вкладку *Hardware* (*Оборудование*), нажать на кнопку *Device Manager* (*Диспетчер устройств*), выбрать устройство *Computer* (*Компьютер*) и посмотреть имя «драйвера устройства» (например, «Однопроцессорный компьютер с *ACPI*»).
3. Надо найти файл `\Winnt\System32\HAL.DLL`, нажать на нем правую кнопку мыши, выбрать пункт контекстного меню «*Свойства*», выбрать вкладку «*Версия*», и выбрать имя элемента «*Внутреннее имя*» – в поле «*Значение*» появится имя загруженного файла (например, *halaacpi.dll*).

## 2.6 Основные компоненты подсистемы ввода-вывода

Как уже говорилось, подсистема ввода-вывода в Windows 2000 состоит из нескольких компонентов исполнительной системы и драйверов устройств [1] (рисунок 2.3).

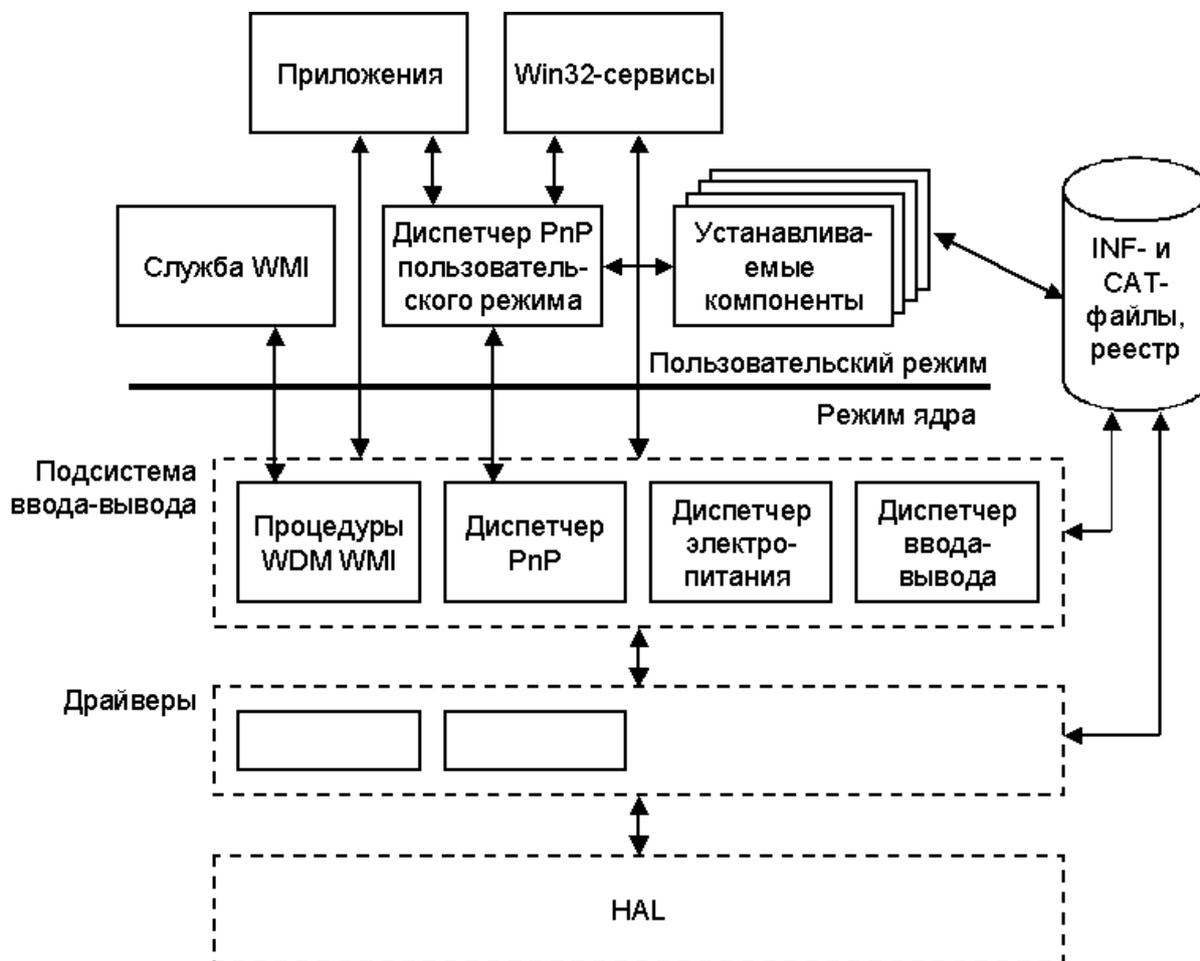


Рисунок 2.3 – Основные компоненты подсистемы ввода-вывода

Каждый из компонентов подсистемы ввода-вывода, изображенных на рисунке 2.3, выполняет свои функции.

- Диспетчер ввода-вывода подключает приложения и системные компоненты к виртуальным, логическим и физическим устройствам, а также определяет инфраструктуру, поддерживающую драйверы устройств.
- Драйвер устройства обычно предоставляет интерфейс ввода-вывода для устройств конкретного типа. Драйверы принимают от

диспетчера ввода-вывода команды для управляемых ими устройств и уведомляют диспетчер ввода-вывода о выполнении этих команд. Драйверы могут использовать диспетчер ввода-вывода для пересылки команд ввода-вывода другим драйверам, реализующим интерфейс того же устройства.

- Диспетчер PnP взаимодействует с диспетчером ввода-вывода и драйверами шин (*Bus Drivers*) – одной из разновидностей драйверов устройств. Он управляет выделением аппаратных ресурсов, распознает устройства и реагирует на их подключение и отключение. Диспетчер PnP и драйверы шин отвечают за загрузку соответствующего драйвера при обнаружении нового устройства. Если устройство добавляется в систему, в которой не установлен нужный драйвер устройства, компоненты исполнительной системы, отвечающие за поддержку PnP, вызывают сервисы установки устройств, поддерживаемый диспетчером PnP пользовательского режима.
- Диспетчер электропитания также взаимодействует с диспетчером ввода-вывода, управляет системой и драйверами устройств при их переходе в различные состояния энергопотребления.
- Процедуры поддержки *Windows Management Instrumentation* – Инструментарий управления Windows (*WMI*) образуют компонент доступа *WDM (Windows Driver Model) WMI*. Они позволяют драйверам устройств выступать в роли компонентов доступа, взаимодействуя со службой *WMI* пользовательского режима через компонент доступа *WDM WMI*.
- Реестр – это база данных, в которой хранится описание основных устройств, подключенных к системе, а также параметры инициализации драйверов и настройки конфигурации.

- *INF*-файлы используются для установки драйверов. Они связывают конкретное аппаратное устройство с драйвером, который и занимается управлением этим устройством. *INF*-файл состоит из инструкций, описывающих соответствующее устройство, исходное и целевое место нахождения файлов драйвера и дополнительную информацию о драйверах. *CAT*-файлы хранят цифровые подписи файлов драйверов, которые прошли испытание в лаборатории *Microsoft Windows Hardware Quality Lab (WHQL)*.
- Уровень аппаратных абстракций (*HAL*) изолирует драйверы от специфических особенностей конкретной аппаратуры.

Большинство операций ввода-вывода не требуют участия всех указанных компонентов. Обычно запрос на ввод-вывод выдается приложением, этот запрос обрабатывается диспетчером ввода-вывода, одним или несколькими драйверами устройств и *HAL*.

В Windows 2000 потоки выполняют операции ввода-вывода над виртуальными файлами. Операционная система абстрагирует все запросы на ввод-вывод, скрывая тот факт, что конечное устройство ввода-вывода может и не быть устройством с файловой структурой. Это позволяет унифицировать интерфейс между приложениями и устройствами. Таким образом, виртуальный файл сопоставляется с любым устройством ввода или вывода, который рассматривается как файл. Приложения пользовательского режима (в любой исполнительной подсистеме) вызывают документированные функции, которые обращаются к внутренним функциям подсистемы ввода-вывода для чтения/записи файла, или для других операций.

Запросы, адресованные виртуальным файлам, диспетчер ввода-вывода направляет соответствующим драйверам устройств. Примерная схема обработки запроса ввода-вывода приведена на рисунке 2.4.



Рисунок 2.4 – Схема обработки типичного запроса ввода-вывода

## 2.7 Диспетчер ввода-вывода

Диспетчер ввода-вывода (*I/O Manager*) определяет модель доставки запросов ввода-вывода драйверам устройств. Подсистема ввода-вывода управляется пакетами. Большая часть запросов ввода-вывода представляется пакетами запросов ввода-вывода (*I/O Request Packets, IRP*), передаваемых от одного компонента подсистемы ввода-вывода к другому. (Исключением является быстрый ввод-вывод, при котором *IRP* не используются.) Подсистема ввода-вывода позволяет потоку приложения управлять сразу несколькими запросами ввода-вывода. *IRP* – это структура данных, которая содержит информацию, описывающую запрос ввода-вывода. Она используется диспетчером ввода-вывода следующим образом.

- Диспетчер ввода-вывода создает *IRP*, представляющий операцию ввода-вывода, передает указатель на *IRP* соответствующему драйверу и удаляет пакет по завершении операции ввода-вывода.
- Драйвер, получивший *IRP*, выполняет указанную в пакете операцию и возвращает *IRP* диспетчеру ввода-вывода.
- Диспетчер ввода-вывода либо завершает эту операцию, либо передает пакет другому драйверу для дальнейшей обработки.

Диспетчер ввода-вывода не только создает и уничтожает *IRP*, но и содержит общий для различных драйверов код, используемый ими при обработке ввода-вывода. Благодаря этому коду драйверы стали проще и компактнее. Например, одна из таких функций диспетчера ввода-вывода позволяет драйверу вызывать другие драйверы. Диспетчер управляет также буферами ввода-вывода и тайм-аутами драйверов, регистрирует устанавливаемые в систему файловые системы. Диспетчер ввода-вывода предоставляет драйверам около сотни готовых функций.

Диспетчер ввода-вывода предоставляет также гибкие сервисы ввода-вывода, используя которые подсистемы окружения (*Win32*, *POSIX*) реализуют свои функции. В их число входят, например, сервисы асинхронного ввода-вывода, позволяющие создавать масштабируемые серверные приложения.

Унификация и модульность интерфейса драйверов позволяет диспетчеру ввода-вывода вызывать любой драйвер, ничего не зная о его внутреннем устройстве и структуре. Как уже говорилось, операционная система обрабатывает запросы ввода-вывода, как работу с файлами. Драйвер преобразует запросы к виртуальному файлу в аппаратно-специфические запросы. Драйверы могут вызывать друг друга (через диспетчер ввода-вывода), обеспечивая многоуровневую независимую обработку запросов ввода-вывода.

Кроме стандартных функций открытия, закрытия, чтения и записи подсистема ввода-вывода Windows 2000 предоставляет ряд дополнительных функций, например, асинхронного, прямого и буферизованного ввода-вывода.

## 2.8 Драйверы устройств

Драйверы устройств являются загружаемыми модулями режима ядра (обычно это файлы с расширением *.SYS*). Драйверы образуют интерфейс между диспетчером ввода-вывода и соответствующим оборудованием. Эти драйверы выполняются в режиме ядра в одном из трех контекстов:

- в контексте пользовательского потока, инициировавшего функцию ввода-вывода,
- в контексте системного потока режима ядра,
- как результат прерывания (то есть, не в контексте процесса или потока, который был текущим на момент прерывания).

Ранее уже говорилось о том, что в Windows 2000 драйверы не работают непосредственно с оборудованием – они вызывают функции *HAL*. Драйверы обычно пишутся на C или на C++, поэтому при грамотном использовании процедур *HAL* они легко переносятся между архитектурами, поддерживаемыми Windows 2000 на уровне исходного кода. На уровне двоичных файлов они могут переноситься внутри семейства с одинаковой архитектурой. В Windows 2000 имеется несколько видов драйверов устройств [1].

- *Драйверы аппаратных устройств*, управляющие (через *HAL*) оборудованием. Они получают от физического устройства или из сети данные ввода или записывают в данные вывода. Такими драйверами являются драйверы шин, интерфейсов, устройств массовой памяти и т. д.
- *Драйверы файловой системы* – это драйверы Windows 2000, обрабатывающие запросы на файловый ввод-вывод и транслирующие их в запросы ввода-вывода для конкретных устройств,

- *Драйверы фильтра файловой системы*, которые отвечают за зеркалирование и шифрование дисков, перехват ввода-вывода и дополнительную обработку информации перед передачей ее на следующий уровень.
- *Сетевые редиректоры и серверы*, являющиеся драйверами файловых систем, которые передают запросы файловой системы на ввод-вывод другим компьютерам в сети и принимают от них аналогичные запросы.
- *Драйверы протоколов*, реализующие сетевые протоколы *NCP/IP*, *NetBEUI* и *IPX/SPX*.
- *Драйверы потоковых фильтров ядра*, предназначенные для обработки потоковых данных, например, при записи и воспроизведении аудио и видеоинформации.

Установка драйвера устройства – единственный способ добавить в систему дополнительный код режима ядра. Поэтому драйверы часто используются лишь для того, чтобы получить доступ к внутренним функциям или структурам данных, недоступных через *Win32 API*.

В Windows 2000 введена поддержка *Plug and Play (PnP)* и энергосберегающие технологии, а также расширена модель драйверов Windows NT, называемая *Windows Driver Model (WDM)*. Windows 2000 может работать и с драйверами, унаследованными от Windows NT, но, очевидно, в этом случае не будет *Plug and Play* и энергосберегающих технологий.

С точки зрения *WDM* существует три типа драйверов:

- *Драйвер шины (Bus Driver)*, обслуживающий контроллер шины, адаптер, мост или любые другие устройства, имеющие дочерние устройства. Для каждого типа шины (*PCI*, *PCMCIA*, *USB*) в системе имеется свой драйвер. Для поддержки новых шин

(*VMEbus, Multibus, Futurebus*) используются драйверы сторонних разработчиков.

- *Функциональный драйвер (Function Driver)* – основной драйвер устройства, предоставляющий его функциональный интерфейс. Функциональный драйвер обладает наиболее полной информацией о своем устройстве. Обычно только этот драйвер имеет доступ к специфическим регистрам устройства.
- *Драйвер фильтра (Filter Driver)* поддерживает дополнительную функциональность устройства (или драйвера) или изменяет запросы ввода-вывода и ответы на них от других драйверов. Такие драйверы не обязательны. Их может быть несколько. Эти драйверы могут работать, как на более высоком уровне, чем функциональный драйвер или драйвер шины,

В среде *WDM* драйвер шины информирует диспетчер *PnP* об устройствах, подключенных к шине, а функциональный драйвер управляет устройством.

Драйвер фильтра низшего уровня обычно модифицирует поведение устройства, а драйвер фильтра высшего уровня придает устройству дополнительную функциональность. Так, драйвер низшего уровня может изменить требования устройства к аппаратным ресурсам, направляемые диспетчеру *PnP*, а драйвер высшего уровня для клавиатуры может обеспечить дополнительную защиту данных.

## 2.9 Резюме

В этой главе показаны общие аспекты системной архитектуры Windows 2000. Рассмотрены ключевые компоненты Windows 2000 и принципы их взаимодействия.

Подсистема ввода-вывода операционной системы Windows 2000, являющаяся одной из важнейших, состоит из ряда компонентов исполнительной системы ОС, которые управляют аппаратными средствами и предоставляют интерфейсы для обращения к ним системе и приложениям пользователя.

Особого внимания заслуживает элемент «*DLL* подсистем». Его присутствие связано с тем, что в Windows 2000 пользовательские приложения не могут напрямую вызывать встроенные сервисы операционной системы – они работают через одну или несколько *DLL подсистем*, которые предназначены для трансляции документированных функций в соответствующие недокументированные внутренние вызовы системных сервисов Windows 2000.

В Windows 2000 имеется три подсистемы окружения: *Win32*, *POSIX* и *OS/2*. Подсистема *Win32* здесь стоит на первом месте не случайно – без нее Windows 2000 работать не может. Эта подсистема обрабатывает все, что связано с клавиатурой, мышью и экраном. Она нужна даже на тех серверах, у которых нет интерактивных пользователей. Подсистема *Win32* работает всегда, а остальные две подсистемы запускаются только по требованию.

Ядро Windows 2000 состоит из набора фундаментальных функций (в том числе, планирование потоков и синхронизация), которые расположены в файле *Ntoskrnl.exe*, и используются компонентами исполнительной системы и низкоуровневыми (аппаратно-зависимыми)

средствами – диспетчерами прерываний и исключений, которые непосредственно «привязаны» к конкретной аппаратной платформе.

Ядро состоит из низкоуровневых примитивов и механизмов операционной системы, поддерживающих функции компонентов исполнительной системы более высокого уровня. Ядро отделено от исполнительной системы – оно реализует системные механизмы, но не участвует в принятии решений, связанных с системной политикой. Все такие решения, кроме планирования и диспетчеризации потоков, принимаются исполнительной системой.

Ключевым компонентом, обеспечивающим переносимость Windows 2000 между различными аппаратными архитектурами, является уровень аппаратных абстракций *HAL*. *HAL* – это загружаемый модуль режима ядра, предоставляющий низкоуровневый интерфейс с аппаратной платформой, на которой выполняется Windows 2000. Он скрывает от операционной системы специфику конкретной аппаратной платформы (интерфейсов ввода-вывода, контроллеров прерываний, механизмов взаимодействия между процессорами и т. д.), то есть, все функции, которые зависят от аппаратной архитектуры и конкретной ЭВМ.

Драйверы устройств являются загружаемыми модулями режима ядра, которые образуют интерфейс между диспетчером ввода-вывода и соответствующим оборудованием. Эти драйверы выполняются в режиме ядра в контексте пользовательского потока, инициировавшего функцию ввода-вывода, в контексте системного потока режима ядра или как результат прерывания (то есть, не в контексте процесса или потока, который был текущим на момент прерывания).

В Windows 2000 введена поддержка *PnP* и энергосберегающие технологии, а также расширена модель драйверов *WDM*. Windows 2000

может работать и с драйверами, унаследованными от Windows NT, но, очевидно, в этом случае не будет *PnP* и энергосберегающих технологий.

## 2.10 Контрольные вопросы

1. На какие два класса разделяются элементы операционной системы Windows 2000?
2. Чем отличаются потоки пользовательского режима от потоков ядра?
3. Какие типы пользовательских процессов существуют?
4. Для чего нужны *DLL* подсистем?
5. Какие компоненты режима ядра есть в Windows 2000?
6. Что такое исполнительная система?
7. Для чего нужно ядро?
8. Для чего нужны драйверы?
9. Что такое уровень аппаратных абстракций?
10. Что такое подсистема окружения?
11. Какие подсистемы окружения есть в Windows 2000?
12. Когда запускаются подсистемы окружения?
13. Какая подсистема окружения Windows 2000 является главной?
14. Из каких подсистем состоит Win32?
15. Для чего нужна подсистема POSIX?
16. Для чего нужна подсистема OS/2?
17. Что такое Remote Procedure Call?
18. Для чего нужен модуль *Ntdll.dll*?
19. Что такое диспетчер системных сервисов?
20. Что такое внутренние функции поддержки?
21. Что такое исполнительная система?
22. Для чего нужна исполнительная система?
23. Из чего состоит исполнительная система?
24. Что такое экспортируемые функции, доступные из пользовательского режима?
25. Что такое экспортируемые функции, доступные из режима ядра?

26. Что такое диспетчер конфигурации?
27. Для чего нужен справочный монитор безопасности?
28. Что делает диспетчер ввода-вывода?
29. Что такое диспетчер *Plug and Play*?
30. Для чего нужен диспетчер электропитания?
31. Что такое Windows Management Instrumentation?
32. Для чего нужен диспетчер кэша?
33. Что такое диспетчер виртуальной памяти?
34. Для чего нужен диспетчер объектов?
35. Что делает механизм *Local Procedure Call*?
36. Из чего состоит ядро Windows 2000?
37. Для чего нужно ядро?
38. Что такое объекты ядра?
39. Что такое управляющие объекты?
40. Что такое *Asynchronous Procedure Call*?
41. Что такое *Deferred Procedure Call*?
42. Для чего нужны объекты диспетчера?
43. В чем состоит главная задача ядра?
44. В чем специфика интерфейсов, поддерживаемых ядром?
45. Для чего нужен уровень аппаратных абстракций?
46. Как определить, какой модуль *HAL* используется в компьютере?
47. Из чего состоит подсистема ввода-вывода Windows 2000?
48. Что такое диспетчер ввода-вывода?
49. Что такое реестр?
50. Для чего нужны *INF*-файлы?
51. Для чего нужны *CAT* файлы?
52. Из чего состоит типичная схема обработки запроса ввода-вывода?
53. Что такое пакет запроса ввода-вывода?

54. Как диспетчер ввода-вывода обрабатывает *IRP*?
55. Что такое драйвер устройства?
56. В каком режиме работают драйверы устройств?
57. В каком контексте выполняются драйверы?
58. Как драйверы устройств Windows 2000 работают с оборудованием?
59. Какие типы драйверов устройств есть в Windows 2000?
60. Что такое драйвер файловой системы?
61. Для чего нужны драйверы фильтра файловой системы?
62. Что такое драйверы протоколов?
63. Для чего нужны драйверы потоковых фильтров ядра?
64. Как можно добавить в систему код режима ядра?
65. Что такое *Plug and Play*?
66. Что такое *WDM*?
67. Что такое драйвер шины?
68. Что такое функциональный драйвер?
69. Что такое драйвер фильтра?
70. Что такое унаследованный драйвер?

### 3 СОЗДАНИЕ ДРАЙВЕРОВ WINDOWS 2000

В Windows 2000, как и во всех операционных системах семейства NT драйверы бывают следующих типов:

- Драйверы режима ядра (*Kernel mode drivers*). Основной тип драйвера. Если точно неизвестно, какого именно типа драйвер нужен, используют именно этот тип.
- Графические драйверы (*Graphics drivers*). Драйверы видеокарт. Обычно они создаются одновременно с самой видеокартой. Очень сложны в написании, так как должны учитывать множество противоречивых требований и поддерживать множество стандартов.
- Мультимедийные драйверы (*Multimedia drivers*). Драйверы для:
  - аудиоустройств – считывание, воспроизведение и сжатие аудиоданных.
  - устройств работы с видео – захват и сжатие видеоданных.
  - позиционных устройств – джойстики, световые перья, планшеты и пр.
- Сетевые драйверы (*Network drivers*) – работа с сетью и сетевыми протоколами на всех уровнях.
- *Virtual DOS Drivers* – драйверы для виртуальных машин MS-DOS.

Наибольший интерес для программиста представляют драйверы режима ядра. Прежде всего, необходимо представить окружение драйвера, среду, в которой он работает.

Как уже было показано, ядро системы представляется набором отдельных изолированных модулей с четко определенными внешними интерфейсами. Все драйверы NT имеют множество *стандартных методов драйвера*, определенных системой, и, возможно, несколько специфических методов, определенных разработчиком.

Существует три типа драйверов ядра, каждый тип имеет четко определенные структуру и функциональность.

- Драйверы устройств (*Device drivers*), такие как драйвер клавиатуры или дисковый драйвер, напрямую общающийся с дисковым контроллером. Эти драйвера также называются *драйверами низкого уровня*, т. к. они находятся в самом низу цепочки драйверов Windows NT.
- Промежуточные драйверы (*Intermediate drivers*), такие как драйвер виртуального или зеркального диска. Они используют драйверы устройств для обращения к аппаратуре.
- Драйверы файловых систем (*File system drivers*). Драйверы файловых систем, таких как FAT, NTFS, CDFS, для доступа к аппаратуре используют промежуточные драйверы и драйверы устройств.

Драйверы Windows NT должны удовлетворять следующим требованиям:

- переносимость с одной платформы на другую,
- программная конфигурируемость,
- прерываемость,
- мультиплатформенность,
- объектно-ориентированность,
- поддержка пакетного ввода-вывода с повторно используемыми *IRP* (запросами ввода-вывода),
- поддержка асинхронного ввода-вывода.

Каждая операционная система имеет модель ввода-вывода для управления потоками данных к периферийным устройствам и от них.

Модель ввода-вывода Windows NT имеет следующие особенности:

- менеджер ввода-вывода NT представляет интерфейс для всех драйверов режима ядра, включая драйверы физических устройств, драйверы логических устройств и драйверы файловых систем,
- операции ввода-вывода послонные, то есть, что вызов, сделанный пользователем, проходит через несколько слоев, генерируя несколько пакетных запросов ввода-вывода и обращаясь к необходимым драйверам,
- менеджер ввода-вывода определяет множество стандартных процедур, которые должны быть реализованы разработчиком драйвера,
- подобно NT в целом, драйверы имеют объектную архитектуру, то есть, их устройства и системное оборудование представлены как объекты Windows NT.

### 3.1 Стандартные процедуры

Любой драйвер должен реализовывать основное множество процедур и, возможно, еще несколько дополнительных подмножеств в зависимости от разновидности драйвера. На рисунке 3.1 показан стандартный цикл работы драйвера, заключающийся в обработке запроса на прерывание *IRP*.

Ниже показаны основные процедуры, которые должен иметь каждый драйвер.

#### DriverEntry

```
NTSTATUS
(*PDRIVER_INITIALIZE) (
    IN PDRIVER_OBJECT DriverObject,
    IN PUNICODE_STRING RegistryPath
);
```

Каждый драйвер должен иметь инициализационную процедуру, которую менеджер ввода-вывода вызывает автоматически, если эта процедура называется *DriverEntry*.

#### Dispatch

```
NTSTATUS
(*PDRIVER_DISPATCH) (
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP Irp
);
```

Каждый драйвер должен иметь, по крайней мере, одну процедуру *Dispatch*.

#### StartIo (или Queue-management)

```
VOID
(*PDRIVER_STARTIO) (
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP Irp
);
```

Если драйвер устройства не может завершить все возможные запросы ввода-вывода в его *Dispatch* процедуре, он должен иметь либо процедуру *StartIo*, либо заводить одну или более внутренних очередей и управлять собственным механизмом отложенных запросов на прерывание.

### 3.2 Дополнительные стандартные процедуры

В зависимости от типа и от уровня, занимаемого драйвером в стеке обработки запроса на прерывание, драйвер может обладать следующими дополнительными процедурами:

#### Reinitialize

```
VOID
(*PDRIVER_REINITIALIZE) (
    IN PDRIVER_OBJECT DriverObject,
    IN PVOID Context,
    IN ULONG Count
);
```

Вдобавок к процедуре *DriverEntry* драйвер может иметь процедуру *Reinitialize*, вызываемую один или несколько раз в процессе загрузки системы после того, как *DriverEntry* вернет управление.

#### InterruptService (ISR)

```
BOOLEAN
(*PKSERVICE_ROUTINE) (
    IN PKINTERRUPT Interrupt,
    IN PVOID ServiceContext // usually points to device object
);
```

Любой драйвер физического устройства, который генерирует прерывания, должен иметь эту процедуру. Этот драйвер всегда самый низкий в стеке.

#### DpcForIsr или CustomDpc

```
VOID
(*PIO_DPC_ROUTINE) (
    IN PKDPC Dpc,
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP Irp,
    IN PVOID Context
);

VOID
(*PKDEFERRED_ROUTINE) (
    IN PKDPC Dpc,
    IN PVOID DeferredContext,
    IN PVOID SystemArgument1,
    IN PVOID SystemArgument2
);
```

Любой драйвер, имеющий *ISR* должен иметь *DpcForIsr* или *CustomDpc*.

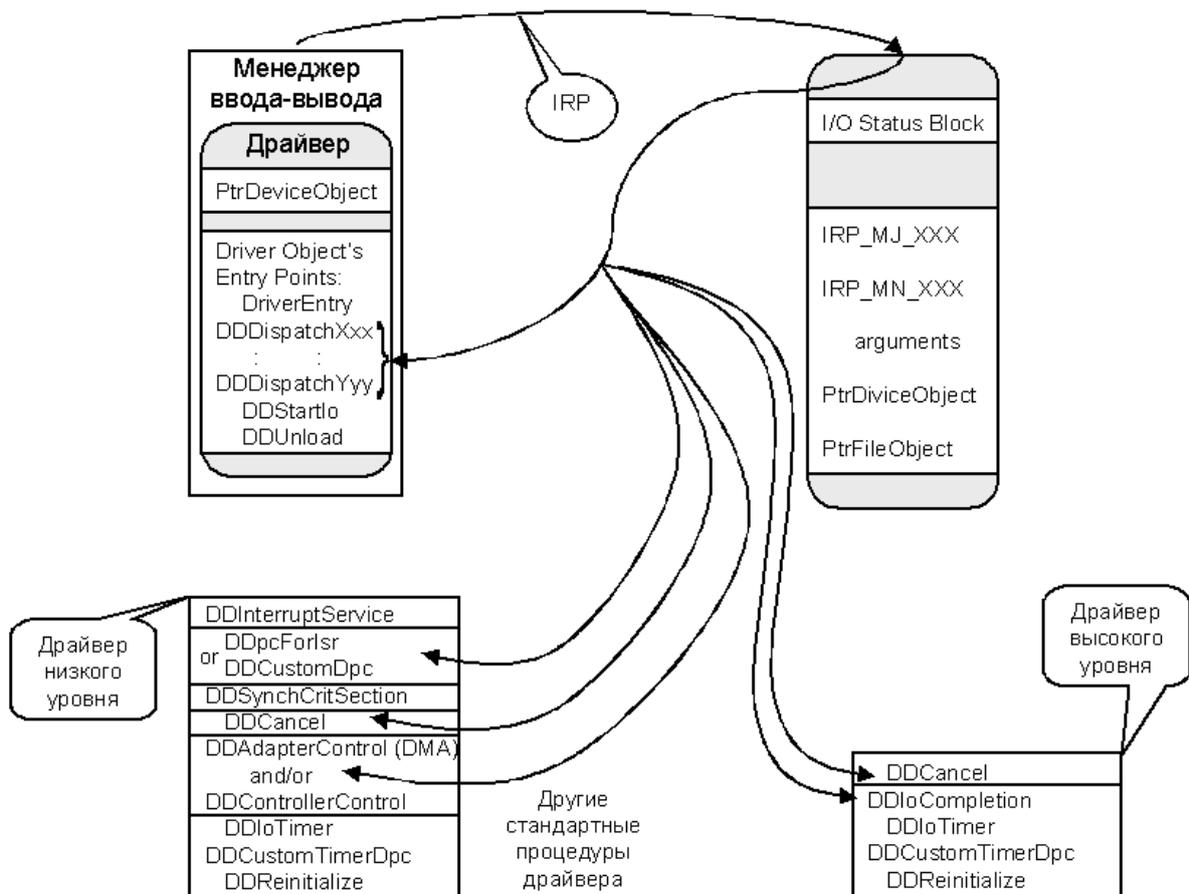


Рисунок 3.1 – Стандартные процедуры драйвера

### SynchCritSection

```

BOOLEAN
(*PKSYNCHRONIZE_ROUTINE) (
    IN PVOID SynchronizeContext
);

```

Любой низкоуровневый драйвер устройства, у которого данные или регистры сопряженного устройства могут изменяться в его *ISR* и других процедурах драйвера, должен иметь одну или более процедур *SynchCritSection*.

### AdapterControl и/или ControllerControl

```

IO_ALLOCATION_ACTION
(*PDRIVER_CONTROL) (
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP Irp,
    IN PVOID MapRegisterBase,

```

```
IN PVOID Context
);
```

Любой драйвер устройства, использующий *DMA*, должен иметь процедуру *AdapterControl*. Любой драйвер устройства, который должен синхронизировать операции с физическим контроллером для нескольких устройств или каналов устройства, должен иметь *ControllerControl*.

### Cancel

```
VOID
(*PDRIVER_CANCEL) (
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP Irp
);
```

Клавиатура, мышь, последовательный, параллельный, звуковой драйверы и драйвер файловой системы имеют процедуру *Cancel*. Любой драйвер, обрабатывающий запрос в течение длительного промежутка времени (в течение которого пользователь может отменить операцию), должен иметь процедуру *Cancel*. Обычно эту процедуру имеет высший драйвер в стеке обработки запроса.

### IoCompletion

```
NTSTATUS
(*PIO_COMPLETION_ROUTINE) (
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP Irp,
    IN PVOID Context
);
```

Любой драйвер верхнего уровня, который создает запросы к более низкоуровневым драйверам, должен иметь, по крайней мере, одну процедуру *IoCompletion* для освобождения всех структур *IRP*, созданных драйвером. Таким образом, любой драйвер высшего уровня должен иметь процедуру *IoCompletion*. Другие процедуры драйвера могут сказать, чтобы *IoCompletion* была вызвана, когда все низкоуровневые драйверы обработают текущий запрос.

## IoTimer и/или CustomTimerDpc

```
VOID
(*PIO_TIMER_ROUTINE) (
    IN PDEVICE_OBJECT DeviceObject,
    IN PVOID Context
);

VOID
(*PKDEFERRED_ROUTINE) (
    IN PKDPC Dpc,
    IN PVOID DeferredContext,
    IN PVOID SystemArgument1,           // reserved for system use
    IN PVOID SystemArgument2           // reserved for system use
);
```

Для отслеживания времени, занимаемого процедурой ввода-вывода, или для других целей, определяемых разработчиком, любой драйвер должен иметь процедуры *IoTimer* и/или *CustomTimerDpc*. *IoTimer* вызывается раз в секунду когда драйвер включает таймер. *CustomTimerDpc* может вызываться в более часто или нерегулярно интервал.

## Unload

```
VOID
(*PDRIVER_UNLOAD) (
    IN PDRIVER_OBJECT DriverObject
);
```

Драйвер должен иметь процедуру *Unload* если он может быть выгружен во время работы системы.

### 3.3 Сервисные системные вызовы

Для выполнения обращений к микроядру, работы с реестром, памятью, объектами, синхронизацией и пр. существует набор функций, называемых *функциями поддержки ядра*. Рассмотрим только самые необходимые.

#### IoCreateDevice

Создает новый объект устройства и инициализирует его для использования драйвером. Объект устройства представляет собой физическое, виртуальное или логическое устройство, которое необходимо драйверу для поддержки динамического управления этим устройством.

NTSTATUS

```
IoCreateDevice(  
    IN PDRIVER_OBJECT DriverObject, указатель на объект драйвера  
    IN ULONG DeviceExtensionSize, размер блока пользовательской информации в  
        байтах  
    IN PUNICODE_STRING DeviceName, имя устройства (иногда опускается)  
    IN DEVICE_TYPE DeviceType, тип устройства (последовательное, диск, мышь и  
        т.д.)  
    IN ULONG DeviceCharacteristics, параметры устройства (вынимаемое и пр.)  
    IN BOOLEAN Exclusive, параллельность доступа к устройству  
    OUT PDEVICE_OBJECT *DeviceObject, указатель на объект создаваемого  
        устройства  
);
```

#### IoCreateSymbolicLink

Создает символическую связь между устройством и видимым пользователем именем.

NTSTATUS

```
IoCreateSymbolicLink(  
    IN PUNICODE_STRING SymbolicLinkName, символическое имя, видимое  
        пользователю  
    IN PUNICODE_STRING DeviceName, имя устройства в пространстве имен ядра  
        Windows  
);
```

### *IoCompleteRequest*

Объявляет менеджеру ввода-вывода, что обработка текущего запроса ввода-вывода закончена.

VOID

*IoCompleteRequest*(

IN PIRP *Irp*, указатель на запрос ввода-вывода

IN CCHAR *PriorityBoost* повышение приоритета драйвера для обработки запроса. Зависит от обрабатываемого устройства.  
O\_NO\_INCREMENT при ошибке или очень быстрой обработке запроса

);

### 3.4 Драйвер виртуального диска

Рассмотрим конкретный пример – драйвер виртуального диска. По мере необходимости будем снабжать текст программы необходимыми описаниями.

Драйвер создает виртуальный диск, отформатированный по умолчанию как FAT. Выбор этого типа драйвера обусловлен следующим:

- простота реализации, не требуется кода для работы с оборудованием,
- легкость демонстрации, пример не требует специфических устройств, и может быть продемонстрирован практически на любой машине,
- малый объем исходного текста, позволяющий изучить основные принципы работы, не вдаваясь в подробности.

В целях экономии места и концентрации на основной теме опустим незначительные, но необходимые детали – инициализацию и освобождение памяти, получение параметров из реестра, обработку ошибок, – заменив их комментариями, вкратце описывающими процесс.

Начнем с рассмотрения функции *DriverEntry* – точки входа в драйвер, служащей для инициализации программы. С нее начинается выполнение драйвера. Функция устанавливает, какие системные вызовы она будет обрабатывать и создает собственно виртуальный диск.

NTSTATUS

```
DriverEntry(  
    IN OUT PDRIVER_OBJECT  DriverObject,  
    IN PUNICODE_STRING     RegistryPath  
)
```

Аргументы:

*DriverObject* – указатель на объект, представляющий этот драйвер.

*RegistryPath* – указатель на строку, задающую точку входа в реестре, отведенную для этого драйвера. Через реестр может осуществляться конфигурирование драйвера.

Возвращаемое значение:

*STATUS\_SUCCESS* если драйвер нормально инициализировался, иначе код ошибки

```
{
    NTSTATUS          ntStatus;
    UNICODE_STRING    paramPath;
    static WCHAR      SubKeyString[] = L"\\Parameters";
```

... Считывание параметров из реестра

...Инициализация объекта драйвера и его точек входа. Соответствующим элементам массива *DriverObject->MajorFunction* присваиваются адреса поддерживаемых функций.

```
    DriverObject->MajorFunction[IRP_MJ_CREATE]          =
    RamDiskCreateClose;
    DriverObject->MajorFunction[IRP_MJ_CLOSE]          =
    RamDiskCreateClose;
    DriverObject->MajorFunction[IRP_MJ_READ] = RamDiskReadWrite;
    DriverObject->MajorFunction[IRP_MJ_WRITE] = RamDiskReadWrite;
    DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] =
    RamDiskDeviceControl;

    ntStatus = RamDiskInitializeDisk(DriverObject, &paramPath);
    ...
    return ntStatus;
}
```

```
NTSTATUS
RamDiskInitializeDisk(
    IN PDRIVER_OBJECT  DriverObject,
    IN PUNICODE_STRING ParamPath
)
```

*RamDiskInitializeDisk* вызывается во время инициализации функцией *DriverEntry()*.

Создает и инициализирует объект устройства для диска. Память для образа выделяется в несвопируемой области. *RamDiskFormatFat* вызывается для создания файловой системы FAT.

Два параметра могут быть заданы при помощи реестра:

- *DiskSize* определяет размер виртуального диска в байтах. Если система не может выделить достаточно памяти, возвращается *STATUS\_INSUFFICIENT\_RESOURCES*. По умолчанию – 1 МБ.
- *DriveLetter* используется для указания имени диска. Строка должна быть буквой или буквой с двоеточием.

Аргументы:

*DriverObject* – указатель на объект, представляющий этот драйвер.

*ParamPath* – указатель на подключ *Parameters* реестра.

Возвращаемое значение:

*STATUS\_SUCCESS* если драйвер нормально инициализировался, иначе код ошибки

```
{
    STRING                ntNameString;  Имя устройства NT
    "\\Device\\RamDisk"
    UNICODE_STRING        ntUnicodeString;  Unicode версия ntNameString
    UNICODE_STRING        Win32PathString;  Имя Win32 "\\DosDevices\\Z:"

    PDEVICE_OBJECT        deviceObject = NULL;  Указатель на объект
устройства
    PRAMDISK_EXTENSION    diskExtension = NULL;  Указатель на
специфические
данные устройства

    NTSTATUS              ntStatus;

    ULONG                 defaultDiskSize = DEFAULT_DISK_SIZE;
    ULONG                 diskSize = DEFAULT_DISK_SIZE;
    UNICODE_STRING        driveLetterString;
    WCHAR                 driveLetterBuffer[sizeof(WCHAR) * 10];
}
```

... Инициализация и считывание параметров конфигурации из реестра

... Создание устройства «виртуальный диск»

```
ntStatus = IoCreateDevice (
    DriverObject,                Наш драйвер устройства
    sizeof( RAMDISK_EXTENSION ), Размер дополнительной информации
    &ntUnicodeString,            Имя устройства "\Device\RamDisk"
    FILE_DEVICE_VIRTUAL_DISK,   Тип устройства
    0,                           Свойства устройства
    FALSE,                       Особое устройство. Драйвер
                                обрабатывает
                                только одного клиента
                                Возвращает указатель на объект
                                устройства
    &deviceObject );
```

... Размещение и обнуление образа диска, установление данных boot сектора, корневого каталога и пр.

... Форматирование файловой системы FAT

```
RamDiskFormatFat (diskExtension, ParamPath);
```

... Создание символической связи между именем устройства "\Device\RamDisk" и именем Win32 "\DosDevices\Z:"

```
ntStatus = IoCreateSymbolicLink (
    &diskExtension->Win32NameString,
    &ntUnicodeString );
```

```
RamDiskInitializeDiskExit:
```

```
    return ntStatus;
}
```

Функция *RamDiskFormatFat* не представляет интереса. Она размечает выделенный образ диска в соответствии со стандартом файловой системы FAT.

NTSTATUS

```
RamDiskCreateClose (
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP Irp
)
```

Функция вызывается системой ввода-вывода каждый раз, когда *RamDisk* открывается или закрывается. Реализация ничего не выполняет, просто корректно закрывая запрос.

Аргументы:

*DeviceObject* – указатель на объект представляемого устройства

*Irp* – указатель на запрос ввода-вывода для этого вызова

Возвращаемое значение:

*STATUS\_INVALID\_PARAMETER* если параметры заданы неверно,  
иначе *STATUS\_SUCCESS*.

```
{  
    Irp->IoStatus.Status = STATUS_SUCCESS;  
    Irp->IoStatus.Information = 0;  
  
    IoCompleteRequest( Irp, IO_NO_INCREMENT );  
  
    return STATUS_SUCCESS;  
}
```

NTSTATUS

```
RamDiskDeviceControl (  
    IN PDEVICE_OBJECT DeviceObject,  
    IN PIRP Irp  
)
```

Функция реализует *ioctl* функции

Аргументы:

*DeviceObject* – указатель на объект устройства

*Irp* – указатель на запрос ввода-вывода

Возвращаемое значение:

*STATUS\_SUCCESS* если поддерживаемый запрос, иначе  
*STATUS\_INVALID\_DEVICE\_REQUEST*.

```
{  
    PRAMDISK_EXTENSION    diskExtension;  
    PIO_STACK_LOCATION    irpSp;  
    NTSTATUS               ntStatus;  
  
    ... Инициализация  
    По умолчанию – неверный запрос  
    Irp->IoStatus.Status = STATUS_INVALID_DEVICE_REQUEST;  
  
    Определяет, какая функция требуется  
    switch ( irpSp->Parameters.DeviceIoControl.IoControlCode )  
    {  
  
    case IOCTL_DISK_GET_MEDIA_TYPES:  
    case IOCTL_DISK_GET_DRIVE_GEOMETRY:  
        ... Возвращает параметры диска, так называемую геометрию
```

```

{
    PDISK_GEOMETRY outputBuffer;

    outputBuffer = ( PDISK_GEOMETRY ) Irp-
AssociatedIrp.SystemBuffer;
    outputBuffer->MediaType = RemovableMedia;
    outputBuffer->Cylinders = RtlConvertUlongToLargeInteger(
        diskExtension->NumberOfCylinders );
    outputBuffer->TracksPerCylinder =
        diskExtension->TracksPerCylinder;
    outputBuffer->SectorsPerTrack = diskExtension-
>SectorsPerTrack;
    outputBuffer->BytesPerSector = diskExtension-
>BytesPerSector;

    Irp->IoStatus.Status = STATUS_SUCCESS;
    Irp->IoStatus.Information = sizeof( DISK_GEOMETRY );
}
break;

case IOCTL_DISK_GET_PARTITION_INFO:
    ... Возвращает информацию о разделе
{
    PPARTITION_INFORMATION outputBuffer;
    PBOOT_SECTOR bootSector =
        (PBOOT_SECTOR) diskExtension->DiskImage;

    outputBuffer = ( PPARTITION_INFORMATION ) Irp-
>AssociatedIrp.SystemBuffer;

    outputBuffer->PartitionType =
        (bootSector->bsFileSystemType[4] == '6') ?
            PARTITION_FAT_16 : PARTITION_FAT_12;

    outputBuffer->BootIndicator = FALSE;
    outputBuffer->RecognizedPartition = TRUE;
    outputBuffer->RewritePartition = FALSE;
    outputBuffer->StartingOffset =
RtlConvertUlongToLargeInteger(0);
    outputBuffer->PartitionLength =
        RtlConvertUlongToLargeInteger(diskExtension-
>DiskLength);
    outputBuffer->HiddenSectors = 1L;

    Irp->IoStatus.Status = STATUS_SUCCESS;
    Irp->IoStatus.Information = sizeof( PARTITION_INFORMATION );
}
break;

case IOCTL_DISK_VERIFY:
{
    Выполняет проверку носителя. Операция идентична чтению
    PVERIFY_INFORMATION verifyInformation;

```

```

verifyInformation = Irp->AssociatedIrp.SystemBuffer;

irpSp->Parameters.Read.ByteOffset.LowPart =
    verifyInformation->StartingOffset.LowPart;
irpSp->Parameters.Read.ByteOffset.HighPart =
    verifyInformation->StartingOffset.HighPart;
irpSp->Parameters.Read.Length = verifyInformation->Length;

ntStatus = RamDiskReadWrite( DeviceObject, Irp );
}
return ntStatus;

default:
    Нераспознанная функция. Ошибка
break;
}

    Закончить операцию ввода-вывода.
ntStatus = Irp->IoStatus.Status;

IoCompleteRequest( Irp, IO_NO_INCREMENT );

return ntStatus;
}

```

NTSTATUS

```

RamDiskReadWrite (
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP Irp
)

```

Эта процедура вызывается для чтения и записи данных.

Аргументы:

*DeviceObject* – указатель объект драйвера

*Irp* – указатель на запрос ввода-вывода

Возвращаемое значение:

*STATUS\_INVALID\_PARAMETER* если параметр неверен, иначе  
*STATUS\_SUCCESS*.

```

{
    PRAMDISK_EXTENSION    diskExtension;
    PIO_STACK_LOCATION    irpSp;
    PUCCHAR                CurrentAddress;

```

... Инициализация и проверка на правильность параметра

... Получить указатель на данные пользователя в системном контексте

```

    CurrentAddress = MmGetSystemAddressForMdl( Irp->MdlAddress
);

Irp->IoStatus.Information = irpSp->Parameters.Read.Length;

switch (irpSp->MajorFunction)
{
case IRP_MJ_READ:
... Чтение
    RtlMoveMemory(
        CurrentAddress,
        diskExtension->DiskImage + irpSp-
>Parameters.Read.ByteOffset.LowPart,
        irpSp->Parameters.Read.Length);
    break;

case IRP_MJ_DEVICE_CONTROL:
... Проверка. Всегда все в порядке
    break;

case IRP_MJ_WRITE:
... Запись
    RtlMoveMemory(
        diskExtension->DiskImage + irpSp-
>Parameters.Read.ByteOffset.LowPart,
        CurrentAddress, irpSp->Parameters.Read.Length);
    break;

default:
... Что-то не поддерживаемое
    Irp->IoStatus.Information = 0;
    break;
}

Irp->IoStatus.Status = STATUS_SUCCESS;
IoCompleteRequest( Irp, IO_NO_INCREMENT );

return STATUS_SUCCESS;
}

VOID
RamDiskUnloadDriver(
    IN PDRIVER_OBJECT DriverObject
)

```

Эта процедура вызывается системой для выгрузки драйвера.  
Требуются освободить все занятые ресурсы

Аргументы:

*DriverObject* – указатель на объект драйвера

```
{  
    ... Освобождает всю выделенную для драйвера память.  
    Удаление объекта устройства. Символическая связь рвется автоматически  
        IoDeleteDevice( deviceObject );  
}
```

## *Некоторые важные сокращения*

- APC** (Asynchronous Procedure Call) – асинхронный вызов процедуры  
**ARPL** – инструкция выравнивания *RPL*  
**CPL** (Graphical User Interface) – графический интерфейс пользователя  
**CR<sub>n</sub>** (Hardware Abstraction Layer) – уровень аппаратных абстракций  
**DDK** (Driver Development Kit) – набор для разработки драйверов  
**DMA** (Direct Memory Access) – прямой доступ к памяти  
**DPC** (Deferred Procedure Call) – отсроченный вызов процедуры  
**DPL** (Descriptor Privilege Level) – уровень привилегий дескриптора  
**DR<sub>n</sub>** – регистр отладки *n*  
**EPL** (Effective Privilege Level) – эффективный уровень привилегий  
**GDI** – графический интерфейс устройства  
**GDT** (Global Descriptor Table) – глобальная таблица дескрипторов  
*gdt<sub>r</sub>* – регистр таблицы глобальных дескрипторов  
**GUI** – графический пользовательский интерфейс  
**HAL** – уровень аппаратных абстракций  
**HLT** – команда остановки процессора  
**IDT** (Interrupt Descriptor Table) – таблица дескрипторов прерываний  
*idtr* – регистр таблицы дескрипторов прерываний  
**IFS** (Installable File System) – встраиваемая файловая система  
**IOPL** – уровень привилегий ввода-вывода (в регистре флагов)  
**IRQ** (Interrupt Request) – запрос прерывания  
**LDT** (Local Descriptor Table) – локальная таблица дескрипторов  
*ldtr* – регистр таблицы локальных дескрипторов  
**LGDT** – команда загрузки таблицы глобальных дескрипторов  
**LIDT** – команда загрузки таблицы дескрипторов прерываний  
**LLDT** – команда загрузки таблицы локальных дескрипторов  
**LMSW** – команда загрузки слова состояния процессора  
**LPC** (Local Procedure Call) – локальный вызов процедуры  
**Mutex** (Mutual Exclusion) – мьютекс, объект, обеспечивающий взаимoisключающий доступ к ресурсу, Fast Mutex – это мьютекс, не допускающий рекурсивного захвата ресурсов  
**PAE** (Page Size Extension) – расширение размера страницы  
**PSE** (Physical Address Extension) – расширение физического адреса  
**RPC** (Remote Procedure Call) – стандарт сетевого программирования, позволяющий создавать приложения, состоящие из произвольного числа процедур, часть из которых выполняется локально, а часть – на удаленных компьютерах через сеть  
**RPL** – запрашиваемый уровень привилегий  
**TLB** (Translation Lookaside Buffer) – буфер ассоциативной трансляции  
**TR<sub>n</sub>** – регистр пошаговой отладки *n*  
**TSS** (Task State Segment) – сегмент состояния задачи  
**Сокет** – конечная точка коммуникационного соединения

## *Литература*

1. Соломон Д., Руссинович М. Внутреннее устройство Microsoft Windows 2000. Мастер-класс/Перев. с англ. – СПб.: Питер; М.: Издательско-торговый дом «Русская редакция», 2004. – 746 стр.: ил.
2. Несвижский В. Программирование аппаратных средств в Windows. – СПб: БХВ-Петербург, 2004. – 880 с.: ил.
3. Мюррей У., Паппас К. Создание переносимых приложений для Windows/ Пер. с англ. – СПб.: ВHV – Санкт-Петербург, 1998. – 816 с.: ил.
4. Гриноу Л. Философия программирования для Windows 95/NT/Пер. с англ. – СПб.: Символ-Плюс, 1998. – 640 с.: ил.
5. Солдатов В.П. Программирование драйверов Windows 95. – М.: ООО «Бином-Пресс», 2004. – 432 с.: ил.
6. Сорокина С.И. и др. Программирование драйверов и систем безопасности: Учеб. пособие / Сорокина С.И., Тихонов А.Ю., Щербаков А.Ю. – СПб.: БХВ-Петербург, М.: Издатель Молгачева С.В., 2002. – 256 с.: ил.
7. Финогенов К.Г. Разработка драйверов для Windows NT. – М.: 2001.
8. Karen Hazzah/ Writing Windows VxDs and Device Drivers/ – R&D Books, Lawrence, KS 66046, 1998. – 480 s.

*Учебное пособие*

*Роцин Алексей Васильевич*

Организация ввода-вывода.  
Часть 2. Драйверы для WINDOWS NT.  
Учебное пособие.

---

Подписано к печати \_\_.\_\_.2006 г.  
Формат 60×84 1/16.  
Объем 7,0 п.л. Тираж 300 экз. Заказ № \_\_

Отпечатано в типографии Московской государственной академии  
приборостроения и информатики  
107076, Москва, ул. Стромьнка 20.