Светлана Сорокина
Андрей Тихонов
Андрей Щербаков

Программирование драйверов и систем безопасности

Учебное пособие

Санкт-Петербург
«БХВ-Петербург»

Москва
Издатель Молгачева С. В.

2003

УДК 681.3.06
ББК 32.973.26-018
С65

Рецензенты:
Московский институт электроники и математики (технический университет),
Московское отделение Пензенского электротехнического института (ПНИЭИ)

Научная редакция:
д.ф.-м.н., проф., заведующий кафедрой "Криптология и дискретная математика" МИФИ,
начальник Департамента науки, высоких технологий, образования и культуры
аппарата Правительства РФ, академик РАО Подуфалов Н. Д.

Сорокина С. И. и др.

С65 Программирование драйверов и систем безопасности: Учеб. пособие / Сорокина С. И., Тихонов А. Ю., Щербаков А. Ю. — СПб.: БХВ-Петербург, М.: Издатель Молгачева С. В., 2003. — 256 с.: ил.

ISBN 5-94157-263-8

ISBN 5-94740-005-7

Учебное пособие содержит оригинальный научный и учебно-методический материал, посвященный созданию систем безопасности для операционных сред Windows NT/2000. Рассматриваются вопросы создания различных драйверов уровня ядра ОС, предназначенных для шифрования трафика и контроля доступа. Учебное пособие используется при организации занятий на факультете информационной безопасности МИФИ.

Для студентов и аспирантов соответствующих специальностей, разработчиков систем защиты информации, а также сотрудников экспертных организаций

УДК 681.3.06
ББК 32.973.26-018

ISBN 5-94157-263-8 (Изд-во "БХВ-Петербург")
ISBN 5-94740-005-7 (Издатель Молгачева С. В.)

Содержание

Предисловие **11**

Введение **12**

1. Что такое драйвер **15**

1.1. Типы драйверов и характеристики 15

1.2. Среда разработки 16

1.3. Утилита BUILD 19

1.4. C и C++. Интегрированная среда разработки 21

2. Общая архитектура Windows NT **22**

2.1. Понятия «пользовательский режим» и «режим ядра» 22

2.1.1. Некоторые понятия защищенного режима 22

2.2. Основные характеристики Windows NT 24

2.2.1. Модель модифицированного микроядра 24

2.2.2. Эмуляция нескольких ОС 25

2.2.3. Независимость от архитектуры процессора 26

2.2.4. Объектная модель 26

2.2.5. Многопоточность 27

2.2.6. Вытесняющая многозадачность (preemptive multitasking) 27

2.2.7. Виртуальная память с подкачкой страниц по требованию 28

2.2.8. Симметричная мультипроцессорная обработка 29

2.2.9. Интегрированная поддержка сети 29

2.3. Структура Windows NT 29

2.3.1. Защищенные подсистемы 31

2.3.2.1. Подсистемы среды 31

2.3.2.1.1. Подсистема среды Win32 32

2.3.2.1.2. «Родной» API для ОС Windows NT (Native Windows NT API) 33

2.3.2.2. Неотъемлемые подсистемы 35

2.3.3. Исполнительная система (The Executive) 35
2.3.3.1. Диспетчер Объектов 39
2.3.3.2. Система ввода/вывода 41
2.3.3.3. Ядро 42
2.3.3.4. Слой абстрагирования от оборудования 43
2.3.4. Система приоритетов 43
2.3.4.1. Уровни запросов прерываний (IRQL) 44
2.3.4.2. Приоритеты планирования 45
2.3.4.2.1.1. Динамические приоритеты и приоритеты реального времени 45
2.3.4.2.2. Базовый приоритет. Класс приоритета и относительный приоритет 46
2.3.4.3. Как используются IRQL 47
2.3.4.3.1. IRQL PASSIVE_LEVEL, APC_LEVEL и DISPATCH_LEVEL 49
2.3.4.3.1.1. Ограничения, налагаемые на код с уровнем IRQL большим или равным DISPATCH_LEVEL 50
2.3.4.3.2. DIRQLs 52
2.3.4.4. Прерывания и планирование 52
2.3.4.5. Определение текущего уровня IRQL 53
2.3.5. Архитектура памяти 54
2.3.5.1. Организация памяти в защищенном режиме работы процессора 54
2.3.5.2. Организация системного адресного пространства 57
2.3.5.3. Типы адресов в NT 58
2.3.5.4. Совместное использование памяти 58
2.3.5.5. Объект Секция 60
2.3.5.6. Таблица описания памяти (Memory Descriptor List, MDL) 60
2.3.5.7. Функции работы с памятью 61
2.3.5.7.1. Выделение памяти 62
2.3.5.7.2. Список заранее выделенных блоков памяти (Lookaside List) 62
2.3.5.7.3. Пространства ввода/вывода и отображение памяти устройств 63
2.3.5.7.4. Управление памятью и MDL 63

2.3.6. Унифицированная модель драйвера ... 64
2.3.6.1. Объект-файл (файловый объект) ... 65
2.3.6.2. Объект-драйвер ... 65
2.3.6.3. Объект-устройство ... 66
2.3.6.3.1. Имя устройства и символическая связь ... 67
2.3.6.4. Взаимосвязь основных объектов ... 67
2.4. Установка, удаление, запуск и остановка драйвера ... 69
2.4.1.1. Структура драйвера ... 70
2.4.1.1.1. Точки входа драйвера ... 70
2.4.1.1.2. Контекст исполнения и уровень IRQL ... 71
2.4.1.2. Ограничения, налагаемые на драйвер ... 72
2.4.1.3. Точка входа DriverEntry ... 72
2.4.1.3.1. Определение конфигурации аппаратного устройства ... 73
2.4.1.3.2. Создание объекта-устройства и символической связи ... 73
2.4.1.4. Передача данных от приложения к драйверу. Асинхронная обработка ... 74
2.4.1.4.1. Выполнение асинхронного запроса ... 76
2.4.2. Пакет запроса ввода/вывода (IRP) ... 78
2.4.2.1. Характеристики подсистемы ввода/вывода ... 78
2.4.2.2. Структура пакета запроса ввода/вывода (IRP) ... 79
2.4.2.2.1. Поля в фиксированной части IRP ... 80
2.4.2.2.2. Поля в стеке размещения ввода/вывода IRP ... 81
2.4.2.3. Описание буфера данных ... 82
2.4.2.4. Коды функции ввода/вывода ... 84
2.4.2.5. Диспетчерские точки входа драйвера ... 85
2.4.2.5.1. Запросы чтения и записи IRP_MJ_READ и IRP_MJ_WRITE ... 86
2.4.2.5.1.1. Пример обработки запросов чтения/записи ... 87
2.4.2.5.2. Запросы IRP_MJ_DEVICE_CONTROL и IRP_MJ_INTERNAL_DEVICE_CONTROL ... 87
2.4.2.5.2.1. Задание кода управления вводом/выводом (IOCTL) ... 88
2.4.2.5.2.2. Получение буфера ... 90
2.4.2.5.2.3. Пример обработки ... 92

2.4.3. Многоуровневая модель драйверов ... 93
2.4.3.1. Реализация уровневых драйверов ... 93
2.4.3.1.1. Объединение драйверов в стек ... 94
2.4.3.1.2. Обработка запросов IRP стеком драйверов ... 95
2.4.3.1.2.1. Получение драйвером вышележащего уровня уведомления о завершении обработки IRP драйвером нижележащего уровня ... 97
2.4.3.2. Реализация драйверов-фильтров ... 98
2.4.3.2.1. Подключение фильтра к устройству ... 98
2.4.4. Сериализация ... 100
2.4.4.1. Задержка обработки запросов IRP и постановка запросов IRP в очередь ... 101
2.4.4.1.1. Системная очередь запросов IRP (System Queuing) ... 101
2.4.4.1.1.1. Обработка пакетов IRP в функции StartIo ... 103
2.4.4.1.2. Очереди, управляемые драйвером ... 104
2.4.4.1.2.1. Функции управления очередью низкого уровня ... 104
2.4.4.1.2.2. Функции управления очередью высокого уровня – «Очередь Устройства» (Device Queue) ... 106
2.4.4.2. Отмена запросов ввода/вывода ... 106
2.4.4.2.1. Отмена IRP и Системная Очередь ... 107
2.4.4.2.2. Отмена IRP и очереди, управляемые драйвером ... 108
2.4.5. Механизмы синхронизации ... 109
2.4.5.1. Спин-блокировки ... 109
2.4.5.1.1. Использование обычных спин-блокировок ... 110
2.4.5.1.2. Проблема взаимоблокировок (deadlocks) ... 111
2.4.5.2. Диспетчерские объекты ... 111
2.4.5.2.1. Ожидание (захват) диспетчерских объектов ... 113
2.4.5.2.2. Мьютексы ядра ... 113
2.4.5.2.3. Семафоры ... 114
2.4.5.2.4. События ... 115
2.4.5.2.5. Быстрые мьютексы ... 116
2.4.5.3. Ресурсы Исполнительной системы ... 116
2.4.5.4. Обобщенная таблица механизмов синхронизации ... 117
2.4.6. Рабочие потоки ... 119
2.4.6.1. Необходимость в создании рабочих потоков ... 119

2.4.6.2. Системные рабочие потоки ... 119
2.4.6.3. Создание потоков драйвером ... 121
2.4.6.4. Потоки как диспетчерские объекты ... 121
2.4.7. Введение в обработку прерываний ... 124
2.4.7.1. Объекты – прерывания ... 124
2.4.7.2. Отложенный вызов процедуры (Deferred Procedure Call, DPC) ... 126
2.4.7.2.1. DPC-объекты ... 128
2.4.7.2.2. Активизация и обслуживание DPC ... 129
2.4.7.2.3. Многочисленные обращения к DPC ... 130
2.4.7.2.4. DPC на многопроцессорных системах ... 130
2.4.7.2.5. Характеристики Объекта DPC ... 131
2.4.7.2.5.1. Важность DPC (DPC Importance) ... 131
2.4.7.2.5.2. Целевой процессор для DPC (DPC Target Processor) ... 132
2.4.7.2.6. DpcForIsr ... 133

3. Сетевая архитектура Windows NT ... 134
3.1. Соответствие сетевой архитектуры Windows NT модели OSI ... 134
3.2. Сетевые API ... 138
3.2.1. Стандартный API ввода/вывода Win32 ... 138
3.2.2. Сетевые API Win32 (Wnet и Net) ... 138
3.2.3. API NetBIOS (Network Basic Input/Output System) ... 139
3.2.4. Win32 API именованных каналов (named pipes) и почтовых ящиков (mailslots) ... 140
3.2.5. API Windows Sockets ... 142
3.2.6. Средство удаленного вызова процедур (Remote Procedure Call, RPC) ... 143
3.2.7. Средство динамического обмена данными (Network Dynamic Data Exchange, NetDDE) ... 144
3.2.8. Сервисы удаленного доступа (Remote Access Services, RAS) ... 146
3.2.9. TAPI (Telephony Application Programming Interface) ... 148
3.2.10. Интерфейс DLC (Data Link Control) ... 150
3.2.11. Протокол SNMP (Simple Network Management Protocol) ... 151
3.2.12. DCOM ... 152

3.3. Сетевые программные компоненты ОС Windows NT 152
3.3.1. Сетевые сервисы 152
3.3.2. Сетевые файловые системы 153
3.3.2.1. Редиректор 153
3.3.2.2. Сетевой сервер 154
3.3.3. Распределенная файловая система (Distributed File System, DFS) 154
3.3.4. Маршрутизатор многосетевого доступа 155
3.3.5. Многосетевой UNC 156
3.3.6. Интерфейс TDI и транспортные протоколы 157
3.3.6.1. Интерфейс TDI 157
3.3.6.2. Транспортные протоколы 158
3.3.6.3. Среда STREAMS 160
3.3.7. Среда NDIS и NDIS драйверы 161
3.3.7.1. Среда NDIS 161
3.3.7.2. Типы NDIS драйверов 162
3.3.7.2.1. Драйверы протоколов верхнего уровня 162
3.3.7.2.2. Драйверы протоколов промежуточного уровня 163
3.3.7.2.2.1. Промежуточные драйверы ndistapi.sys и ndiswan.sys 163
3.3.7.2.3. Драйверы сетевых карт 164
3.3.8. Обобщенная схема сетевой архитектуры Windows NT 164
3.4. Связь между сетевыми программными компонентами 167

4. Анализ сетевой архитектуры ОС Windows NT с точки зрения возможностей реализации средств защиты и анализа сетевого трафика 169

4.1. Используемые средства построения объединенных сетей и их влияние на уровень расположения средства защиты (согласно модели OSI) 169
4.2. Объем информации, проходящей через средство защиты 170
4.3. Возможности реализации средств защиты сетевой информации на пользовательском уровне 171
4.3.1. Реализация защиты на уровне приложений и собственных DLL 172
4.3.2. Реализация защиты на уровне системных DLL, предоставляющих приложениям различные сетевые интерфейсы 173

4.3.3. Реализация защиты на уровне сетевых сервисов 176
4.3.4. Реализация защиты на уровне «родного» API для ОС Windows NT 177
4.4. Возможности реализации средств защиты сетевой информации на уровне ядра 178
4.4.1. Драйверы – фильтры 178
4.4.2. Реализация защиты на уровне драйвера MUP 181
4.4.3. Реализация защиты на уровне драйверов файловых систем 182
4.4.4. Реализация защиты на уровне транспортного драйвера 185
4.4.5. Реализация защиты с помощью перехвата функций NDIS – библиотеки 186
4.4.6. Реализация защиты на уровне сетевого драйвера промежуточного уровня, поддерживающего интерфейс NDIS 187
4.4.7. Реализация защиты на уровне драйверов сетевых устройств 189
4.5. Сравнительный анализ способов реализации защиты 190
4.5.1.1. Зависимость способа реализации средства защиты от предъявляемых к нему требований 192
4.6. Особенности реализации NDIS-драйверов 194
4.6.1. Синхронизация 195
4.6.2. Структура NDIS-пакетов 196
4.6.3. Завершающие функции 197
4.6.4. Точки входа промежуточного NDIS- драйвера 198
4.6.4.1. Точка входа DriverEntry 199
4.6.4.2. Точки входа ProtocolXxx 200
4.6.4.3. Точки входа MiniportXxx 201
4.7. Пример реализации драйвера шифрования сетевых пакетов 203

5. Общие вопросы обеспечения безопасности в операционной среде Windows NT/2000 210

5.1. Механизм идентификации и аутентификации в ОС Windows NT. Общее описание 210
5.2. Основные сведения о процессе Winlogon и его состояниях 211
5.3. Протокол взаимодействия процесса Winlogon и библиотеки GINA 212

5.4. Локальная аутентификация пользователя в Windows NT 225
5.5. Сетевая аутентификация пользователя в Windows NT 230
5.6. Основные подходы к созданию изолированной программной среды 230
5.7. Макет системы защиты от несанкционированного доступа 231
5.7.1. Драйвер контроля доступа 232
5.7.1.1. Перехват операций открытия, создания и удаления файлов 232
5.7.1.2. Собственный обработчик создания файла и собственный обработчик открытия файла 233
5.7.1.3. Определение имени процесса 234
5.7.1.4. Вывод сообщений на «синий» экран 235
5.7.1.5. Процедура распределения IRP_MJ_DEVICE_CONTROL драйвера контроля доступа 236
5.7.2. Модифицированная библиотека Gina 238

Список источников информации 242

Предисловие

Первая глава книги дает читателю общее представление о драйверах ядра операционной системы Windows NT/2000 и средствах их разработки.

Вторая глава книги посвящена обзору общей архитектуры Windows NT/2000, здесь рассматриваются ключевые архитектурные особенности, основные понятия ОС, а также основы программирования драйверов ядра для нее.

В третьей главе дано подробное описание сетевой архитектуры ОС Windows NT/2000, рассмотрено ее соответствие модели OSI, предоставляемые сетевые интерфейсы. Описаны сетевые программные компоненты, их назначения и взаимосвязи. Представлена обобщенная схема сетевой архитектуры.

Четвертая глава посвящена анализу сетевой архитектуры ОС Windows NT/2000 с точки зрения предоставляемых возможностей для реализации и встраивания средств защиты сетевой информации на различных уровнях сетевой архитектуры, начиная от высокоуровневых компонентов пользовательского режима и заканчивая низкоуровневыми компонентами режима ядра. Представлен сравнительный анализ различных способов реализации защиты и приведены основные факторы, влияющие на выбор конкретного способа реализации средства защиты.

Пятая глава посвящена общим вопросам обеспечения безопасности в ОС Windows NT/2000, в ней рассматриваются особенности штатной процедуры идентификации и аутентификации пользователей, а также вопросы синтеза системы защиты от несанкционированного доступа.

Цель книги – представить читателю основы программирования драйверов ядра для ОС Windows NT/2000 и ознакомить с возможными способами собственных реализаций средств защиты и анализа сетевого трафика.

Введение

Операционная система Windows NT и ее следующий представитель Windows 2000, благодаря своим современным принципам построения, защищенности, гибкости, а также встроенной сетевой поддержке и мощным сетевым возможностям, получила широкое распространение. Поэтому встает насущная проблема реализации систем защиты, которые могли бы встраиваться в ОС Windows NT, расширяя ее возможности и обеспечивая функции защиты сетевой информации.

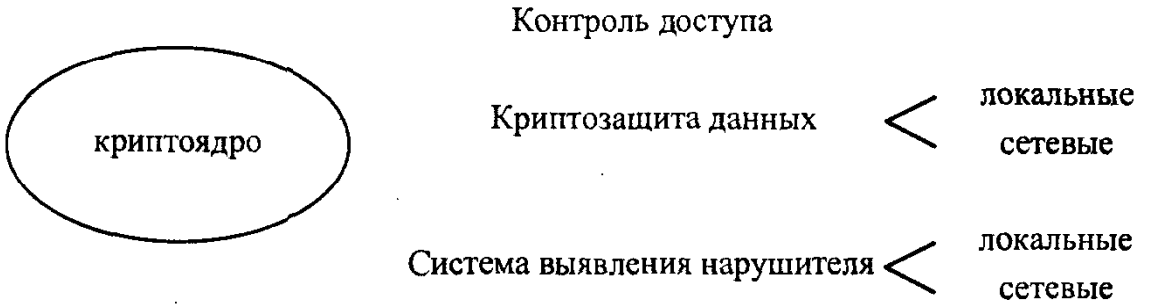
Отметим сразу, что базовая архитектура ядра ОС Windows NT практически не изменилась при переходе к Windows 2000, поэтому почти все, что описано в этой книге верно как для ОС Windows NT, так и для Windows 2000.

В книге помимо базовых основ написания драйверов, являющихся неотъемлемыми компонентами средств защиты информации, представлена общая и сетевая архитектура ОС Windows NT. Описание архитектуры необходимо для определения предоставляемых возможностей по реализации и встраиванию средств защиты сетевой информации, а также для сравнения возможных способов реализации защиты и определения наиболее предпочтительных способов. Исследование архитектуры ОС Windows NT позволяет определить не только то, как и куда можно встроить средство защиты, но и то, как этому средству предоставить наибольшие возможности со стороны операционной системы, поскольку от этого зависит решение конкретных задач по защите, которые оно сможет реализовать.

Знание архитектуры ОС и драйверов применительно к сфере защиты информации необходимо с различных точек зрения:

- С точки зрения средств защиты информации, в том числе и нестандартных (не упомянутых в документации к ОС).
- С точки зрения преодоления средств защиты.

Допустим, нам необходимо разработать средство защиты. Его типовая структура на данный момент выглядит примерно так:



Реализация почти всех перечисленных элементов системы защиты для ОС NT возможна только с применением драйверов:

- Защита локальных данных – либо драйвер шифрующей файловой системы, либо драйвер-фильтр для прозрачного шифрования, либо перехват вызовов системных сервисов.

- Защита сетевых данных – драйвер протокола, NDIS-драйвер промежуточного уровня, собственная сетевая служба, фильтр стандартной сетевой службы (такие службы реализованы как файловые системы), перехват вызовов системных сервисов.
- Выявление нарушителя для всех вышеприведенных вариантов – анализ событий, регистрация в журнале, запрос на подтверждение подозрительных действий.

Для специализированного оборудования должны быть разработаны драйверы для интеграции в эту схему.

В зависимости от конкретной постановки задачи должен быть выбран наиболее подходящий способ ее решения. Одного ответа на все случаи жизни быть не может, так как два основных критерия выбора решения:

- критерий максимальной простоты реализации;
- критерий максимальной универсальности

противоречат друг другу: чем более универсальным может быть вариант реализации, тем сложнее его реализация и наоборот.

В качестве примера можно привести задачу реализации защиты сетевого трафика. Наиболее универсальным (с точки зрения полноты контролируемого трафика) и, на первый взгляд, простым в этом случае будет являться NDIS-драйвер промежуточного уровня. Однако, по мере усложнения системы и приближения ее к коммерческой реализации, будут возникать более серьезные проблемы. Например, такая система должна знать форматы всех протоколов в системе, в том числе и тех, которые в данный момент даже не созданы.

При выборе способа реализации системы жизненно важным может быть также вопрос документированности этого способа, при этом надо учитывать, что большая часть ОС NT не документирована.

Драйверы ядра можно разбить на 2 больших класса: драйверы аппаратных устройств и драйверы виртуальных устройств:

1) архитектура NT исключает использование устройства, если для него нет драйвера. Прикладному ПО доступ к аппаратуре запрещен. Поэтому самым очевидным примером драйверов является драйвер аппаратного устройства, предоставляющий прикладным программам интерфейс доступа к устройству. В области защиты информации задача написания таких драйверов весьма актуальна.
2) Драйвер виртуального устройства не работает с каким либо специализированным аппаратным устройством, однако, предоставляет прикладным программам такие возможности по работе со стандартными ресурсами компьютера и ОС (процессор, память, порты, регистры, служебные структуры ОС), которые без драйвера были бы недоступны.

Что конкретно можно сделать с помощью драйвера виртуального устройства?

Как будет видно из следующих разделов, системная архитектура NT представляет собой набор модулей, связанных друг с другом стандартными, но далеко не всегда документированными интерфейсами. Благодаря этим интерфейсам можно производить как замену стандартных модулей на собственные, так и вставлять новые модули

в «разрыв» связей между старыми. Такое устройство ОС позволяет разрабатывать новые модули (драйверы) для различных целей:

- «прозрачная», то есть невидимая для прикладных программ, обработка данных, например, шифрование и компрессия данных на диске или в компьютерной сети;
- расширение набора предоставляемых ОС сервисов;
- «прозрачное» сканирование на наличие вирусов;
- написание вирусов и закладок;
- средства сбора статистики о событиях для различных компонентов системы.

Следует иметь в виду, что помимо общих правил разработки и взаимодействия драйверов существуют специальные правила для особых типов драйверов. В качестве примера можно привести драйверы файловой системы FSD и сетевые драйверы – архитектура NDIS и TDI.

Необходимо остановиться на состоянии изученности данной области. Руководств по разработке драйверов для ОС Windows NT, издаиных на русском языке, а также информации в области архитектуры ОС Windows NT гораздо меньше, чем для обычного программирования. В зарубежной литературе также очень мало книг, посвященных данной тематике. Что касается подмножества этой тематики – сетевой архитектуры, сетевых интерфейсов и сетевых драйверов, то соответствующей специализированной литературы практически нет. Основным источником информации является документация Microsoft Windows NT Device Driver Kit Version 4.0 и документация Microsoft Platform Software Development Kit.

Перечисленные источники дают информацию об общей и сетевой архитектуре, но никак не затрагивают вопросы реализации средств защиты сетевой информации.

При чтении этой книги настоятельно рекомендуется иметь под рукой:

- справочник по Visual C++;
- MSDN library;
- справочник по командам ассемблера.

1. Что такое драйвер

Понять, что такое драйвер, мы попробуем на типовом примере взаимодействия прикладной программы с драйвером.

Код прикладной программы исполняется в пользовательском режиме работы процессора. В этом случае имеется ряд серьезных ограничений, связанных с доступом к памяти, аппаратным обеспечением и привилегированными инструкциями процессора. Когда возникает необходимость в преодолении этих ограничений, прикладная программа обращается к ядру ОС, код которого исполняется процессором в режиме ядра. Режим ядра лишен всех упомянутых ограничений. Для расширения функциональных возможностей ядра служат драйверы ядра (kernel mode drivers). Как они работают?

В отличие от прикладной программы, драйвер не является процессом и не имеет потока исполнения. Вместо этого любая функция драйвера выполняется в контексте того потока и процесса, в котором она была вызвана. При этом вызов может происходить от прикладной программы или драйвера, либо возникать в результате прерывания. В первом случае контекст исполнения драйвера точно известен – это прикладная программа. В третьем случае контекст исполнения случайный, поскольку прерывание (и, соответственно, исполнение кода драйвера) может произойти при выполнении любой прикладной программы. Во втором случае контекст исполнения может быть как известным, так и случайным – это зависит от контекста исполнения функции вызывающего драйвера.

Под вызовом драйвера здесь подразумевается не обычный вызов функции, а передача так называемого запроса ввода/вывода.

Различают несколько классов драйверов:

- Драйвер, получающий запросы ввода/вывода из прикладной программы, называют драйвером высшего уровня. Если такой драйвер не пользуется услугами других драйверов, он называется монолитным.
- Драйвер, получающий запросы ввода/вывода от другого драйвера, называют промежуточным, если он пользуется услугами других драйверов, или драйвером низшего уровня, если он не пользуется услугами других драйверов.

1.1. Типы драйверов и характеристики

В NT существует два типа драйверов: драйверы пользовательского режима и драйверы режима ядра. В дальнейшем, говоря «драйвер», мы будем подразумевать драйвер режима ядра. Такие драйверы являются частью исполнительной системы, а более точно – элементами диспетчера ввода/вывода (архитектура NT и ее компоненты будут обсуждаться ниже). Как следует из названия, при работе драйвера режима ядра процессор находится в режиме ядра (RING 0 – см. любой справочник по защищенному режиму работы процессора).

Драйвер NT располагается в файле с расширением .sys и имеет стандартный PE-формат (PE – Portable Executable).

Драйверы реализованы как самостоятельные модули с четко определенным интерфейсом взаимодействия с ОС. Все драйверы имеют определенный системой набор стандартных функций драйвера (standard driver routines) и некоторое число внутренних функций, определенных разработчиком.

Драйверы режима ядра можно разбить на три типа:

- драйверы высшего уровня (highest level drivers);
- драйверы промежуточного уровня (intermediate drivers);
- драйверы низшего уровня (lowest level drivers).

Как будет показано ниже, такое разбиение обусловлено многоуровневой моделью драйверов (layered driver model). Для сохранения общности изложения, монолитный драйвер можно включить в эту схему, хотя он не использует многоуровневую архитектуру. В этом случае он будет «гибридом» – драйвером, принадлежащим одновременно к нескольким типам. Например, монолитный драйвер, имеющий интерфейс с приложением и осуществляющий доступ к оборудованию, будет одновременно драйвером высшего и низшего уровня.

Кроме того, в зависимости от назначения драйвера, он может являться каким-либо специализированным драйвером, то есть удовлетворять дополнительному набору требований к своей структуре. Можно привести следующие типы специализированных драйверов:

- драйверы файловой системы;
- сетевые драйверы.

Отдельно необходимо упомянуть архитектуру WDM – Windows Driver Model. Эта архитектура позволяет создавать драйверы для Windows 98 и Windows 2000, совместимые на уровне двоичного кода.

Характеристики драйверов – это совокупность следующих вопросов:

1) Поддержка динамической загрузки и выгрузки (однако могут быть исключения).
2) Необходимость следовать определенным протоколам взаимодействия с системой, нарушение которых чаще всего ведет к «синему экрану» (Blue Screen Of Death, BSOD).
3) Возможность «наслоения» драйверов поверх друг друга. В Win2000 эта возможность возведена в абсолют, хотя монолитные драйвера все еще поддерживаются.
4) Поскольку драйвера являются частью ядра ОС, они могут сделать с системой все, что угодно, поэтому основная проблема – это закрытость архитектуры ОС.

1.2. Среда разработки

В этом разделе мы рассмотрим, какое программное обеспечение необходимо для разработки и отладки драйверов, а также его установку и настройку.

Необходимое ПО:

1) операционная система Windows NT или Windows 2000, Service Pack и отладочная информация;

2) компилятор;
3) SDK;
4) DDK;
5) средства отладки и вспомогательные средства.

Операционная система имеет два варианта поставки:

- Checked build (Debug build);
- Free build (Retail build).

Free build – обычная поставка. Включена полная оптимизация, отсутствуют специализированные отладочные возможности.

Checked build – специальная поставка для использования разработчиками драйверов. Оптимизации почти нет, что способствует лучшему пониманию кода при работе под отладчиком. Специализированный отладочный код встроен во многие функции для проверки правильности параметров и перехвата ошибочных ситуаций. Поставляется только в составе подписки MSDN.

В комплекте с ОС нам понадобится отладочная информация (файлы с расширением .dbg и .pdb). Она содержит сопоставление адресов внутри конкретного исполняемого файла с символическими именами функций и переменных и может быть использована отладчиками.

Необходимо подчеркнуть, что символьная информация различна для checked и free версий системы.

После установки ОС необходимо установить последнюю версию SP. Надо помнить, что SP заменяет почти все системные файлы, и поэтому для них необходима новая символьная информация.

Для checked и free версий системы требуются отдельные версии SP. Кроме того, ОС и SP могут различаться по поддержке криптоалгоритмов (40-128 бит), что может влиять на возможность установки SP.

Компилятор. Хотя принципиально могут использоваться компиляторы различных производителей, структура заголовочных файлов и переменных окружения, поставляемых Microsoft для создания драйверов оптимизирована для использования компилятора Microsoft Visual C. Версия компилятора должна быть не ниже 4.1, однако реально необходимая версия будет зависеть от двух других компонентов – SDK и DDK.

MSDN Library. При установке Developer Studio запрашивается установка MSDN Library. Этот продукт предоставляет информацию о разработке ПО на всех поддерживаемых платформах Microsoft.

SDK. В ранних версиях комплект назывался **Win32 SDK**, сейчас – **Platform SDK**. Это необязательный, но желательный для разработки драйверов компонент. Содержит заголовочные файлы, lib-файлы, документацию и примеры программирования на пользовательском уровне с использованием подсистемы Win32.

DDK. Существуют DDK для Windows 95, Windows 98, Windows NT 4.0 и Windows 2000. DDK должен соответствовать платформе, для которой предполагается создание драйвера, что не обязательно для той, на которой производится создание. Мы будем пользоваться DDK для Windows NT 4.0. DDK содержит заголовочные файлы, доку-

ментацию и примеры написания драйверов, за исключением драйверов файловой системы.

IFS Kit. Пакет для создания драйверов файловой системы. Поставляется как отдельный от подписки MSDN продукт. Существуют версии для Windows 98, Windows NT 4.0 и Windows 2000. Последние версии включают в себя DDK, но с другим набором примеров. Более ранние версии требовали предварительной установки DDK.

Между перечисленным набором компонентов имеется взаимосвязь.

Первым ставится компилятор. Как уже говорилось, хотя существует возможность использования компиляторов других фирм, SDK и DDK предполагают наличие именно Visual C, причем в зависимости от времени выхода SDK и DDK подразумеваются различные версии компилятора (при линковке будут указаны библиотеки различных версий – этим грешит SDK, либо будут некорректно запускаться командные файлы инициализации переменных окружения – этим грешит DDK). Кроме того, ранние версии DDK требовали обязательного наличия установленного SDK. Из возможных проблем стоит указать и то, что при использовании ОС Windows NT Workstation могут не устанавливаться системные переменные окружения.

Расположение командных файлов для установки переменных окружения:

- VC98\bin\vcvars32.bat;
- Mstools\setenv.bat;
- Ddk\bin\setenv.bat.

При наличии версии DDK, требующей наличия SDK, из файла Ddk\bin\setenv.bat должны быть исключены строки проверки наличия SDK и запуска файла setenv.bat, а также прописан вызов vcvars32.bat.

Средства отладки и вспомогательные средства.

Выбор средства отладки – важный момент, который может влиять на набор необходимых аппаратных средств.

Вместе с продуктами Microsoft поставляются четыре отладчика:

1) **KD** – консольная программа для отладки драйверов режима ядра, находится в директории *bin* пакета DDK для NT4 и Win2000. (i386kd.exe, ia64kd.exe, alphakd.exe, mipskd.exe).
2) **NTSD** – консольная программа для отладки программ и драйверов пользовательского режима, находится в директории *system32* ОС Windows 2000.
3) **CDB** – вариант NTSD, содержится в директории *bin* пакета DDK для Windows 2000.
4) **WinDbg** – графический отладчик для отладки кода как пользовательского режима, так и режима ядра, содержится в директории *bin* пакета DDK для Win2000 и Platform SDK.

Из всех перечисленных вариантов упоминания достоин лишь отладчик WinDbg. Он предоставляет удобный пользовательский интерфейс, однако очень неустойчив в работе, плохо документирован и не имеет поддержки от Microsoft. При использования этого продукта для отладки драйверов необходимы два компьютера – Development Platform и Test Platform. Отладчик доступен для всех поддерживаемых платформ, при

этом возможна кроссплатформенная отладка. Поддерживается работа на мультипроцессорных системах.

Лучшим отладчиком для отладки ОС и драйверов многие разработчики с полным основанием считают **SoftICE фирмы NuMega**, стабильный при работе, хорошо документированный, с поддержкой от фирмы. Отладка осуществляется на том же компьютере, на котором проводилась разработка, однако возможна и удаленная отладка посредством dos-программы serial.exe. Недостатком можно считать пользовательский интерфейс, однако – это дело привычки. Более серьезными недостатками является ограничение поддержки процессоров только платформой intel, а также отсутствие поддержки мультипроцессорных систем (однако система все еще активно развивается).

Ниже перечислены средства, которые могут выступать к качестве «наглядного пособия» при написании драйверов (часть этих средств снабжена исходными текстами):

1) Monitor – просмотр трассировочной информации, выводимой драйверами и прикладными программами;
2) Winobj – просмотр пространства имен диспетчера объектов;
3) Handleex – информация о запущенных процессах, всех открытых ими описателях и подгруженных модулях dll;
4) Filemon – просмотр активности файловых систем;
5) Regmon – отслеживание обращений к реестру, в том числе на этапе загрузки системы;
6) Portmon – отслеживание обращений к последовательным и параллельным портам;
7) Tdimon – отслеживание запросов TDI;
8) Tokenmon – отслеживание работы системы безопасности.

1.3. Утилита BUILD

Для построения драйверов и связанных с ними прикладных программ используется утилита BUILD, входящая в состав DDK. Эта утилита позволяет создавать любой тип исполняемого файла, поддерживаемый NT с использованием командной строки. *Стандартного* (и поддерживаемого Microsoft) способа использования *Интегрированной Среды Разработки* для написания драйвера не существует. (Варианты, иллюстрирующие то, как это можно сделать, мы рассмотрим в следующем разделе.)

Возможны два варианта построения драйвера:

- Checked build – эквивалент Debug build в интегрированной среде;
- Free build, также называемый Retail build – эквивалент Release build в интегрированной среде.

Для осуществления конкретного построения необходимо запустить файл setenv.bat с соответствующими параметрами. С целью автоматизации этого процесса при установке DDK создаются ярлыки «Checked Build Environment» и «Free Build Environment». Запуск любого из них вызывает командную оболочку, из которой необходнмо вызывать команду build.

Для успешной работы команды build в текущей директории должны находиться два специальных файла: SOURCES и DIRS.

Файл SOURCES является аналогом понятия проект. В нем определяется имя создаваемого модуля, его тип, корневая директория, где будет размещен результат, путь поиска include-файлов, список подключаемых библиотек и список файлов с исходным текстом.

Файл DIRS определяет список поддиректорий, которые должны быть обработаны командой build, прежде чем перейти к текущей директории. Директория, в которой запускается команда build, может содержать либо файл SOURCES, либо файл DIRS, либо оба вместе. Отсутствие файлов является ошибкой.

Директория, в которой содержится только файл DIRS, не будет обработана командой build, но будут обработаны все поддиректории из файла DIRS.

Директория, в которой содержится только файл SOURCES, будет обработана командой build, но ни одна поддиректория обработана не будет.

Директория, в которой содержатся оба файла, будет обработана командой build только после обработки всех поддиректорий из файла DIRS.

Примеры файлов можно посмотреть в DDK, а полное описание утилиты build и структуры файлов – в документации к IFS Kit (В принципе, можно воспользоваться документацией к DDK, однако там описание менее понятно.)

Теперь рассмотрим некоторые отличия checked и free build. Как ясно из названия, это отладочный и окончательный варианты драйвера. Соответственно, они отличаются режимами оптимизации кода. Для режима checked создается отладочная информация в формате dbg, которую можно использовать для отладки в символьном режиме с помощью SoftIce. В режиме checked на этапе компиляции определено имя DBG, которое может быть использовано директивами

```
#if DBG
...
#else
...
#endif
```

для выполнения действий, специфичных для checked или free версий драйвера. Типичным примером является обрамление таким способом функции DbgPrint(), которая выводит трассировочную информацию. Функция работает и в checked, и в free build, однако с помощью такой проверки ее можно исключить из free build.

Хорошо известная техника Microsoft по написанию кода – использование макроса ASSERT(). Этот макрос проверяет условие. Если оно ложно, генерируется прерывание. При этом отладчику сообщается имя файла с исходным текстом и номер строки с макросом. Макрос работает только в checked-версии драйвера, установленного на checked-версии ОС. Во всех остальных случаях ничего не происходит.

При создании серьезного продукта цикл разработки драйвера выгладит примерно так:

1) написание кода;

2) проверка и отладка отладочной версии драйвера на отладочной версии ОС;
3) проверка и отладка отладочной версии драйвера на рабочей версии ОС;
4) проверка рабочей версии драйвера на рабочей версии ОС;
5) проверка работы на многопроцессорной системе.

Особо обратите внимание на последний пункт. Нет никакого другого способа убедиться в корректной работе драйвера на многопроцессорной системе, кроме его тестирования на такой системе, даже при условии корректной работы драйвера на однопроцессорной системе.

1.4. C и C++. Интегрированная среда разработки

Необходимо особо отметить, что драйверы предполагается писать на C, а не на C++. Microsoft не поддерживает использование C++ для компонентов ядра. Для этого имеется ряд причин:

- отсутствие библиотеки времени исполнения (runtime library), а, следовательно, и определяемых в ней глобальных операторов new и delete4;
- отсутствие поддержки исключительныых ситуаций C++;
- нет поддержки инициализации глобальных экземпляров классов.

В принципе, все эти проблемы разрешимы. Не будем останавливаться на описании конкретных способов. Об этом вы можете узнать в статье «C++ Runtime Support for the NT DDK», а также из анализа заголовочных файлов в продукте DriverWorks (в особенности файла vdw.h).

Как было сказано выше, интегрированная среда Developer Studio не имеет поддержки для создания драйверов. Драйверы компилируются из командной строки с использованием утилиты BUILD, поставляемой в составе DDK.

Реализовать поддержку драйверов из интегрированной среды можно несколькими способами:

- реализацией собственного AppWizard (см. AppWizard Programming Reference);
- созданием проекта на основе make-файла с вызовом собственного командного файла.

Этот файл должен:

- произвести настройку переменных окружения с помощью вызова setenv.bat из DDK;
- перейти в директорию с исходным текстом и вызвать утилиту build (см. также статью «Integrating BUILD and Developer Studio» в директории NT Insider).

Реализация собственного AppWizard – довольно непростая задача, однако, можно воспользоваться готовым из DriverWorks. Последовательность действий такая: выберите меню Developer Studio File\New... . В появившемся окне на закладке Projects выберите NT/WDM Driver (DriverWorks). В появившемся окне Мастера укажите тип драйвера NT и следуйте инструкциям, внося минимальные изменения. По завершении работы мастера удалите все созданные им cpp- и h-файлы, и вставьте собственные c- и h-файлы.

2. Общая архитектура Windows NT

В этой главе рассматриваются ключевые архитектурные особенности и характеристики ОС Windows NT. Эти сведения необходимы для получения представления о назначении различных компонентов ОС, их взаимодействии друг с другом, а также для ознакомления с терминологией, используемой в данной книге. Кроме того, знание общей архитектуры ОС позволяет понять, какие возможности операционной системы могут задействовать средства защиты, расположенные на том или ином уровне архитектуры ОС.

Большое внимание в этой главе уделено модели драйвера, его структуре и характеристикам, а также взаимосвязи с другими драйверами и прикладными программами.

2.1. Понятия «пользовательский режим» и «режим ядра»

При обсуждении архитектуры ОС Windows NT постоянно используются понятия «режим пользователя» и «режим ядра», поэтому стоит определить, что это значит. Начнем с обсуждения разницы между пользовательским режимом и режимом ядра (user mode/kernel mode).

Пользовательский режим – наименее привилегированный режим, поддерживаемый NT; он не имеет прямого доступа к оборудованию и у него ограниченный доступ к памяти.

Режим ядра – привилегированный режим. Те части NT, которые исполняются в режиме ядра, такие как драйверы устройств и подсистемы типа Диспетчера Виртуальной Памяти, имеют прямой доступ ко всей аппаратуре и памяти.

Различия в работе программ пользовательского режима и режима ядра поддерживаются аппаратными средствами компьютера (а именно – процессором).

Большинство архитектур процессоров обеспечивают, по крайней мере, два аппаратных уровня привилегий. Аппаратный уровень привилегий процессора определяет возможное множество инструкций, которые может вызывать исполняемый в данный момент процессором код. Хотя понятия «режим пользователя» и «режим ядра» часто используются для описания кода, на самом деле это уровни привилегий, ассоциированные с процессором. Уровень привилегий накладывает три типа ограничений: 1) возможность выполнения привилегированных команд, 2) запрет обращения к данным с более высоким уровнем привилегий, 3) запрет передачи управления коду с уровнем привилегий, не равным уровню привилегий вызывающего кода.

2.1.1. Некоторые понятия защищенного режима

Защищенный режим является основным и наиболее естественным режимом работы 32-разрядных процессоров. Этот режим был в полной мере реализован в процессорах серии i386 и с тех пор существенных изменений не претерпел.

Защищенный режим 32-разрядных процессоров реализует поддержку следующих механизмов:

- Организация памяти, при которой используются два механизма преобразования памяти: сегментация и разбиение на страницы.
- Четырехуровневая система защиты пространства памяти и ввода/вывода.
- Переключение задач.

Сегмент – это блок пространства памяти определенного назначения, внутри которого применяется линейная адресация. Максимальный размер сегмента при 32-разрядной адресации составляет 4 Гб (2^{32} байт). Максимальное число таких сегментов равно 2^{13} (8192). Сегмент может иметь произвольную длину в допустимых границах.

Каждый сегмент характеризуется 8-байтной структурой данных – дескриптором сегмента, в котором, в числе прочего, указаны:

- Права доступа, которые определяют возможность чтения, записи и исполнения сегмента.
- Уровень привилегий (относится к четырехуровневой системе защиты).

На сегментации основана защита памяти. При этом не допускается:

- использовать сегменты не по назначению (нарушение прав доступа);
- обращаться к сегменту, не имея достаточных привилегий;
- адресоваться к элементам, выходящим за границы сегмента.

Страничная организация памяти позволяет использовать большее пространство памяти. При этом базовым объектом памяти служит блок фиксированного размера 4 Кб.

Физический адрес памяти, получаемый на выходе сегментного и страничного преобразования памяти, является 32-разрядным, позволяя адресовать, таким образом, до 4 Гб реально доступной физической памяти.

Четырехуровневая система привилегий предназначена для управления использованием привилегированных инструкций, а также для защиты пространства памяти и ввода/вывода.

Уровни привилегий нумеруются от 0 до 3, нулевой уровень соответствует максимальным (неограниченным) возможностям доступа и отводится для ядра ОС. Уровень 3 имеет самые ограниченные права и обычно предоставляется прикладным задачам.

Систему защиты обычно изображают в виде колец, соответствующих уровням привилегий, а сами уровни привилегий иногда называют кольцами защиты.

В зависимости от уровня привилегий осуществляется защита по доступу к привилегированным командам, по доступу к данным с более высоким уровнем привилегий и по передаче управления коду с уровнем привилегий, отличным от текущего.

Защищенный режим предоставляет средства переключения задач. Состояние каждой задачи (значения всех связанных с ней регистров процессора) может быть сохранено в специальном сегменте состояния задачи. Там же хранится карта разрешения ввода/вывода, указывающая для каждого из 64К адресов портов ввода/вывода возможность обращения к нему.

ОС NT использует два кольца защиты – 0 и 3, имея соответственно режим работы в 0 кольце – kernel mode, в 3 кольце – user mode.

2.2. Основные характеристики Windows NT

ОС NT характеризуется поддержкой следуюших механизмов:

1) модель модифицированного микроядра;
2) эмуляция нескольких ОС;
3) независимость от архитектуры процессора;
4) объектная модель;
5) многопоточность;
6) вытесняющая многозадачность;
7) виртуальная память с подкачкой страниц по требованию;
8) мультипроцессорная обработка;
9) интегрированная поддержка сети.

2.2.1. Модель модифицированного микроядра

На NT иногда ссылаются как на операционную систему на основе микроядра (microkernel-based operating system). Идея, лежащая в основе концепции микроядра, состоит в том, что все компоненты ОС за исключением небольшой основы (собственно, микроядра) исполняются как процессы пользовательского режима. Базовые компоненты в микроядре исполняются в привилегированном режиме.

Архитектура микроядра обеспечивает в системе возможность конфигурации и устойчивость к ошибкам. Поскольку подсистемы ОС типа *Диспетчера Виртуальной Памяти* исполняются как отдельные программы в архитектуре микроядра, их можно заменить различными реализациями, экспортирующими такой же интерфейс. Если в *Диспетчере Виртуальной Памяти* происходит ошибка, то, благодаря устойчивости к ошибкам в дизайне микроядра, операционная система может перезапустить его с минимальным воздействием на остальную систему.

Недостаток чистой архитектуры микроядра – низкая производительность. Любое взаимодействие между компонентами ОС при такой схеме нуждается в межпроцессном сообщении с длительными переключениями между задачами.

NT использует уникальный подход, известный как модифицированное микроядро. Он является промежуточным между чистым микроядром и монолитной структурой.

При этом подходе в пользовательском режиме работают прикладные программы и набор подсистем, относящихся к одному из двух классов – подсистемы окружения и неотъемлемые подсистемы. Подсистемы и прикладные программы реализованы как процессы, однако способ создания подсистем и интеграции их с ОС не документирован.

Подсистемы окружения предоставляют прикладным программам интерфейс программирования, специфичный для некоторых ОС (WIN32, POSIX, OS/2, DOS).

Неотъемлемые подсистемы исполняют важные функции ОС. Среди таких подсистем – подсистема безопасности, служба рабочей станции и служба сервера.

При выполнения задач, которые не могут быть выполнены в пользовательском режиме, все подсистемы ОС NT полагаются на системные сервисы, экспортируемые режимом ядра. Эти сервисы известны как «родной» API. Такой API состоит примерно из 250 функций, доступных через модуль ntdll.dll.

Прикладная программа использует интерфейс программирования, предоставляемый какой-либо одной подсистемой окружения, либо использует напрямую собственный интерфейс программирования.

В режиме ядра работает *Исполнительная система* NT (NT Executive). Она, сама по себе, является законченной ОС со своим интерфейсом программирования, как для пользовательского режима, так и для режима ядра.

Исполнительная система состоит из набора подсистем, *Микроядра* и *Слоя Абстрагирования от Оборудования* (HAL). Подсистемы *Исполнительной системы* и *Микроядро* находятся в едином модуле – ntoskrnl.exe. *Слой Абстрагирования от Оборудования* находится в модуле hal.dll. Все загруженные системой драйверы также являются частью исполнительной системы.

Каждый компонент исполнительной системы экспортирует набор функций для использования другими компонентами. Кроме того, каждый компонент исполнительной системы, за исключением *диспетчера Кэша* и *Слоя Абстрагирования от Оборудования*, реализует набор системных сервисов.

2.2.2. Эмуляция нескольких ОС

Подсистемы окружения операционной системы NT реализованы как системы типа клиент/сервер. Как часть процесса компиляции, прикладные программы прикрепляются на этапе компоновки к API операционной системы, который экспортируют подсистемы окружения ОС NT . Связывание на этапе компоновки подключает прикладную программу к клиентским DLLs подсистем окружения, которые осуществляют экспорт API. Например, Win32 программа – это клиент подсистемы окружения Win32, поэтому она связана с клиентскими DLL Win32, включая Kernel32.dll, gdi32.dll, и user32.dll. Программа POSIX связана с клиентской DLL POSIX – psxdll.dll.

Клиентские DLL выполняют задачи от имени их серверов, но они выполняются, как часть клиентского процесса. Как показано рис. 1, в некоторых случаях пользовательская DLL может полностью реализовывать API без необходимости обращения к помощи сервера; в других случаях сервер должен помочь. Помощь сервера обычно необходима только когда должна быть модифицирована общая информация, связанная с подсистемой окружения. Когда пользовательская DLL требует помощи от сервера, DLL посылает сообщение, известное, как вызов локальной процедуры (LPC) на сервер. Когда сервер завершает указанный запрос и возвращает ответ, DLL может завершить функцию и возвратить управление клиенту. И пользовательская DLL и сервер могут использовать «родной» API, когда это необходимо. API подсистем окружения дополняют «родной» API специфическими функциональными возможностями или семантикой.

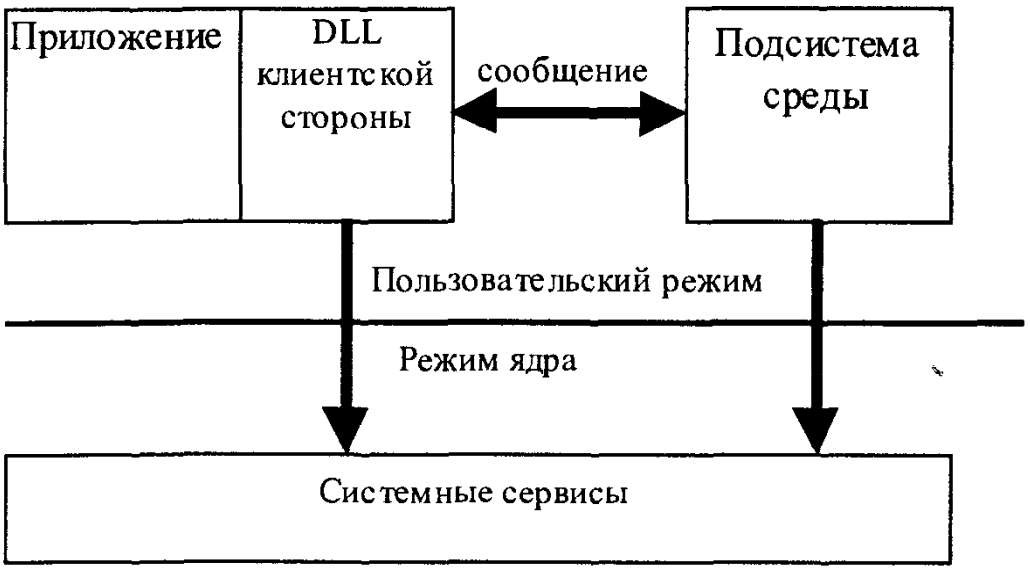


Рис. 1

2.2.3. Независимость от архитектуры процессора

Микроядро и *Слой Абстрагирования от Оборудования* (HAL) изолируют подсистемы *Исполнительной Системы* от конкретной архитектуры процессора.

Другой аспект независимости от архитектуры состоит в том, что правильно написанный драйвер (общающийся с внешним миром только посредством функций, предоставляемых различными компонентами исполнительной системы) переносим между всеми поддерживаемыми NT платформами на уровне исходных текстов.

Микроядро OS Windows NT обеспечивает единый интерфейс для использования ресурсов, общих для определенной аппаратной платформы, на которой может работать OS. Например, микроядро обеспечивает интерфейсы к обработке и управлению прерываниями, сохранению и восстановлению контекста потоков и мультипроцессорной синхронизации.

HAL обеспечивает поддержку и отвечает за предоставление стандартного интерфейса к ресурсам процессора, которые могут меняться в зависимости от модели внутри одного семейства процессоров. Возможность замены слоя HAL обеспечивает всем вышележащим слоям операционной системы независимость от аппаратной архитектуры.

2.2.4. Объектная модель

В исполнительной системе объект (object) – это отдельный образец статически определенного типа объектов, существующий во время выполнения. Тип объектов, иногда называемый классом объектов, включает определенный системой тип данных, объектные сервисы, работающие с образцами этого типа, и набор атрибутов объекта.

Атрибут объекта – это поле данных внутри объекта, частично определяющее его состояние. Объектные сервисы – способы манипулирования объектами – обычно считывают или изменяют атрибуты объектов.

Windows NT использует объекты для унификации представления и управления системными ресурсами. Каждый системный ресурс, который могут совместно использовать несколько процессов, такой, как файл, память или физическое устройство, реализован как объект и обрабатывается объектными сервисами. Доступ ОС к ресурсам и работа с ними унифицированы. Создание, удаление и ссылка на объект осуществляется с использованием описателей (handle) объектов. Контроль использования ресурсов сводится к отслеживанию создания и использования объектов. Для всех объектов контроль доступа к ним осуществляется одинаково с помощью подсистемы защиты. Два процесса совместно используют объект тогда, когда каждый из них открыл его описатель. ОС может отслеживать количество описателей, открытых для данного объекта, чтобы определить, действительно ли они все еще используется, и может удалить объекты, которые более не используются.

2.2.5. Многопоточность

Каждая исполняющаяся в NT программа представляется как процесс.

Процесс (process) – это программа (статическая последовательность команд и данные) и системные ресурсы, необходимые для ее работы. ОС предоставляет каждому процессу адресное пространство, выделенное для программы, и гарантирует, что программа каждого процесса будет направляться на выполнение в определенном порядке и в нужное время. Чтобы процесс смог заработать, он должен включать, по крайней мере, один поток исполнения (thread of execution).

Поток (thread) – единица исполнения в NT. Поток – это сущность внутри процесса, которую ядро направляет на исполнение, он может принадлежать только одному процессу. Поток состоит из указателя текущей команды, пользовательского стека, стека ядра и набора значений регистров. Все потоки процесса имеют одинаковый доступ к его адресному пространству, описателям объектов и другим ресурсам. Потоки реализованы как объекты-потоки.

Начальный поток возникает при создании процесса, и затем он может создать дополнительные потоки.

Каждый поток имеет свой собственный приоритет, в соответствии с которым, ОС будет принимать решение о его запуске. При этом принадлежность потока к конкретному процессу не учитывается.

2.2.6. Вытесняющая многозадачность (preemptive multitasking)

NT позволяет нескольким единицам исполнения – потокам – выполняться одновременно, быстро переключаясь между ними. Такое поведение называется **многозадачностью** (multitasking).

Каждому потоку на исполнение выделяется квант времени процессора. По истечении этого времени операционная система насильственно отдаст время процессора другому потоку (говорят, что поток будет вытеснен). Такое поведение называется вытесняющей многозадачностью (в отличие от невытесняющей многозадачности, когда поток сам должен освободить процессор).

Необходимо определить еще два термина: диспетчеризация и планирование.

Диспетчеризация (dispatching) – механизм переключения с одного потока исполнения на другой.

Планирование (sheduling) – механизм определения потока, который должен выполняться следующим на текущем процессоре.

Таким образом, по истечении кванта времени некоторого потока на основе механизма планирования осуществляется выбор следующего потока для исполнения, а на основе механизма диспетчеризации происходит переключение на этот поток.

Каждый поток имеет также приоритет, называемый приоритетом планирования, что подчеркивает его важность при выборе для исполнения очередного потока.

2.2.7. Виртуальная память с подкачкой страниц по требованию

Виртуальное адресное пространство (virtual address space) процесса – это набор адресов, которые могут использовать потоки процесса, оно равно четырем гигабайтам (2^{32} байт), два из которых предназначены для использования программой, а другие два зарезервированы для ОС.

Во время выполнения потока диспетчер памяти при помощи аппаратных средств транслирует (отображает) виртуальные адреса в физические, по которым данные хранятся на самом деле. Посредством контроля над процессом отображения ОС может гарантировать, что процессы не будут пересекаться друг с другом и не повредят ОС.

Когда физической памяти не хватает, диспетчер памяти выгружает часть содержимого памяти на диск. При обращении потока по виртуальному адресу, соответствующему переписанным на диск данным, диспетчер памяти снова загружает эти данные с диска в память.

В Windows NT код ОС располагается в верхней части виртуального адресного пространства, а пользовательский код и данные – в нижней. Можно выгружать всю пользовательскую память. Код пользовательского режима не может производить запись и чтение системной памяти.

Часть системной памяти, называемая невыгружаемым (резидентным) пулом (nonpaged pool), никогда не выгружается на диск и используется для хранения некоторых объектов и других важных структур данных. Другая часть системной памяти, которая может быть выгружена на диск, называется выгружаемым (нерезидентным) пулом (paged pool).

2.2.8. Симметричная мультипроцессорная обработка

NT поддерживает только архитектуру с симметричной мультипроцессорной обработкой – SMP.

Системы с симметричной мультипроцессорной обработкой позволяют коду операционной системы выполняться на любом свободном процессоре или на всех процессорах одновременно, причем каждому из процессоров доступна вся память. Чтобы гарантировать правильную работу системы, код таких ОС должен следовать строгим правилам. Windows NT обладает свойствами, которые принципиально важны для мультипроцессорной ОС:

- Код ОС может выполняться на любом из доступных процессоров и на нескольких процессорах одновременно. За исключением кода ядра, которое выполняет планировку потоков и обработку прерываний, весь код ОС может быть вытеснен потоком с более высоким приоритетом.
- В одном процессе может быть несколько потоков управления. Потоки позволяют процессу выполнять разные части его программы на нескольких процессорах одновременно.
- Серверные процессы могут иметь несколько потоков для одновременной обработки запросов от нескольких клиентов.

Имеются механизмы совместного использования объектов потоками разных процессов, и гибкие возможности коммуникации между потоками разных процессов, включая совместно используемую память и оптимизированное средство передачи сообщений.

2.2.9. Интегрированная поддержка сети

Windows NT разработана со встроенной сетевой поддержкой и включает широкую поддержку сети, интегрированную с системой ввода/вывода и интерфейсом Win32 API.

Четырьмя основными типами сетевого программного обеспечения являются сетевые сервисы, сетевые API, протоколы и драйверы сетевых карт, располагающиеся друг под другом, формируя сетевой стек.

Windows NT предоставляет хорошо определенные интерфейсы для каждого слоя в стеке, чтобы в дополнение к поставляющемуся с Windows NT множеству различных сетевых интерфейсов API, протоколов и драйверов сетевых карт, пользователи могли расширять сетевые возможности ОС путем разработки собственного сетевого программного обеспечения.

2.3. Структура Windows NT

Всю операционную систему Windows NT можно разделить на следующие части (см. рис. 2):

1) защищенные подсистемы (protected subsystems), работающие в пользовательском режиме, тогда как остальная часть ОС исполняется в режиме ядра;

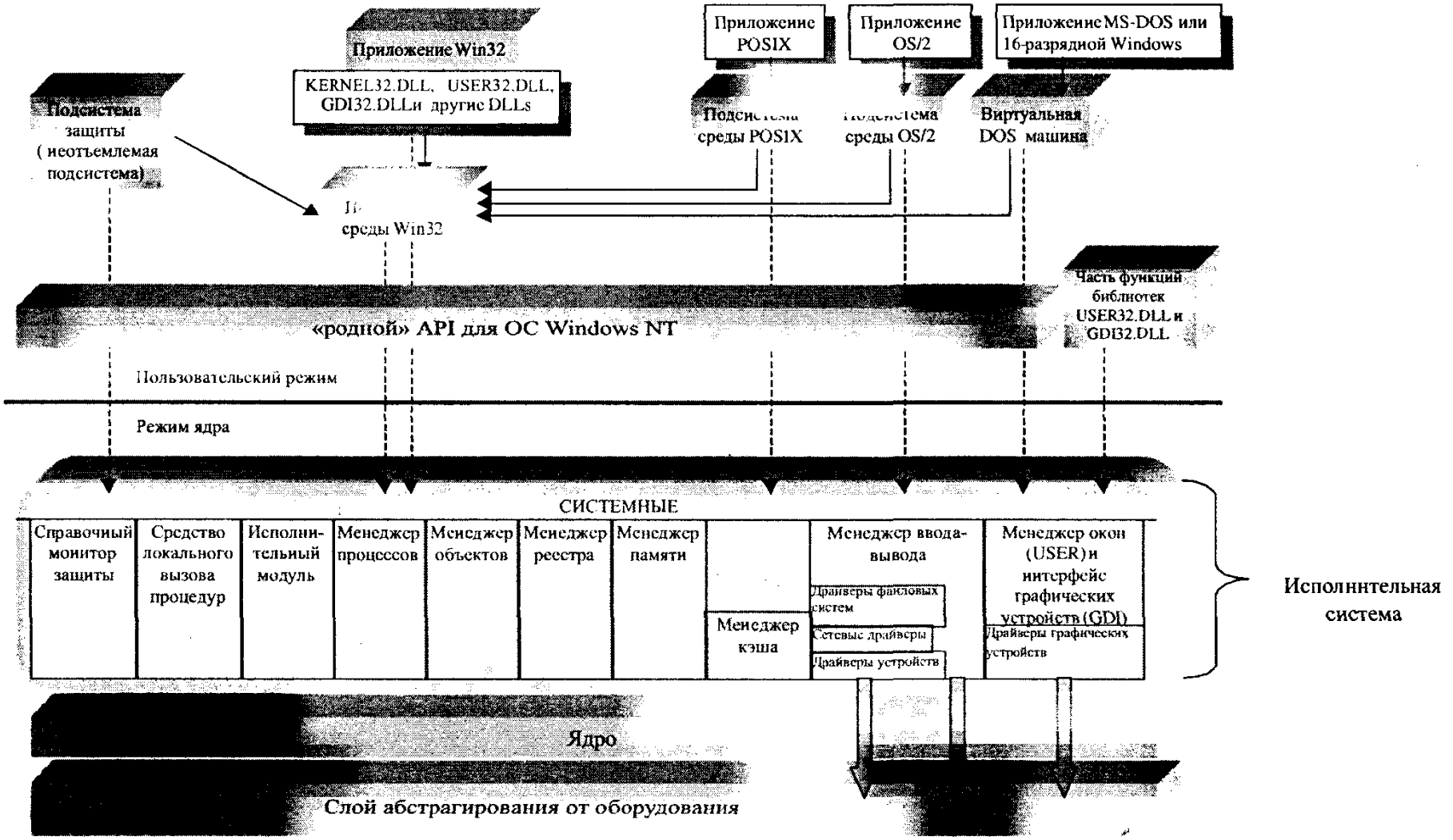


Рис. 2. Общая архитектура ОС Windows NT

2) исполнительная система (executive);
3) ядро (kernel);
4) слой абстрагирования от оборудования (Hardware Abstraction Layer, HAL).

2.3.1. Защищенные подсистемы

Серверы Windows NT называются защищенными подсистемами, так как каждый из них – это отдельный процесс, память которого защищена от других процессов системой виртуальной памяти исполнительной системы NT. Каждая защищенная подсистема обеспечивает интерфейс прикладным программам (API) посредством DLLs клиентской стороны. Когда приложение или другой сервер вызывает некоторую процедуру API, соответствующая DLL упаковывает параметры функции API в сообщение и с помощью средства локального вызова процедур (Local Procedure Call, LPC) посылает его серверу, реализующему данную процедуру. Сервер же, выполнив вызов, посылает ответное сообщение вызывающей программе. Передача сообщений остается невидимой для прикладного программиста. Используя такую процедуру, вызывающая программа никогда не получает прямого доступа к адресному пространству подсистемы.

Надо отметить, что далеко не все функции API реализуются сервером, например, большая часть функций API Win32 оптимизирована в DLL клиентской стороны, и в действительности не обращается к подсистеме Win32.

Защищенные подсистемы подразделяются на подсистемы среды (environment subsystems) и неотъемлемые подсистемы (integral subsystems).

2.3.2.1. Подсистемы среды

Подсистема среды – это сервер пользовательского режима, реализующий API некоторой ОС. Самая важная подсистема среды в Windows NT – это подсистема среды Win32 (рассматриваемая ниже), которая предоставляет прикладным программам интерфейс API 32-разрядной Windows. В Windows NT также имеются подсистемы среды: POSIX, OS/2 и виртуальная DOS машина (virtual DOS machine, VDM), эмулирующая 16-разрядную Windows и MS-DOS.

Данные подсистемы предоставляют свои API, но используют для получения пользовательского ввода и отображения результатов подсистему Win32, то есть перенаправляют видеовывод своих приложений подсистеме Win32 для отображения.

Говоря о подсистемах окружения, необходимо отметить также следующее. Каждая прикладная программа (и даже более того – каждый модуль, будь то exe, dll, sys или что-то другое) может относиться только к какой-то одной подсистеме окружения, либо не относиться ни к одной из них. Эта информация прописывается в любом исполняемом модуле на этапе его компиляции и может быть получена через утилиту «Быстрый просмотр» (Quick View) в пункте «Subsystem» (варианты: The image does not require subsystem, Win32 GUI, Win32 Console, ...).

2.3.2.1.1. Подсистема среды Win32

Подсистема среды Win32 делится на серверный процесс (csrss.exe – Client/Server Runtime SubSystem) и клиентские DLLs (user32.dll, gdi32.dll, kernel32.dll), которые связаны с программой, использующей Win32 API. Win32. API разделен на три категории:

- Управление окнами (windowing) и передача сообщений (messaging). Эти интерфейсы оконных процедур и процедур сообщений включают, например, такие функции, как CreateWindow(), SendMessage(), и предоставляются прикладной программе через библиотеку user32.dll.
- Рисование (drawing). Например, функции BitBit() и LineTo() являются Win32-функциями рисования и предоставляются библиотекой gdi32.dll.
- Базовые сервисы. Базовые сервисы включают весь ввод/вывод Win32, управление процессами и потоками, управление памятью, синхронизацию и предоставляются библиотекой kernel32.dll.

Когда Win32-приложение вызывает функцию API Win32, управление передается одной из клиентских DLLs подсистемы Win32. Эта DLL может:

- Выполнить функцию самостоятельно без обращения к системным сервисам ОС и вернуть управление вызывающей программе.
- Послать сообщение Win32-серверу для обработки запроса в том случае, если сервер должен участвовать в выполнении заданной функции. Так, функция CreateProcess(), экспортируемая библиотекой kernel32.dll, требует взаимодействия с Win32-сервером, который, в свою очередь, вызывает функции «родного» API.
- Вовлечь «родной» интерфейс API для выполнения заданной функции. Последний вариант встречается наиболее часто. В версиях Windows NT ниже 4.0 функции окон (windowing) и рисования (drawing) были расположены в Win32-сервере (csrss.exe). Это означало, что, когда приложение использовало такие функции, посылались сообщения указанному серверу. В версии 4.0 эти функции были перенесены в компонент режима ядра, называемый win32k.sys. Теперь вместо того, чтобы посылать сообщение серверу, клиентская DLL обращается к этому компоненту ядра, уменьшая затраты на создание сообщений и переключение контекстов потоков различных процессов. Это увеличило производительность графического ввода/вывода. Библиотеки gdi32.dll и user32.dll стали вторым «родным» API, но оно менее загадочно, чем первое, так как хорошо документировано.

Функциями библиотеки kernel32.dll, вызывающими «родной» интерфейс API напрямую, являются функции ввода/вывода, синхронизации и управления памятью. Фактически, большинство экспортируемых библиотекой kernel32.dll функций используют «родной» API напрямую. На рис. 3 иллюстрируется передача управления от Win32-приложения, выполнившего вызов Win32-функции CreateFile(), библиотеке kernel32.dll, затем функции NtCreateFile() в ntdll.dll и далее режиму ядра, где управление передает-

ся системному сервису, реализующему создание/открытие файла. Подробнее вызов системных сервисов рассматривается в следующем параграфе.

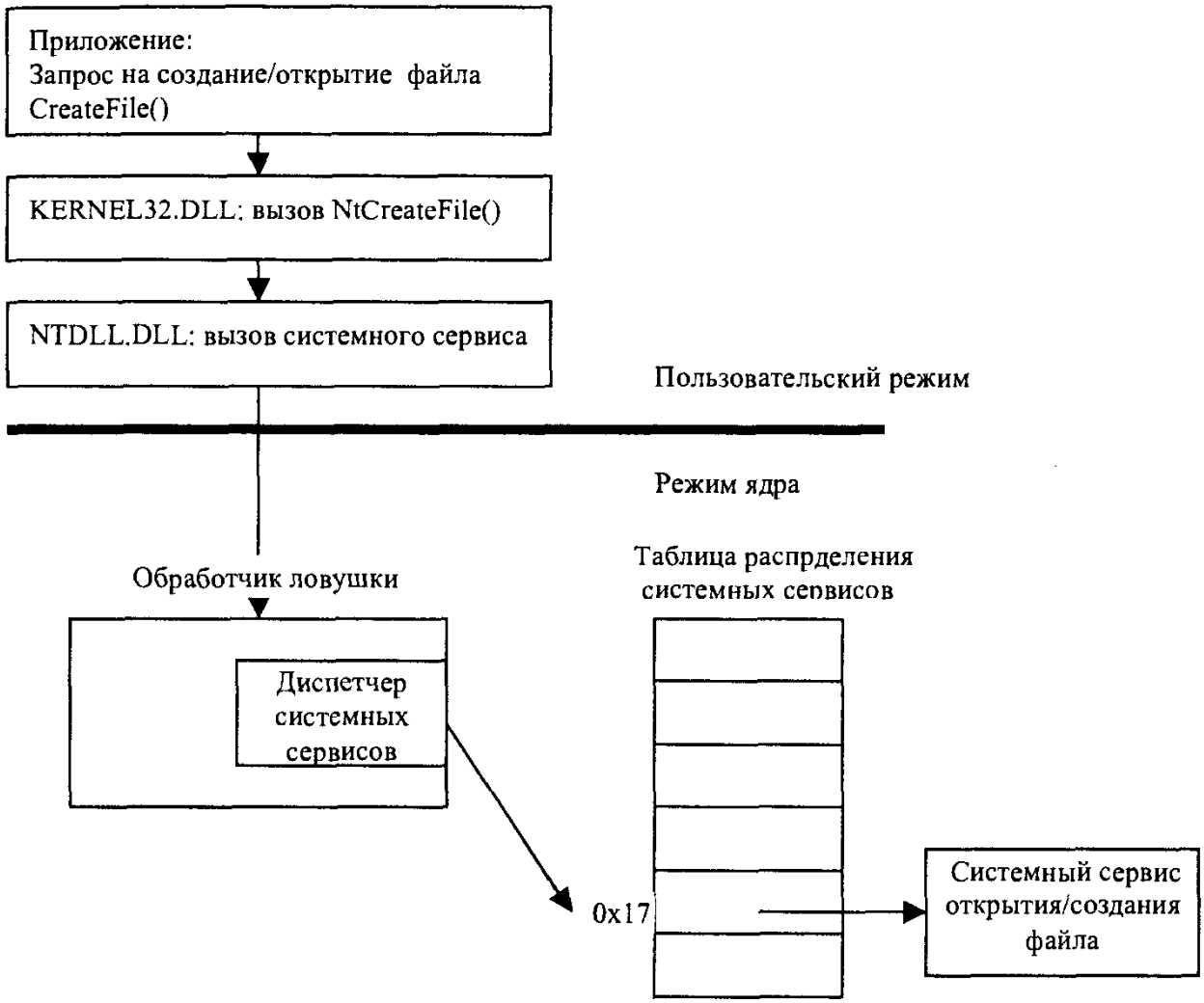


Рис. 3. Вызов системных сервисов через "родной" API (ntdll.dll)

2.3.2.1.2. «Родной» API для ОС Windows NT (Native Windows NT API)

«Родной» API для Windows NT является средством, которое реализует контролируемый вызов системных сервисов, исполняемых в режиме ядра. Так, например, если программа, исполняющаяся в пользовательском режиме, захочет выполнить операцию ввода/вывода, зарезервировать или освободить регион в виртуальном адресном пространстве, запустить поток или создать процесс, – она должна запросить (естественно, не напрямую) один или несколько системных сервисов, расположенных в режиме ядра.

Этот интерфейс API является интерфейсом системных вызовов и не предназначается для непосредственного использования пользовательскими программами, кроме того, его документация ограничена. В Windows NT «родной» интерфейс спрятан от прикладных программистов под интерфейсами API более высокого уровня, таких как Win32, OS/2, POSIX, DOS/Win16.

«Родной» API предоставляется коду пользовательского режима библиотекой ntdll.dll. Библиотека ntdll.dll, имеющая точки входа в «родной» API для кода пользовательского режима, содержит также код загрузки модуля и запуска потока процесса. Однако большинство входов в «родной» API являются заглушками, которые просто передают управление режиму ядра. Это осуществляется путем генерации программного исключения, например, ассемблерный код функции NtCreateFile() в библиотеке ntdll.dll, выглядит следующим образом:

```
mov eax, 0x00000017
lea edx, [esp+04]
int 0x2E
ret 0x2C
```

Другие вызовы выглядят почти также. Первая инструкция загружает регистр процессора индексным номером конкретной функции «родного» API (каждая функция «родного» API имеет уникальный индексный номер). Вторая инструкция загружает в регистр указатель на параметры вызова. Следующая инструкция – команда генерации программного исключения. ОС регистрирует обработчик ловушки для перехвата управления, переключения из пользовательского режима в режим ядра и передачи управления в фиксированную точку ОС при возникновении прерывания или исключения. В случае вызова системного сервиса (на процессорах x86 программное исключение для вызова системных сервисов генерируется кодом 0x2E), этот обработчик ловушки передает управление диспетчеру системных сервисов. Последняя инструкция забирает параметры из стека вызывающего потока.

Диспетчер системных сервисов определяет, является ли корректным индексный номер функции «родного» API. Индексный номер, переданный из пользовательского режима, используется для входа в таблицу распределения системных сервисов (KeServiceDescriptorTable). Каждый элемент этой таблицы включает указатель на соответствующий системный сервис и число параметров. Диспетчер системных сервисов берет параметры, переданные в стеке пользовательского режима (указатель стека находится в регистре edx) и помещает их в стек ядра, а затем передает управление для обработки запроса соответствующему системному сервису, который исполняется в режиме ядра и находится в ntoskrnl.exe.

Введенные в Windows NT версии 4.0 интерфейсы API Win32 управления окнами и рисованием управляются тем же диспетчером системных сервисов, но индексные номера Win32-функций указывают на то, что должен использоваться второй массив указателей системных сервисов. Указатели во втором массиве ссылаются на функции в win32k.sys.

Большинство системных сервисов должно выполнять проверку параметров, переданных им из пользовательского режима. Некоторые параметры являются указателя-

ми, а передача неверного указателя в режим ядра без предварительной проверки может привести к краху системы. Проверка параметров обязательна, но оказалось, что некоторые функции «родного» API (13 системных сервисов из win32k.sys) не выполняют обстоятельную проверку, что приводит в некоторых случаях к падению системы. Microsoft закрыла эти дыры в Service Pack 1. В дальнейшем оказалось, что при тестировании параметров, соответствующих граничным условиям, вызовы еще 40 функций «родного» API, из которых 25 из win32k.sys, вызвали падение системы. Эти дыры были закрыты в SP4.

После проверки параметров, системные сервисы обычно вызывают функции, реализуемые компонентами исполнительной системы: диспетчером процессов, диспетчером памяти, диспетчером ввода/вывода и средством локального вызова процедур.

Все функции прикладного уровня, вызывающие это прерывание, сосредоточены в модуле ntdll.dll и имеют в своем названии префикс Nt либо Zw, например, NtCreateFile()/ZwCreateFile(). Точка входа для двух таких имен одна. Вызов многих функций различных подсистем рано или поздно приведет к вызову соответствующей функции из ntdll.dll. При этом не все, что есть в ntdll.dll вызывается из подсистемы Win32.

Вызов системных сервисов возможен не только из прикладной программы, но и из ядра ОС, то есть из драйверов. Имена соответствующих функций ядра имеют префикс либо Zw, либо Nt (ZwCreateFile(), NtCreateFile()). Функции с префиксом Zw обращаются к сервисам посредством прерывания 2E, тогда как функции с префиксом Nt являются собственно точками входа стандартных системных сервисов. Из этого следует, что число функций с префиксом Nt неизменно, а множество этих функций является подмножеством функций с префиксом Zw. Не путайте функции с префиксами Nt и Zw режима ядра и пользовательского режима. В режиме ядра они находятся в модуле ntoskrnl.exe (микроядро), в пользовательском режиме – в модуле ntdll.dll («родной» API, вызывают int 2E).

2.3.2.2. Неотъемлемые подсистемы

Другой тип защищенных подсистем – неотъемлемые подсистемы – это серверы, выполняющие важные функции ОС. Примером неотъемлемой подсистемы является подсистема защиты, исполняющаяся в пользовательском режиме и реализующая правила контроля доступа, определенные для локального компьютера. Некоторые компоненты сетевого обеспечения Windows NT также реализованы как защищенные подсистемы, например, сервис рабочей станции реализует API для доступа и управления сетевым редиректором.

2.3.3. Исполнительная система (The Executive)

Надо отметить, что в разных источниках понятие исполнительной системы интерпретируется по-разному. Например, в документации DDK исполнительная сис-

тема – это совокупность компонентов, исполняющихся в привилегированном режиме – режиме ядра, и формирующих законченную ОС за исключением пользовательского интерфейса. В данном случае к компонентам исполнительной системы относятся также само ядро и слой абстрагирования от оборудования (HAL). В других источниках ядро и HAL рассматриваются как отдельные модули. В этой книге решено было следовать именно этому разделению, хотя бы потому, что HAL не предоставляет системных сервисов, к которым могут обращаться защищенные подсистемы.

Подсистемы Исполнительной Системы NT составляют наиболее существенный слой в режиме ядра, и они исполняют большую часть функций, традиционно связанных с операционными системами. В Таблице 1 перечислены подсистемы Исполнительной Системы NT, и рис. 2 показывает их позицию в архитектуре NT. Эти подсистемы имеют разные обязанности и названия, так что Вы могли бы подумать, что они являются различными процессами. Например, когда программа типа Microsoft Word запрашивает обслуживание операционной системы типа распределения памяти, поток управления передается от программы Word в режим ядра через «родной» интерфейс системных сервисов NT. Тогда обработчик системного сервиса для распределения памяти напрямую вызывает соответствующую функцию Диспетчера Виртуальной Памяти. Запрошенное распределение памяти выполняется в контексте процесса Word, который запросил его, то есть нет никакого переключения контекста к другому системному процессу.

Драйверы Windows NT, включая драйверы устройств, промежуточные драйверы и драйверы файловых систем, после загрузки рассматриваются как часть исполнительной системы, а точнее как часть системы ввода/вывода.

Все компоненты, кроме диспетчера кэша, предоставляют определенное множество системных сервисов, к которым могут обращаться защищенные подсистемы. И каждый компонент исполнительной системы реализует множество внутренних процедур, доступных только компонентам исполнительной системы.

Префиксы в названиях внутренних процедур соответствуют названиям компонентов исполнительной системы, обеспечивающих эти процедуры, например: «**Ex**» для функций, реализуемых компонентом **Ex**(ecutive) Support – исполнительным модулем, **Ps** – диспетчером процессов, **Ob** – диспетчером объектов, **Io** – диспетчером ввода/вывода, **Mm** – диспетчером памяти, **Cc** – диспетчером кэша, **Se** – монитором безопасности.

Исполнительная система не исполняется постоянно в собственном процессе, а работает в контексте некоторого существующего процесса, завладевая выполняющимся потоком, когда происходит важное системное событие. Например, когда поток вызывает системный сервис, в результате чего происходит программное прерывание, или когда внешнее устройство генерирует прерывание, ядро получает управление потоком, который выполнялся процессором. Оно выполняет соответствующий системный код для обработки события и затем возвращает управление коду, выполнявшемуся перед прерыванием.

Исполнительный модуль (executive support) – это особый компонент исполнительной системы ОС Windows NT, давший свое имя целой группе модулей операционной системы. Он отвечает за многие разнообразные функции, включая управление очередями (их блокирование), управление резидентной и нерезидентной системной областью памяти, увеличение/уменьшение значения глобальной переменной и др. Этот компонент обеспечивает также системные рабочие потоки, которые драйверы NT, особенно драйверы файловых систем, используют для выполнения необходимой работы.

Таблица 1. Подсистемы Исполнительной Системы NT и их предназначение

Подсистема исполнительной системы	**Предназначение**
Диспетчер Объектов (Object Manager)	Управляет ресурсами и реализует глобальное пространство имен
Монитор Безопасности (Security Reference Monitor)	Реализует модель безопасности NT на основе Идентификаторов Безопасности (SID) и Списков Разграничительного Контроля Доступа (Discretionary Access Control List - DACL)
Диспетчер Виртуальной Памяти (Virtual Memory Manager)	Определяет адресное пространство процесса и распределяет физическую память
Диспетчер Ввода/Вывода (I/O Manager)	Служит интерфейсом между прикладными программами и драйверами устройств
Диспетчер Кэша (Cache Manager)	Реализует глобальный файловый кэш
Средство Вызова Локальных Процедур (Local Procedure Call (LPC) Facility)	Обеспечивает эффективную межпроцессную коммуникацию
Диспетчер Конфигурации (Configuration Manager)	Управляет Реестром
Диспетчер Процессов (Process Structure)	Экспортирует программные интерфейсы (API) процессов и потоков
Поддержка среды Win32 (Win32 Support)	Реализует Win32-функции обмена сообщениями и рисования (новые для NT 4.0)
Диспетчер Plug-and-Play (Plug-and-Play Manager)	Уведомляет драйверы устройств о включении или отключении устройства (новые для NT 5.0)
Диспетчер Электропитания (Power Manager)	Контролирует состояние электропитания компьютера (появился в NT 5.0)
Исполнительный модуль (Executive Support)	Реализует управление очередями, системной областью памяти, обеспечивает системные рабочие потоки

Ниже перечислены компоненты исполнительной системы и их области ответственности.

Справочный монитор защиты (security reference monitor) отвечает за реализацию единой политики защиты на локальном компьютере. Оберегает ресурсы ОС, обеспечивая защиту объектов и аудит во время выполнения доступа к ним. Справочный монитор защиты использует для реализации единой системной политики безопасности списки контроля доступа (Access Control Lists, ACL), содержащие информацию о том, какие процессы имеют доступ к конкретному объекту и какие действия они могут над ним выполнять, и идентификаторы безопасности (Security Identifiers, SI). Он поддерживает уникальный для каждого потока профиль защиты, и проверку полномочий при попытке доступа к объектам. При открытии потоком описателя объекта активизируется подсистема защиты, сверяя ACL, связанный с объектом, с запрашиваемыми потоком действиями над этим объектом. Другим аспектом справочного монитора защиты является поддержка имперсонации (impersonation), которая позволяет одному потоку передать другому право использования своих атрибутов защиты. Это наиболее часто используется во время клиент-серверных операций, когда сервер использует атрибуты защиты клиента.

Диспетчер процессов (process manager или process structure) отвечает за создание и уничтожение процессов и потоков. Диспетчер процессов взаимодействует с диспетчером объектов для построения объекта-процесса и объекта-потока, а также взаимодействует с диспетчером памяти для выделения виртуального адресного пространства для процесса.

Средство локального вызова процедур (LPC) организует взаимодействие между клиентскими и серверными процессами, расположенными на одном и том же компьютере. LPC – это гибкая, оптимизированная версия удаленного вызова процедур (Remote Procedure Call, RPC), средства коммуникации между клиентскими и серверными процессами по сети. LPC поддерживает передачу данных между клиентом и сервером посредством использования объектов-портов, которые в качестве атрибутов имеют указатель на очередь сообщений и описатель секции разделяемой памяти. API, необходимый для доступа к LPC, не документирован. Интересно, что запрос RPC между приложениями, исполняющимися на одном компьютере, в действительности будет использовать механизм LPC.

Диспетчер памяти и диспетчер кэша (memory manager и cache manager). Диспетчер памяти и диспетчер кэша вместе формируют подсистему виртуальной памяти. Эта подсистема виртуальной памяти реализует 32-разрядную страничную организацию памяти. Подсистема виртуальной памяти поддерживает совместное использование страниц физической памяти между несколькими процессами. Она поддерживает разделяемый сегмент памяти «только для чтения», а также «чтения-записи». Подсистема виртуальной памяти отвечает за реализацию механизма кэширования данных. Данные файла могут быть доступны через диспетчера ввода/вывода при использовании стандартных операций чтения и записи в файл, или через диспетчер памяти посредством проецирования данных файла напрямую в виртуальное пространство процесса. Чтобы гарантировать согласованность между этими двумя методами доступа, диспетчер кэша поддерживает единый глобальный общий кэш. Этот единый кэш ис-

пользуется для кэширования, как страниц процесса, так и страниц файла. Диспетчер памяти реализует схему управления памятью, которая предоставляет каждому процессу 4-гигабайтное собственное виртуальное адресное пространство и защищает его от других процессов. Диспетчер памяти реализует механизм подкачки страниц (paging) – перенос страниц физической памяти на диск и обратно. Диспетчер кэша повышает производительность файлового ввода/вывода, сохраняя информацию, считанную с диска последней, в системной памяти. Диспетчер кэша использует средство подкачки страниц диспетчера памяти для автоматической записи информации на диск в фоновом режиме.

Поддержка среды Win32 (Win32 support) включает диспетчера окон (window manager), интерфейс графических устройств (Graphic Device Interface, GDI), драйверы графических устройств (graphic device drivers). Эти компоненты поддержки среды Win32 были перенесены в режим ядра в версии NT 4.0, а ранее они принадлежали подсистеме среды Win32. Эти средства взаимодействуют между GUI – приложениями и графическими устройствами.

Диспетчер конфигурации (configuration manager) определяет тип объект-ключ (key object) и манипулирует этими объектами. Тип объект-ключ представляет элемент реестра Windows NT. Каждый экземпляр типа объект-ключ представляет либо некоторый узел реестра, являющийся частью пути к множеству подключей, либо он содержит именованные поля с соответствующими значениями.

2.3.3.1. Диспетчер Объектов

Диспетчер Объектов (object manager), который является вероятно наименее известной из подсистем Исполнительной Системы NT, является также одним из наиболее важных. Главная роль операционной системы – это управление физическими и логическими ресурсами компьютера. Другие подсистемы Исполнительной Системы используют Диспетчер Объектов, чтобы определять и управлять объектами, которые представляют ресурсы.

Диспетчером объектов нельзя манипулировать из пользовательского режима напрямую, а его пространство имен является невидимым.

В таблице 2 приведен список объектов, определенных в NT 4.0, и подсистем исполнительной системы, которые управляют ими.

Диспетчер Объектов исполняет обязанности:

- Поддержание единого пространства имен для всех именованных объектов системы.
- Отвечает за создание, удаление и управление именованными и неименованными объектами ОС, представляющими системные ресурсы.

Обязанности по управлению объектами включают в себя идентификацию и подсчет ссылок. Когда прикладная программа открывает ресурс, Диспетчер Объектов или определяет местонахождение связанного с ресурсом объекта, или создает новый объект. Вместо возвращения прикладной программе, которая открыла ресурс, указателя на объект, Диспетчер Объектов возвращает непрозрачный (не имеющий смысла) идентиф-

фикатор, называемый дескриптором. Значение дескриптора уникально в рамках прикладной программы, которая открыла ресурс, но не уникально между различными прикладными программами.

Таблица 2. Типы объектов и подсистемы исполнительной системы, которые ими управляют

Тип Объекта	Какой ресурс представляет	Подсистема
Тип Объекта (Object type)	Объект типа объекта	Диспетчер объектов
Директория (Directory)	Пространство имен объектов	Диспетчер объектов
Символическая Связь (SymbolicLink)	Пространство имен объектов	Диспетчер объектов
Событие (Event)	Примитив синхронизации	Исполнительный модуль
Пара Событий (Event-Pair)	Примитив синхронизации	Исполнительный модуль
Мутант (Mutant)	Примитив синхронизации	Исполнительный модуль
Таймер (Timer)	Таймерное предупреждение	Исполнительный модуль
Семафор (Semaphore)	Примитив синхронизации	Исполнительный модуль
Станция Windows (Windows Station)	Интерактивный вход в систему	Поддержка среды Win32
Рабочий Стол (Desktop)	Рабочий Стол Windows	Поддержка среды Win32
Файл (File)	Отслеживание открытых файлов	Диспетчер ввода/вывода
Завершение ввода/вывода (I/O Completion)	Отслеживание завершения ввода/вывода	Диспетчер ввода/вывода
Адаптер (Adapter)	Ресурс прямого Доступа к Памяти (DMA)	Диспетчер ввода/вывода
Контроллер (Controller)	Контроллер DMA	Диспетчер ввода/вывода
Устройство (Device)	Логическое или физическое устройство	Диспетчер ввода/вывода
Драйвер (Driver)	Драйвер устройства	Диспетчер ввода/вывода
Ключ (Key)	Вход в реестре	Диспетчер конфигурации
Порт (Port)	Канал связи	Средство LPC
Секция (Section)	Отображение в памяти	Диспетчер памяти
Процесс (Process)	Активный процесс	Диспетчер процессов
Поток (Thread)	Активный поток	Диспетчер процессов
Маркер (Token)	Профиль безопасности процесса	Диспетчер процессов
Профиль (Profile)	Измерение производительности	Ядро

Прикладная программа использует дескриптор, чтобы идентифицировать ресурс в последующих операциях. Когда прикладная программа закончила работу с объектом, она закрывает дескриптор. Диспетчер Объектов использует подсчет ссылок, чтобы проследить сколько элементов системы, включая прикладные программы и подсистемы Исполнительной Системы, обращаются к объекту, который представляет ресурс. Когда счетчик ссылок обнуляется, объект больше не используется как представление ресурса, и Диспетчер Объектов удаляет объект (но не обязательно ресурс).

Для обеспечения идентификации объектов, Диспетчер Объектов реализует пространство имен NT. Все разделяемые ресурсы в NT имеют имена, располагающиеся в этом пространстве имен. Например, когда программа открывает файл, Диспетчер Объектов анализирует имя файла для выявления драйвера файловой системы (FSD) для диска, который содержит файл. Точно так же, когда прикладная программа открывает ключ Реестра, Диспетчер Объектов по имени ключа Реестра определяет, что должен быть вызван Диспетчер Конфигурации.

Рассмотрим следующий пример:

Прикладная программа вызывает функцию Win32 – CreateFile() с именем файла «c:\mydir\file.txt». При этом происходят следующие действия:

1) Вызов системного сервиса NtCreateFile(). В качестве имени ему будет передано «\??\c:\mydir\file.txt». Такой формат имени является «родным» для NT, точнее – это формат имени в пространстве имен Диспетчера Объектов.

2) Диспетчер Объектов начнет последовательно разбирать переданное имя. Первым будет разобран элемент «\??». Корень пространства имен содержит объект с таким именем. Тип объекта – «Directory». В этой директории будет произведен поиск объекта с именем «c:». Это – «SymbolicLink» – ссылка на имя «\Device\Harddisk0\ Partition1». Дальнейшему разбору будет подвергнуто имя «\Device\Harddisk0\Partition1\ mydir\file.txt». Разбор будет закончен при достижении объекта, не являющегося директорией или символической связью. Таким объектом будет «Partition1», имеющий тип «Device». Этому объекту для дальнейшей обработки будет передано имя «\mydir\file.txt».

2.3.3.2. Система ввода/вывода

Система ввода/вывода исполнительной системы – это часть кода ОС, получающая запросы ввода/вывода от процессов пользовательского режима и передающая их, в преобразованном виде, устройствам ввода/вывода. Между сервисами пользовательского режима и аппаратурой ввода/вывода располагается несколько отдельных системных компонентов, включая законченные файловые системы, многочисленные драйверы устройств и драйверы сетевых транспортов.

Система ввода/вывода управляется пакетами запроса ввода/вывода (I/O Request Packet, IRP). Каждый запрос ввода/вывода представляется в виде пакета IRP во время его перехода от одной компоненты системы ввода/вывода к другой. IRP – это структура данных, управляющая обработкой операции ввода/вывода на каждой стадии ее выполнения.

В систему ввода/вывода входят следующие компоненты:

1) Диспетчер ввода/вывода (I/O manager). Реализует средства ввода/вывода, не зависящие от типа устройства, и устанавливает модель для ввода/вывода исполнительной системы. Диспетчер ввода/вывода осуществляет создание, чтение, запись, установку и получение информации, и многие другие операции над файловыми объектами. Диспетчер ввода/вывода реализует асинхронную подсистему ввода/вывода, основанную на передаче пакетов запроса ввода/вывода (I/O Request Packet, IRP). Диспетчер ввода/вывода также отвечает за поддержку и обеспечение операционной среды для драйверов.

2) Файловые системы. Драйверы, принимающие запросы файлового ввода/вывода и транслирующие их в запросы, привязанные к конкретному устройству. Сюда же входят сетевые файловые системы, состоящие из двух компонентов: сетевого редиректора (network redirector), реализуемого как драйвер файловой системы и передающего удаленные запросы ввода/вывода на машины в сети, и сетевого сервера (network server), являющегося обычным драйвером, принимающим и обрабатывающим такие запросы.

3) Сетевые драйверы, которые могут загружаться в ОС и рассматриваться как часть системы ввода/вывода.

4) Драйверы устройств. Низкоуровневые драйверы, напрямую работающие с оборудованием.

Диспетчер ввода/вывода (I/O manager) определяет порядок, по которому запросы ввода/вывода доставляются драйверам. В обязанности диспетчера входит:

1) Получение запроса на ввод/вывод и создание пакета IRP.

2) Передача IRP соответствующему драйверу. Драйвер, получив IRP, выполняет указанную в нем операцию ввода/вывода, и, либо возвращает его диспетчеру ввода/вывода для завершения обработки, либо передает другому драйверу для продолжения операции ввода/вывода.

3) Сопровождение IRP по стеку драйверов.

4) Завершение IRP по окончании операции ввода/вывода и возвращение результатов обработки инициатору запроса ввода/вывода.

5) Также диспетчер ввода/вывода реализует общие процедуры, к которым обращаются драйверы во время обработки ввода/вывода, и предоставляет системные сервисы, позволяющие защищенным подсистемам реализовать свои API ввода/вывода.

2.3.3.3. Ядро

Ядро ОС Windows NT реагирует на прерывания и исключения, занимается планированием потоков, сохранением и восстановлением контекстов потоков, направляет потоки на выполнение, выполняет межпроцессорную синхронизацию, предоставляет набор сервисов, элементарных объектов и интерфейсов, используемых компонентами исполнительной системы. Большая часть ядра зависит от типа процессора.

Драйверы NT и компоненты исполнительной системы вызывают внутренние процедуры, обеспечиваемые ядром, названия которых начинаются с префикса **Ke**(rnel) – ядро.

Ядро экспортирует два основных типа объектов ядра: объекты-диспетчеры (dispatcher objects) и управляющие объекты (control objects). Объекты ядра отличаются от объектов исполнительного уровня, создаваемых и управляемых менеджером объектов, и зачастую являются базисом для них.

Объект-диспетчер используется для планирования и синхронизации и имеет атрибут, определяющий его состояние – «занят» или «свободен». Объектами-диспетчерами являются: события, мьютексы, семафоры и таймеры.

Управляющие объекты используются для управления системными операциями. Управляющими объектами являются: APC-объект (Asynchronous Procedure Call), содержащий адрес процедуры асинхронного вызова и указатель на объект-поток, который будет исполнять данный вызов; DPC-объект (Deferred Procedure Call), содержащий адрес процедуры отложенного вызова; объект-прерывание, отвечающий за установление соответствия между определенным вектором прерывания и процедурой обработки прерывания (Interrupt Service Routine, ISR) драйвера устройства.

2.3.3.4. Слой абстрагирования от оборудования

Слой абстрагирования от оборудования (Hardware Abstraction Layer, HAL) является относительно тонким слоем кода, взаимодействующим напрямую с процессором, шинами и другим оборудованием, и отвечает за обеспечение стандартного интерфейса к платформенно-зависимым ресурсам для ядра, диспетчера ввода/вывода и драйверов устройств.

Вместо того чтобы обращаться к аппаратуре непосредственно исполнительная система сохраняет максимальную переносимость, обращаясь к функциям HAL, когда ей нужна платформенно-зависимая информация (некоторый объем кода, который зависит от конкретной архитектуры, располагается не только в HAL, но и в ядре и в менеджере памяти). Драйверы устройств содержат, конечно же, код, зависящий от устройств, но избегают кода, зависящего от процессора или платформы, вызывая процедуры ядра и HAL.

HAL обеспечивает поддержку и отвечает за предоставление стандартного интерфейса к ресурсам процессора, которые могут меняться в зависимости от модели внутри одного семейства процессоров. Возможность замены слоя HAL обеспечивает всем вышележащим слоям операционной системы независимость от аппаратной архитектуры.

Внутренние процедуры, обеспечиваемые слоем абстрагирования от оборудования, начинаются с префикса **Hal.**

2.3.4. Система приоритетов

Windows NT имеет двухуровневую модель приоритетов (см. рис. 4).

- Приоритеты высшего уровня (**уровни запросов прерываний** – Interrupt ReQuest Level – IRQL) управляются аппаратными и программными прерываниями.
- Приоритеты низшего уровня (**приоритеты планирования**) управляются планировщиком.

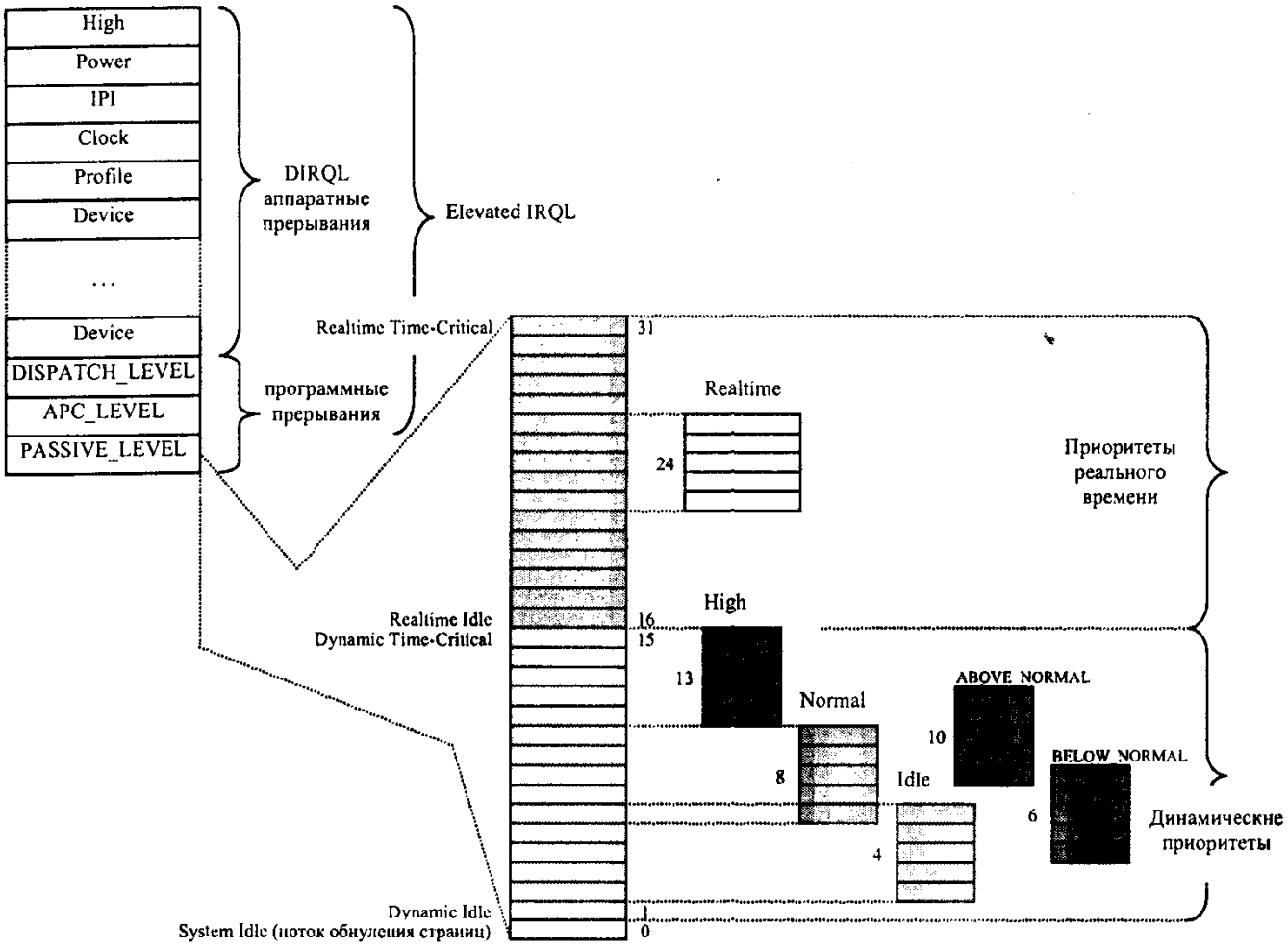


Рис. 4. Система приоритетов

2.3.4.1. Уровни запросов прерываний (IRQL)

В любое время исполняющийся код будет иметь определенный уровень IRQL. Этот уровень определяет, что позволено делать коду, применяется ли к коду механизм квантования времени планировщика и каковы его взаимосвязи с другими потоками исполнения.

Наивысшие из уровней IRQL – **уровни запросов прерываний устройств** (*Device Interrupt Request Levels* – **DIRQLs**). Это уровни IRQL, соответствующие аппаратным прерываниям. Другие уровни IRQL реализованы программно.

Уровень IRQL прерывания контролирует то, когда прерывание может быть обработано. Прерывание никогда не будет обработано, пока процессор занят обработкой прерывания более высокого уровня. Уровни IRQL располагаются в порядке убывания от HIGH_LEVEL до PASSIVE_LEVEL.. Уровни в подмножестве от HIGH_LEVEL до APC_LEVEL называют **повышенными** (elevated IRQLs). DISPATCH_LEVEL и APC_LEVEL реализованы программно.

Модель приоритетов низшего уровня управляет исполнением потоков, выполняющихся на уровне IRQL PASSIVE_LEVEL. Этот уровень контролируется **планировщиком** (*scheduler*) (его также называют **диспетчером** – *dispatcher*), который планирует исполнение потоков (но не процессов). Планировщик планирует исполнение прикладных и системных потоков, используя для наблюдения и контроля исполнения потоков системные часы.

2.3.4.2. Приоритеты планирования

Каждому потоку назначается приоритет планирования. Имеется 32 уровня приоритетов планирования со значениями 0–31. Низший приоритет планирования со значением 0 зарезервирован для потока обнуления страниц (*Zero Page Thread*), который выполняется в случае, когда больше нечего исполнять. Этот поток является компонентом диспетчера памяти, и его работа состоит в обнулении страниц из списка свободных страниц. Когда диспетчер памяти получает запрос на выдачу обнуленной страницы памяти, диспетчер памяти вначале попробует выделить страницу, обнуленную потоком обнуления страниц, и только если таких страниц нет, он потратит время на обнуление.

2.3.4.2.1.1. Динамические приоритеты и приоритеты реального времени

Приоритеты планирования делятся на две главных группы:

- динамические приоритеты (dynamic priorities);
- приоритеты реального времени (real-time priorities).

Динамические приоритеты имеют значения в диапазоне 1-15. Они названы динамическими, потому что ОС может динамически изменять приоритет потока в этом диапазоне.

Приоритеты реального времени имеют значения в диапазоне 16-31. ОС не может изменять значение приоритета потока, находящееся в этом диапазоне.

Имеется два важных отличия между динамическими приоритетами и приоритетами реального времени.

Поток с приоритетом реального времени может сохранять контроль над процессором до тех пор, пока не появится поток с большим или равным значением приоритета. Таким образом, пока выполняется поток реального времени, потоки с меньшим значением приоритета никогда не получат шанса исполниться (механизм вытесняющей многозадачности не задействован). Такой поток должен сам освободить процессор.

Однако в любом случае при появлении потока с большим или равным значением приоритета задействуется механизм вытесняющей многозадачности.

В случае потоков с динамическими приоритетами, потоки с меньшими приоритетами также не могут получить шанса на исполнение, пока готовы к исполнению потоки с большими приоритетами.

Однако, в ряде случаев планировщик повышает приоритет потоков в диапазоне динамических приоритетов. Это дает возможность рано или поздно выполниться любому потоку с приоритетом в этом диапазоне.

К механизму повышения приоритетов применимы следующие утверждения:

1) Система никогда не меняет приоритет потоков из диапазона приоритетов реального времени.
2) Повышение приоритета не может вызвать его переход в диапазон приоритетов реального времени, то есть превысить значение 15.
3) Повышение приоритета операционной системой является временным. Каждый раз, когда поток исчерпывает отведенный ему квант времени, значение его приоритета уменьшается на единицу. Так происходит до достижения значения базового приоритета.
4) Операционная система не может снизить приоритет ниже уровня базового приоритета.
5) Повышение приоритета может происходить несколько раз подряд.

2.3.4.2.2. Базовый приоритет. Класс приоритета и относительный приоритет

Ядро NT предоставляет функции для назначения потоку любого из 31 уровня приоритетов (кроме зарезервированного нулевого уровня). Программно назначенное потоку значение приоритета называют **базовым приоритетом**.

Подсистема Win32 не позволяет непосредственно назначать потоку базовое значение приоритета. Вместо этого используется комбинация двух значений:

- Класс приоритета процесса, назначаемый процессу при его создании (дальше мы будем ссылаться на этот термин как на **класс приоритета**).
- Относительное значение приоритета потока внутри класса приоритета процесса (**относительный приоритет**).

На рис. 4 показаны группы взаимосвязанных приоритетов. Максимальный (31) и минимальный (1) из возможных приоритетов определяются в Win32 как THREAD_PRIORITY_TIME_CRITICAL и THREAD_PRIORITY_IDLE соответственно.

В таблице 3 приведены все возможные классы приоритетов. По умолчанию, процесс имеет класс NORMAL_PRIORITY_CLASS, если при вызове функции CreateProcess() не было указано другого. Прикладная программа может получать/изменять класс приоритета процесса с помощью функций Win32-API *GetPriotityClass*() / *SetPriorityClass*().

Таблица 3. Классы приоритетов

Класс приоритета	Базовый приоритет	Примечание
REALTIME_PRIORITY_CLASS	24	
HIGH_PRIORITY_CLASS	13	
ABOVE_NORMAL_PRIORITY_CLASS	10	Только Win 2000
NORMAL_PRIORITY_CLASS	8	
BELOW_NORMAL_PRIORITY_CLASS	6	Только Win 2000
IDLE_PRIORITY_CLASS	4	

Поток может иметь одно из 7 значений (см. таблицу 4): 5 значений, относительных внутри каждого класса приоритетов, и 2 значения, относительных внутри диапазонов динамического приоритета и приоритетов реального времени.

Таблица 4. Относительный приоритет

Относительный приоритет	
THREAD_PRIORITY_TIME_CRITICAL	15(31)
THREAD_PRIORITY_HIGHEST	+2
THREAD_PRIORITY_ABOVE_NORMAL	+1
THREAD_PRIORITY_NORMAL	+0
THREAD_PRIORITY_BELOW_NORMAL	-1
THREAD_PRIORITY_LOWEST	-2
THREAD_PRIORITY_IDLE	1(16)

Два значения, обозначающие минимальное и максимальное значение приоритета внутри диапазона динамических приоритетов и приоритетов реального времени – это THREAD_PRIORITY_IDLE и THREAD_PRIORITY_ TIME_CRITICAL. Для диапазона динамических приоритетов они обозначают базовые приоритеты 1 и 15, а для диапазона приоритетов реального времени – 16 и 31 соответственно.

Любой поток всегда создается с относительным приоритетом THREAD_ PRIORITY_NORMAL. Соответствующие значения базового приоритета в зависимости от класса приоритета указаны в таблице 3.

Относительный приоритет потока может быть получен/изменен с помощью WIN32-функций GetThreadPriority()/SetThreadPriority().

Необходимо отметить, что служебные потоки ОС, производящие операции с мышью и клавиатурой, а также некоторые файловые операции, работают с приоритетом реального времени. Поэтому использование пользовательскими потоками таких приоритетов может повлиять на корректность функционирования ОС.

2.3.4.3. Как используются IRQL

NT управляет прерываниями путем отображения уровней прерывания контроллера прерываний в собственную аппаратно-независимую таблицу уровней прерываний. Осуществляет отображение слой абстрагирования от оборудования (HAL – модуль NT, специально написанный для конкретных контроллеров прерываний, материнских плат, либо наборов микросхем процессоров). В мультипроцессорных системах принять прерывание может любой процессор, поэтому NT поддерживает *независимый IRQL для каждого процессора*. IRQL процессора представляет уровень прерывания, который маскируется в данный момент процессором и прямо соответствует (аналогичен) прерываниям, которые маскирует контроллер прерываний CPU. Поскольку уровни IRQL NT не привязаны к какой-либо спецификации оборудования, NT также может отображать в свою иерархию приоритетов неаппаратные типы прерываний. Операционная

система использует программные прерывания в основном для запуска операций планирования, таких как переключение потоков или обработка завершения ввода/вывода.

Когда NT обслуживает аппаратное прерывание, NT устанавливает IRQL процессора в соответствующее значение таблицы IRQL NT. NT программирует контроллер прерываний так, чтобы он маскировал прерывания с более низким приоритетом, и драйверы устройств (так же как и NT) могут запрашивать IRQL для определения его значения. (Как мы увидим позднее, NT позволяет выполнять некоторые операции только когда IRQL меньше определенных значений.)

Размер таблицы IRQL разнится между архитектурами процессоров (Intel, Alpha и др.) для того, чтобы лучше отображать уровни прерываний, предоставляемые контроллерами прерываний, однако уровни прерываний, которые могут найти интересными разработчики драйверов устройств и разработчики NT, имеют символические имена. Таблица 5 представляет символические имена IRQL и соответствующие им числовые значения в архитектурах Intel и Alpha.

Низшие уровни IRQL (от passive_level до dispatch_level) используются для синхронизации программных частей операционной системы. Эти IRQL сконструированы как программные прерывания. Уровни IRQL выше dispatch_level, имеют ли они конкретные мнемонические имена или нет, отражают приоритеты аппаратных прерываний. Таким образом, эти аппаратные IRQL часто упоминаются как уровни IRQL Устройства (или DIRQL).

Конкретные значения, назначенные мнемоническим именам IRQL, изменяются от системы к системе. Взаимоотношения между программными уровнями IRQL от системы к системе остаются постоянными; верным также остается положение о том, что программные уровни IRQL имеют более низкий приоритет, чем аппаратные IRQL. Таким образом, IRQL passive_level всегда является самым низким уровнем IRQL в системе, apc_level всегда выше, чем passive_level, и dispatch_level всегда выше, чем apc_level. Все эти уровни IRQL всегда ниже, чем самый низкий уровень DIRQL.

Таблица 5. Символические и числовые определения IRQL

Символическое имя	**Предназначение**	**Уровень Intel**	**Уровень Alpha**
HIGH_LEVEL	Наивысший уровень прерывания	31	7
POWER_LEVEL	Power event	30	7
IPI_LEVEL	Межпроцессорный сигнал	29	6
CLOCK_LEVEL	такт системных часов	28	5
PROFILE_LEVEL	Контроль производительности	27	3
DEVICE LEVEL	Обычные прерывания устройств	3-26	3-4
DISPATCH_LEVEL	Операции планирования и отложенные вызовы процедур (DPC)	2	2
APC_LEVEL	Асинхронные вызовы процедур (APC)	1	1
PASSIVE_LEVEL	Нет прерываний	0	0

В отличие от программных IRQL, значения и отношения аппаратных IRQL могут изменяться в зависимости от реализации аппаратной части системы. Например, в архитектурах на основе x86, уровень IRQL profile_level ниже, чем IRQL ipi_level, который является в свою очередь ниже, чем IRQL power_level. Однако, на MIPS системах, IRQL power_level и IRQL ipi_level имеют то же самое значение, и оба ниже, чем IRQL profile_level.

Уровни IRQL являются главным методом, используемым для расположения по приоритетам действий операционной системы Windows NT. Повышение уровня IRQL позволяет подпрограмме операционной системы как управлять повторной входимостью (реентерабельность) так и гарантировать, что она может продолжать работу без приоритетного прерывания (вытеснения) некоторыми другими действиями. Следующие разделы описывают, как используются наиболее распространенные уровни IRQL.

2.3.4.3.1. IRQL PASSIVE_LEVEL, APC_LEVEL и DISPATCH_LEVEL

Наименьший приоритет IRQL в таблице 5 – **Passive Level**. Этот уровень является обычным уровнем IRQL, на котором производится работа в операционной системе, как в пользовательском режиме, так и в режиме ядра. Когда процессор находится в этом состоянии, не происходит никакой деятельности по обработке прерываний. Подпрограмма, выполняющаяся на уровне IRQL passive_level, может быть подвергнута прерыванию и вытеснению почти всем, чем угодно еще случившемся в системе. Так, потоки, выполняющиеся на IRQL passive_level, подвергаются вытеснению Диспетчером (планировщиком) по истечении их кванта времени.

Большинство подпрограмм уровня исполнительной системы Windows NT (то есть подпрограммы режима ядра, не принадлежащие Микроядру и HAL) стремятся держать уровень IRQL как можно более низким. В большинстве случаев, это приводит к выполнению большинства подпрограмм на уровне IRQL passive_level. Эта стратегия повышает возможность выполнения действий с высоким уровнем IRQL.

Следующие два уровня IRQL выше Passive Level (APC Level и Dispatch Level) – программные уровни прерываний, связанные с планировщиком.

Когда система находится на уровне **APC Level**, исполняющийся поток не будет получать запросы APC, которые NT обычно использует для операций завершения ввода/вывода.

IRQL dispatch_level используется внутри Windows NT для двух различных действий:

- Обработка Отложенных Вызовов Процедур (DPCs);
- Выполнение Диспетчера (планировщик NT).

DPC обработка обсуждена позже в этой главе, в собственном разделе. Следовательно, мы ограничимся обсуждением Диспетчера. Диспетчер, является планировщиком потоков Windows NT. Он отвечает за реализацию алгоритма планирования, который выбирает, какой поток будет выполняться, и осуществляет приоритетное прерывание (выгрузку) в конце кванта времени.

Диспетчер (планировщик) Windows NT получает запросы, чтобы выполнить операцию перепланирования на уровне IRQL dispatch_level. Когда операционная система решает заменить поток, который выполняется на текущем процессоре, она иногда может вызывать Диспетчер напрямую. Однако, когда система выполняется на уровне IRQL выше, чем dispatch_level, она запрашивает программное прерывание уровня dispatch_level. Результатом явится запуск на текущем процессоре Диспетчера в следующий раз, когда уровень dispatch_level станет наиболее приоритетным для обслуживания системой.

Рассмотрим, например, случай потока, выполняющегося в режиме пользователя. Так как он выполняется в режиме пользователя, этот поток, конечно, выполняется на IRQL passive_level. В то время как поток выполняется, часы периодически генерируют прерывания, чтобы указать операционной системе прохождение промежутка времени. С каждым переданным тиком часов, программа обработки прерывания часов уменьшает остающийся у выполняющегося в данный момент потока квант времени. Когда оставшийся у потока квант уменьшается до нуля, программа обработки прерывания часов генерирует прерывание уровня dispatch_level, чтобы запросить запуск Диспетчера и выбор им следующего потока для выполнения. Так как программа обработки прерывания часов выполняется на уровне IRQL большем, чем dispatch_level (она выполняется на уровне IRQL CLOCK2_LEVEL на x86 процессорах), обработка запроса для Диспетчера откладывается.

После генерирования прерывания уровня dispatch_level, программа обработки прерывания часов заканчивает любую другую работу, которую она должна сделать и управление возвращается Микроядру.

Затем Микроядро распознает следующее самое высоко приоритетное прерывание, которое находится в режиме ожидания. Каждое прерывание обслуживается по очереди. Когда для обслуживания не остается никаких прерываний уровня выше dispatch_level, выполняется программа обработки прерывания уровня dispatch_level. Эта программа обработки прерывания обрабатывает список DPC (обсуждаемый позже), и задействует Диспетчер, чтобы выбрать новый поток выполнения.

Когда задействуется Диспетчер, он обращает внимание, что квант времени текущего потока был уменьшен до нуля. Затем Диспетчер осуществляет алгоритм планирования Windows NT, чтобы определить следующий поток, который нужно запланировать. Если выбран новый поток (мог бы быть перепланирован предыдущий поток), происходит переключение контекста. Если нет никаких ожидающих вызовов APC для вновь выбранного потока, код потока будет выполнен, когда система возвратится обратно к уровню IRQL PASSIVE_LEVEL.

2.3.4.3.1.1. Ограничения, налагаемые на код с уровнем IRQL большим или равным DISPATCH_LEVEL

IRQL dispatch_level имеет важное значение в Windows NT. Как уже говорилось выше, Диспетчер (планировщик) Windows NT получает запросы, чтобы выполнить

операцию перепланирования, на уровне IRQL dispatch_level. Этот факт имеет три важных следствия:

- Любой поток с IRQL >= DISPATCH_LEVEL не подвержен механизму планирования.
- Любой поток с IRQL >= DISPATCH_LEVEL не может использовать никакие функции ожидания Диспетчерских Объектов ядра с отличным от нуля временем ожидания.
- Любой поток с IRQL >= DISPATCH_LEVEL не должен приводить к ошибкам отсутствия страниц памяти (что происходит при обращении к участку памяти, находящемуся на выгруженной на диск странице памяти). Иными словами, на таких уровнях IRQL может быть использована только невыгружаемая память (**nonpaged pool**, организация памяти будет рассмотрена в следующем разделе).

Рассмотрим эти пункты более подробно.

Так как Диспетчер выполняется на уровне IRQL DISPATCH_LEVEL, любая подпрограмма, которая выполняется на IRQL dispatch_level или выше, не подчиняется приоритетному прерыванию (выгрузке). Таким образом, когда квант времени потока истекает, если этот поток выполняется в настоящее время на IRQL dispatch_level или выше, он продолжит выполняться, пока не попытается понизить IRQL текущего процессора ниже dispatch_level. Это должно быть очевидно, так как исполнение на некотором уровне IRQL блокирует распознавание других событий, запрошенных на этом же или более низком уровне IRQL.

Что может быть менее очевидно, так это то, что, когда код выполняется на уровне IRQL dispatch_level или выше, он не может ждать никакие диспетчерские объекты (Dispatcher Object – см. раздел «Механизмы синхронизации»), которые еще не переведены в сигнальное состояние (состояние «свободен»). Таким образом, например, код, выполняющийся на уровне IRQL dispatch_level или выше, не может ожидать установки объектов событие или мьютекс. Так происходит потому, что действие освобождения процессора (которое происходит, когда поток переходит в режим ожидания события) требует (по крайней мере, концептуально) запуска Диспетчера. Однако, если подпрограмма выполняется на уровне dispatch_level или выше, прерывание уровня dispatch_level (по которому запускается Диспетчер) будет маскировано и, следовательно, распознано не сразу. В результате происходит возврат обратно к коду, который вызвал операцию ожидания!

Еще менее очевидным может быть тот факт, что код, выполняющийся на уровне IRQL dispatch_level или выше не должен приводить к ошибкам отсутствия страниц (page faults). Это означает, что любой такой код сам должен быть невыгружаемым, и должен обращаться только к невыгружаемым структурам данных. В основном это объясняется тем, что код, выполняющийся на IRQL dispatch_level или выше не может ждать освобождения диспетчерского объекта. Таким образом, даже если бы страничный запрос был обработан, поток с ошибкой отсутствия страницы не мог бы быть приостановлен, пока необходимая страница читалась с диска.

2.3.4.3.2. DIRQLs

Все уровни IRQL выше Dispatch Level относятся к аппаратным прерываниям. Аппаратные прерывания периферийных устройств системы (например, дисков, клавиатур, последовательных портов) отображаются на уровни IRQL в диапазоне Device Level. Из Таблицы 5 вы можете видеть, что на процессорах Intel этот диапазон лежит от 3 до 26, а на машинах Alpha – от 3 до 4. Тот факт, что существует такая разница между двумя диапазонами, имеет следствием то, что NT в действительности не располагает обычные прерывания устройств в соответствии с приоритетом. Даже на процессорах Intel, где аппаратные прерывания могут иметь различные значения IRQL, назначение IRQL является случайным.

Так как это может быть важным моментом для разработчиков драйвера устройства в некоторых системах, необходимо повторить снова: связь между двумя IRQ, назначенными двум определенным устройствам не обязательно сохраняется, когда IRQL назначены этим устройствам. Назначен ли устройству с более важным IRQ более высокий (то есть более важный) уровень IRQL, полностью зависит от HAL. В действительности, в большинстве HAL стандартных многопроцессорных систем x86, для систем, которые используют архитектуры APIC, связь между IRQ и IRQL не сохраняется.

Уровни IRQL выше Device Level имеют предопределенные связи с определенными прерываниями. Profile Level относится к таймеру профилирования ядра (механизму измерения производительности системы), Clock Level относится к такту системных часов, IPI Level относится к сигналам, посылаемым от одного CPU к другому, и Power Level относится к событиям сбоя в питании.

NT резервирует, но в настоящий момент не использует IRQL High Level.

IRQL high_level всегда определяется как самый высокий уровень IRQL в системе Windows NT. Этот уровень IRQL используется для NMI (Немаскируемого Прерывания) и других прерываний очень высокого приоритета. В редких случаях, когда драйвер устройства нуждается в блокировании прерываний на конкретном процессоре на короткий период, драйвер может поднимать IRQL до уровня high_level, но такое повышение уровня IRQL драйвером устройства считается очень решительным шагом, и в Windows NT это почти не требуется.

Подъем IRQL до уровня HIGH_LEVEL большинству драйверов Windows NT желательно никогда не делать. Блокирование прерываний является широко используемым методом для достижения синхронизации на других операционных системах (типа DOS или Win9x). Однако, в Windows NT, простое поднятие до IRQL HIGH_LEVEL в целях синхронизации не будет работать на многопроцессорных системах. Код режима ядра выполняет сериализацию, используя спин-блокировки, которые подробно описаны в разделе "Механизмы синхронизации".

2.3.4.4. Прерывания и планирование

Сведения об IRQL в Таблице 5 показывают, что уровень Dispatch Level связан с операциями планирования. Когда уровень IRQL соответствует Dispatch Level или выше,

NT маскирует программные прерывания планировщика, что означает, что NT отключает планировщик. Фактически, драйверы устройств (и NT) не должны осуществлять операции, требующие немедленного ответа от планировщика, когда процессор находится на уровне IRQL большем или равном Dispatch Level.

Это ограничение включает в себя выполнение всего, что может указывать NT, что текущий поток уступает CPU для ожидания осуществления некоторого события, так как эта акция заставит планировщик искать новый поток для исполнения.

Другое действие, которое требует вмешательства планировщика, это ошибка отсутствия страницы. Когда поток обращается к виртуальной памяти, ссылающейся на данные в страничном файле, NT обычно блокирует поток до тех пор, пока данные не будут прочитаны.

Поэтому на уровне Dispatch Level или выше NT не позволяет доступ к памяти, не заблокированной в физической памяти.

Если вы когда либо видели код останова синего экрана IRQL_NOT_LESS_OR_EQUAL, вы вероятно были свидетелем эффекта от нарушения драйвером этих правил.

Отключение планировщика во время обработки прерывания имеет другой, менее очевидный эффект: NT подсчитывает время, затраченное функциями ISR (процедурами обработки прерываний) и DPC (вызовами отложенных процедур) на фоне величины времени, которое поток был активным к моменту, когда CPU получил прерывание.

Например допустим, что Word выполняет операцию проверки правописания, от устройства поступает прерывание, и драйвер устройства имеет DPC, которое отбирает весь квант времени программы Word (а затем еще сколько-то). Когда IRQL процессора упадет ниже уровня Dispatch Level, планировщик может решить переключиться на другой поток другого приложения, чувствительно наказывая Word за обработку прерывания.

Хотя такая практика кажется несправедливой, почти каждый раз, когда NT распределяет прерывание, оно равномерно распределяется между прикладными программами в системе.

2.3.4.5. Определение текущего уровня IRQL

Текущий уровень IRQL свой у каждого CPU. Код режима ядра может определить IRQL, в котором он выполняется, посредством вызова функции KeGetCurrentIrql (), прототип которой:

```
KIRQL KeGetCurrentIrql ();
KeGetCurrentIrql() возвращает IRQL текущего CPU.
```

Большинство подпрограмм драйвера устройства вызывается Диспетчером ввода/вывода на определенном архитектурой уровне IRQL. То есть разработчик драйвера знает уровень (или уровни) IRQL, на котором будет вызываться данная функция. Подпрограммы режима ядра могут изменять IRQL, на котором они выполняются, вызывая функции KeRaiseIrql() и KeLowerIrql(), прототипы которых:

```
VOID KeRaiseIrql (IN PKIRQL NewIrql, OUT PKIRQL OldIrql);
```

Где: *Newlrql* – значение, до которого должен быть поднят уровень IRQL текущего процессора; *Oldlrql* – указатель на место, в которое будет помещен IRQL, на котором текущий процессор выполнялся перед тем, как был поднят к Newlrql.

```
VOID KeLowerIrql (IN KIRQL Newlrql);
```

Где: *Newirql* – значение, до которого должен быть понижен IRQL текущего процессора.

Так как уровни IRQL являются методом синхронизации, большинство подпрограмм режима ядра (в особенности драйверы устройств) никогда не должны понижать свой уровень IRQL ниже того, на котором они вызывались. Таким образом, драйверы могут вызывать KeRaiselrql (), чтобы поднять IRQL до более высокого уровня, и затем вызывать KeLowerlrql(), чтобы возвратиться обратно к первоначальному уровню IRQL, на котором они были вызваны (например, из Диспетчера ввода/вывода). Однако драйвер никогда не должен вызывать функцию KeLowerlrql(), чтобы понизить IRQL до уровня, меньшего, чем тот, на котором он был вызван. Такое поведение может привести к крайне непредсказуемой работе операционной системы, которая наверняка закончится полным отказом системы.

2.3.5. Архитектура памяти

2.3.5.1. Организация памяти в защищенном режиме работы процессора

Ранее мы кратко рассмотрели работу процессоров серии i386 и выше в защищенном режиме, использующем организацию памяти, при которой используются два механизма преобразования памяти:

- сегментация;
- разбиение на страницы.

ОС NT в различной мере использует оба этих механизма.

Как уже говорилось, в защищенном режиме может быть определено до 2^{13} (8192) **сегментов**. Каждый сегмент может иметь размер до 4 Гб (2^{32} байт). Таким образом, максимальный размер виртуального адресного пространства составляет 64 Тб.

Каждый сегмент описывается 8-байтной структурой данных – **дескриптором** сегмента. Дескрипторы находятся в специальной таблице дескрипторов (GDT, см. рис. 5). Для указания конкретного сегмента используется 16-битный **селектор**. Он является индексом внутри таблицы дескрипторов. Младшие 2 бита селектора определяют номер привилегированного режима (DPL – уровень привилегий дескриптора), который может воспользоваться данным селектором для доступа к дескриптору, третий бит определяет локальную/глобальную дескрипторную таблицу. (отсюда максимальное число селекторов 2^{13}).

ОС NT, хотя и использует селекторы, но использует их в минимальной степени. NT реализует плоскую 32-разрядную модель памяти с размером линейного адресного пространства 4 Гб (2^{32} байт). Это сделано следующим образом:

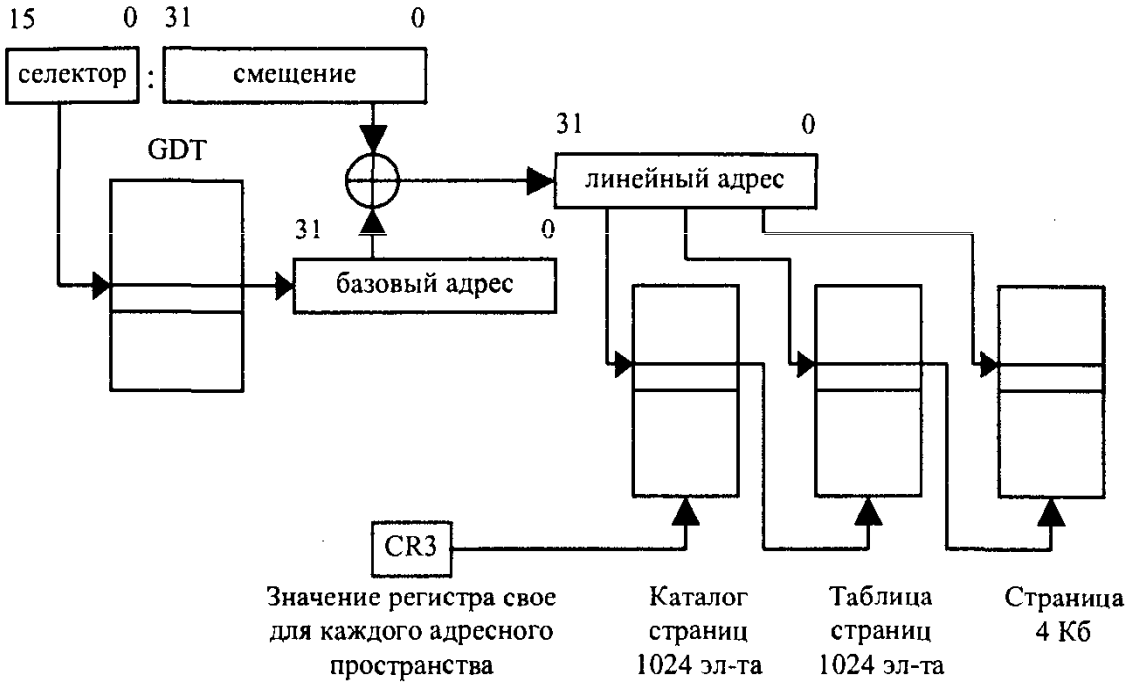


Рис. 5

В NT определено 11 селекторов, из которых нас будут интересовать всего 4:

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

Эти четыре селектора позволяют адресовать все 4Гб линейного адресного пространства, причем для всех селекторов при фиксированном контексте памяти производится трансляция в одни и те же физические адреса. Разница только в режиме доступа.

Первые два селектора имеют DPL=0 и используются драйверами и системными компонентами для доступа к системному коду, данным и стеку. Вторые два селектора используются кодом пользовательского режима для доступа к коду, данным и стеку пользовательского режима. Эти селекторы являются константами для ОС NT.

Сегментное преобразование пары селектор:смещение дает 32-битный линейный адрес (лежащий в диапазоне 4 Гб линейного адресного пространства). При этом линейный адрес совпадает со значением смещения виртуального адреса. Фактически, при такой организации памяти виртуальный и линейный адреса совпадают.

Наличие поля тип, определяющего возможность чтения/записи/исполнения кода в соответствующем сегменте может навести на мысль, что именно на этом уровне про-

изводится защита памяти от нецелевого использования. Например, при работе прикладной программы в пользовательском режиме ее код находится в сегменте с селектором 1b. Для этого сегмента разрешены операции чтения и исполнения. Используя селектор 1b, программа не сможет модифицировать свой собственный код. Однако, как уже было сказано, для всех сегментов производится трансляция в одни и те же физические адреса. Поэтому при обращении к данным или стеку (селектор 23) прикладная программа обнаружит свой код по тому же смещению, что и для селектора 1b, причем режим доступа к сегменту позволяет производить чтение/запись. (При этом важно помнить: одно и то же смещение в разных адресных пространствах указывает на разную физическую память.) Таким образом, способ использования сегментации в ОС NT не обеспечивает защиту кода от нецелевого использования.

Далее задействуется механизм страничной организации памяти и переключения контекста памяти.

Каждый контекст памяти (адресное пространство процесса) представляется собственной таблицей трансляции линейного адреса (совпадающего с виртуальным) в физический.

Каждый элемент таблицы страниц содержит бит, указывающий на возможность доступа к странице из пользовательского режима. При этом все страницы доступны из режима ядра.

Кроме того, каждый элемент таблицы страниц содержит бит, указывающий на возможность записи в соответствующую страницу памяти.

Эти два бита используются для управления доступом к страницам памяти и формируют следующий набор правил:

1) страница всегда может быть прочитана из режима ядра;
2) на страницу может быть произведена запись из режима ядра, только если установлен бит разрешения записи;
3) страница может быть прочитана из пользовательского режима, только если установлен бит доступа к странице из пользовательского режима;
4) на страницу может быть произведена запись из пользовательского режима, если установлены оба бита (разрешение записи и доступ из пользовательского режима);
5) если страница может быть прочитана, она может быть исполнена.

Страницы памяти с исполняемым кодом не будут иметь разрешения на запись, если не предпринять никаких дополнительных действий. Поэтому при попытке использования селектора данных для модификации кода будет сгенерирована исключительная ситуация.

Для упрощения организации памяти, для всех контекстов памяти NT осуществляет одинаковую трансляцию для некоторого диапазона виртуальных адресов. Таким образом, при переключении контекста памяти, то есть переходе на новую таблицу трансляции виртуального адреса в физический, некоторая часть элементов этой таблицы останется неизменной.

Это нужно для того, чтобы ядро операционной системы (компоненты ОС и драйвера) всегда располагалось по фиксированным виртуальным адресам вне зависимости

от текущего контекста памяти. Таким неизменяемым диапазоном адресов являются верхние 2 Гб памяти.

Для защиты кода ОС соответствующие элементы таблицы трансляции виртуального адреса в физический 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).

Во избежание путаницы между терминами **адресное пространство процесса** и **пользовательское/системное адресное пространство**, где это возможно, вместо термина **адресное пространство процесса** мы будем пользоваться термином **контекст памяти.**

При этом возможная путаница вполне допустима, так как с точки зрения прикладного программиста пользовательское адресное пространство – и есть текущий контекст памяти.

2.3.5.2. Организация системного адресного пространства

Как уже отмечалось, системное адресное пространство сильно отличается от пользовательского:

- Системное адресное пространство одинаково вне зависимости от текущего контекста памяти, то есть от содержимого пользовательского адресного пространства.
- В системном адресном пространстве имеются диапазоны памяти как выгружаемые на диск, так и не выгружаемые.

На рис. 6 показана приблизительная организация системного адресного пространства для платформы x86.

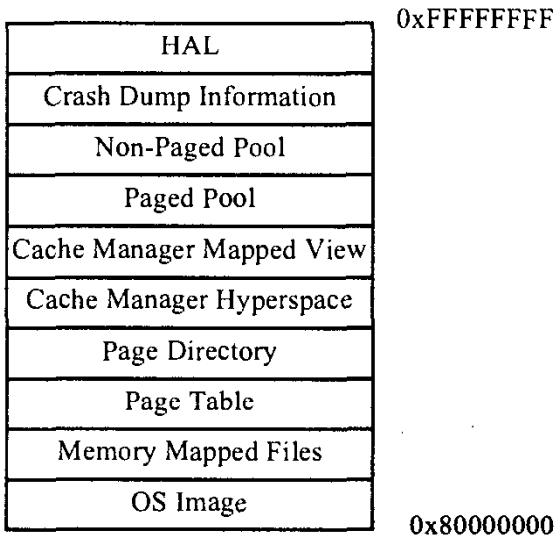


Рис. 6

2.3.5.3. Типы адресов в NT

Как мы уже отмечали, линейный и виртуальный адреса в NT совпадают. Здесь и далее мы будем пользоваться термином **виртуальный адрес**.

Виртуальный адрес транслируется в **физический адрес**. Этот адрес соответствует физической памяти.

Кроме этих двух типов адресов существует еще один – **логический адрес**, реализуемый на уровне HAL.

HAL поддерживает гибкую модель для адресации аппаратных устройств. В соответствии с этой моделью, устройства подключаются к шинам, каждая из которых имеет свое собственное адресное пространство. Реально эти адреса могут быть как в пространстве портов ввода/вывода, так и в пространстве памяти.

Прежде чем может быть произведено обращение к некоторому адресу устройства (посредством функции HAL), адрес должен быть переведен из относительного адреса для шины в некоторый транслированный модулем HAL адрес. Этот транслированный адрес и есть **логический адрес**. Он имеет смысл только для HAL и не имеет ничего общего с конкретным адресом для работы с оборудованием.

Для получения логического адреса из шинного адреса служит функция HalTranslateBusAddress(). Полученный адрес будет находиться либо в пространстве портов ввода/вывода, либо в обычном пространстве памяти. В последнем случае для использования в драйвере полученный логический адрес должен быть преобразован к адресу в невыгружаемой области системного адресного пространства. Это делается посредством вызова функции MmMapIoSpace().

2.3.5.4. Совместное использование памяти

Схему организации памяти на рис. 5 можно представить более упрощенно, а именно: каталог страниц и соответствующие ему таблицы страниц рассматривать как единую таблицу страниц для трансляции виртуального адреса в физический. Каждый контекст памяти при таком представлении определяется своей таблицей страниц.

Из рис. 7 видно, что разные таблицы страниц могут иметь ссылку на одну и ту же физическую страницу памяти. Эта возможность позволяет приложениям совместно использовать одну и ту же физическую память (обращаясь при этом к различным виртуальным адресам).

Кроме того, как уже говорилось выше, на уровне таблиц страниц производится управление доступом к страницам памяти. При этом к одной и той же физической странице памяти может быть предоставлен различный уровень доступа даже внутри одного адресного пространства (никто не запрещает нескольким записям одной таблицы страниц указывать на одну физическую страницу).

Для совместно используемой памяти может быть задействован механизм Copy-On-Write. Запись в таблице страниц для такой памяти указывает запрет на модификацию и задействование Copy-On-Write. Пока два процесса совместно используют такую память для чтения, ничего не происходит. При попытке записи одним из процес-

сов в такую память генерируется исключение (защита от записи). Диспетчер памяти анализирует исключение, обнаруживает, что для страницы установлен механизм Copy-On-Write, создает новую страницу в физической памяти, копирует в нее первоначальную страницу и модифицирует элемент таблицы страниц так, чтобы он указывал на новую страницу, а затем производит первоначальную запись в память. Таким образом, каждый процесс получит свою копию первоначальной страницы каждый со своими изменениями.

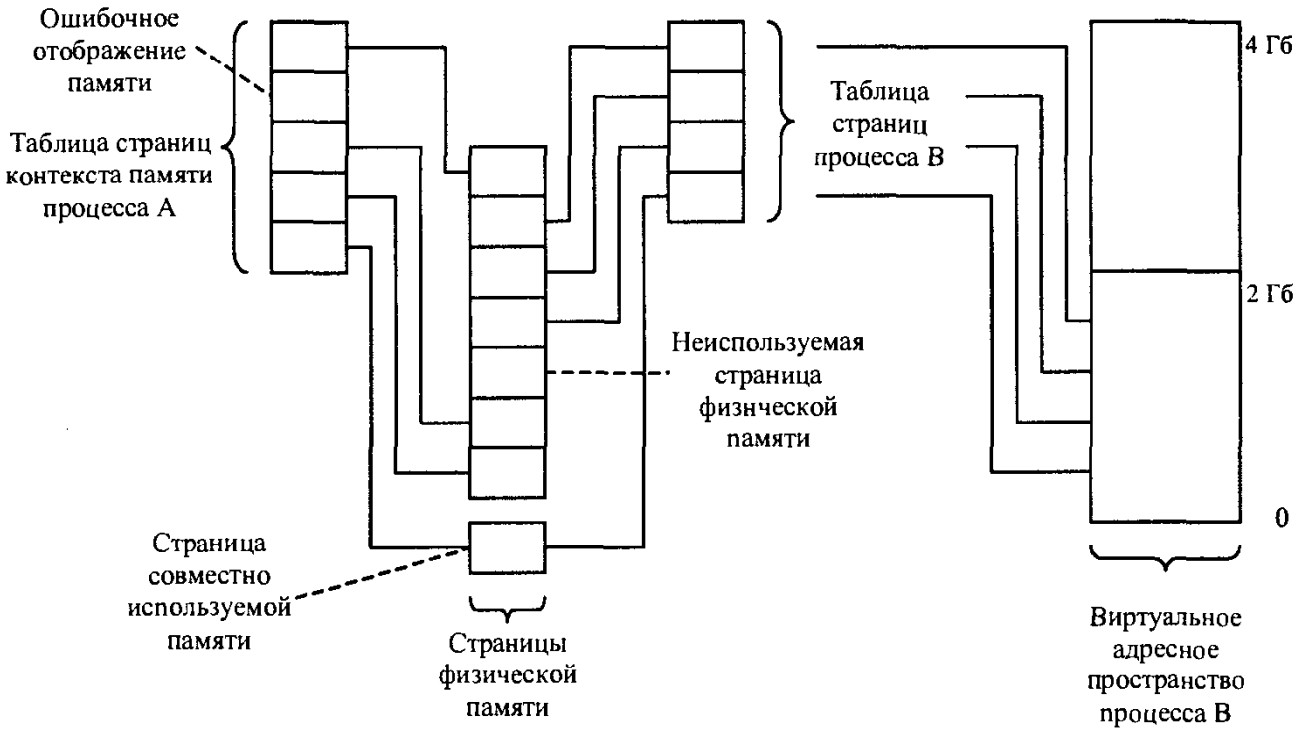


Рис. 7

Все исполняемые модули в ОС NT находятся в совместно используемой памяти с задействованием Copy-On-Write. Это означает, например, что при отладке кода DLL, используемой в некотором процессе, при установке точки прерывания (что осуществляется записью в код), она будет присутствовать только в этом адресном пространстве.

Для постановки точки прерывания на код в DLL так, чтобы она срабатывала вне зависимости от адресного пространства, необходимо предпринять специальные действия:

1) Для данного виртуального адреса нужно получить физический адрес.
2) Для физического адреса создать новую запись в таблице страниц, устанавливающих для страницы доступ на запись. Создание записи в таблице страниц означает получение нового виртуального адреса.

3) При записи по полученному виртуальному адресу будет модифицирована страница физической памяти, соответственно это изменение увидится во всех контекстах памяти, на которые отображена эта страница.

Естественно, первые два шага можно сделать только в коде режима ядра, то есть из драйвера.

2.3.5.5. Объект Секция

Диспетчер Памяти в NT экспортирует единственную структуру для контроля данных – Объект-Секцию. Подобно другим объектам, Объект-Секция может быть именованным, то есть имя будет видимо в пространстве имен Диспетчера Объектов.

Секция может быть использована драйвером для отображения участка памяти системного адресного пространства (в том числе невытесняемого) в пользовательское адресное пространство. В этом случае прикладная программа и драйвер получат в свое распоряжение совместно используемую область памяти, что может быть полезно, например, при необходимости передачи большого объема данных от драйвера к приложению.

Секция используется для описания всего, что может быть отображено в память. Например, для каждого отображаемого в память файла создается объект-Секция. При этом, сколько бы не было открытий такого файла, объект-Секция всегда одна. Поскольку все исполняемые файлы загружаются посредством механизма отображения памяти, единственность объекта-секции всегда гарантирует наличие только одной копии такого файла в памяти.

При создании секции указывается режим доступа (чтение/запись/исполнение). Этот режим доступа будет влиять на записи в таблице страниц, относящиеся к секции.

Функции работы с секциями:

- ZwOpenSection();
- ZwMapViewOfSection();
- ZwUnmapViewOfSection().

2.3.5.6. Таблица описания памяти (Memory Descriptor List, MDL)

Диспетчер памяти использует структуру MDL для описания набора страниц физической памяти, составляющих буфер виртуальной памяти в контексте памяти некоторого процесса. Интерпретация MDL не зависит от контекста памяти, поскольку MDL оперирует со страницами физической памяти. Получив для данного буфера описание в виде MDL, драйвер в дальнейшем может использовать буфер в контексте памяти любого процесса. Для того, чтобы обращаться к такой памяти, необходимо получить для MDL адрес памяти в системном адресном пространстве. Сделать это можно с помощью функции MmGetSystemAddressForMdl().

Кроме того, буфер, описанный с помощью MDL, может быть использован для операций DMA. Для этого физический адрес внутри MDL должен быть транслирован в

логический адрес (имеющий смысл только для данного устройства DMA) с помощью функции IoMapTransfer().

Интересно отметить следующий момент. MDL предназначен для описания буфера данных, непрерывного в виртуальной памяти. Однако страницы физической памяти, список которых собственно и содержит MDL, могут располагаться в памяти произвольным образом. Это дает возможность «собирать» непрерывный в виртуальной памяти буфер из различных фрагментов физической памяти без копирования памяти.

В основном, мы будем встречаться с MDL при передаче данных в драйвер посредством пакетов IRP (которые будут описаны в последующих разделах).

2.3.5.7. Функции работы с памятью

Из всего вышеизложенного нужно выделить основные моменты:

1) Все адресное пространство процесса (контекст памяти) делится на две области – системное адресное пространство (верхние 2 Гб) и пользовательское адресное пространство (нижние 2 Гб):
   - Системное адресное пространство всегда одинаково, вне зависимости от текущего контекста памяти.
   - Пользовательское адресное пространство разное для каждого контекста памяти.
2) Код пользовательского режима и режима ядра пользуется для доступа к памяти разными селекторами:
   - Для одного контекста памяти селекторы адресуют одну и ту же физическую память, но с разными режимами доступа.
   - Селекторы не имеют ничего общего с контекстом памяти.
   - По селекторам для ядра можно получить доступ (по крайней мере, для чтения) ко всему адресному пространству в данном контексте памяти, а по селекторам для ядра – только для пользовательской области памяти.
3) Код в пользовательском адресном пространстве всегда вытесняемый, если не предприняты меры по созданию новой записи в каталоге страниц, указывающих на невытесняемую память.
4) Код, работающий на уровне IRQL большем или равном DISPATCH_LEVEL, может использовать только невыгружаемую память (соответственно, сам код должен находиться в невыгружаемой памяти).
5) В ОС существуют функции для выполнения любой работы с памятью:
   - Выделение/освобождение памяти в выгружаемой/невыгружаемой, кешируемой/некешируемой памяти.
   - Преобразование адресов памяти (виртуальный в физический, физический в виртуальный).
   - Проверка доступности памяти.

Рассмотрим эти функции более подробно.

2.3.5.7.1. Выделение памяти

```
PVOID ExAllocatePool(
  IN POOL_TYPE PoolType,
  IN ULONG NumberOfBytes);
PVOID ExAllocatePoolWithTag(
  IN POOL_TYPE PoolType,
  IN ULONG NumberOfBytes, IN ULONG Tag);
```

Где: POOL_TYPE принимает следующие значения:

Тип памяти (PoolType)	Описание
NonPagedPool	Обычное выделение памяти из Nonpaged Pool.
NonPagedPoolCacheAligned	Выделение памяти из Nonpaged Pool будет выровнено по линии кеша.
NonPagedPoolMustSucceed	Используется в специальных случаях драйверами, необходимыми для загрузки системы.
NonPagedPoolCacheAlignedMustSucceed	
PagedPool	Обычное выделение памяти из Paged Pool.
PagedPoolCacheAligned	Выделение памяти из Paged Pool будет выровнено по линии кеша.

```
VOID ExFreePool(IN PVOID address);
PVOID MmAllocateNonCachedMemory(IN ULONG NumberOfBytes);
VOID MmFreeNonCachedMemory(   IN PVOID BaseAddress,
       IN ULONG NumberOfBytes);
PVOID MmAllocateContiguousMemory(IN ULONG NumberOfBytes,
       IN PHYSICAL_ADDRESS HighestAcceptableAddress);
VOID MmFreeContiguousMemory(IN PVOID BaseAddress);
```

2.3.5.7.2. Список заранее выделенных блоков памяти (Lookaside List)

Во многих случаях, выделение и освобождение временного буфера памяти должно происходить очень часто, для уменьшения накладных расходов служит Lookaside List – список заранее выделенных блоков памяти фиксированного размера.

Первоначально, память выделяется только под небольшой заголовок со служебной информацией. При каждом запросе на выделение памяти проверяется, есть ли в списке свободные блоки. Если их нет – они выделяются из выгружаемой или невыгружаемой памяти. Если есть, то помечаются, как используемые и выдаются для использования.

```
VOID ExInitializeNPagedLookasideList(
     IN PNPAGED_LOOKASIDE_LIST Lookaside,
     IN PALLOCATE_FUNCTION Allocate OPTIONAL,
     IN PFREE_FUNCTION Free OPTIONAL,
     IN ULONG Flags,
```

```
    IN ULONG Size,
    IN ULONG Tag,
    IN USHORT Depth );
  VOID ExInitializePagedLookasideList(
    IN PPAGED_LOOKASIDE_LIST Lookaside,
    IN PALLOCATE_FUNCTION Allocate OPTIONAL,
    IN PFREE_FUNCTION Free OPTIONAL,
    IN ULONG Flags,
    IN ULONG Size,
    IN ULONG Tag,
    IN USHORT Depth );
  PVOID ExAllocateFromNPagedLookasideList(IN PNPAGED_LOOKASIDE_LIST Lookaside);
  PVOID ExAllocateFromPagedLookasideList(IN PPAGED_LOOKASIDE_LIST Lookaside);
  VOID ExFreeToNPagedLookasideList(
    IN PNPAGED_LOOKASIDE_LIST Lookaside,
    IN PVOID Entry);
  VOID ExFreeToPagedLookasideList(
    IN PPAGED_LOOKASIDE_LIST Lookaside,
    IN PVOID Entry);
  VOID ExDeleteNPagedLookasideList(IN PNPAGED_LOOKASIDE_LIST
Lookaside);
  VOID ExDeletePagedLookasideList(IN PPAGED_LOOKASIDE_LIST
Lookaside);
```

2.3.5.7.3. Пространства ввода/вывода и отображение памяти устройств

Смотри раздел «Типы адресов в NT».

```
BOOLEAN HalTranslateBusAddress(IN INTERFACE_TYPE InterfaceType,
    IN ULONG BusNumber,
    IN PHYSICAL_ADDRESS BusAddress,
    IN OUT PULONG AddressSpace,
    OUT PPHYSICAL_ADDRESS TranslatedAddress);
PVOID MmMapIoSpace(IN PHYSICAL_ADDRESS PhysicalAddress,
    IN ULONG NumberOfBytes,
    IN BOOLEAN CacheEnable);
VOID MmUnmapIoSpace(IN PVOID BaseAddress,
    IN ULONG NumberOfBytes);
```

2.3.5.7.4. Управление памятью и MDL

```
  BOOLEAN MmIsAddressValid(IN PVOID VirtualAddress );
  VOID MmProbeAndLockPages(IN OUT PMDL Mdl,
```

```
  IN KPROCESSOR_MODE AccessMode,
  IN LOCK_OPERATION Operation);
PVOID MmGetSystemAddressForMdl(IN PMDL Mdl)
MmGetPhysicalAddress(IN PVOID BaseAddress);
VOID MmUnlockPages(IN PMDL mdl);
PMDL IoAllocateMdl(IN PVOID VirtualAddress,
  IN ULONG Length,
  IN BOOLEAN SecondaryBuffer,
  IN BOOLEAN ChargeQuota,
  IN OUT PIRP Irp);
VOID MmPrepareMdlForReuse(IN PMDL Mdl);
VOID IoFreeMdl(IN PMDL Mdl);
```

2.3.6. Унифицированная модель драйвера

В исполнительной системе драйвер устройства и файловая система строятся и выглядят для остальной части ОС одинаково. Более того, именованные каналы и сетевые редиректоры рассматриваются, как файловые системы, и реализованы в виде соответствующих драйверов. Каждый драйвер – это автономный компонент, который можно динамически загружать и выгружать из системы в зависимости от потребностей пользователя.

Унифицированный модульный интерфейс, предоставляемый драйверами, позволяет диспетчеру ввода/вывода не видеть их структуру или внутренние детали. Драйверы могут вызывать друг друга через диспетчер ввода/вывода, что обеспечивает независимую обработку запроса ввода/вывода на нескольких уровнях.

Драйверы являются модульными и могут располагаться слоями один над другим, что позволяет, например, драйверам разных файловых систем использовать для доступа к файлам один и тот же драйвер диска. Послойная модель драйверов позволяет также вставлять в иерархию новые драйверы.

Драйвер – это особый тип динамически подключаемой библиотеки. Фактически, это DLL, удовлетворяющая ряду дополнительных требований и имеющая расширение «.sys».

Как и любая DLL, драйвер имеет свою точку входа – функцию, вызываемую при загрузке исполняемого файла в память. Адрес этой точки входа содержится в служебной информации в самом модуле. При создании модуля в процессе компиляции настройки среды разработки предполагают, что имя соответствующей функции будет **DriverEntry**, хотя оно может быть заменено на любое другое. Момент загрузки драйвера определяется соответствующими данному драйверу настройками в реестре (ключ Start). Этими настройками управляет Service Control Manager (SCM), хотя они могут быть изменены и вручную.

Прежде чем перейти к описанию структуры драйвера желательно ознакомиться с такими важными понятиями как объект-файл, объект-драйвер и объект-устройство.

2.3.6.1. Объект-файл (файловый объект)

Код пользовательского режима может получить доступ к файлам на диске или всевозможным устройствам (физическим, логическим и виртуальным) только через описатели файловых объектов, обеспечиваемых менеджером ввода/вывода по запросу от пользовательской программы на открытие/создание файла или устройства. После открытия/создания виртуального файла, обозначающего любой источник или приемник ввода/вывода (работа с которым идет так, как если бы он был обычным файлом на диске), программы могут осуществлять ввод/вывод в этот виртуальный файл, манипулируя им посредством описателя.

Итак, **файловый объект** – это объект, видимый из режима пользователя, который представляет всевозможные открытые источники или приемники ввода/вывода: файл на диске или устройство (физическое, логическое, виртуальное). Физическим устройством может быть, например, последовательный порт, физический диск; логическим – логический диск; виртуальным – виртуальный сетевой адаптер, именованный канал, почтовый ящик.

Всякий раз, когда некоторый поток открывает файл, создается новый файловый объект с новым набором атрибутов. В любой момент времени сразу несколько файловых объектов могут быть ассоциированы с одним разделяемым виртуальным файлом, но каждый такой файловый объект имеет уникальный описатель, корректный только в контексте процесса, поток которого инициировал открытие файла. Возможны ситуации, когда два процесса имеют разные описатели, ссылающиеся на один и тот же файловый объект: 1) когда процесс дублирует описатель файлового объекта для другого процесса; 2) когда дочерний процесс наследует описатель от родительского.

Файловые объекты, как и другие объекты, имеют иерархические имена, охраняются объектной защитой, поддерживают синхронизацию и обрабатываются системными сервисами.

2.3.6.2. Объект-драйвер

Драйверы скрыты от программ пользовательского режима. Устройства (физические, логические и виртуальные), создаваемые и управляемые драйверами, видны программам пользовательского режима как именованные файловые объекты. Как уже отмечалось ранее, код пользовательского режима может получить доступ к устройству только через описатель, возвращаемый менеджером ввода/вывода во время открытия/создания файлового объекта, представляющего устройство. (В структуре FILE_OBJECT есть указатель на DEVICE_OBJECT, ассоциированный с данным файловым объектом.)

Объект-драйвер представляет в системе некоторый драйвер Windows NT и хранит для диспетчера ввода/вывода адреса стандартных процедур (точки входа), которые драйвер может или должен иметь в зависимости от того, является ли он драйвером верхнего или нижнего уровней. Объект-драйвер описывает также, где драйвер загружен в физическую память и размер драйвера. Объект-драйвер описывается частично документированной структурой данных DRIVER_OBJECT.

Диспетчер ввода/вывода определяет тип объект-драйвер и использует экземпляры этого типа для регистрации и отслеживания информации о загруженных образах драйверов. Этот объект создается менеджером ввода/вывода при загрузке драйвера в систему, после чего диспетчер вызывает процедуру инициализации драйвера DriverEntry и передает ей указатель на объект-драйвер. Эта процедура устанавливает стандартные и опциональные точки входа процедур драйвера, содержащиеся в структуре DRIVER_OBJECT, чтобы в дальнейшем диспетчер ввода/вывода мог направлять пакеты запроса ввода/вывода IRP соответствующей процедуре. В DriverEntry устанавливаются также точки входа для других процедур, например для StartIo, Unload. Во время инициализации также может происходить считывание информации в поля объекта-драйвера из базы данных реестра диспетчера конфигурации.

Пакеты IRP, направляемые стандартным процедурам, содержат, кроме всего прочего, указатель на объект – устройство, который является устройством назначения для конкретного запроса ввода/вывода.

Объект-драйвер является скрытым для кода пользовательского уровня, то есть только определенные компоненты уровня ядра (в том числе и диспетчер ввода/вывода) знают внутреннюю структуру этого типа объекта и могут получать доступ ко всем данным, содержащимся в объекте, напрямую.

Microsoft не гарантирует неизменность недокументированных полей любых своих структур в последующих версиях ОС. Однако фактически, некоторые недокументированные в DDK элементы различных системных структур документированы в другом пакете для разработчика – IFS Kit.

2.3.6.3. Объект-устройство

Диспетчер ввода/вывода определяет тип объекта – **объект-устройство**, используемый для представления физического, логического или виртуального устройства, чей драйвер был загружен в систему. Формат объекта-устройство определяется частично документированной структурой данных DEVICE_OBJECT. Хотя объект-устройство может быть создан в любое время посредством вызова функции IoCreateDevice(), обычно он создается внутри DriverEntry.

Объект-устройство описывает характеристики устройства, такие как требование по выравниванию буферов и местоположение очереди устройства, для хранения поступающих пакетов запросов ввода/вывода.

Процедура инициализации драйвера DriverEntry создает по одному объекту-устройству для каждого устройства, управляемого данным драйвером, посредством вызова процедур, предоставляемых менеджером ввода/вывода. Процедура инициализации драйвера должна создавать, по крайней мере, один объект-устройство, некоторые драйверы должны создавать более одного объекта-устройства в зависимости от уровня, на котором они располагаются в цепочке драйверов.

В каждом объекте-устройстве есть указатель на следующий объект-устройство (либо NULL) и указатель на объект-драйвер, управляющий данным устройством. Благодаря этому диспетчер ввода/вывода может определить, процедуру какого драйвера он должен

вызвать при получении запроса к устройству. Диспетчер ввода/вывода поддерживает также указатель на объект-устройство, созданный драйвером, в объекте-драйвере.

Большинство драйверов NT используют структуру данных DeviceExtension, указатель на которую хранится в объекте-устройстве, а ее размер и содержимое определяется при создании объекта-устройства во время инициализации драйвера. Внутренняя структура DeviceExtension определяется разработчиками драйверов и используется для хранения информации о состоянии устройства, некоторого контекста текущего запроса ввода/вывода, для хранения указателей на описатели различных объектов ядра, то есть для хранения любых данных, которые нужны драйверу. Диспетчер ввода/вывода выделяет память под DeviceExtension из резидентной системной области памяти.

Так как большинство процедур драйверов исполняются в контексте произвольного потока, то есть потока, оказавшегося текущим на момент вызова процедуры драйвера, то область DeviceExtension является одним из первых мест, где драйвер хранит необходимые ему данные.

2.3.6.3.1. Имя устройства и символическая связь

При создании объекта-устройства также может быть указано его имя, которое будет видимо в директории «\Device» пространства имен диспетчера объектов. Объекты- устройства используются в Windows NT как точки входа в пространства имен, не контролируемые менеджером объектов. Если при разборе имени объекта диспетчер объектов встречает объект-устройство, то он вызывает метод разбора, связанный с этим устройством.

Если имя не указано, объект-устройство может быть использован только внутри драйвера и недоступен извне. Если быть более точным, такой объект-устройство можно использовать, если иметь указатель на описывающую его структуру.

Именованный объект-устройство доступен для использования через вызов системного сервиса NtCreateFile().

Для обращения к именованному объекту-устройству из подсистемы Win32 посредством функции CreateFile() должны быть предприняты дополнительные действия. Функция CreateFile() ищет имя устройства в директории Диспетчера Объектов «\??» (NT 4.0 и Win2000), либо «\DosDevices»(NT 3.51). Поэтому в соответствующей директории, а для большей совместимости это должна быть «\DosDevices», должен быть создан объект (символическая связь), указывающий на имя устройства в директории «\Device». Обычно связь создается в самом драйвере, хотя это можно сделать и из прикладной программы пользовательского режима с помощью Win32-функции DefineDosDevice().

2.3.6.4. Взаимосвязь основных объектов

Что происходит при успешном открытии объекта-устройства (неважно, с помощью какой функции: CreateFile() или NtCreateFile())? Для описания состояния работы с каждым объектом, открытым с помощью этих функций, создается новый экземпляр **объекта-файла**. Именно отсюда название функции – CreateFile.

С точки зрения прикладного уровня ОС функции открытия файла могут применяться для открытия файлов, устройств, именованных каналов, почтовых слотов и т.п. Однако, с точки зрения ядра ОС, объекта-файла (в смысле файл на жестком диске) не существует, как не существует объектов именованный канал или почтовый слот. В действительности объект-файл будет связан с некоторым объектом-устройством, для которого имеет смысл, допустим, понятие «файл» (в смысле файл файловой системы).

Соответственно, каждая операция ввода/вывода на прикладном уровне будет производиться с некоторым объектом-файлом. Причем конкретный объект будет указан не напрямую, а через так называемый **описатель** (HANDLE), возвращаемый как результат работы функции CreateFile().

Важно отметить, что каждый процесс имеет свою собственную таблицу описателей, обеспечивая уникальность описателей только в рамках своей таблицы. Это означает, например, что для двух процессов – A и B – описателю со значением 1 будут в общем случае соответствовать два различных объекта-файла (см. рис. 8). Кроме того, ОС поддерживает **механизм наследования описателей**. В этом случае два разных процесса через два разных описателя будут разделять один и тот же объект-файл.

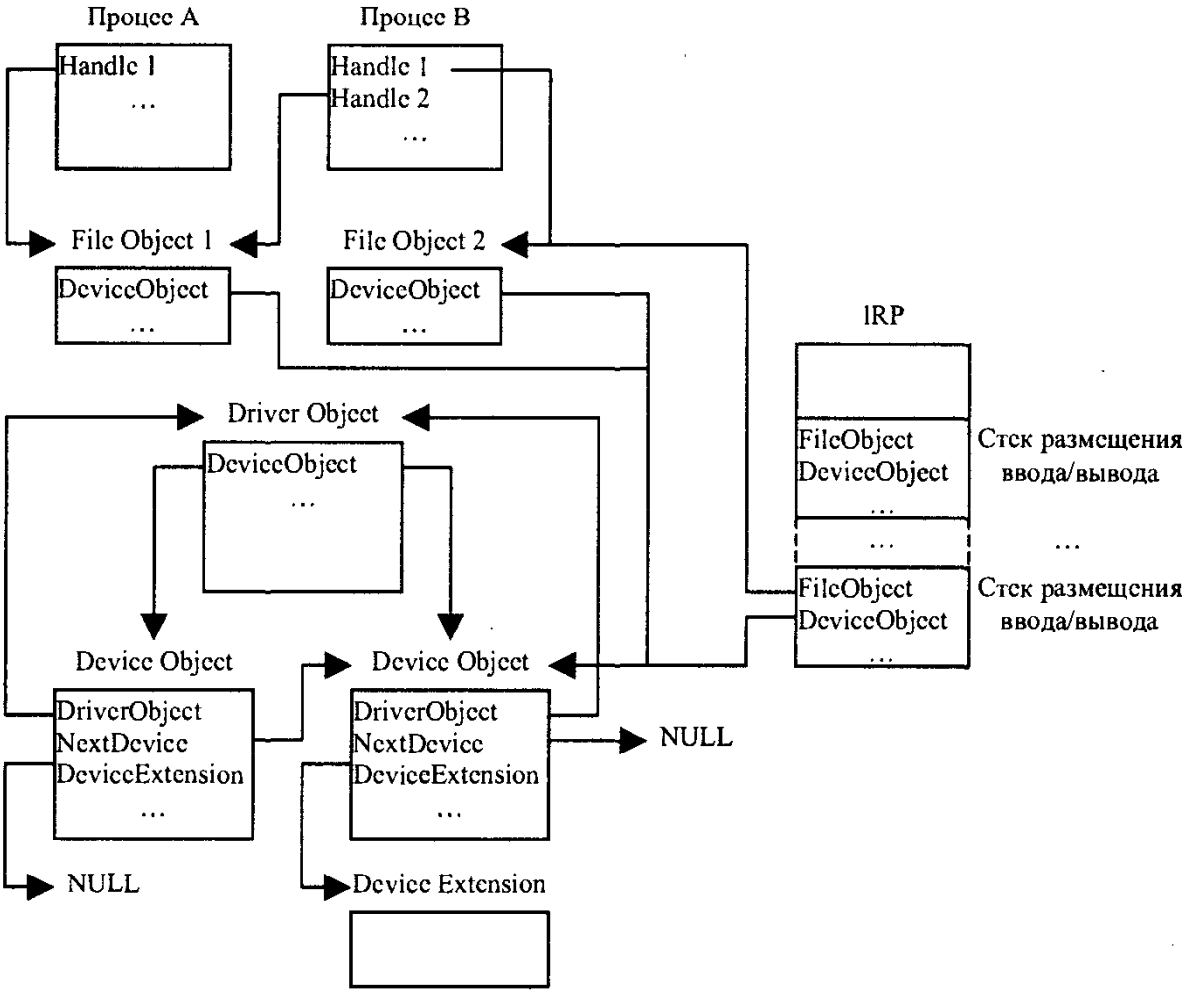


Рис. 8

Практически уникальность описателя только в контексте данного процесса означает невозможность использования описателя драйвером при работе в случайном контексте. Описатель обязательно должен быть переведен в указатель на объект посредством вызова функции диспетчера объектов ObReferenceObjectByHandle().

2.4. Установка, удаление, запуск и остановка драйвера

Сейчас мы коротко рассмотрим операции установки и управления драйверами. Драйверы в NT поддерживают динамическую загрузку и выгрузку. Информация о драйвере, такая, как его имя, тип, местонахождение, способ загрузки и др. находится в реестре в ключе HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\Service_name. Подробно обо всех подключах, которые могут там находиться, вы можете узнать в статье «Using The NT Registry for Driver Install» в директории NTInsider, либо в DDK Help\Programmers Guide\Driver Installation\Configuration Registry.

Управлением сервисами и драйверами в системе занимается Service Control Manager (SCM). Он управляет базой данных установленных сервисов и драйверов, обеспечивает единый способ контроля над ними, а также предоставляет API.

Подробную информацию о функционировании SCM и предоставляемом им API можно получить в MSDN Library в разделе Platform SDK\Base Services\DLLs, Processes and Threads\Services.

Примерная последовательность действий при установке/удалении запуске/остановке драйвера следующая:

1) открытие SCM – OpenSCManager();
2) получение описателя для вновь созданного или уже существующего драйвера – CreateService() или OpenService();
3) запуск\остановка\удаление драйвера – StartService(), StopService(), DeleteService().

Установленный в системе драйвер также может быть запущен/остановлен с помощью команды net start\net stop.

Рассмотрим другие способы установки драйверов:

- **Text Setup**. Этот механизм используют драйверы, устанавливаемые при установке ОС. Этот механизм требует создания скрипт-файла txtsetup.oem. Его формат описан в DDK, имеются примеры в \ddk\src\setup. В этом файле программе установки NT указывается, какие файлы и куда копировать и какие ключи реестра создавать.
- **GUI Setup**. Драйверы для стандартных устройств, устанавливаемые по окончании установки ОС, используют inf-файлы, формат которых и примеры также приведены в DDK.
- **Custom Setup**. Прикладная программа, использующая функции SCM.

2.4.1.1. Структура драйвера

2.4.1.1.1. Точки входа драйвера

При написании любого драйвера необходимо помнить четыре основных момента:

1) возможные точки входа драйвера;
2) контекст, в котором могут быть вызваны точки входа драйвера;
3) последовательность обработки типичных запросов;
4) Уровень IRQL, при котором вызывается точка входа, и, следовательно, ограничения на использование некоторых функций ОС каждая (!!!) функция ОС может быть вызвана только при определенных уровнях IRQL (см. описание любой функции в DDK, там всегда указаны эти уровни).

Архитектура драйвера Windows NT использует модель точек входа, в которой Диспетчер Ввода/вывода вызывает специфическую подпрограмму в драйвере, когда требуется, чтобы драйвер выполнил специфическое действие. В каждую точку входа передается определенный набор параметров для драйвера, чтобы дать возможность ему выполнить требуемую функцию.

Базовая структура драйвера состоит из набора точек входа, наличие которых обязательно, плюс некоторое количество точек входа, наличие которых зависит от назначения драйвера.

Далее перечисляются точки входа либо классы точек входа драйвера:

1) DriverEntry. Диспетчер Ввода/вывода вызывает эту функцию драйвера при первоначальной загрузке драйвера. Внутри этой функции драйверы выполняют инициализацию как для себя, так и для любых устройств, которыми они управляют. Эта точка входа требуется для всех NT драйверов.

2) Диспетчерские (Dispatch) точки входа. Точки входа Dispatch драйвера вызываются Диспетчером Ввода/вывода, чтобы запросить драйвер инициировать некоторую операцию ввода/вывода.

3) Unload. Диспетчер Ввода/вывода вызывает эту точку входа, чтобы запросить драйвер подготовиться к немедленному удалению из системы. Только драйверы, которые поддерживают выгрузку, должны реализовывать эту точку входа. В случае вызова этой функции, драйвер будет выгружен из системы при выходе из этой функции вне зависимости от результата ее работы.

4) Fast I/O. Вместо одной точки входа, на самом деле это набор точек входа. Диспетчер Ввода/вывода или Диспетчер Кэша вызывают некоторую функцию быстрого ввода/вывода (Fast I/O), для инициирования некоторого действия "Fast I/O". Эти подпрограммы поддерживаются почти исключительно драйверами файловой системы.

5) Управление очередями запросов IRP (сериализация – процесс выполнения различных транзакций в нужной последовательности). Два типа очередей: Системная очередь (StartIo) и очереди, управляемые драйвером. Диспетчер Ввода/вывода использует точку входа StartIo только в драйверах, которые используют механизм Системной Очереди (System Queuing). Для таких драйверов, Диспетчер Ввода/вывода вызывает эту точку входа, чтобы начать новый запрос ввода/вывода.

6) Reinitialize. Диспетчер Ввода/вывода вызывает эту точку входа, если она была зарегистрирована, чтобы позволить драйверу выполнить вторичную инициализацию.

7) Точка входа процедуры обработки прерывания (ISR – Interrupt Service Routine). Эта точка входа присутствует, только если драйвер поддерживает обработку прерывания. Как только драйвер подключен к прерываниям от своего устройства, его ISR будет вызываться всякий раз, когда одно из его устройств запрашивает аппаратное прерывание.

8) Точки входа вызовов отложенных процедур (DPC – Deferred Procedure Call). Два типа DPC: DpcForIsr и CustomDpc. Драйвер использует эти точки входа, чтобы завершить работу, которая должна быть сделана в результате появления прерывания или другого специального условия. Процедура отложенного вызова выполняет большую часть работы по обслуживанию прерывания от устройства, которое не требует высокого уровня прерывания IRQL, ассоциированного с процессором.

9) SynchCritSection. Диспетчер Ввода/вывода вызывает эту точку входа в ответ на запрос драйвера на захват одной из спин-блокировок его ISR.

10) AdapterControl. Диспетчер Ввода/вывода вызывает эту точку входа, чтобы указать, что общедоступные DMA-ресурсы драйвера доступны для использования в передаче данных. Только некоторые драйверы устройств DMA реализуют эту точку входа.

11) Cancel. Драйвер может определять точку входа Cancel для каждого IRP, который он содержит во внутренней очереди. Если Диспетчер Ввода/вывода хочет отменить конкретный IRP, он вызывает подпрограмму Cancel, связанную с этим IRP.

12) IoCompletion. Эту точку входа для каждого IRP может устанавливать драйвер верхнего уровня при многоуровневой организации. Диспетчер Ввода/вывода вызывает эту подпрограмму после того, как все драйверы нижнего уровня завершили IRP.

13) IoTimer. Для драйверов, которые инициализировали и запустили поддержку IoTimer, Диспетчер Ввода/вывода вызывает эту точку входа приблизительно каждую секунду.

14) CustomTimerDpc. Эта точка входа вызывается, когда истекает время для запрошенного драйвером таймера.

2.4.1.1.2. Контекст исполнения и уровень IRQL

Говоря о точках входа в драйвер, необходимо отметить контекст, при котором эти точки входа могут быть вызваны.

Вначале необходимо определиться с тем, что мы подразумеваем под контекстом исполнения?

Контекст исполнения определяется двумя составляющими:

- исполняемый в настоящее время поток (контекст планирования потока – thread scheduling context);
- контекст памяти процесса, которому принадлежит поток.

Текущий контекст исполнения может принадлежать одному из трех классов:

- контекст процесса «System» (далее – **системный контекст**);
- контекст конкретного потока и процесса;
- контекст случайного потока и процесса (далее – **случайный контекст**).

Различные точки входа в драйвер могут вызываться в контексте, принадлежащем одному из этих классов.

DriverEntry всегда вызывается в системном контексте.

Диспетчерские функции для драйверов верхнего уровня (то есть получающих запрос от прикладных программ) вызываются в контексте инициирующего запрос потока.

Диспетчерские функции драйверов остальных уровней, получающие запрос от драйвера верхнего уровня, вызываются в случайном контексте.

Все точки входа, связанные с сериализацией запросов ввода/вывода или с обработкой прерываний и DPC, вызываются в случайном контексте.

DriverEntry всегда вызывается на IRQL, равным PASSIVE_LEVEL.

Диспетчерские точки входа вызываются на IRQL, равным PASSIVE_LEVEL или APC_LEVEL.

Вызов отложенных процедур – на DISPATCH_LEVEL

Функции обработки прерываний – на одном из DIRQL.

2.4.1.2. Ограничения, налагаемые на драйвер

1) Драйвер режима ядра не может использовать API пользовательского уровня или стандартные библиотеки времени исполнения языка C. Можно использовать только функции ядра.

2) Драйвер не может осуществлять операции с числами с плавающей точкой. Попытка сделать это может вызвать аварийную остановку системы. Причина – в основе реализации архитектуры MMX. Не вдаваясь в подробности можно сказать, что в этой архитектуре для обозначения регистров MMX использованы те же обозначения, что и для использования регистров FPU. Переключение между использованием регистров MMX/FPU, производимое на пользовательском уровне, невидимо для драйвера.

3) Драйвер не может манипулировать физической памятью напрямую. Однако он может получить виртуальный адрес для любого физического адреса и манипулировать им.

4) Код драйвер не должен долгое время работать на повышенных уровнях IRQL.

Другие ограничения можно посмотреть в [Developing Windows NT Device Driver, chapter 5, Driver Limitation].

Последующие разделы будут посвящены описанию различных точек входа драйвера.

2.4.1.3. Точка входа DriverEntry

Диспетчер ввода/вывода вызывает точку входа DriverEntry при загрузке драйвера. В NT может существовать только один экземпляр драйвера, вне зависимости от числа физических устройств, контролируемых им. Таким образом, DriverEntry вызывается только один раз, на уровне IRQL равном PASSIVE_LEVEL в системном контексте.

Прототип DriverEntry:

```
NTSTATUS DriverEntry(IN PDRIVER_OBJECT DriverObject,
                     IN PUNICODE_STRING RegistryPath);
```

Где: *DriverObject* – указатель на объект-драйвер, соответствующий загружаемому драйверу; *RegistryPath* – указатель на строку в формате Unicode с именем ключа реестра, соответствующего загружаемому драйверу.

Возвращаемое значение имеет тип NTSTATUS. Если возвращается успешный статус завершения, диспетчер ввода/вывода немедленно позволяет производить обработку запросов к объектам-устройствам, созданным драйвером. Во всех остальных случаях драйвер не загружается в память, и запросы к нему не передаются.

В функции DriverEntry обычно происходит:

- определение всех других точек входа драйвера (их перечень см. в предыдущем разделе);
- определение конфигурации аппаратного устройства;
- создание одного или нескольких объектов-устройств.

Информация об определении всех других точек входа драйвера будет описана в следующих разделах.

2.4.1.3.1. Определение конфигурации аппаратного устройства

Довольно обширную тему, связанную с подготовкой драйвера к использованию аппаратного устройства мы пропустим. Интересующиеся могут обратиться к [Device Driver Development, chapter 13. Driver Entry].

Тем не менее, один часто используемый в драйверах физических устройств момент, а именно – использование реестра – необходимо упомянуть.

В соответствии с принятыми соглашениями драйверы хранят настроечные параметры в ключе реестра \HKLM\CurrentControlSet\Services*DrvName*\Para-meters или ...*DrvName*\Device*X*\Parameters.

Наиболее простой способ запроса содержимого реестра предоставляет функция RtlQueryRegistryValues(). Имя ключа реестра \HKLM\CurrentCont-rolSet\Services\ *DrvName* содержится во втором параметре функции DriverEntry.

2.4.1.3.2. Создание объекта-устройства и символической связи

Как уже говорилось, объект-устройство представляет физическое, логическое или виртуальное устройство, которое должно быть использовано в качестве получателя запросов ввода/вывода. Именованный объект-устройство может быть использован либо из прикладного уровня посредством вызова (Nt)CreateFile(), либо из другого драйвера посредством вызова IoGetDeviceObjectPointer().

Создается объект-устройство с помощью вызова функции IoCreateDevice().

```
NTSTATUS IoCreateDevice(IN PDRIVER_OBJECT DriverObject,
  IN ULONG DeviceExtensionSize,
  IN PUNICODE_STRING DeviceName,
  IN DEVICE_TYPE DeviceType,
  IN ULONG DeviceCharacteristics,
  IN BOOLEAN Exclusive,
  OUT PDEVICE_OBJECT *DeviceObject);
```

Стоит упомянуть, что при необходимости создания нескольких именованных объектов-устройств одного типа (параметр DeviceType) стандартные драйверы NT пользуются определенным соглашением (которое не является обязательным). А именно: к фиксированному имени прибавляется номер устройства, начиная с нуля. Например, COM0, COM1,... Это позволяет открывать устройства, про которые заранее не известно, существуют они или нет. Для получения следующего номера, для которого еще не создано устройство, драйвер может использовать функцию IoGetConfiguration Information(). Эта функция возвращает указатель на структуру, содержащую число устройств текущего драйвера для каждого обнаруженного типа устройства.

Параметр «тип устройства» (DeviceType) может принимать или одно из предопределенных в ntddk.h значений, начинающихся с FILE_DEVICE_, либо значение в диапазоне 32768...65535, зарезервированное для нестандартных устройств.

Необходимо отметить, что при определении кода запроса ввода/вывода к устройству(Device i/o control code, будет рассмотрен в следующем разделе) следует использовать тот же тип устройства.

Параметр «характеристики устройства» (DeviceCharacteristics) является набором флагов и представляет интерес в основном для разработчиков стандартных типов устройств. Среди возможных значений флагов – FILE_REMO-VABLE_MEDIA, FILE_READ_ONLY_DEVICE, FILE_REMOTE_DEVICE.

Параметр «эксклюзивность устройства» (Exclusive). Когда этот параметр установлен в TRUE, для устройства может быть создан единственный объект-файл. Как видно из рис. 8, взаимосвязь различных объектов может быть довольно сложной. При этом довольно часто в драйвере необходимо представлять, от какого файлового объекта поступил запрос. Для упрощения ситуации можно разрешить использование только одного файлового объекта, то есть в один момент времени устройство будет использоваться только из одного места.

Как уже говорилось, при открытии устройства из подсистемы Win32 для имени устройства в директории \Device в пространстве имен диспетчера объектов должна быть создана символическая связь в директории \?? или \DosDevices. Это можно сделать в драйвере с помощью функции IoCreateSymbolicLink(), либо из прикладной программы с помощью Win32-функции DefineDosDevice().

2.4.1.4. Передача данных от приложения к драйверу. Асинхронная обработка

Код пользовательского уровня не может напрямую вызвать код режима ядра. Для этого существуют специальные прерывания. Одним из них является прерывание 2E – вызов системного сервиса. Диспетчер ввода/вывода обрабатывает вызовы системных сервисов специальным образом (см. рис. 9). В своем обработчике системного сервиса он создает специальный запрос ввода/вывода IRP и передает его на обработку некоторому объекту-устройству, после чего работа обработчика может завершиться, но обработка IRP при этом может быть не закончена.

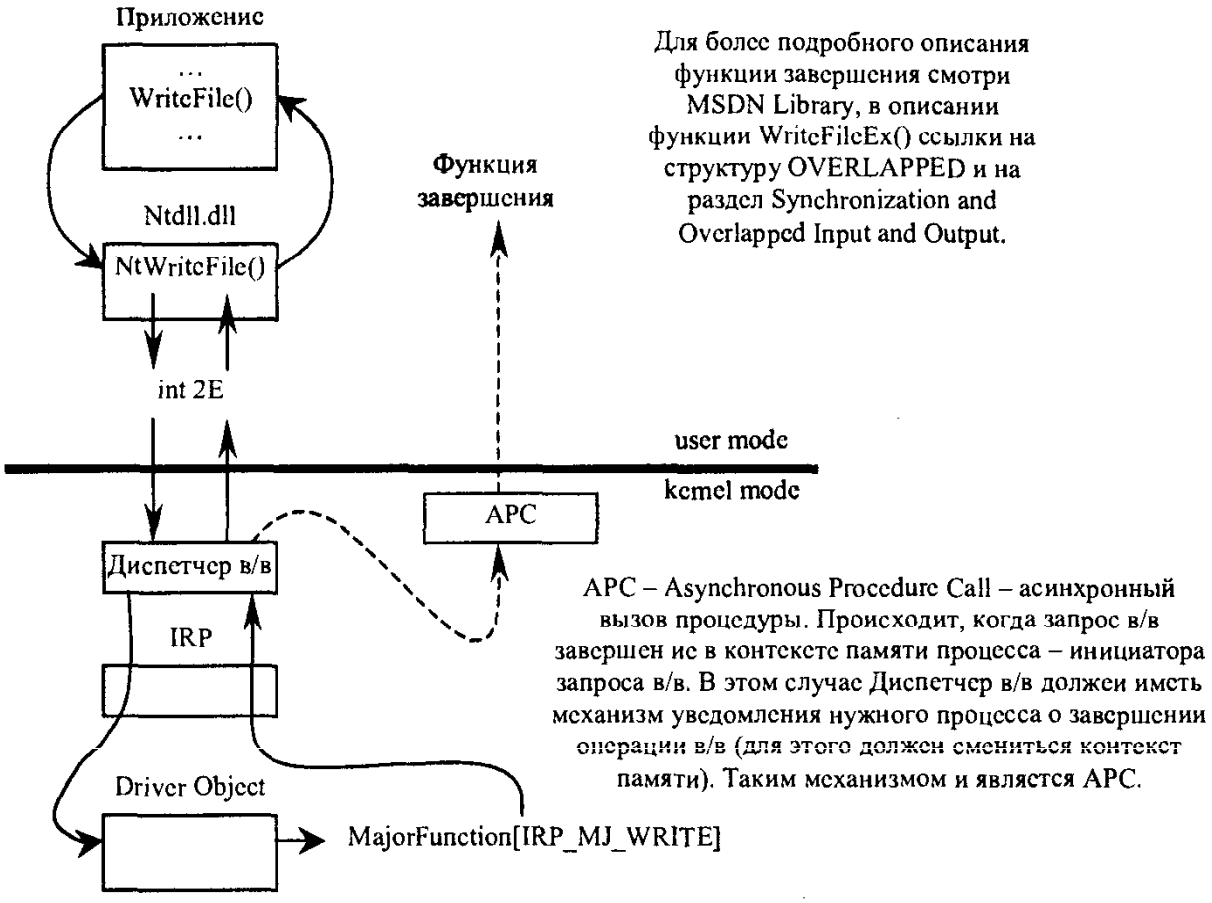


Рис. 9

Модель ввода/вывода, предусматривающая завершение функции ввода/вывода до завершения запроса ввода/вывода называется асинхронным вводом/выводом. Асинхронный ввод/вывод позволяет программе запросить выполнение операции ввода/вывода, после чего продолжать выполнение другой операции, пока устройство не закончит пересылку данных. Система ввода/вывода автоматически уведомляет программу о завершении операции ввода/вывода.

Модель ввода/вывода, обеспечиваемая Диспетчером ввода/вывода, асинхронна всегда, хотя этот факт может быть скрыт функциями подсистемы окружения. Например, так происходит в случае функции CreateFile(). Эта функция не является асинхронной, хотя пользуется асинхронной функцией NtCreateFile().

При асинхронном вводе/выводе более поздний по времени запрос ввода/вывода может быть закончен до завершения более ранних запросов.

Вызов прерывания и переключение процессора в режим ядра не влияют на текущий контекст памяти. Из этого следует, что при обработке вызова системного сервиса и Диспетчер памяти, и соответствующая точка входа драйвера работают в контексте памяти процессора – инициатора запроса ввода/вывода.

В случае, когда устройство обрабатывает запрос ввода/вывода сразу при его поступлении, диспетчеру ввода/вывода не требуется переключение контекста памяти для уведомления процесса о завершении запроса.

В случае, когда устройство отложило запрос в очередь запросов, в тот момент, когда подошла очередь запроса на обработку, контекст памяти будет неизвестен – случайный контекст памяти. При этом диспетчеру ввода/вывода понадобится механизм уведомления нужного процесса о завершении запроса ввода/вывода. Таким механизмом является механизм Асинхронного Вызова Процедуры (Asynchronous procedure call, APC). Вкратце он состоит в том, что прикладная программа предоставляет диспетчеру ввода/вывода адрес функции, которая должна быть вызвана при завершении запроса ввода/вывода. При запросе APC диспетчер ввода/вывода указывает этот адрес и поток, в котором должна быть вызвана эта функция. APC запрашивается с помощью генерации специального прерывания на уровне IRQL равном APC_LEVEL. Запрос APC откладывается в очередь APC и будет выполнен, когда управление получит нужный поток и текущий уровень IRQL будет меньше APC_LEVEL.

2.4.1.4.1. Выполнение асинхронного запроса

Выполняя асинхронный ввод/вывод, поток пользовательского режима может использовать для синхронизации с моментом завершения операции ввода/вывода: 1) ожидание у описателя файла; 2) ожидание у объекта-события, используемого для каждого запроса ввода/вывода; 3) асинхронный вызов процедуры (Asynchronous procedure call, APC) пользовательского режима.

Выполнение асинхронного запроса приводится на примере запроса на запись, проходящего через несколько слоев драйверов. При этом добавляется один или несколько уровней обработки.

Вместо повторного использования одного IRP, драйвер верхнего уровня может создать группу ассоциированных IRP, которые будут управлять одним запросом параллельно. Этот драйвер отслеживает завершение всех ассоциированных IRP, и только потом завершает исходный IRP.

1) Подсистема среды или клиентская DLL вызывает функцию «родного» API NtWriteFile() с описателем файлового объекта.

2) Диспетчер ввода/вывода создает IRP, в котором он сохраняет указатель на файловый объект и код функции, указывающий драйверу верхнего уровня тип операции. Далее диспетчер ввода/вывода отыскивает драйвер верхнего уровня и передает ему IRP.

3) Драйвер верхнего уровня определяет по информации в соответствующей ему области стека IRP, какую операцию он должен выполнить. Драйвер верхнего уровня может, либо разбить первоначальный запрос на несколько запросов, путем размещения новых пакетов IRP (возможно для нескольких драйверов устройств), либо может повторно использовать первоначальный IRP, заполнив область стека IRP, соответствующую нижележащему драйверу. Затем драйвер передает этот IRP (или несколько ассоциированных IRP) посредством вызова диспетчера ввода/вывода.

4) Вызванный драйвер нижнего уровня (пусть это будет уже драйвер устройства) проверяет свою область стека IRP, чтобы определить какую операцию он должен выполнить на устройстве, и в поле статуса операции ввода/вывода в пакете IRP ставит код «ввод/вывод выполняется». Затем он вызывает функцию IoStartPacket(), реализуемую менеджером ввода/вывода.

5) Диспетчер ввода/вывода определяет, занято ли уже устройство обработкой другого IRP, и если да, то ставит текущий IRP в очередь устройства и возвращает управление. Если нет, то диспетчер ввода/вывода передает IRP соответствующей процедуре драйвера устройства, которая начинает операцию ввода/вывода на устройстве. Начав операцию на устройстве, драйвер устройства возвращает управление. Заметьте, что асинхронный запрос ввода/вывода возвращает управление вызывающей программе немедленно, даже если очередь устройства пуста.

Вторая стадия обработки запроса ввода/вывода, состоит в обслуживании прерывания от устройства:

1) После завершения передачи данных устройство генерирует прерывание для обслуживания. Диспетчер прерываний ядра передает управление процедуре обслуживания прерываний (Interrupt Service Routine, ISR) драйвера устройства.

2) ISR выполняет минимум работы по сохранению необходимого контекста операции. ISR также вызывает соответствующие сервисы диспетчера ввода/вывода для постановки в очередь отложенного вызова процедуры (Deferred Procedure Call, DPC). DPC – асинхронная процедура драйвера устройства, выполняющая завершение требуемой операции при более низком уровне IRQL процессора.

Прерывания от устройств имеют высокий IRQL, но IRQL процессора, выполняющего ISR, остается на этом уровне только на время, необходимое для того, чтобы запретить новые прерывания от устройства. После завершения ISR, ядро понижает IRQL процессора до уровня, на котором тот находился до прерывания.

3) Если IRQL процессора понизится ниже уровня планирования потоков и обработки отложенного вызова процедуры (Dispatch level), возникнет прерывание DPC, и диспетчер прерываний передаст управление процедуре DPC драйвера устройства.

4) Процедура DPC использует для завершения операции сохраненный в процедуре ISR контекст операции, запоминает статус завершившейся операции и выбирает из очереди следующий пакет IRP, запустив тем самым новый запрос ввода/вывода. Затем устанавливает статус только что завершенной операции в поле статуса IRP и вызывает диспетчер ввода/вывода для завершения обработки запроса и удаления IRP.

5) Диспетчер ввода/вывода обнуляет область стека IRP, соответствующую драйверу устройства, и вызывает завершающую процедуру драйвера верхнего уровня. Эта процедура проверяет поле статуса операции, чтобы определить, нужно ли повторить запрос, или можно освободить этот IRP (если это IRP было размещено этим драйвером). Драйвер верхнего уровня собирает информацию о статусах операций для всех размещенных им пакетов IRP, чтобы установить общий статус в первоначальном пакете IRP и завершить его.

6) Диспетчер ввода/вывода ставит в очередь APC-объект (Asynchronous Procedure Call – асинхронный вызов процедуры) для завершения ввода/вывода в контексте потока – инициатора запроса. (Подсистема ввода/вывода должна скопировать некоторые данные из системной области памяти в виртуальное адресное пространство процесса, поток которого инициировал запрос ввода/вывода, во время исполнения такого потока, это достигается путем пересылки APC-объекта режима ядра в очередь APC-объектов этого потока.)

7) Когда поток – инициатор запроса, начнет исполняться в следующий раз, то возникнет прерывание обработки асинхронного вызова процедуры. Диспетчер прерываний передает управление процедуре APC режима ядра (процедуре APC диспетчера ввода/вывода).

8) Процедура APC режима ядра записывает данные в адресное пространство потока-инициатора запроса, устанавливает описатель файла в состояние «свободен», устанавливает в очередь на исполнение APC-объект пользовательского режима (если нужно) и удаляет IRP.

2.4.2. Пакет запроса ввода/вывода (IRP)

2.4.2.1. Характеристики подсистемы ввода/вывода

В предыдущем разделе мы рассмотрели схему использования системных сервисов, то есть прохождение запроса ввода/вывода от приложения к драйверу и обратно. Компонентом ОС, отвечающим за реализацию этой схемы, является Диспетчер ввода/вывода. Диспетчер ввода/вывода является компонентом более общей модели – подсистемы ввода/вывода. Подсистема ввода/вывода включает в себя все компоненты, которые обеспечивают возможность осуществления ввода/вывода. В число этих компонент входит Диспетчер ввода/вывода и все драйверы режима ядра. В числе характеристик подсистемы ввода/вывода NT принято выделять следующие:

1) согласованность и высокая структурированность;
2) переносимость между процессорными архитектурами;
3) конфигурируемость;
4) вытесняемость и прерываемость;
5) поддержка многопроцессорности;
6) объектная базированность (но не объектная ориентированность);
7) асинхронность;
8) подсистема ввода/вывода управляется пакетами;
9) подсистема ввода/вывода многоуровневая (послойная модель).

Как уже говорилось, подсистема ввода/вывода NT управляется пакетами. При таком подходе каждый запрос ввода/вывода описывается своим собственным пакетом запроса ввода/вывода (I/O Request Packet – IRP). При задействовании системного сервиса (например, при запросе на чтение или запись в файл) Диспетчер ввода/вывода обрабатывает этот запрос путем создания пакета IRP, описывающего запрос, и затем передает указатель на этот пакет драйверу для обработки.

2.4.2.2. Структура пакета запроса ввода/вывода (IRP)

При осуществлении операции ввода/вывода диспетчер ввода/вывода создает специальный пакет, описывающий эту операцию – **пакет запроса ввода/вывода** (I/O Request Packet, IRP). Как будет показано ниже, обработка такого пакета может происходить поэтапно несколькими объектами-устройствами.

IRP содержит всю необходимую информацию для полного описания запроса Ввода – вывода Диспетчеру Ввода/вывода и драйверам устройств. IRP описывается стандартной структурой типа "IRP", показанной на рис. 10.

Структура IRP специально разработана для поддержки многоуровневой модели ввода/вывода, при которой запрос ввода/вывода последовательно обрабатывается стеком из нескольких драйверов.

Для обеспечения этого каждый пакет запроса ввода/вывода состоит из двух частей: **"фиксированной" части** и **Стека Ввода/вывода**. **Фиксированная часть** IRP содержит информацию относительно той части запроса, которая или не изменяется от драйвера к драйверу, или которую не надо сохранять при передаче IRP от одного драйвера к другому. **Стек Ввода/вывода** содержит набор **Стеков Размещения Ввода/вывода**, каждый из которых содержит информацию, специфическую для каждого драйвера, который может обрабатывать запрос.

В стеке размещения ввода/вывода для каждого устройства, которое должно принимать участие в обработке пакета, содержатся указатели на объект-устройство, которое будет обрабатывать запрос, и на объект-файл, для которого была инициирована операция ввода/вывода (см. рис. 10).

Пакеты IRP всегда выделяются из невыгружаемой системной памяти (nonpaged pool), поэтому к ним может осуществляться обращение из функций, работающих на любом уровне IRQL.

Как уже говорилось, драйверы подразделяются на три класса по их положению в стеке драйверов: драйверы высшего уровня, драйверы промежуточного уровня и драйверы низшего уровня.

Драйвер высшего уровня – это верхний драйвер в стеке драйверов, получающий запросы через Диспетчер ввода/вывода от компонентов прикладного уровня.

Драйвер высшего уровня (или, что более правильно, устройство высшего уровня) имеет один или несколько стеков размещения ввода/вывода.

Число стеков размещения ввода/вывода устанавливается Диспетчером ввода/вывода в поле StackSize объекта-устройства. По умолчанию это значение равно 1. Присваивание происходит при создании устройства функцией IoCreateDevice(). Если вы создаёте многоуровневый драйвер, вы должны установить StackSize на 1 больше, чем StackSize объекта-устройства, над которым будете размещать свое устройство. В случае, если ваше устройство будет использовать больше одного устройства уровнем ниже, поле StackSize этого устройства должно быть на 1 больше максимального значения StackSize из всех устройств уровнем ниже.

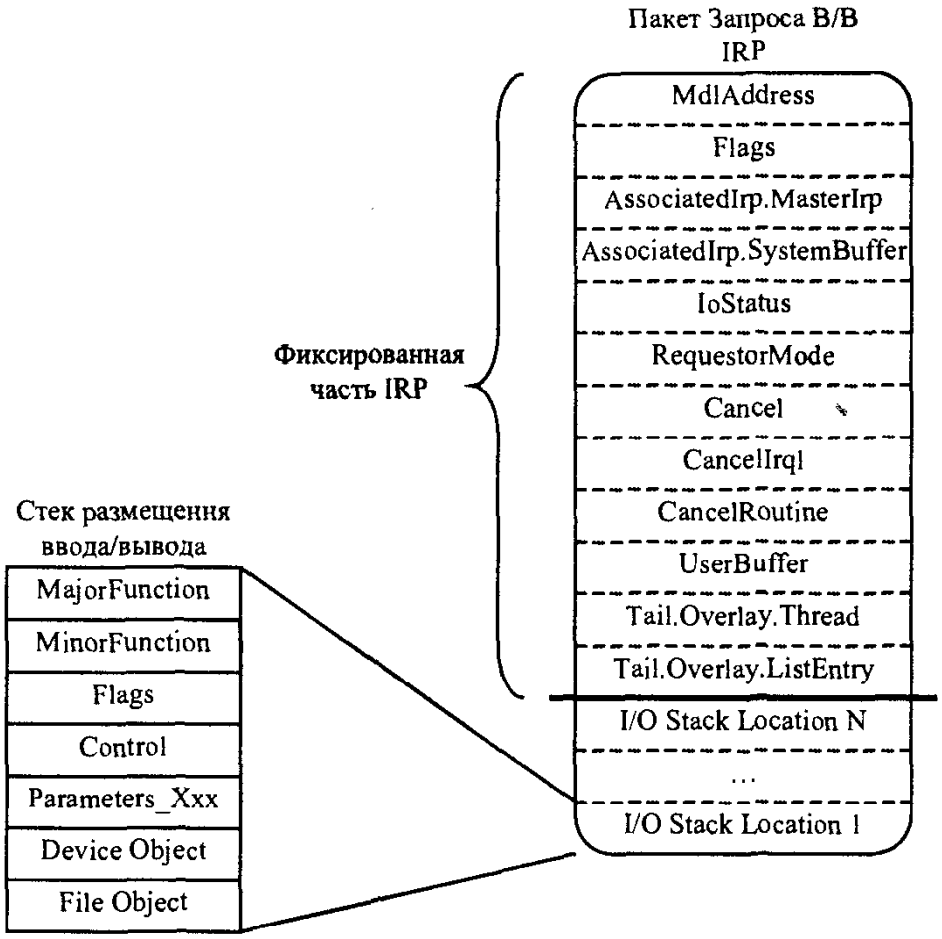


Рис. 10

2.4.2.2.1. Поля в фиксированной части IRP

Как было уже сказано, фиксированная часть IRP содержит информацию, которая или не изменяется от драйвера к драйверу или не должна сохраняться, когда IRP передается от одного драйвера к другому. Особенно интересными или полезными поля в фиксированной части IRP являются следующие:

1) MdlAddress. Это поле указывает на Таблицу Описания Памяти (MDL), которая описывает буфер запроса, когда драйвер использует Прямой ввода/вывода (Direct I/O – представлен далее в этом разделе).

2) Flags. Как подразумевает название, это поле содержит флаги, которые (обычно) описывают запрос ввода/вывода. Например, если в этом поле установлен флаг IRP_PAGING_IO, это указывает на то, что операция чтения или операция записи, описанная IRP, есть страничный запрос. Точно так же бит IRP_NOCACHE указывает, что запрос должен быть обработан без промежуточной буферизации. Поле **Flags** обычно представляют интерес только для файловых систем.

3) AssociatedIrp.MasterIrp. В связанном (associated) IRP это указатель на главный (master) IRP, с которым связан этот запрос. Это поле представляет интерес только драйверам верхнего уровня, типа драйверов файловых систем.

4) AssociatedIrp.SystemBuffer. Это место указывает на промежуточный буфер в невыгружаемой памяти, содержащий данных запроса в случае, когда драйвер использует буферизированный ввод/вывод (Buffered I/O).

5) IoStatus. Это Блок Состояния Ввода/вывода, который описывает состояние завершения обработки IRP. Когда IRP завершен, драйвер помещает в поле **IoStatus.Status** состояние завершения операции ввода/вывода, а в поле **IoStatus.Information** – любую дополнительную информацию, которую нужно передать обратно инициатору запроса ввода/вывода. Как правило, поле **IoStatus.Information** содержит фактическое число байтов, прочитанных или записанных запросом передачи данных.

6) RequestorMode. Это поле указывает режим работы процессора (режим ядра или пользовательский режим), из которого был инициирован запрос ввода/вывода.

7) Cancel, CancelIrql и CancelRoutine. Эти поля используются, если IRP может быть отменен в процессе обработки. Cancel – поле типа BOOLEAN, значение которого устанавливается Диспетчером ввода/вывода. Установка в TRUE указывает, что была запрошена отмена операции ввода/вывода, описанная этим IRP. CancelRoutine – это указатель на функцию драйвера (точка входа драйвера), вызываемую Диспетчером Ввода/вывода для того, чтобы драйвер мог корректно отменить IRP. Точка входа CancelRoutine вызывается на IRQL DISPATCH_LEVEL, CancelIrql является тем уровнем IRQL, к которому драйвер должен возвратиться. Более подробно обработка отмены запроса ввода/вывода будет обсуждаться в разделе, посвященном сериализации.

8) UserBuffer. Это поле содержит виртуальный адрес буфера данных инициатора запроса, связанного с запросом Ввода/вывода, если такой буфер имеется.

9) Tail.Overlay.DeviceQueueEntry. Это поле используется Диспетчером Ввода/вывода для постановки IRP в очередь в случае использования системной очереди (System Queuing). Системная очередь будет обсуждаться в разделе, посвященном сериализации.

10) Tail.Overlay.Thread. Это поле является указателем на управляющий блок потока инициатора запроса (ETHREAD).

11) TailOverlay.ListEntry. Когда драйвер сам создал IRP, он может использовать это поле для соединения одного IRP с другим.

2.4.2.2.2. Поля в стеке размещения ввода/вывода IRP

Каждый Стек размещения Ввода/вывода в IRP содержит информацию для конкретного драйвера относительно запроса Ввода/вывода. Стек размещения Ввода/вывода определяется структурой IO_STACK_LOCATION. Для определения местонахождения текущего Стека Размещения Ввода/вывода внутри данного IRP, драйвер должен использовать функцию IoGetCurrentIrp StackLocation(). Единственным параметром при вызове является указатель на IRP. Возвращаемым значением будет указатель на текущий Стек размещения Ввода/вывода. Когда Диспетчер Ввода/вывода создает IRP и инициализирует его фиксированную часть, он также инициализирует в IRP первый Стек Размещения Ввода/вывода. В него помещается информация, которую нужно передать первому драйверу в стеке драйверов, которые будут обрабатывать этот запрос. Поля в Стеке Размещения Ввода/вывода включают следующее:

1) MajorFunction. Это поле указывает главный код функции ввода/вывода, связанный с запросом ввода/вывода. Тем самым указывается тип операции ввода/вывода, которая должна быть выполнена.

2) MinorFunction. Это поле указывает второстепенный код функции ввода/вывода, связанный с запросом. При использовании, это поле переопределяет главный функциональный код. Второстепенные функции используются почти исключительно сетевыми транспортными драйверами и файловыми системами и игнорируются большинством драйверов устройств.

3) Flags. Это поле содержит флаги обработки, определенные для выполняемой функции ввода/вывода. Это поле представляет интерес главным образом для драйверов файловых систем.

4) Control. Это поле является набором флагов, которые устанавливаются и читаются Диспетчером Ввода/вывода, указывая, как надо обработать данный пакет IRP. Например, в этом поле с помощью обращения драйвера к функции IoMarkIrpPending() может быть установлен бит SL_PENDING, указывающий Диспетчеру Ввода/вывода, что завершение обработки пакета IRP отложено на неопределенное время. Точно так же флажки SL_INVOKE_ON_CANCEL, SL_INVOKE_ON_ERROR и SL_INVOKE_ON_SUCCESS указывают, когда для этого IRP должна быть вызвана Подпрограмма Завершения Ввода/вывода драйвера.

5) Parameters. Это поле включает несколько подполей, каждое из которых зависит от главной функции Ввода – вывода, которая будет выполняться.

6) DeviceObject. Это поле содержит указатель на объект-устройство, который является получателем запроса Ввода/вывода.

7) FileObject. Это поле содержит указатель на объект-файл, связанный с запросом Ввода/вывода.

После того, как фиксированная часть IRP и первый Стек размещения Ввода/вывода в IRP инициализированы, Диспетчер Ввода/вывода вызывает верхний драйвер в стеке драйверов в его точке входа dispatch, которая соответствует главному функциональному коду для запроса. Таким образом, если Диспетчер Ввода/вывода сформировал IRP для описания запроса чтения, он вызовет первый драйвер в стеке драйверов в его диспетчерской точке входа для чтения (IRP_MJ_READ). При этом Диспетчер Ввода/вывода передает следующие параметры:

- указатель на IRP, который был только что сформирован;
- указатель на обьект-устройство, который соответствует устройству, для которого драйвер должен обработать запрос.

2.4.2.3. Описание буфера данных

Описатель для буфера данных инициатора запроса находится в фиксированной части IRP. Для осуществления операции ввода/вывода NT предусматривает три различных метода передачи буфера данных, принадлежащего инициатору запроса:

- **Прямой Ввод/вывод (Direct I/O)**. Буфер находится в виртуальном адресном пространстве инициатора запроса. Для передачи буфера драйверу Диспетчер ввода/вывода создает таблицу описания памяти (MDL), описывающую размещение буфера в физической памяти.

- **Буферизированный Ввод/вывод (Buffered I/O)**. Для передачи буфера драйверу Диспетчер ввода/вывода создает в невыгружаемой системной памяти копию первоначального буфера. Драйверу передается указатель на этот новый буфер. Выделенная память будет освобождена Диспетчером ввода/вывода при завершении запроса ввода/вывода.
- **«Никакой» Ввод/вывод (Neither I/O)**. В драйвер передается виртуальный адрес буфера инициатора запроса.

Коды функции Ввода/вывода и IOCTLs освещены более подробно ниже.

Таблица 6. Характеристики Прямого ввода/вывода, Буферизованного ввода/вывода и «никакого» ввода/вывода

	Прямой ввод/вывод (Direct I/O)	**Буферизованный ввод/вывод (Buffered I/O)**	**«Никакой» ввод/вывод (Neither I/O)**
Буфер инициатора запроса	Описывается с помощью MDL	Скопирован во временный буфер в невыгружаемой системной памяти	Описывается виртуальным адресом инициатора запроса
Состояние буфера инициатора запроса в процессе обработки запроса ввода/вывода	Буфер блокирован в памяти Диспетчером ввода/вывода	Буфер не блокирован	Буфер не блокирован
Описание буфера в IRP	Irp->MdlAddress содержит указатель на MDL	Irp->AssociatedIrp.SystemBuffer содержит виртуальный адрес временного буфера в системной области памяти в области невыгружаемой памяти (nonpaged pool)	Irp->UserBuffer содержит не проверенный на доступность виртуальный адрес буфера инициатора запроса ввода/вывода
Контекст, при котором буфер может быть использован	Случайный контекст	Случайный контекст	Только контекст потока - инициатора запроса
Уровень IRQL, при котором буфер может быть использован	IRQL < DISPATCH_LEVEL	Любой	IRQL < DISPATCH_LEVEL

Для описания всех запросов чтения и записи, которые посылаются конкретному устройству, после создания объекта-устройства в его поле Flags Диспетчеру ввода/

вывода должен быть указан единственный метод для использования. Однако для каждого кода Управления Вводом/выводом Устройства (I/O Control Code, IOCTL), поддерживаемого драйвером, может использоваться любой другой метод.

2.4.2.4. Коды функции ввода/вывода

NT использует коды функции ввода/вывода для определения конкретной операции ввода/вывода, которая будет иметь место для конкретного объекта-файла. Коды функции ввода/вывода Windows NT разделены на коды главной и второстепенной функции ввода/вывода. Оба кода находятся в IRP в Стеке Размещения Ввода/вывода драйвера. Главные функциональные коды определены символами, которые начинаются с IRP_MJ_. Перечислим некоторые из главных кодов функции ввода/вывода:

1) IRP_MJ_CREATE. Этот главный функциональный код соответствует созданию нового объекта-файла, либо при обращении к существующему устройству или файлу, либо при создании нового файла. Этот функциональный код представляет запросы, идущие через функцию Win32 CreateFile() или базовый системный сервис NtCreateFile().

2) IRP_MJ_CLOSE. Этот главный функциональный код соответствует уничтожению предварительно созданного объекта-файла. Этот функциональный код представляет запросы, идущие через функцию Win32 CloseHandle() или базовый системный сервис NtClose(). К появлению этого запроса ввода/вывода может привести не каждый вызов CloseHandle(), так как на соответствующий файловый объект могут ссылаться другие, еще не закрытые описатели. Объект не может быть уничтожен, пока для него есть описатели. Кроме того, для каждого объекта диспетчер объектов ведет подсчет ссылок, и объект не может быть уничтожен, пока его число ссылок не равно нулю.

3) IRP_MJ_READ. Этот главный функциональный код выполняет операцию чтения для существующего объекта-файла. Этот функциональный код представляет запросы, идущие через функцию Win32 ReadFile() или базовый системный сервис NtReadFile().

4) IRP_MJ_WRITE. Этот главный функциональный код выполняет операцию записи для существующего объекта-файла. Этот функциональный код представляет запросы, идущие через функцию Win32 WriteFile() или системный сервис NtWriteFile().

5) IRP_MJ_DEVICE_CONTROL. Этот главный функциональный код выполняет определенную драйвером функцию для существующего объекта-файла. Этот функциональный код представляет запросы, идущие через функцию Win32 DeviceIoControl() или базовый системный сервис NtDeviceIoControlFile().

6) IRP_MJ_INTERNAL_DEVICE_CONTROL. Этот главный функциональный код выполняет определенную драйвером функцию для существующего объекта-файла. Никаких API уровня пользователя, соответствующих этой функции, нет. Эта функция используется, когда один драйвер посылает запрос ввода/вывода другому драйверу.

Законченный список кодов функции ввода/вывода представлен в NTDDK.H. Второстепенные коды функции ввода/вывода в NT определены символами, которые начи-

наются с IRP_MN_. NT обычно избегает использование второстепенных функциональных кодов для перезагрузки главной функции для драйверов устройства, приветствуя вместо этого использование Кодов Управления вводом/выводом (I/O Control Codes, IOCTL). Поэтому, почти все IRP, полученные драйверами устройства, имеют второстепенный функциональный код IRP_MN_NORMAL (который имеет значение 0x00). Вообще, второстепенные коды функции ввода/вывода используются исключительно файловыми системами и сетевыми транспортами. Например, одним из второстепенных кодов функции ввода/вывода, специфичным для файловой системы, является IRP_MN_COMPRESSED, указывающий, что данные должны быть записаны на том в сжатом формате.

Главные и второстепенные коды функции ввода/вывода, связанные с конкретным IRP, сохранены в полях MajorFunction и MinorFunction текущего Стека Размещения Ввода/вывода в IRP. На эти поля можно ссылаться, как показано в примере проверки главных и второстепенных функциональных кодов IRP:

```
IoStack = IoGetCurrentIrpStackLocation(Irp);
if (IoStack->MajorFunction == IRP_MJ_READ)
{
    if (IoStack->MinorFunction == IRP_MN_NORMAL)
    {
    // что-то делать
    }
}
```

2.4.2.5. Диспетчерские точки входа драйвера

Информация, требуемая для выполнения запроса ввода/вывода, содержится в различных элементах как фиксированной части IRP, так и стека размещения ввода/вывода. Рассмотрим эти элементы. Структура поля Parameters в стеке размещения ввода/вывода зависит от кода главной и второстепенной функции ввода/вывода. Нас в основном будет интересовать структура поля Parameters для запросов чтения, записи и пользовательских запросов ввода/вывода:

1) **IRP_MJ_READ**. Параметры для этого функционального кода содержат следующее:
   - Parameters.Read.Length (ULONG) содержит размер в байтах буфера инициатора запроса.
   - Parameters.Read.Key (ULONG) содержит ключевое значение, которое нужно использовать при чтении. Обычно представляет интерес только для драйверов файловой системы.
   - Parameters.Read.ByteOffset (LARGE_INTEGER) содержит смещение (обычно в файле), с которого должна начаться операция чтения.
2) **IRP_MJ_WRITE**. Параметры для этого функционального кода следующие:
   - Parameters.Write.Length (ULONG) содержит размер в байтах буфера инициатора запроса.

- Parameters.Write.Key (ULONG) содержит ключевое значение, которое нужно использовать при записи. Обычно представляет интерес только для драйверов файловой системы.
- Parameters.Write.ByteOffset (LARGE_INTEGER) содержит смещение (обычно в файле) с которого должна начаться операция записи.

3) **IRP_MJ_DEVICE_CONTROL**. Параметры для этого функционального кода следующие:
   - Parameters.DeviceIoControl.OutputBufferLength (ULONG) содержит длину в байтах буфера OutBuffer.
   - Parameters.DeviceIoControl.InputBufferLength (ULONG) содержит длину в байтах буфера InBuffer.
   - Parameters. DeviceIoControl.ControlCode (ULONG) содержит код управления вводом/выводом, идентифицирующий запрашиваемую функцию управления устройством. Этот управляющий код обычно предварительно определен драйвером с использованием макрокоманды CTL_CODE.
   - Parameters.DeviceIoControl.Type3InputBuffer (PVOID) содержит виртуальный адрес буфера инициатора запроса InBuffer (см. функцию Win32 API DeviceIoControl()). Адрес обычно используется только тогда, когда IOCTL использует METHOD_NEITHER.

2.4.2.5.1. Запросы чтения и записи IRP_MJ_READ и IRP_MJ_WRITE

Метод передачи буфера, используемый в запросах чтения и записи, контролируется полем **Flags** объекта-устройства. После создания объекта-устройства с помощью функции IoCreateDevice() необходимо инициализировать это поле. Поле может иметь установленными несколько флагов, при этом применяются следующие правила:

1) Если установлены флаги DO_BUFFERED_IO или DO_DIRECT_IO, метод передачи буфера будет соответственно буферизованным или прямым.
2) Если поле флагов не инициализировано (никакие флаги не установлены), используется метод передачи буфера Neither («никакой» ввода/вывода).
3) Одновременная установка флагов DO_BUFFERED_IO и DO_DIRECT_IO запрещена и будет являться ошибкой.
4) Установленный полем Flags метод передачи будет использован и запросом чтения, и запросом записи.

Расположение буфера в зависимости от метода его передачи для запросов чтения и записи полностью определяется таблицей 6.

Для завершения запроса IRP на чтение/запись, необходимо установить поле Irp->IoStatus.Information равным числу прочитанных/записанных в буфер байт. В случае буферизованного ввода/вывода это поле укажет Диспетчеру ввода/вывода, сколько байт нужно скопировать из промежуточного буфера в невыгружаемой области системного адресного пространства в пользовательский буфер.

2.4.2.5.1.1. Пример обработки запросов чтения/записи

Данный пример обработки запросов чтения/записи демонстрирует получение адреса буфера для чтения/записи и его длины. Такой код вставляется в обработчик диспетчерских функций MajorFunction[IRP_MJ_READ], MajorFunction[IRP_MJ_WRITE].

```
//получение адреса буфера для чтения/записи
//в случае буферизованного ввода/вывода
BufferAddress = Irp->AssociatedIrp.SystemBuffer;
     //в случае прямого ввода/вывода
BufferAddress = MmGetSystemAddressForMdl(Irp->MdlAddress);
//в случае Neither i/o
BufferAddress = Irp->AssociatedIrp.UserBuffer;
//получение длины буфера для чтения/записи
stack = = IoGetCurrentIrpStackLocation( Irp );
     BufferLength = stack->Parameters.Read.Length;
```

2.4.2.5.2. Запросы IRP_MJ_DEVICE_CONTROL и IRP_MJ_INTERNAL_DEVICE_CONTROL

Как говорилось выше, точка входа драйвера IRP_MJ_DEVICE_CONTROL вызывается при вызове пользовательской программой функции DeviceIoControl(). Прототип этой функции:

```
BOOL DeviceIoControl(
     HANDLE hDevice,                  // описатель открытого
                                      // устройства
     DWORD dwIoControlCode,           // контрольный код
                                      // запрашиваемой
                                      // операции
     LPVOID lpInBuffer,               // адрес буфера со входными
                                      // данными
     DWORD nInBufferSize,             // размер входного буфера
     LPVOID IpOutBuffer,              // адрес буфера для приема
                                      // выходных данных
     DWORD nOutBufferSize,            // размер выходного буфера
     LPDWORD IpBytesReturnedy         // адрес переменной
                                      // для получения
                                      // числа реально
                                      // переданных байтов данных
     LPOVERLAPPED IpOverlapped        // адрес структуры
                                      // для обеспечения
                                      // асинхронности
                                      // ввода/вывода
);
```

Зачем нужен IRP_MJ_DEVICE_CONTROL? Когда драйвер поддерживает определенный тип устройства, такое устройство обычно имеет набор специализированных возможностей, которые могут управляться через драйвер. Эти возможности не могут быть задействованы использованием стандартных кодов функций IRP_MJ_CREATE, IRP_MJ_CLOSE, IRP_MJ_READ и IRP_MJ_WRITE. Например, драйвер устройства для лентопротяжного устройства SCSI должен обеспечить механизм, который дает возможность пользователям послать запрос на стирание ленты. Такие зависящие от устройства запросы описываются, используя главный функциональный код IRP_MJ_DEVICE_CONTROL. Устройство обычно может принимать несколько разнотипных команд. Для драйвера тип такой команды указывается **Кодом Управления ввода/вывода (IOCTL)**, который передается как часть запроса ввода/вывода.

Возвращаясь к функции DeviceIoControl(), код управления ввода/вывода **(IOCTL)** указывается во втором параметре этой функции. Код управления ввода/вывода имеет специальный формат, указывающий метод передачи буфера и другую информацию. Этот формат будет рассмотрен в следующем разделе. Путаница может возникнуть при задании буфера для ввода/вывода.

Как видно из прототипа функции DeviceIoControl(), она может передавать два буфера (третий и пятый параметры). Несмотря на названия буферов – входной и выходной буфер – выходной буфер может быть использован как для передачи данных в драйвер, так и для приема данных из драйвера. Разница будет в используемом методе передачи буфера. Использование буферов будет подробно рассмотрено в разделе «Получение буфера».

2.4.2.5.2.1. Задание кода управления вводом/выводом (IOCTL)

Формат кода управления ввода/вывода показан на рис. 11.

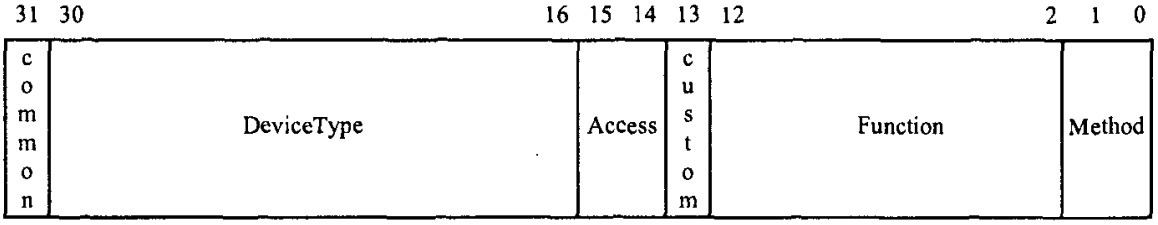


Рис. 11. Формат кода управления вводом/выводом

CTL_CODE(*DeviceType*, *Function*, *Method*, *Access*) – специальный макрос, определенный в заголовочных файлах ntddk.h и windows.h, для задания кода в формате, представленном на рис. 11.

Рассмотрим составляющие кода управления ввода/вывода:

1) Поле **DeviceType** определяет тип объекта-устройства, которому предназначен запрос. Это тот самый тип устройства, который передается функции IoCreateDevice()

при создании устройства. Как уже говорилось, существует два диапазона значений типов устройств: 0-32767 – зарезервированные значения для стандартных типов устройств, 32768-65535 – диапазон значений типов устройств для выбора разработчиком. Следует отметить, что несколько разных устройств могут иметь одинаковое значение типа устройства. Поскольку каждый запрос ввода/вывода предназначен конкретному устройству, совпадение типов устройств не приводит к неприятностям. Также необходимо отметить, что тип устройства в коде управления ввода/вывода может не совпадать с типом устройства объекта-устройства, и это не будет являться ошибкой.

2) Поле **Function** идентифицирует конкретные действия, которые должно предпринять устройство при получении запроса. Значения поля Function должны быть уникальны внутри устройства. Как и для типов устройств, существует два диапазона значений поля Function: 0-2047 – зарезервированный диапазон значений, и 2048-4095 – диапазон значений, доступный разработчикам устройств.

3) Поле **Method** указывает метод передачи буферов данных. Для понимания этого поля вернемся к функции DeviceIoControl(). Функция передает два буфера – InBuffer и OutBuffer. Буфер InBuffer передает данные драйверу, буфер OutBuffer может передавать данные в обоих направлениях (к драйверу и от драйвера).

В следующей таблице приведены возможные значения поля Method и методы передачи буферов InBuffer и OutBuffer:

Значение поля Method	Использование OutBuffer	Используемый метод передачи буфера	
		InBuffer	OutBuffer
METHOD_BUFFERED		Буферизованный ввод/вывод (Buffered I/O)	
METHOD_IN_DIRECT	Передача данных к драйверу	Буферизованный ввод/вывод	Прямой ввод/вывод. Осуществляется проверка буфера на доступ по чтению
METHOD_OUT_DIRECT	Приема данных от драйвера	Буферизованный ввод/вывод	Прямой ввод/вывод. Осуществляется проверка буфера на доступ по записи
METHOD_NEITHER		Neither I/O	

Местоположение буферов данных в пакете IRP будет рассмотрено в следующем разделе («Получение буфера»).

4) Поле Access указывает тип доступа, который должен был быть запрошен (и предоставлен) при открытии объекта-файла, для которого передается данный код Управления вводом/выводом. Возможные значения для этого параметра следующие:

- **FILE_ANY_ACCESS**. Это значение указывает, что при вызове CreateFile() мог быть запрошен любой доступ.

- **FILE_READ_ACCESS**. Это значение указывает, что должен был быть запрошен доступ для чтения.
- **FILE_WRITE_ACCESS**. Это значение указывает, что должен был быть запрошен доступ для записи.

Заметим, что file_read_access и file_write_access могут быть указаны одновременно, чтобы указать, что при открытии устройства должен быть предоставлен и доступ на чтение, и доступ на запись.

Параметр Access требуется потому, что операции управления ввода/вывода, по своей сути не есть операция чтения или записи. Прежде, чем будет разрешена запрашиваемая операция ввода/вывода, Диспетчер Ввода/вывода должен знать, какой режим доступа нужно проверить в таблице описателей.

2.4.2.5.2.2. Получение буфера

При использовании буферизованного метода, Диспетчер ввода/вывода выделяет в системной невыгружаемой памяти промежуточный буфер, размер которого равен максимальному из размеров буферов InBuffer и OutBuffer. Если при запросе был определен InBuffer и его длина не нулевая, содержание InBuffer копируется в промежуточный буфер. В любом случае, адрес промежуточного буфера помещается в IRP в поле AssociatedIrp.SystemBuffer. Затем IRP, содержащий запрос, передается драйверу.

Данные, находящиеся в промежуточном буфере, могут читаться и перезаписываться драйвером. Затем драйвер размещает в промежуточном буфере данные, которые нужно вернуть в OutBuffer.

При завершении запроса ввода/вывода, если OutBuffer был определен при запросе ввода/вывода и его длина не нулевая, Диспетчер ввода/вывода копирует из промежуточного буфера в OutBuffer столько байтов, сколько было указано в поле Irp->IoStatus.Information. После этого, как и при любом буферизированном запросе Ввода/вывода, Диспетчер ввода/вывода освобождает промежуточный буфер.

При использовании методов METHOD_IN_DIRECT и METHOD_OUT_DIRECT, буфер InBuffer, если он определен в запросе ввода/вывода и его длина не нулевая, обрабатывается в точности так же, как и при буферизованном вводе/выводе. В этом случае выделяется промежуточный буфер, в него копируется InBuffer, указатель на промежуточный буфер помещается в IRP в поле AssociatedIrp.SystemBuffer.

Буфер OutBuffer, если он определен в запросе ввода/вывода и его длина не нулевая, обрабатывается в соответствии с прямым вводом/выводом. В этом случае адрес проверяется на возможность доступа (запись или чтение), производится закрепление физических страниц в памяти, и создается таблица описания памяти MDL, описывающая OutBuffer. Указатель на MDL передается в поле Irp->MdlAddress.

При использовании метода METHOD_NEITHER, оба буфера передаются в соответствии с методом Neither. То есть, не производится проверка доступности памяти, не выделяются промежуточные буфера и не создаются MDL. В пакете IRP передаются виртуальные адреса буферов в пространстве памяти инициатора запроса ввода/вывода. Адрес буфера OutBuffer передается в фиксированной части IRP в поле Irp-

>UserBuffer, адрес буфера InBuffer передается в стеке размещения ввода/вывода в поле stack->Parameters.DeviceControl.Type3InputBuffer.

Положение буферов показано в таблице 7.

Таблица 7

		METHOD_ BUFFERED	METHOD_IN_ DIRECT	METHOD_ OUT_ DIRECT	METHOD_ NEITHER
InBuffer	**Метод передачи**	Buffered I/O	Buffered I/O	Buffered I/O	Виртуальный адрес инициатора запроса
	Если существует, то где расположен	Адрес промежуточного буфера в фиксированной части IRP в поле Irp->AssociatedIrp.SystemBuffer			В стеке размещения ввода/вывода виртуальный адрес инициатора запроса в Parameters. DeviceIoControl. Type3InputBuffer
	Длина	Длина в байтах в поле Parameters.DeviceIoControl.InputBuffer Length в текущем стеке размещения ввода/вывода.			
Out-Buffer	**Метод передачи**	Buffered I/O	Direct I/O	Direct I/O	Виртуальный адрес инициатора запроса
	Если существует, то где расположен	Адрес промежуточного буфера в фиксированной части IRP в поле Irp->AssociatedIrp.SystemBuffer	MDL, адрес в Irp->MdlAddress	MDL, адрес в Irp->MdlAddress	Виртуальный адрес инициатора запроса в Irp->UserBuffer
	Длина	Длина в байтах в поле Parameters.DeviceIoControl.OutputBufferLength в текущем стеке размещения ввода/вывода.			

Для завершения запроса IRP необходимо установить поле Irp->IoStatus.Information равным числу прочитанных/записанных в буфер байт. В случае буферизованного ввода/вывода это поле укажет Диспетчеру ввода/вывода, сколько

байт нужно скопировать из промежуточного буфера в невыгружаемой области системного адресного пространства в пользовательский буфер.

2.4.2.5.2.3. Пример обработки

Пример получения адресов и длин буферов в диспетчерской функции драйвера, обрабатывающей функциональные коды IRP_MJ_CREATE, IRP_MJ_CLOSE и IRP_MJ_DEVICE_CONTROL:

```
stack = IoGetCurrentIrpStackLocation(Irp);
switch (pIrpStack->MajorFunction)
{
case IRP_MJ_CREATE:
case IRP_MJ_CLOSE:
   break;

case IRP_MJ_DEVICE_CONTROL:
    switch (stack->Parameters.DeviceIoControl.IoControlCode)
   {
   case IOCTL_MY_BUFFERED:
      InBuffer = Irp->AssociatedIrp.SystemBuffer;
     InLength = stack->Parameters.DeviceIoControl.InputBuffer.Length;
      OutBuffer = Irp->AssociatedIrp.SystemBuffer;
      OutLength = stack->Parameters.DeviceIoControl.OutputBufferLength;
   case IOCTL_MY_IN_DIRECT:
      //OutBuffer доступен только для чтения
      InBuffer = Irp->AssociatedIrp.SystemBuffer;
     InLength = stack->Parameters.DeviceIoControl.InputBufferLength;
      OutBuffer = MmGetSystemAddressForMdl( Irp->MdlAddress
);
      OutLength = stack->Parameters.DeviceIoControl.OutputBufferLength;
      break;
   case IOCTL_MY_OUT_DIRECT:
      //OutBuffer доступен для чтения/записи
      InBuffer = Irp->AssociatedIrp.SystemBuffer;
     InLength = stack->Parameters.DeviceIoControl.InputBufferLength;
      OutBuffer = MmGetSystemAddressForMdl( Irp->MdlAddress
);
      OutLength = stack->Parameters.DeviceIoControl.OutputBufferLength;
      break;

   case IOCTL_MY_NEITHER:
      InBuffer = irpStack->Parameters.DeviceIoControl.Type3InputBuffer;
```

```
    InLength = irpStack->Parameters.DeviceIoControl.InputBufferLength;
     OutBuffer = Irp->UserBuffer;
   OutLength = irpStack->Parameters.DeviceIoControl.OutputBufferLength;
     break;
}
```

2.4.3. Многоуровневая модель драйверов

Ранее в качестве одной из характеристик подсистемы ввода/вывода упоминалась ее многоуровневость. Что это такое?

NT позволяет выстраивать драйверы в соответствии с некоторой функциональной иерархией. При этом, например, одни драйверы имеют своей единственной целью обмен данными с некоторым физическим устройством. Что это за данные и что с ними делать, такие драйвера не знают. Другие драйвера выполняют обработку данных, но не знают в точности, как эти данные получены и как будут отправлены. Такая концепция разделения полномочий драйверов носит название **многоуровневой (или послойной) модели драйверов (layered driver model)**, а сами драйвера – **уровневыми драйверами (layered drivers)**.

В NT 4.0 концепция многоуровневых драйверов занимает важное место, но ее использование не является обязательным требованием.

В Win2000 все драйвера, считающиеся родными, будут уровневыми (для того, чтобы драйвер считался родным для Win2000, он должен как минимум поддерживать управление питанием, а для этого он должен быть уровневым). Большинство драйверов, которые в NT4 мы считали монолитными, в Win2000 будут по своей сути уровневыми.

Будем выделять следующие типы драйверов:

1) драйверы, представляющие некоторый уровень в многоуровневой архитектуре. Далее именно эти драйверы мы будем называть **уровневыми** драйверами;
2) драйверы-фильтры;
3) драйверы файловой системы (File System Driver, FSD);
4) мини-драйверы (mini-driver).

Для каждого типа драйверов существует свой протокол реализации многоуровневой структуры. Мы рассмотрим только уровневые драйверы и драйверы-фильтры.

2.4.3.1. Реализация уровневых драйверов

Стек драйверов обычно создается самими драйверами. Корректное создание стека зависит от правильной последовательности и момента загрузки каждого драйвера из стека. Первыми должны грузиться драйвера самого нижнего уровня и т.д.

При рассмотрении организации стека драйверов необходимо понимание трех моментов:

- объединение драйверов в стек;
- обработка запросов IRP стеком;
- освобождение драйверов стека.

2.4.3.1.1. Объединение драйверов в стек

Для объединения драйверов в стек обычно используется функция IoGetDeviceObject Pointer(). Функция вызывается драйвером вышележащего уровня для получения указателя на объект-устройство драйвера нижележащего уровня по его имени.

Функция имеет следующий прототип:

```
NTSTATUS  IoGetDeviceObjectPointer(
  IN PUNICODE_STRING ObjectName,
  IN ACCESS_MASK DesiredAccess,
  OUT PFILE_OBJECT FileObject,
  OUT PDEVICE_OBJECT DeviceObjct);
```

Где:

ObjectName – Имя требуемого Объекта-устройства;

DesiredAccess – Требуемый доступ к указанному Объекту-устройству;

FileObject – Указатель на Объект-файл, который будет использоваться для обращения к устройству;

DeviceObject – Указатель на Объект-устройство с именем ObjectName.

Функция IoGetDeviceObjectPointer() принимает имя Объекта-устройства, и возвращает указатель на Объект-устройство с этим именем. Функция работает, посылая запрос CREATE на названное устройство. Этот запрос будет неудачным, если никакого устройства по имени *ObjectName* не существует, или вызывающая программа не может предоставить доступ, указанный в параметре DesiredAccess. Если запрос CREATE успешен, создается Объект-файл, что увеличивает счетчик ссылок Объекта-устройства, с которым связан Объект-файл. Затем Диспетчер ввода/вывода искусственно увеличивает счетчик ссылок на Объект-файл на единицу, и посылает на устройство запрос CLOSE. В результате всего этого процесса, Объект-устройство (чей указатель возвращен в DeviceObject) не может быть удален, пока не обнулится счетчик ссылок соответствующего ему Объекта-файла. Таким образом, Объект-устройство нижнего уровня не может быть удален, в то время как драйвер вышележащего уровня имеет указатель на него.

Выделим из всего вышесказанного, что функция предусматривает в качестве выходного параметра указатель на Объект-файл специально для того, чтобы при выгрузке стека драйверов освободить устройство нижележащего уровня. Это должно быть сделано в функции Unload драйвера вышележащего уровня с помощью вызова функции ObDereferenceObject(*FileObject*).

После получения указателя на объект-устройство драйвера нижележащего уровня, драйвер вышележащего уровня должен установить корректное значение полей Characteristics, StackSize, Flags и AlignmentRequirement своего объекта-устройства. Поля Characteristics, Flags и AlignmentRequirement объектов-устройств всех драйверов в стеке должны совпадать, а значение поля StackSize вышележащего устройства должно быть на 1 больше значения этого поля у нижележащего устройства.

2.4.3.1.2. Обработка запросов IRP стеком драйверов

Запрос ввода/вывода приходит в виде пакета IRP самому верхнему драйверу в стеке драйверов (драйверу верхнего уровня). При этом возможны следующие варианты обработки IRP:

1) Обработка IRP полностью в драйвере верхнего уровня.
2) После выполнения своей части обработки IRP драйвер отправляет первоначальный пакет IRP драйверу нижележащего уровня.
3) После выполнения своей части обработки IRP драйвер создает один или несколько новых пакетов IRP и отправляет их драйверу нижележащего уровня.
4) После выполнения своей части обработки IRP драйвер создает один или несколько новых пакетов IRP, ассоциированных с первоначальным пакетом IRP, и отправляет их драйверу нижележащего уровня.

Все варианты, кроме последнего, возможны при обработке пакета IRP драйвером любого уровня. Последний вариант – создание ассоциированных пакетов IRP – возможен только при обработке IRP драйвером верхнего уровня.

Самостоятельная обработка IRP драйвером. Если драйвер может завершить обработку пакета IRP самостоятельно, он так и должен сделать. При этом обработка может быть завершена либо сразу при поступлении запроса ввода/вывода, либо после постановки запроса ввода/вывода в очередь.

Примером немедленного завершения обработки пакета IRP драйвером может служить случай обнаружения некорректного параметра в IRP.

Передача первоначального пакета IRP драйверу нижележащего уровня. Если драйвер не может самостоятельно обработать пакет IRP, он может передать его нижележащему драйверу. Для этого необходимо заполнить параметры в IRP в стеке размещения ввода/вывода, относящегося к нижележащему драйверу. Указатель на стек размещения ввода/вывода нижележащего драйвера возвращается функцией IoGetNext IrpStackLocation(). После заполнения необходимых параметров IRP передается драйверу нижележащего уровня с помощью вызова функции IoCallDriver(). Прототип этой функции:

```
NTSTATUS IoCallDriver (IN PDEVICE_OBJECT DeviceObject, IN OUT PIRP Irp);
```

Где:

DeviceObject – указатель на объект-устройство, принадлежащий драйверу нижележащего уровня в стеке драйверов, которому должен быть послан запрос;

IRP – указатель на IRP, который будет послан драйверу нижележащего уровня.

При вызове функции IoCallDriver() Диспетчер ввода/вывода напрямую вызывает соответствующую диспетчерскую функцию требуемого драйвера. Это означает, что IoCallDriver() завершится только после завершения соответствующей диспетчерской функции.

Создание новых пакетов IRP для передачи драйверу нижележащего уровня. Вариантом передачи IRP драйверу нижележащего уровня является создание одного или нескольких новых пакетов IRP и передача этих IRP драйверу нижележащего уров-

ня. Пакет IRP может быть создан драйвером различными способами, наиболее общим из которых является вызов функции IoAllocateIrp().

```
PIRP IoAllocateIrp (IN CCHAR StackSize, IN BOOLEAN ChargeQuota);
```

Где: *StackSize* – Число Стеков размещения Ввода/вывода, требуемых в IRP;

ChargeQuota – Указывает, должна ли квота текущего процесса быть загружена.

Функция IoAllocateIrp() возвращает частично инициализированный пакет IRP. По умолчанию, IoAllocateIrp() предполагает, что вызванный драйвер не хочет иметь собственный Стек размещения Ввода/вывода. Драйвер, распределяющий IRP, может факультативно просить IoAllocateIrp() создать IRP с достаточным количеством Стеков Размещения Ввода/вывода, которые он мог иметь один. В этом случае, драйвер должен вызвать IoSetNextIrpStackLocation() (макрокоманда в NTDDK.H), чтобы установить Стек Размещения. Драйвер может затем вызвать функцию IoGetCurrentIrpStack Location().

Причина, по которой драйверу может потребоваться свой собственный Стек Размещения Ввода /вывода состоит в том, что драйвер может использовать стек для передачи информации к подпрограмме завершения. Подпрограмма Завершения (Completion Routine) обсуждается позже в этом разделе.

Как только IRP создан, драйвер может вызывать IoGetNextIrpStackLocation(), чтобы получить указатель на Стек размещения для драйвера нижележащего уровня в стеке драйверов. Затем драйвер заполняет параметры для Стека Ввода/вывода. Если для запроса требуется буфер данных, драйвер должен или установить MDL или SystemBuffer, как того требует используемый нижележащим драйвером способ передачи буферов в пакете IRP. IRP посылается нижележащему драйверу с помощью функции IoCallDriver(), как описано выше.

Создание новых ассоциированных пакетов IRP для передачи драйверу нижележащего уровня. Немного другой подход к созданию нового пакета IRP для передачи драйверу нижележащего уровня состоит в создании **ассоциированного** пакета IRP с помощью функции IoMsakeAssociatedIrp().

```
PIRP IoMakeAssociatedIrp (IN PIRP MasterIrp, IN CCHAR StackSize);
```

Где: *StackSize* – Число Стеков размещения Ввода/вывода, требуемых в IRP;

MasterIrp – Указатель на IRP, с которым должен быть связан создаваемый пакет IRP.

IoMakeAssociatedIrp() позволяет создавать IRP, которые "связаны» с некоторым "главным" IRP. Драйвер, который вызывает IoMakeAssociatedIrp(), должен вручную инициализировать поле **Irp.IrpCount** главного IRP счетчиком ассоциированных с ним IRP, которые созданы до вызова IoMakeAssociatedIrp. Ассоциированные IRP являются особым видом пакетов IRP, при завершении которых значение поля IrpCount в главном IRP уменьшается. Когда значение поля IrpCount в главном IRP становится равным нулю, Диспетчер ввода/вывода автоматически завершает главный IRP.

Ассоциированные пакеты IRP могут создаваться только драйвером высшего уровня. Драйвер высшего уровня, использующий ассоциированные пакеты IRP, может вернуть управление диспетчеру ввода/вывода после вызова IoCallDriver() для каж-

дого из ассоциированных IRP и вызова IoMarkIrpPending() для главного IRP. Если такой драйвер устанавливает процедуру завершения для созданного им ассоциированного IRP (с помощью вызова IoSetCompletion Routine(), описанного ниже), Диспетчер ввода/вывода не завершит автоматически главный IRP. В этом случае процедура завершения должна напрямую завершить главный IRP посредством вызова IoCompleteRequest().

2.4.3.1.2.1. Получение драйвером вышележащего уровня уведомления о завершении обработки IRP драйвером нижележащего уровня

В некоторых случаях драйвер вышележащего уровня может захотеть получить уведомление о завершении обработки переданного им запроса ввода/вывода драйвером нижележащего уровня(Completion Notification). Это может быть сделано с помощью вызова функции IoSetCompletionRoutine() перед передачей пакета IRP драйверу нижележащего уровня любым указанным выше способом.

```
VOID IoSetCompletionRoutine(IN PIRP Irp,
IN PIO_COMPLETION_ROUTINE CompletionRoutine,
INPVOID Context,
IN BOOLEAN InvokeOnSuccess,
IN BOOLEAN InvokeOnError,
IN BOOLEAN InvokeOnCancel);
```

Где: *Irp* – Указатель на IRP, при завершении которого вызывается точка входа CompletionRoutine;

CompletionRoutine – Указатель на точку входа драйвера, вызываемую при завершении IRP;

Context – определенное драйвером значение, которое нужно передать как третий параметр для точки входа *CompletionRoutine;*

InvokeOnSuccess, *InvokeOnError*, *IwokeOnCancel* – Параметры, которые, указывают должна ли точка входа CompletionRoutine быть вызвана при завершении IRP с указанным состоянием.

В качестве второго параметра в вызове IoSetCompletionRoutine() передается адрес точки входа драйвера, которая должна быть вызвана при завершении указанного в первом параметре пакета IRP. Прототип функции – точки входа драйвера:

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

Где: *DeviceObject* – Указатель на объект-устройство, которому предназначался закончившийся пакет IRP;

IRP – Указатель на закончившийся пакет IRP;

Context – Определенное драйвером значение контекста, переданное, когда была вызвана функция IoSetCompletionRoutine().

При вызове IoSetCompletionRoutine() указатель на функцию завершения сохраняется в IRP в следующем после текущего Стеке Размещения ввода/вывода (то есть в

Стеке Размещения ввода/вывода нижележащего драйвера). Из этого следуют два важных вывода:

- Если драйвер установит функцию завершения для некоторого IRP и завершит этот IRP, функция завершения не будет вызвана.
- Драйвер низшего уровня (либо монолитный драйвер) не может устанавливать функции завершения, так как, во-первых, это бессмысленно (см. предыдущий вывод), а во-вторых, это ошибка, так как следующего (за текущнм) стека размещения ввода/вывода для таких драйверов не существует.

Функция завершения вызывается при том же уровне IRQL, при котором нижележащим драйвером была вызвана функция завершения обработки IRP – IoComplete Request(). Это может быть любой уровень IRQL меньший либо равный IRQL_ DISPATCH_LEVEL.

Если драйвер вышележащего уровня создавал новые пакеты IRP для передачи драйверу нижележащего уровня, он обязан использовать функцию завершения для этих IRP, причем параметры *InvokeOnSuccess*, *InvokeOnError*, *IwokeOnCancel* должны быть установлены в TRUE. В этих случаях функция завершения должна освободить созданные драйвером IRP с помощью функции IoFreeIrp() и завершить первоначальный пакет IRP.

Требования к реализации функции завершения достаточно сложные. Эти требования можно найти в [Developing Windows NT Device Drivers, pages 481-485].

2.4.3.2. Реализация драйверов-фильтров

Диспетчер ввода/вывода предоставляет возможность одному драйверу «подключить» один из своих объектов-устройств к объекту-устройству другого драйвера. Результатом будет перенаправление пакетов IRP, предназначавшихся некоторому объекту-устройству, на объект-устройство драйвера-фильтра. Драйвер-фильтр может просмотреть, модифицировать, завершить или передать полученный IRP первоначальному драйверу. Драйверы-фильтры могут быть подключены к любому драйверу в стеке драйверов.

2.4.3.2.1. Подключение фильтра к устройству

Имеется пара различных способов подключения драйвера-фильтра к другому драйверу. Первый способ – вначале получить указатель на объект-устройство, к которому необходимо подключиться, с помощью функции IoGetDeviceObjectPointer() (см. раздел «Объединение драйверов в стек и освобождение драйверов стека»). Подключение драйвера-фильтра к найденному устройству производится с помощью вызова функции IoAttachDeviceToDeviceStack().

```
PDEVICE_OBJECT IoAttachDeviceToDeviceStack (
IN PDEVICE_OBJECT SourceDevice, IN PDEVICE_OBJECT TargetDevice);
```

Где: *SourceDevice* – Указатель на первоначальный Объект-устройство, к которому будем прикреплять фильтр;

TargetDevice – Указатель на Объект-устройство фильтра.

Каждый объект-устройство имеет поле AttachedDevice, которое указывает на объект-устройство первого драйвера-фильтра, который был прикреплен к этому объекту-устройству. Если поле AttachedDevice объекта-устройства нулевое, нет никаких прикрепленных устройств. Если поле AttachedDevice не нулевое, оно указывает на объект-устройство драйвера-фильтра. IoAttachDeviceToDeviceStack() находит конец списка AttachedDevice для объекта-устройства, указанного параметром TargetDevice, и устанавливает в поле AttachedDevice этого конечного объекта-устройства указатель на объект-устройство драйвера-фильтра.

Возвращаемым значением функции IoAttachDeviceToDeviceStack() является указатель на объект-устройство, к которому был прикреплен объект-устройство драйвера фильтра. Драйвер-фильтр может использовать этот указатель, чтобы передавать запросы первоначальному устройству. Хотя этот указатель обычно указывает на то же устройство, что и SourceDevice, он может быть указателем на другой драйвер-фильтр, прикрепленный к устройству SourceDevice.

Другим способом прикрепления фильтра к устройству является вызов функции IoAttachDevice(). Эта функция просто объединяет функциональные возможности, обеспечиваемые функциями IoGetDeviceObjectPointer() и IoAttachDeviceToDeviceStack().

```
NTSTATUS IoAttachDevice (IN PDEVICE_OBJECT SourceDevice,
IN PUNICODE_STRING TargetDevice,
OUT PDEVICE_OBJECT *AttachedDevice);
```

Где: *SourceDevice* – Указатель на Объект-устройство драйвера Фильтра;

TargetDevice – строка Unicode с именем устройства, к которому будет прикреплено устройство *SourceDevice*;

AttachedDevice – Указатель на объект-устройство, к которому было прикреплено устройство *SourceDevice*.

Эффект подсоединения фильтра заключается в том, что при запросе на открытие некоторого устройства (например, через CreateFile()) для этого устройства просматривается список присоединенных устройств. Если он пуст, как обычно, открывается запрошенное устройство. Если он не пуст, открывается последнее устройство в списке. После этого все запросы ввода/вывода направляются именно этому устройству.

Из сказанного следует, что для успешного подключения фильтра к некоторому устройству в стеке, подключение нужно делать до того момента, когда вышележащий драйвер вызовет функцию IoGetDeviceObjectPointer(), и, тем самым, пошлет запрос на открытие нижележащего устройства.

Выгрузка драйвера-фильтра. Для отсоединения подсоединенного устройства служит функция IoDetachDevice.

```
VOID IoDetachDevice(IN OUT PDEVICE_OBJECT TargetDevice);
```

Соответственно, функция выгрузки драйвера-фильтра DriverUnload должна делать примерно следующее:

- С помощью вызова ObDereferenceObject() уменьшить счетчик ссылок на объект-файл, полученный в результате вызова IoGetDeviceObjectPointer().
- Для каждого объекта-устройства, принадлежащего выгружаемому драйверу-фильтру, вызвать IoDetachDevice() и IoDeleteDevice().

2.4.4. Сериализация

Сериализация – это процесс выполнения различных операций в нужной последовательности.

Функция драйвера для обработки запроса ввода/вывода может быть вызвана из различных потоков различных процессов. Неэксклюзивное устройство может быть одновременно открыто несколькими прикладными программами, каждая из которых может послать один и тот же запрос в одно и то же время. Нам необходимо, чтобы обработка каждого такого запроса началась как можно быстрее, но не раньше, чем завершится обработка предыдущего запроса.

Если устройство эксклюзивное и вся обработка завершается в диспетчерской функции, сериализация обеспечивается операционной системой. Диспетчерские функции работают в контексте памяти потока-инициатора запроса ввода/вывода, следовательно, пока не завершится диспетчерская функция, код прикладной программы не получит управления и не инициирует очередной запрос ввода/вывода.

Однако даже эксклюзивное устройство может не суметь обработать запрос ввода/вывода в диспетчерской функции сразу при его поступлении. Если драйвер будет ждать в диспетчерской функции возможности обработать и завершить запрос, прикладная программа – инициатор запроса будет «висеть» – ее код не выполняется. Более того, если такой драйвер является уровневым и не является драйвером верхнего уровня, его диспетчерские функции будут вызываться в случайном контексте потока. «Завесив» свою диспетчерскую функцию, драйвер «завесит» произвольный поток произвольной прикладной программы! Чтобы такого не происходило, диспетчерская функция должна завершиться как можно скорее. При этом диспетчеру ввода/вывода необходимо указать, что соответствующий запрос ввода/вывода не был завершен, однако это не ошибка, и этот запрос ввода/вывода будет завершен когда-то позже. При этом возникают вопросы:

- как указать, что запрос ввода/вывода не завершен;
- где хранить незавершенные пакеты ввода/вывода;
- как быть, если запрос ввода/вывода не завершен, а инициировавшее запрос приложение закрылось.

Завершение запроса ввода/вывода. Успешное завершение запроса ввода/вывода в диспетчерской функции показано в следующем листинге:

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

При неудачном завершении обработки запроса ввода/вывода в поле Irp->IoStatus.Status устанавливается код ошибки (значение, не равное STATUS_SUCCESS), оно же возвращается оператором return.

2.4.4.1. Задержка обработки запросов IRP и постановка запросов IRP в очередь

Когда немедленное завершение запроса ввода/вывода в диспетчерской функции невозможно, драйвер должен указать Диспетчеру ввода/вывода, что обработка запроса продолжается. Для этого возвращаемым значением диспетчерской функции должно быть значение STATUS_PENDING. Перед этим в самом пакете IRP в текущем стеке размещения ввода/вывода должен быть установлен флаг SL_PENDING_RETURNED, помечающий запрос как неоконченный. Для этого служит функция IoMarkIrpPending().

```
VOID IoMarkIrpPending(IN PIRP Irp);
```

Установка этого флага указывает, что IRP будет освобожден драйвером с помощью вызова IoCompleteRequest() когда-то потом.

Для постановки пакета IRP в очередь диспетчерская функция должна:

1) пометить IRP как неоконченный;
2) установить функцию отмены IRP;
3) использовать один из нижеприведенных способов для постановки IRP в очередь;
4) завершиться со статусом STATUS_PENDING.

NT предусматривает два способа организации очередей пакетов IRP:

- Использование диспетчера ввода/вывода для постановки запросов в управляемую им очередь (Системная Очередь).
- Создание и управление очередью самостоятельно (Очереди, управляемые драйвером).

2.4.4.1.1. Системная очередь запросов IRP (System Queuing)

Простейший способ, с помощью которого драйвер может организовать очередь IRP – использовать Системную Очередь. Для этого драйвер предоставляет диспетчерскую точку входа StartIo (DriverObject->DriverStartIo). При получении пакета IRP, который необходимо поставить в Системную Очередь, такой пакет необходимо пометить как отложенный с помощью вызова IoMarkIrpPending(), а затем вызвать функцию IoStartPacket().

```
VOID IoStartPacket(IN PDEVICE_OBJECT DeviceObject,
  IN PIRP Irp,
  IN PULONG Key,
  IN PDRIVER_CANCEL CancelFunction);
```

Где: *DeviceObject* – Указатель на устройство, которому направлен запрос ввода/вывода;

Irp – Указатель на пакет IRP, описывающий запрос ввода/вывода;

Key – Необязательный указатель на значение, определяющее позицию в очереди IRP, в которую будет вставлен пакет IRP. Если указатель NULL, IRP помещается в конец очереди;

CancelFunction – Необязательный указатель на функцию – точку входа драйвера, которая будет вызвана при отмене данного запроса ввода/вывода диспетчером ввода/вывода.

Вызов этой функции ставит пакет IRP для данного устройства в Системную Очередь. При этом, если в момент вызова IoStartPacket() функция StartIo не выполняется, происходит выборка из очереди очередного IRP.

При выборке очередного пакета IRP из очереди, если очередь не пуста, для очередного IRP будет вызвана функция StartIo. Функция имеет такой же прототип, как и все диспетчерские функции, но вызывается в случайном контексте потока на уровне IRQL DISPATCH_LEVEL:

```
NTSTATUS StartIo(IN PDEVICE_OBJECT DeviceObject, IN PIRP Irp).
```

Структура функции StartIo практически такая же, как у диспетчерской функции, не использующей организацию очередей, за двумя важными исключениями:

- StartIo вызывается в случайном контексте потока на уровне IRQL DISPATCH_LEVEL. Поэтому необходимо: а) соблюдать все ограничения для уровня IRQL DISPATCH_LEVEL; б) не использовать метод передачи буфера Neither, так как такой метод передачи предполагает знание контекста памяти процесса, из которого был инициирован запрос ввода/вывода (см. раздел «Описание Буфера Данных»).
- Какой бы ни был результат обработки пакета IRP, после завершения его обработки StartIo обязана вызвать функцию IoStartNextPacket() для указания диспетчеру ввода/вывода необходимость выбора из системной очереди очередного пакета IRP (то есть вызвать для него StartIo) сразу после завершения StartIo для текущего пакета.

Прототип функции IoStartNextPacket():

```
VOID IoStartNextPacket(IN PDEVICE_OBJECT DeviceObject,
IN BOOLEAN Cancelable);
```

Где: *DeviceObject* – Указатель на устройство, для которого необходимо выбрать следующий пакет из системной очереди IRP;

Cancelable – Указывает, может ли выбираемый из очереди пакет IRP быть отменен в процессе обработки. Если при вызове IoStartPacket() была указана ненулевая функция отмены, параметр *Cancelable* обязан быть равным TRUE.

Вместо IoStartNextPacket() может быть использована функция IoStartNext PacketByKey():

```
VOID IoStartNextPacketByKey(IN PDEVICE_OBJECT DeviceObject,
IN BOOLEAN Cancelable, IN ULONG Key);
```

В этом случае из очереди будет выбран очередной пакет IRP с заданным значением *Key* (это значение могло быть установлено при помещении пакета в очередь при вызове IoStartPacket()).

При использовании системной очереди диспетчер ввода/вывода отслеживает, когда данный объект-устройство свободен или занят обработкой очередного IRP. Это осуществляется с помощью специального объекта ядра Очередь Устройства (Device Queue Object, тип – структура KDEVICE_QUEUE), помещенного внутрь объекта-устройства в поле DeviceQueue. Поле DeviceQueue.Busy устанавливается диспетчером ввода/вывода равным TRUE непосредственно перед вызовом функции StartIo, и FALSE, если вызвана функция IoStartNextPacket(ByKey)(), а системная очередь не содержит пакетов IRP для данного устройства. При обработке очередного IRP указатель на него находится в объекте-устройстве в поле CurrentIrp.

2.4.4.1.1.1. Обработка пакетов IRP в функции StartIo

Как должна происходить обработка пакетов из системной очереди? Как уже говорилось, StartIo работает на уровне IRQL DISPATCH_LEVEL. Пока выполняется эта функция, ни один поток с более низким значением IRQL не может получить управление (если в системе один процессор). Следовательно, новые запросы ввода/вывода от прикладных программ попасть в очередь не могут (потоки не выполняются). Если завершение очередного пакета ввода/вывода **всегда** происходит в функции StartIo, системная очередь **всегда** содержит не более одного пакета ввода/вывода. Если пакет ввода/вывода не может быть обработан в тот момент, когда он попал в функцию StartIo, функция просто должна завершиться, не завершая запрос ввода/вывода и не вызывая IoStartNextPacket(). В этом случае устройство остается «занятым». Поле **pDeviceObject->DeviceQueue.Busy** все еще TRUE, а в поле **pDeviceObject->CurrentIrp** находится указатель на этот пакет IRP. Такой пакет может быть обработан, например, при поступлении прерывания от аппаратного устройства (или при возникновении другого ожидаемого события). Функция, которая завершит обработку такого пакета, обязана вызвать IoStartNextPacket(), чтобы инициировать выборку очередного пакета из системной очереди. Заметим, что пока устройство остается «занятым», функция StartIo для обработки пакетов из системной очереди не может быть вызвана.

Несмотря на простоту использования системной очереди, имеется существенное ограничение. Оно состоит в том, что очередь одна на все типы запросов ввода/вывода (чтение, запись, управление устройством). В каждый конкретный момент обрабатывается только какой-то один пакет IRP.

Могут быть ситуации, когда такое ограничение неприемлемо. Классическим примером является драйвер полнодуплексного устройства, которое одновременно позволяет как отправлять, так и получать данные. В этом случае необходимо начать обработку следующего запроса чтения при завершении текущего запроса чтения, и следующего запроса записи при завершении текущего запроса записи. При этом важно понимать, что в этом случае одновременно (то есть в контекстах разных потоков) могут выполняться один запрос чтения и один запрос записи. Необходимы две очереди: одна – для запросов чтения, другая – для запросов записи.

2.4.4.1.2. Очереди, управляемые драйвером

Вместо использования системной очереди, можно предусмотреть свой собственный механизм организации очереди. Это можно сделать двумя способами:

- использовать для создания очереди функции управления очередью низкого уровня;
- использовать функции управления очередью высокого уровня. Этот способ очень редко используется и является промежуточным между использованием системной очереди и функций управления очередью низкого уровня.

Используя очереди, управляемые драйвером, драйвер получает полный контроль над очередью. Например, драйвер может организовать одновременную обработку трех запросов записи и двух запросов чтения при условии, что в данный момент не выполняется запрос управления устройством.

Как и в случае использования системной очереди, при получении пакета IRP, который необходимо поставить в очередь, такой пакет необходимо пометить как отложенный с помощью вызова IoMarkIrpPending(). Затем используется любой способ помещения указателя на пакет IRP в очередь.

2.4.4.1.2.1. Функции управления очередью низкого уровня

Для организации очереди с помощью функций низкого уровня используется стандартная структура LIST_ENTRY.

```
typedef struct _LIST_ENTRY {
struct _LIST_ENTRY *volatile Flink; // Указатель
                                    // на следующий элемент списка
struct _LIST_ENTRY *volatile Blink; // Указатель
                                    // на предыдущий элемент списка
} LIST_ENTRY, *PLIST_ENTRY;
```

Очередь, как это видно из определения структуры, двунаправленная.

В структуре DeviceExtension обычно создается экземпляр структуры LIST_ENTRY, представляющей голову очереди, который затем инициализируется с помощью функции InitializeListHead(). После этого можно добавлять или удалять записи в очередь. Для этого используются функции InsertHeadList(), InsertTailList(), RemoveHeadList(), RemoveTailList(), RemoveEntryList().

Пакеты IRP добавляются в очередь в диспетчерской функции, скорее всего работающей на уровне IRQL равном PASSIVE_LEVEL. При этом выниматься из очереди они могут функциями, работающими на любом уровне IRQL, причем функции, выбирающие IRP из очереди, могут вытеснять функции, помещающие IRP в очередь. Возникает проблема синхронизации. Если ее не решить, незаконченная операция помещения IRP в очередь может быть прервана операцией выборки IRP из очереди, что приведет к появлению синего экрана.

Синхронизация доступа к разделяемому ресурсу производится с помощью спинблокировки (механизмы синхронизации будут рассмотрены в следующем разделе). Поскольку операции добавления и удаления записей в очередь на уровнях IRQL PASSIVE_LEVEL и DISPATCH_LEVEL очень распространены, для их безопасного

осуществления предусмотрена специальная функция: ExInterlocked...List(). Для использования этой функции должна быть создана и инициализирована спин-блокировка. Создается она обычно там же, где и голова очереди (обычно в DeviceExtension), и инициализируется после инициализации головы очереди. Например:

```
typedef struct _DEVICE_EXTENSION
{
...
LIST_ENTRY ListHead;
KSPIN_LOCK ListLock;
...
}DEVICE_EXTENSION, *PDEVICE_EXTENSION;
//В функции DriverEntry
InitializeListHead(&(pDeviceExtension->ListHead));
KeInitializeSpinLock(&(pDeviceExtension->ListLock));

//после этого можно добавлять и удалять записи
PLIST_ENTRY pOldHead = ExInterlockedInsertHeadList(
        &(pDeviceExtension->ListHead),
        pNewListEntry,
        &(pDeviceExtension->ListLock));
```

Как видно из определения структуры LIST_ENTRY, она не содержит полей для хранения собственно данных (например, указателя на пакет IRP). Поэтому распространенный способ использования структуры LIST_ENTRY – включение ее экземпляра в состав более общей структуры.

Для организации очереди пакетов IRP, в каждом пакете IRP в поле **Tail.Overlay.ListEntry** содержится экземпляр структуры LIST_ENTRY. При этом встает вопрос, как, зная указатель на структуру LIST_ENTRY, получить указатель на структуру IRP, в состав которой входит LIST_ENTRY. Для этого DDK предоставляет специальный макрос CONTAINING_RECORD:

```
#define CONTAINING_RECORD (address, type, field) \
((type *)((PCHAR)(address) - (ULONG_PTR)(&((type *)0)->field)))
```

Где: *Address* – Известный адрес некоторого поля структуры, адрес которой необходимо получить;

Type – Тип структуры, адрес которой необходимо получить;

Field – Имя поля внутри искомой структуры, адрес этого поля передан в параметре *address*.

Применительно к IRP, мы должны будем написать что-то вроде:

```
PListEntry = ExInterlockedRemoveHeadList(
     &(pDeviceExtension->ListHead),
     &(pDeviceExtension->ListLock));
pIrp = CONTAINING_RECORD(pListEntry, IRP, Tail.Overlay.ListEntry);
//далее - обработка IRP
```

Организация очереди пакетов IRP показана на рис. 12.

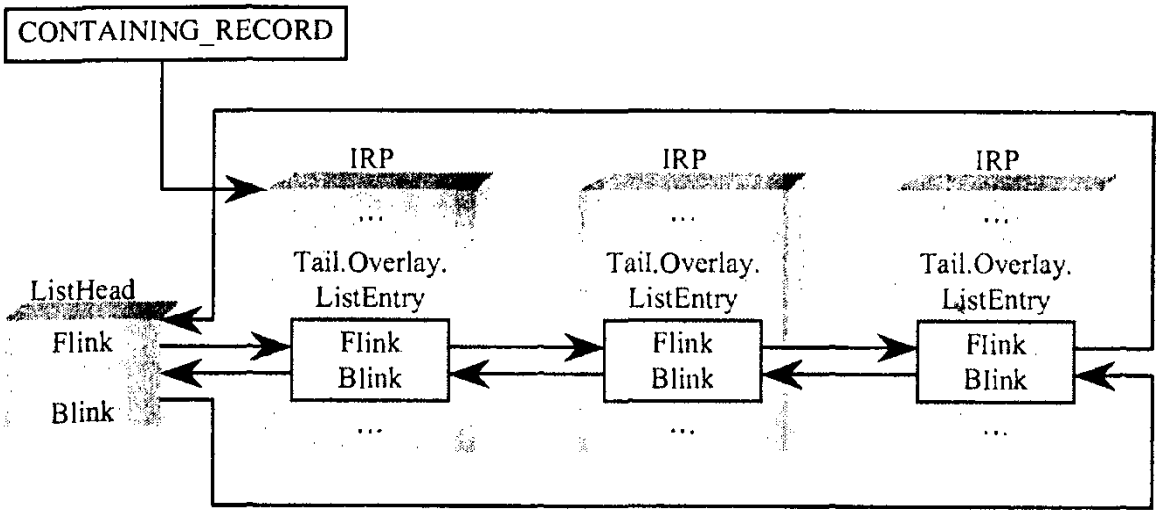


Рис. 12

2.4.4.1.2.2. Функции управления очередью высокого уровня – «Очередь Устройства» (Device Queue)

Драйвер создает дополнительные Очереди Устройства с помощью выделения памяти из невыгружаемой памяти под дополнительные объекты-Очереди Устройства (KDEVICE_QUEUE) и инициализирует эти объекты с помощью функции KeInitializeDeviceQueue(). Добавление пакетов IRP в эти очереди производится с помощью функции KeInsertDeviceQueue() или KeInsertByKeyDeviceQueue(), а выборка пакетов из очереди – KeRemoveDeviceQueue(), KeRemoveByKeyDeviceQueue() или KeRemoveEntryDeviceQueue().

Для организации очереди пакетов IRP используется структура типа KDEVICE_QUEUE_ENTRY, указатель на которую содержится в пакете IRP в поле Tail.Overlay.DeviceQueueEntry.

2.4.4.2. Отмена запросов ввода/вывода

Всякий раз, когда запрос ввода/вывода удерживается драйвером в течение продолжительного отрезка времени, драйвер должен быть готов к отмене данного запроса. В случае закрытия потока диспетчер ввода/вывода пытается отменить все запросы ввода/вывода, отправленные этим потоком и еще не завершенные. Пока все такие запросы ввода/вывода не будут завершены устройством, ему не придет запрос IRP_MJ_CLOSE, и, следовательно, не освободится объект-файл, а впоследствии – и само устройство (драйвер никогда не получит запрос DriverUnload).

Для обеспечения отмены запроса ввода/вывода в пакете IRP, представляющем такой запрос, должен быть указан адрес диспетчерской точки входа драйвера, собственно отменяющей запрос. Для этого служит функция IoSetCancelRoutine().

```
PDRIVER_CANCEL IoSetCancelRoutine(IN PIRP Irp,
PDRIVER_CANCEL CancelRoutine);
```

Где функция CancelRoutine() имеет такой же прототип, как и все диспетчерские функции.

```
VOID CancelRoutine(IN PDEVICE_OBJECT DeviceObject,
IN PIRP Irp);
```

Она вызывается на уровне IRQL DISPATCH_LEVEL в случайном контексте потока, однако перед ее вызовом происходит захват специальной системной спин-блокировки. До тех пор, пока системная спин-блокировка не будет освобождена, функция CancelRoutine() работает на уровне IRQL DISPATCH_LEVEL. Уровень IRQL, на который нужно перейти после освобождения блокировки указывается при вызове IoReleaseCancelSpinLock() (см. ниже). Принцип реализации функции следующий:

- Если указанный пакет IRP не может быть отменен или не принадлежит драйверу, то надо освободить системную спин-блокировку и завершить работу функции.
- В противном случае:
  1) удалить пакет IRP из любых очередей, в которых он присутствует;
  2) установить функцию отмены IRP в NULL с помощью IoSetCancelRoutine();
  3) освободить системную спин-блокировку;
  4) установить в IRP поле IoStatus.Information равном 0, а поле IoStatus.Status равном STATUS_CANCELLED;
  5) завершить IRP с помощью IoCompleteRequest().

Системная спин-блокировка отмены IRP освобождается с помощью IoRelease CancelSpinLock():

```
VOID IoReleaseCancelSpinLock(IN KIRQL Irql);
```

Где: *Irql* – уровень IRQL, на который система должна вернуться после освобождения спин-блокировки. Это значение хранится в IRP в поле CancelIrql.

2.4.4.2.1. Отмена IRP и Системная Очередь

Пример функции отмены IRP драйвера, использующего системную очередь, показан в следующем листинге. Необходимо отметить, что для удаления IRP из системной очереди используется функция KeRemoveEntryDeviceQueue() так, как это показано в листинге.

```
VOID Cancel(IN PDEVICE_OBJECT DeviceObject, IN PIRP Irp)
{
    // Обрабатывается ли отменяемый запрос в данный момент?
    if (Irp == DeviceObject->CurrentIrp)
    {
    // Да. Освободить системную спин-блокировку и указать
    // диспетчеру ввода/вывода начать обработку следующего
    // пакета. Отмена IRP - в конце функции
        IoReleaseCancelSpinLock(Irp->CancelIrql);
```

```
        IoStartNextPacket(DeviceObject, TRUE);
    }
    else
    {
    // Нет. Отменяемый IRP находится в очереди.
    // Удалить его из очереди
        KeRemoveEntryDeviceQueue(&DeviceObject->DeviceQueue,
            &Irp->Tail.Overlay.DeviceQueueEntry);
        IoReleaseCancelSpinLock(Irp->CancelIrql);
    }
    // Отменить IRP
    Irp->IoStatus.Status = STATUS_CANCELLED;
    Irp->IoStatus.Information = 0;
    IoCompleteRequest(Irp, IO_NO_INCREMENT);
    return;
}
```

2.4.4.2.2. Отмена IRP и очереди, управляемые драйвером

```
VOID Cancel(IN PDEVICE_OBJECT DeviceObject, IN PIRP Irp)
{
    PIRP irpToCancel;
    PDEVICE_EXT devExt;
    KIRQL oldIrql;

    // обнулить указатель на функцию отмены
    IoSetCancelRoutine(Irp, NULL);
    // Освободить системную спин-блокировку
    // как можно быстрее
    IoReleaseCancelSpinLock(Irp->CancelIrql);
    devExt = DeviceObject->DeviceExtension;
    // Захватить спин-блокировку доступа к очереди,
    // удалить IRP и освободить
    // спин-блокировку
    KeAcquireSpinLock(&devExt->QueueLock, &oldIrql);
    RemoveEntryList(&Irp->Tail.Overlay.ListEntry);
    KeReleaseSpinLock(&devExt->QueueLock, oldIrql);

    // Отменить IRP
    Irp->IoStatus Status = STATUS_CANCELLED;
    Irp->IoStatus.Information = 0;
    IoCompleteRequest(Irp, IO_NO_INCREMENT);
}
```

2.4.5. Механизмы синхронизации

В операционной системе с вытесняющей многозадачностью, да еще поддерживающей несколько процессоров, остро встает задача синхронизации доступа к совместно используемым ресурсам компьютера, будь то аппаратное устройство или структура в памяти.

2.4.5.1. Спин-блокировки

Спин-блокировка – простейший механизм синхронизации. Спин-блокировка может быть **захвачена**, и **освобождена**. Если спин-блокировка была захвачена, последующая попытка захватить спин-блокировку любым потоком приведет к бесконечному циклу с попыткой захвата спин-блокировки (состояние потока *busy-waiting*). Цикл закончится только тогда, когда прежний владелец спин-блокировки освободит ее. Использование спин-блокировок безопасно на мультипроцессорных платформах, то есть гарантируется, что, даже если ее запрашивают одновременно два потока на двух процессорах, захватит ее только один из потоков.

Спин-блокировки предназначены для защиты данных, доступ к которым производится на различных, в том числе повышенных уровнях IRQL. Теперь представим такую ситуацию: код, работающий на уровне IRQL PASSIVE_ LEVEL захватил спин-блокировку для последующего безопасного изменения некоторых данных. После этого код был прерван кодом с более высоким уровнем IRQL DISPATCH_LEVEL, который попытался захватить ту же спин-блокировку, и, как следует из описания спин-блокировки, вошел в бесконечный цикл ожидания освобождения блокировки. Этот цикл никогда не закончится, так как код, который захватил спин-блокировку и должен ее освободить, имеет более низкий уровень IRQL и никогда не получит шанса выполниться! Чтобы такая ситуация не возникла, необходим механизм, не позволяющий коду с некоторым уровнем IRQL прерывать код с более низким уровнем IRQL в тот момент когда код с более низким уровнем IRQL владеет спин-блокировкой. Таким механизмом является повышение текущего уровня IRQL в момент захвата спин-блокировки до некоторого уровня IRQL, ассоциированного со спин-блокировкой, и восстановление старого уровня IRQL в момент ее освобождения. Из сказанного следует, что код, работающий на повышенном уровне IRQL, не имеет права обращаться к ресурсу, защищенному спин-блокировкой, если уровень IRQL спин-блокировки ниже уровня IRQL производящего доступ к ресурсу кода. При попытке таким кодом захватить спин-блокировку его уровень IRQL будет понижен до уровня IRQL спин-блокировки, что приведет к непредсказуемым последствиям.

В NT имеется два вида спин-блокировок:

- Обычные спин-блокировки, особым случаем которых являются спин-блокировки отмены запроса ввода/вывода, используемые при организации очередей запросов ввода/вывода (см. раздел «Отмена запросов ввода/вывода»).
- Спин-блокировки синхронизации прерываний.

С обычными спин-блокировками связан IRQL DISPATCH_LEVEL, то есть:

- все попытки их захвата должны производиться на уровне IRQL, меньшим или равным DISPATCH_LEVEL;
- в случае захвата спин-блокировки текущий уровень IRQL поднимается до уровня DISPATCH_LEVEL.

Со спин-блокировками синхронизации прерываний связан один из уровней DIRQL. Использование обычных спин-блокировок будет описано ниже (за исключением спин-блокировок отмены запросов ввода/вывода, которые были описаны в предыдущем разделе). Использование спин-блокировок синхронизации прерываний будет описано в разделе, посвященном обработке прерываний.

2.4.5.1.1. Использование обычных спин-блокировок

1) VOID KeInitializeSpinLock(IN PKSPIN_LOCK SpinLock); Эта функция инициализирует объект ядра KSPIN_LOCK. Память под спин-блокировку уже должна быть выделена в невыгружаемой памяти.

2) VOID KeAcquireSpinLock(IN PKSPIN_LOCK SpinLock, OUT PKIRQL OldIrql); Эта функция захватывает спин-блокировку. Функция не вернет управление до успеха захвата блокировки. При завершении функции уровень IRQL повышается до уровня DISPATCH_LEVEL. Во втором параметре возвращается уровень IRQL, который был до захвата блокировки (он должен быть <= DISPATCH_LEVEL).

3) VOID KeReleaseSpinLock(IN PKSPIN_LOCK SpinLock, OUT PKIRQL NewIrql); Эта функция освобождает спин-блокировку и устанавливает уровень IRQL в значение параметра NewIrql. Это должно быть то значение, которое вернула функция KeAcquireSpinLock() в параметре OldIrql.

4) VOID KeAcquireLockAtDpcLevel(IN PKSPIN_LOCK SpinLock); Эта оптимизированная функция захватывает спин-блокировку кодом, уже работающем на уровне IRQL DISPATCH_LEVEL. В этом случае изменение уровня IRQL не требуется. На однопроцессорной платформе эта функция вообще ничего не делает, так как синхронизация обеспечивается самой архитектурой IRQL.

5) VOID KeReleaseLockFromDpcLevel(IN PKSPIN_LOCK SpinLock); Эта функция освобождает спин-блокировку кодом, захватившим блокировку с помощью функции KeAcquireLockAtDpcLevel(). На однопроцессорной платформе эта функция ничего не делает.

Пример использования обычных спин-блокировок:

```
typedef struct _DEVICE_EXTENSION
{
   ...
   KSPIN_LOCK spinlock
}DEVICE_EXTENSION, *PDEVICE_EXTENSION;

NTSTATUS DriverEntry(....)
{
```

```
    KeInitializeSpinLock(&extension->spinlock);
}
NTSTATUS DispatchReadWrite( .... )
{
    KIRQL OldIrql;
    ...
    KeAcquireSpinLock(&extension->spinlock, &OldIrql);
    // произвести обработку данных,
    // защищенных спин-блокировкой
    KeReleaseSpinLock(&extension->spinlock, OldIrql);
}
```

2.4.5.1.2. Проблема взаимоблокировок (deadlocks)

Если поток попробует захватить спин-блокировку повторно, он войдет в бесконечный цикл ожидания – «повиснет». Такая же ситуация возникнет, если два потока используют две спин-блокировки. Поток 1 захватывает блокировку 1, одновременно с этим поток 2 захватывает блокировку 2. Затем поток 1 пробует захватить блокировку 2, а поток 2 – блокировку 1. Оба потока «виснут». Эту ситуацию можно распространить на произвольное число потоков, она широко известна и носит название взаимоблокировки (**deadlocks**).

Решение этой проблемы очень простое. Все блокировки, которые могут захватываться одновременно, помещаются в список в порядке убывания частоты использования. При необходимости захвата блокировок они должны быть захвачены в том порядке, в котором они указаны в списке. Таким образом, мы создали **иерархию блокировок**.

2.4.5.2. Диспетчерские объекты

Спин-блокировки абсолютно необходимы в случаях, когда требуется синхронизация кода, работающего на повышенных уровнях IRQL. Но основное правило в NT – работать на повышенных уровнях IRQL в течение как можно более короткого времени значит, использование спин-блокировок следует по возможности избегать. Диспетчерские объекты NT – это набор механизмов синхронизации, рассчитанных на применение в основном для уровня IRQL PASSIVE_LEVEL.

Базой для любого диспетчерского объекта является структура DISPATCHER_HEADER (определена в ntddk.h). Общим свойством любого **диспетчерского объекта** является то, что в каждый момент времени такой объект находится в одном из двух состояний – **сигнальном** или **несигнальном**, а также то, что поток, ожидающий захвата диспетчерского объекта, **блокирован** и помещен в список ожидания, находящийся в структуре DISPATCHER_HEADER.

Блокирование потока означает его особое состояние, при котором он не занимает время процессора. Блокированный поток не будет поставлен планировщиком в оче-

редь на исполнение до тех пор, пока не будет выведен из состояния блокирования. Это фундаментально отличает ожидание освобождения любого диспетчерского объекта от попытки захвата спин-блокировки. В последнем случае, как было сказано, поток, захватывающий спин-блокировку, «крутится» в бесконечном цикле до момента успешного захвата (отсюда и название спин-блокировка – блокировка вращения в бесконечном цикле).

Единственным отличием одного диспетчерского объекта от другого является правило, в соответствии с которым меняется состояние объекта (переход в сигнальное или несигнальное состояние). В таблице 8 перечислены диспетчерские объекты и моменты изменения их состояния.

Таблица 8. Диспетчерские объекты

Тип Объекта	**Переход в сигнальное состояние**	**Результат для ожидающих потоков**
Мьютекс (Mutex)	Освобождение мьютекса	Освобождается один из ожидающих потоков
Семафор (Semaphore)	Счетчик захватов становится ненулевым	Освобождается некоторое число ожидающих потоков
Событие синхронизации (Synchronization events)	Установка события в сигнальное состояние	Освобождается один из ожидающих потоков
Событие оповещения (Notification event)	Установка события в сигнальное состояние	Освобождаются все ожидающие потоки
Таймер синхронизации (Synchronization timer)	Наступило время или истек интервал	Освобождается один из ожидающих потоков
Таймер оповещения (Notification timer)	Наступило время или истек интервал	Освобождаются все ожидающие потоки
Процесс	Завершился последний поток процесса	Освобождаются все ожидающие потоки
Поток	Поток завершился	Освобождаются все ожидающие потоки
Файл	Завершена операция ввода/вывода	Освобождаются все ожидающие потоки

Диспетчерские объекты управляются Диспетчером объектов. Как и все объекты Диспетчера объектов, они могут иметь имена в пространстве имен Диспетчера объектов. С помощью этого имени различные драйвера и прикладные программы могут обращаться к соответствующему объекту. Кроме того, каждый процесс имеет таблицу описателей, связанных с конкретным объектом. Как уже говорилось, описатель в таблице описателей уникален и имеет смысл только в контексте конкретного процесса. Однако Диспетчер объектов предоставляет функ-

цию ObReferenceObjectByHandle(), которая дает возможность получения указателя на объект по его описателю. Эту функцию, как следует из вышесказанного, можно использовать только в контексте известного процесса (для которого создавался описатель), а полученный указатель на объект уже можно использовать в случайном контексте. Чтобы такой объект впоследствии мог быть удален, по окончании его использования должна быть вызвана функция ObDereference Object().

2.4.5.2.1. Ожидание (захват) диспетчерских объектов

Как уже было сказано, каждый диспетчерский объект всегда находится в одном из двух состояний – сигнальном (свободном) или несигнальном (занятым). Термины «свободный» и «занятый» довольно вольные, поэтому лучше использовать термины сигнальный и несигнальный. Для ожидания момента перехода объекта из несигнального в сигнальное состояние служат специальные функции ожидания: KeWaitFor SingleObject() и KeWaitForMultipleObjects(). Важной особенностью этих функций служит то, что в качестве одного из их параметров указывается интервал времени, в течение которого необходимо ждать.

Если указан нулевой интервал времени (но не NULL !!!), вызов функции ожидания не блокирует поток. В этом случае она работает как функция проверки состояния диспетчерского объекта, и может быть вызвана на уровне IRQL меньшем или равном DISPATCH_LEVEL.

Если интервал времени не указан (NULL в качестве параметра), или указан ненулевой интервал времени, функции ожидания можно вызывать на уровне IRQL строго меньшем DISPATCH_LEVEL. В противном случае будет сделана попытка блокирования потока, но, как мы говорили раньше, механизм диспетчеризации на уровнях IRQL меньших или равных DISPATCH_LEVEL не работает. Переключение контекста потока не сможет произойти, и функция ожидания завершит работу, как будто ожидаемый диспетчерский объект находится в сигнальном состоянии.

Отличие функции KeWaitForMultipleObjects() от KeWaitForSingleObject() в том, что она может ожидать перехода в сигнальное состояние сразу всех указанных в ней диспетчерских объектов, либо любого одного из них.

2.4.5.2.2. Мьютексы ядра

Слово Мьютекс (mutex = Mutually EXclusive) означает взаимоисключение, то есть мьютекс обеспечивает нескольким потокам взаимоисключающий доступ к совместно используемому ресурсу.

Вначале отметим, что кроме мьютексов ядра, есть еще быстрые мьютексы, являющиеся объектами исполнительной системы и не являющиеся диспетчерскими объектами. Мьютексы ядра обычно называют просто мьютексами.

Мьютексы ядра – это диспетчерские объекты, эквиваленты спин-блокировок. Двумя важными отличиями мьютексов от спин-блокировок являются:

- **Захват мьютекса является уникальным в рамках конкретного контекста потока. Поток, в контексте которого произошел захват мьютекса, является его владельцем, и может впоследствии рекурсивио захватывать его. Драйвер, захвативший мьютекс в конкретном контексте потока, обязан освободить его в том же коитексте потока, нарушение этого правила приведет к появлению «синего экрана».**
- **Для мьютексов предусмотрен механизм исключения взаимоблокировок. Ои заключается в том, что при инициализации мьютекса функцией KeInitializeMutex() указывается уровень (level) мьютекса. Если потоку требуется захватить несколько мьютексов одновремеиио, он должеи делать это в порядке возрастания значения level.**

Функции работы с мьютексами ядра:

1) VOID KeInitializeMutex(IN PKMUTEX Mutex, IN ULONG Level); Эта функция инициализирует мьютекс. Память под мьютекс уже должна быть выделена. После инициализации мьютекс находится в сигнальном состоянии.
2) LONG KeReleaseMutex(IN PKMUTEX Mutex, IN BOOLEAN Wait); Эта функция освобождает мьютекс, с указанием того, последует ли сразу после этого вызов функции ожидания мьютекса. Если параметр Wait равен TRUE, сразу за вызовом KeReleaseMutex() должен следовать вызов одной из функций ожидания **KeWait**Xxx(). В этом случае гарантируется, что пара функций – освобождение мьютекса и ожидание – будет выполнена как одна операция, без возможного в противном случае переключения контекста потока. Возвращаемым значением будет 0, если мьютекс был освобожден, то есть переведен из несигнального состояния в сигнальное. В противном случае возвращается ненулевое значение.
3) LONG KeReadStateMutex(IN PKMUTEX Mutex); Эта функция возвращает состояние мьютекса – сигнальное или несигнальное.

2.4.5.2.3. Семафоры

Семафоры являются более гибкой формой мьютексов. В отличие от мьютексов, программа имеет контроль над тем, сколько потоков одновременно могут захватывать семафор.

Семафор инициализируется с помощью функции KeInitializeSemaphore():

```
VOID KeInitializeSemaphore(
    IN PKSEMAPHORE Semaphore,
    IN LONG Count,
    IN LONG Limit);
```

Где:

Count – начальное значение, присвоенное семафору, определяющее число свободных в данный момент ресурсов. Если Count=0, семафор находится в несигнальном состоянии (свободных ресурсов нет), если >0 – в сигнальном;

Limit – максимальное значение, которое может достигать Count (максимальное число свободных ресурсов).

Функция KeReleaseSemaphore() увеличивает счетчик семафора Count на указанное в параметре функции значение, то есть освобождает указанное число ресурсов. Если при этом значение Count превышает значение Limit, значение Count не изменяется и генерируется исключение STATUS_SEMAPHORE_COUNT_EXCEEDED.

При вызове функции ожидания счетчик семафора уменьшается на 1 для каждого разблокированного потока (число свободных ресурсов уменьшается). Когда он достигает значения 0, семафор переходит в несигнальное состояние (свободных ресурсов иет). Использование семафора не зависит от контекста потока или процесса в том смысле, что занять ресурс семафора может один поток, а освободить его – другой, но драйвер не должен использовать семафоры в случайном контексте потока, так как в этом случае будет заблокирован случайный поток, не имеющий к драйверу никакого отношения. Семафоры следует использовать в ситуациях, когда драйвер создал собственные системные потоки.

2.4.5.2.4. События

События(events) позволяют проводить синхронизацию исполнения различных потоков, то есть один или несколько потоков могут ожидать перевода события в сигнальное состояние другим потоком.

При этом события могут быть двух видов:

- События, при переводе которых в сигнальное состояние будет разблокирован только один поток, после чего событие автоматически переходит в несигнальное состояние. Такие события носят название события синхронизации (synchronization events).
- События, при переводе которых в сигнальное состояние будут разблокированы все ожидающие их потоки. Событие должно быть переведено в несигнальное состояние вручную. Такие события носят название оповещающих (notification event).

Функции работы с событиями:

1) KeInitializeEvent() инициализирует событие. Память под событие уже должна быть выделена. При инициализации указывается тип – синхронизация или оповещение, а также начальное состояние – сигнальное или несигнальное. Имя события задать нельзя. Функция может быть использована в случайном контексте памяти на уровне IRQL PASSIVE_LEVEL.

2) IoCreateNotificationEvent(), IoCreateSynchronizationEvent() создают новое или открывает существующее событие с заданным именем. Если объект с таким именем существует, он открывается, если не существует, то создается. Имя события обычно указывается в директории диспетчера объектов «**\BaseNamedObjects**». Именно в этой директории содержатся имена событий, создаваемых или открываемых Win32-функциями CreateEvent()/OpenEvent().

Функция возвращает как указатель на объект-событие, так и его описатель в таблице описателя текущего процесса. Для уничтожения объекта необходимо использовать функцию ZwClose() с описателем в качестве параметра. Описатель должен быть использован в контексте того процесса, в котором он был получен на уровне IRQL PASSIVE_LEVEL.

3) KeClearEvent() и KeResetEvent() сбрасывают указанное событие в несигнальное состояние. Отличие между функциями в том, что KeResetEvent() возвращает состояние события до сброса. Функции могут быть вызваны на уровне IRQL меньшем или равном DISPATCH_LEVEL.

4) KeSetEvent() переводит событие в сигнальное состояние и получает предыдущее состояние. Одним из параметров является логическая переменная, указывающая, будет ли за вызовом KeSetEvent() немедленно следовать вызов функции ожидания. Если параметр TRUE, то гарантируется, что вызов этих двух функций будет выполнен как одна операция.

В случае событий оповещения сброс события в несигнальное состояние должен быть сделан вручную. Обычно это делает тот же код, который перевел событие в сигнальное состояние.

Следующий код корректно уведомляет все блокированные потоки о наступлении ожидаемого ими события:

```
KeSetEvent(&DeviceExt->Event, 0, NULL);
KeClearEvent(&DeviceExt->Event);
```

2.4.5.2.5. Быстрые мьютексы

Быстрый мьютекс являются урезанным вариантом мьютекса, который не может быть рекурсивно захвачен. Поскольку быстрый мьютекс не является диспетчерским объектом, он не может использоваться функцией KeWaitForSingleObject() или KeWaitForMultipleObjects(). Вместо этого нужно использовать функцию ExAcquireFast Mutex(). Эквивалента быстрым мьютексам на пользовательском уровне нет, поэтому они могут использоваться только для синхронизации кода режима ядра.

Функции работы с быстрыми мьютексами:

1) VOID ExInitializeFastMutex(IN PFAST_MUTEX FastMutex);
2) VOID ExAcquireFastMutex(IN PFAST_MUTEX FastMutex);
3) BOOLEAN ExTryToAcquireFastMutex(IN PFAST_MUTEX FastMutex);
4) VOID ExReleaseFastMutex(IN PFAST_MUTEX FastMutex);
5) VOID ExAcquireFastMutexUnsafe(IN PFAST_MUTEX FastMutex);
6) VOID ExReleaseFastMutexUnsafe (IN PFAST_MUTEX FastMutex).

2.4.5.3. Ресурсы Исполнительной системы

Ресурсы являются вариантом быстрого мьютекса. Ресурсы не являются диспетчерскими объектами, поэтому они не могут иметь имя и использоваться в функции

KeWaitForSingleObject() или KeWaitForMultipleObjects(). Ресурсы предоставляют две формы захвата:

- Эксклюзивный – в этом случае ресурс ведет себя как обычный мьютекс – поток, который попытается захватить такой ресурс для эксклюзивного или совместного использования, будет блокирован.
- Совместно используемый – в этом случае ресурс может быть одновременно захвачен для совместного использования любым числом потоков.

Ресурсы идеально подходят для защиты структур данных, которые могут одновременно читаться несколькими потоками, но должны модифицироваться в каждый момент времени только одним потоком.

Для работы с ресурсами существуют функции запроса эксклюзивного доступа, неэксклюзивного доступа и преобразования уже полученного неэксклюзивного доступа в эксклюзивный и, наоборот, без промежуточных операций освобождения ресурса и запроса нового режима доступа. Все функции должны вызываться на уровне IRQL меньшем DISPATCH_LEVEL.

Функции работы с ресурсами:

1) NTSTATUS ExInitializeResourceLite(IN PERESOURCE *Resource*);
2) VOID ExReinitializeResourceLite(IN PERESOURCE *Resource*);
3) BOOLEAN ExAcquireResourceExclusiveLite(IN PERESOURCE *Resource*,IN BOOLEAN *Wait*);
4) BOOLEAN ExTryToAcquireResourceExclusiveLite(IN PERESOURCE *Resource*);
5) BOOLEAN ExAcquireResourceSharedLite(IN PERESOURCE *Resource*,IN BOOLEAN *Wait*);
6) BOOLEAN ExAcquireSharedStarveExclusive(IN PERESOURCE *Resource*,IN BOOLEAN *Wait*);
7) BOOLEAN ExAcquireSharedWaitForExclusive(IN PERESOURCE *Resource*,IN BOOLEAN *Wait*);
8) VOID ExConvertExclusiveToSharedLite(IN PERESOURCE *Resource*);
9) BOOLEAN ExIsResourceAcquiredExclusiveLite(IN PERESOURCE *Resource*);
10) USHORT ExIsResourceAcquiredSharedLite(IN PERESOURCE *Resource*);
11) ULONG ExGetExclusiveWaiterCount(IN PERESOURCE *Resource*);
12) ULONG ExGetSharedWaiterCount(IN PERESOURCE *Resource*);
13) NTSTATUS ExDeleteResourceLite(IN PERESOURCE *Resource*);
14) VOID ExReleaseResourceForThreadLite(IN PERESOURCE *Resource*;
15) IN ERESOURCE_THREAD *ResourceThreadId*).

2.4.5.4. Обобщенная таблица механизмов синхронизации

В таблице 9 представлены механизмы синхронизации и особенности использования каждого из них.

Таблица 9

Объект синхронизации	Уровень IRQL, на котором может работать запрашивающий синхронизацию поток		Уровень IRQL, на котором будет работать запросивший синхронизацию поток при освобождении объекта синхронизации или его переходе в сигнальное состояние
	Запрос без блокирования потока	Запрос с блокированием потока.	
Стандартная спин-блокировка (Standard Spin Lock)	<= DISPATCH_LEVEL		DISPATCH_LEVEL
Спин-блокировка для ISR, определенная по умолчанию (Default ISR Spin Lock)	<= DIRQL		DIRQL
Спин-блокировка для синхронизации с ISR (ISR Synchronize Spin Lock)	<= Specified DIRQL		Specified DIRQL
Мьютекс (Mutex)	<=DISPATCH_LEVEL	<DISPATCH_LEVEL	<=DISPATCH_LEVEL
Семафор (Semaphore)	<=DISPATCH_LEVEL	<DISPATCH_LEVEL	<=DISPATCH_LEVEL
Событие синхронизации (Synchronization Event)	<=DISPATCH_LEVEL	<DISPATCH_LEVEL	<=DISPATCH_LEVEL
Событие уведомления (Notification Event)	<=DISPATCH_LEVEL	<DISPATCH_LEVEL	<=DISPATCH_LEVEL
Таймер синхронизации (Synchronization Timer)	<=DISPATCH_LEVEL	<DISPATCH_LEVEL	-
Таймер уведомления (Notification Timer)	<=DISPATCH_LEVEL	<DISPATCH_LEVEL	-
Процесс (Process)	<=DISPATCH_LEVEL	<DISPATCH_LEVEL	-
Поток (Thread)	<=DISPATCH_LEVEL	<DISPATCH_LEVEL	-
Файл (File)	<=DISPATCH_LEVEL	<DISPATCH_LEVEL	-
Ресурсы (Resources)	< DISPATCH_LEVEL	<DISPATCH_LEVEL	<=DISPATCH_LEVEL

2.4.6. Рабочие потоки

2.4.6.1. Необходимость в создании рабочих потоков

Любой исполняемый код, как и код драйвера, работает в контексте некоторого потока. Мы пока не обсуждали способы, с помощью которых драйвер может создать собственный поток, поэтому предполагается, что поток, в котором выполняется код драйвера, принадлежит некоторой прикладной программе. Это означает, что прикладная программа создала такой поток для выполнения своего кода, а не кода нашего драйвера. Если код драйвера производит длительную обработку, либо драйвер использует механизм синхронизации с ожиданием освобождения некоторого ресурса, код прикладной программы, для выполнения которого и создавался поток, не выполняется. Если этот поток единственный в прикладном процессе, то прикладная программа «висит».

Если описанная ситуация имеет место в диспетчерской функции драйвера верхнего уровня, мы «всего лишь» «подвесили» прикладную программу, непосредственно взаимодействующую с драйвером. В этом случае прикладная программа знает о такой возможности, и может поместить операции взаимодействия с драйвером (чтение, запись, отправка кодов управления) в отдельный поток. В этом случае драйвер может не беспокоиться о прикладной программе. Однако, такая ситуация довольно редка. Очень часто код драйвера работает в контексте случайного потока, то есть любого произвольного потока в системе. Такой поток ничего не знает о нашем драйвере и вышеописанная ситуация неприемлема. В этом случае драйвер должен создать свой собственный поток, в котором и производить длительную обработку, либо ожидание освобождения ресурсов.

Возможна другая ситуация, требующая обязательного создания потоков, когда драйверу необходимо выполнить операции на уровне IRQL меньшем DISPATCH_LEVEL, а код драйвера работает на повышенных уровнях IRQL, больших или равных DISPATCH_LEVEL.

2.4.6.2. Системные рабочие потоки

В процессе системной инициализации NT создает несколько потоков в процессе System. Эти потоки служат исключительно для выполнения работы, затребованной другими потоками. Такие потоки наиболее удобны в случаях, когда потоку с повышенным уровнем IRQL требуется выполнить работу на уровне IRQL PASSIVE_LEVEL.

В принципе, можно создать новый поток, однако создание нового потока и его планирование планировщиком является более ресурсоемким, чем использование существующего потока. Большинство стандартных компонент ОС, таких как компоненты файловой системы, используют для своих нужд готовые системные рабочие потоки.

Имеются ситуации, при которых использование системных рабочих потоков неприемлемо в силу их организации. Такими ситуациями являются необходимость в дли-

тельной (несколько сотен микросекунд) обработке внутри потока, либо длительное ожидание освобождения ресурса или наступления события. В этих ситуациях драйвер должен создавать свой собственный поток.

Как организованы системные рабочие потоки? Как уже было сказано, в процессе системной инициализации NT создает несколько системных рабочих потоков. Число этих потоков фиксировано. Для всех потоков существует единая очередь, из которой поток выбирает адрес функции драйвера, которая должна быть выполнена в данном потоке. Такая функция называется **рабочим элементом** (WorkItem). Функция выполняется в потоке до своего завершения, после чего поток выбирает из очереди следующий рабочий элемент. Если очередь пуста, поток блокируется до появления в очереди очередного рабочего элемента.

Существует три типа системных рабочих потоков: Delayed (замедленные), Critical (критические) и HyperCritical (сверхкритические). Все типы потоков создаются на уровне IRQL PASSIVE_LEVEL. Для каждого типа потоков будут различны:

- число потоков данного типа;
- базовый приоритет планирования потока, относящегося к данному типу;
- очередь рабочих элементов.

Число потоков каждого типа зависит от объема памяти и типа ОС. В таблице 10 указано число потоков и базовый приоритет планирования для ОС Win2000 Professional и Server.

Таблица 10. Число Системных Рабочих Потоков

Тип рабочего потока	**Объем системной памяти**			**Базовый приоритет планирования**
	12-19 MB	***20-64 MB***	***> 64 MB***	
Delayed	3	3	3	Значение в диапазоне динамических приоритетов
Critical	3	Professional: 3 Server: 6	Professional: 5 Server: 10	Значение в диапазоне приоритетов реального времени
HyperCritical	1	1	1	Не документирован

Следует отметить, что использование единственного потока типа HyperCritical не документировано. ОС использует этот поток для выполнения функции – чистильщика, которая освобождает потоки при их завершении.

При постановке рабочего элемента в очередь указывается тип потока, которому предназначен рабочий элемент.

Для работы с системными рабочими потоками существует два набора функций – функции с префиксом **Ex**, и функции с префиксом **Io**. Функции с префиксом Ex использовались в ОС NT 4.0 и более ранних версиях, и в Win2000 считаются устаревшими. В любом случае, вначале драйвер должен инициализировать рабочий элемент с