**<u>МИНИСТЕРСТВО ОБОРОНЫ РОССИЙСКОЙ ФЕДЕРАЦИИ</u>**
**ВОЕННАЯ АКАДЕМИЯ**
**ВОЙСКОВОЙ ПРОТИВОВОЗДУШНОЙ ОБОРОНЫ**
**ВООРУЖЕННЫХ СИЛ РОССИЙСКОЙ ФЕДЕРАЦИИ**

# ОПЕРАЦИОННЫЕ СИСТЕМЫ

Учебное пособие

Утверждено
начальником академии
в качестве учебного пособия
для курсантов

Издание академии

2005

Учебное пособие предназначено для самостоятельной работы курсантов при изучении дисциплины “Операционные системы”, а также при выполнении курсового и дипломного проектирования.

Учебное пособие охватывает основные вопросы теории операционных систем и соответствует по объему и порядку изложения учебной программе указанной дисциплины.

Авторы: кандидат технических наук, доцент Стальнов А. Ф.,
кандидат технических наук, доцент Фомин А. И.

Ответственный за выпуск: Стальнов А. Ф.

Редактор Гусева М. Н.

207. Подписано в печать 07.06.2005. Формат 60×84/16.
Уч.-изд. л. 12,8. Усл. печ. л. 16,3. Печ. л. 17,6. Зак. № 197-2005. Бесплатно

Типография ВА войсковой ПВО ВС РФ

# ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ ... 9
ГЛАВА 1. ОБЩАЯ ХАРАКТЕРИСТИКА ОПЕРАЦИОННЫХ СИСТЕМ ... 13
1. Назначение, состав и основные функции операционных систем и системного программного обеспечения ... 13
1.1. Назначение и состав системного программного обеспечения ... 13
1.2. Назначение и основные функции операционных систем ... 15
1.3. Основные понятия операционных систем ... 17
2. Классификация операционных систем ... 21
2.1. Способы классификации операционных систем ... 21
2.2. Классификация операционных систем по функциональным возможностям ... 23
2.2.1. ДОС (Дисковые Операционные Системы) ... 24
2.2.2. ОС общего назначения ... 24
2.2.3. Системы реального времени ... 25
2.2.4. Средства кросс-разработки ... 25
2.2.5. Системы виртуальных машин ... 26
2.2.6. Системы промежуточных типов ... 26
2.3. Семейства операционных систем ... 27
3. Принципы построения операционных систем ... 29
3.1. Основные принципы построения ОС ... 29
3.2. Функциональные компоненты операционной системы ... 33
3.2.1. Управление процессами ... 33
3.2.2. Управление памятью ... 35
3.2.3. Управление файлами и внешними устройствами ... 36
3.2.4. Защита данных и администрирование ... 38
3.2.5. Интерфейс прикладного программирования ... 39
3.2.6. Пользовательский интерфейс ... 40
3.3. ОС Windows ... 41
3.3.1. Привилегированный и пользовательский режимы ... 41
3.3.2. Уровень абстрагирования от аппаратных средств ... 42
3.3.3. Исполняющая система ... 43
3.3.4. Защищенные подсистемы ... 43
ГЛАВА 2. ОРГАНИЗАЦИЯ ОПЕРАЦИОННЫХ СИСТЕМ ... 45
1. Архитектура операционной системы ... 45
1.1. Ядро и вспомогательные модули ОС ... 45
1.1.1. Назначение ядра и вспомогательных модули ОС ... 45
1.1.2. Ядро в привилегированном режиме ... 48
1.1.3. Многослойная структура ОС ... 51
1.2. Микроядерная архитектура ... 59

1.2.1. Концепция ........ 59
1.2.2. Преимущества и недостатки, микроядерной архитектуры ........ 61
1.3. Аппаратная зависимость и переносимость ОС ........ 54
1.3.1. Типовые средства аппаратной поддержки ОС ........ 55
1.3.2. Машинно-зависимые компоненты ОС ........ 57
1.3.3. Переносимость операционной системы ........ 58
2. Ресурсы операционной системы ........ 63
2.1. Характеристика ресурсов и способов их использования ........ 64
2.1.1. Аппаратные ресурсы ........ 65
2.1.2. Программные ресурсы ........ 67
2.2. Понятие и задачи управления ресурсами ........ 68
2.3. Дисциплины распределения ресурсов, используемые в операционных системах ........ 70
3. Принципы построения интерфейсов операционных систем ........ 75
3.1. Интерфейс прикладного программирования ........ 76
3.1.1. Понятие интерфейса прикладного программирования ........ 76
3.1.2. Реализация функций API на уровне ОС ........ 78
3.1.3. Реализация функций API на уровне системы программирования ........ 79
3.1.4. Реализация функций API с помощью внешних библиотек ........ 80
3.2. Платформенно-независимый интерфейс POSIX ........ 82
4. Принципы построения и организации ОС АСУ войсковой ПВО ........ 84
4.1. Общие сведения об операционных системах реального времени ........ 84
4.2. Требования, предъявляемые к ОС реального времени ........ 87
4.3. Операционные системы образцов АСУ войсковой ПВО ........ 89
4.3.1. Операционная система VxWorks ........ 89
4.3.2. Операционная система ос2000 (ОСРВ “Багет”) ........ 90
4.3.3. Операционная система QNX ........ 92
4.3.4. Операционная система Linux (МС ВС) ........ 95
ГЛАВА 3. УПРАВЛЕНИЕ ПРОЦЕССОМ ........ 97
1. Процессы и потоки ........ 97
1.1. Мультипрограммирование ........ 97
1.1.1. Мультипрограммирование в системах пакетной обработки ........ 98
1.1.2. Мультипрограммирование в системах разделения времени ........ 99
1.1.3. Мультипрограммирование в системах реального времени ........ 100
1.1.4. Мультипроцессорная обработка ........ 100
1.2. Понятия процесса и потока ........ 104

1.3. Идентификатор и дескриптор процесса ... 113
2. Понятие прерываний и их организация в ОС ... 116
2.1. Назначение и типы прерываний ... 116
2.2. Механизм прерываний ... 117
3. Диспетчеризация и синхронизация процессов ... 123
3.1. Планирование и диспетчеризация процессов и задач ... 124
3.1.1. Стратегии планирования ... 125
3.1.2. Дисциплины диспетчеризации ... 125
3.2. Приемы и средства синхронизации процессов ... 130
3.2.1. Независимые и взаимодействующие вычислительные процессы ... 130
3.2.2. Средства синхронизации и связи при проектировании взаимодействующих вычислительных процессов ... 135
4. Проблема тупиков и методы борьбы с ними ... 145
4.1. Понятие тупиковой ситуации и причины их возникновения ... 145
4.2. Примеры тупиковых ситуаций и причины их возникновения ... 148
4.3. Методы борьбы с тупиками ... 153
4.3.1. Предотвращение тупиков ... 153
4.3.2. Обход тупиков ... 154
4.3.3. Обнаружение тупика ... 158
ГЛАВА 4. УПРАВЛЕНИЕ ПАМЯТЬЮ И ДАННЫМИ ... 161
1. Организация и распределение памяти ... 161
1.1. Задачи управления памятью ... 161
1.2. Способы распределения памяти ... 162
1.2.1. Память и отображения, виртуальное адресное пространство ... 162
1.2.2. Простое непрерывное распределение и распределение с перекрытием (оверлейные структуры) ... 166
1.2.3. Распределение статическими и динамическими разделами ... 167
1.3. Распределение оперативной памяти в современных операционных системах для ПК ... 172
1.3.1. Распределение оперативной памяти в MS-DOS ... 172
1.3.2. Распределение оперативной памяти в Microsoft Windows 95/98 ... 175
1.3.3. Распределение оперативной памяти в Microsoft Windows NT ... 178
2. Механизмы реализации виртуальной памяти ... 180
2.1. Сегментный способ организации виртуальной памяти ... 180
2.2. Страничный способ организации виртуальной памяти ... 185
2.3. Сегментно-страничный способ организации виртуальной памяти ... 189

3. Управление вводом-выводом ........ 191
3.1. Основные понятия и концепция организации ввода-вывода в ОС ........ 192
3.2. Режимы управления вводом-выводом ........ 195
3.2.1. Сущность основных режимов ввода-вывода ........ 195
3.2.2. Закрепление устройств, общие устройства ввода/вывода ........ 197
3.2.3. Основные системные таблицы ввода/вывода ........ 198
3.2.4. Синхронный и асинхронный ввод/вывод ........ 202
3.2.5. Кэширование операций ввода/вывода при работе с накопителями на магнитных дисках ........ 204
4. Управление данными ........ 208
4.1. Задачи управления данными ........ 208
4.2. Понятие и функции файловой системы ........ 209
4.3. Логическая организация файловой системы ........ 211
4.3.1. Типы файлов ........ 211
4.3.2. Иерархическая структура файловой системы ........ 212
4.3.3. Имена файлов ........ 213
4.3.4. Монтирование ........ 215
4.3.5. Атрибуты файлов ........ 217
4.3.6. Логическая организация файла ........ 219
5. Физическая организация файловой системы ........ 222
5.1. Структура магнитного диска ........ 222
5.2. Файловые системы современных операционных систем ........ 228
5.2.1. Физическая организация и адресация файла ........ 228
5.2.2. Файловая система FAT ........ 231
5.2.3. Файловая система NTFS ........ 237
ГЛАВА 5. ЗАЩИТА ПАМЯТИ И ДАННЫХ ........ 245
1. Методы защиты памяти и данных ........ 245
1.1. Защита памяти ........ 245
1.1.1. Защита по граничным адресам ........ 246
1.1.2. Защита по ключам ........ 247
1.2. Защита данных ........ 249
1.2.1. Задание доступности и полномочий ........ 250
1.2.2. Реализация защиты ........ 252
2. Система безопасности операционной системы ........ 257
2.1. Основные понятия безопасности ........ 258
2.1.1. Конфиденциальность, целостность и доступность данных ........ 258
2.1.2. Классификация угроз ........ 260
2.1.3. Системный подход к обеспечению безопасности ........ 262
2.1.4. Политика безопасности ........ 263
2.2. Базовые технологии безопасности ........ 265
2.2.1. Шифрование ........ 265

2.2.2. Аутентификация, авторизация, аудит ............................. 274
2.2.3. Технология защищенного канала .................................... 279
3. Аутентификация в современных операционных системах ............. 281
3.1. Аутентификация пользователей ................................................ 281
3.1.1. Сетевая аутентификация на основе многоразового пароля ........................................................................ 281
3.1.2. Аутентификация с использованием одноразового пароля ........................................................................ 283
3.1.3. Аутентификация на основе сертификатов....................... 286
3.2. Аутентификация информации .................................................... 290
3.2.1. Цифровая подпись............................................................ 291
3.2.2. Аутентификация программных кодов ............................ 293
ЗАКЛЮЧЕНИЕ ....................................................................................... 295
ЛИТЕРАТУРА......................................................................................... 303

# ВВЕДЕНИЕ

Уже давно прошло время, когда операционных систем вообще не существовало. Сегодня операционные системы применяются практически на всех вычислительных машинах – от гигантских супер-ЭВМ до миниатюрных персональных компьютеров. Операционные системы зачастую даже в большей степени определяют представление пользователя о машине, чем сама аппаратура этой машины. Например, в сфере персональных компьютеров де-факто стандартной стали операционные системы семейства CP/M, так что очень многие фирмы – изготовители компьютеров создают новые аппаратные средства с ориентацией на эту операционную систему. Пользователь видит аппаратуру компьютера – скажем дисплей на электроннолучевой трубке и клавиатуру, – однако функциональные возможности машины ему обеспечивает операционная система.

Операционная система в наибольшей степени определяет облик всей вычислительной системы в целом. Несмотря на это, пользователи, активно использующие вычислительную технику, зачастую испытывают затруднения при попытке дать определение операционной системе. Частично это связано с тем, что ОС выполняет две по существу малосвязанные функции: обеспечение пользователю-программисту удобств посредством предоставления для него расширенной машины и повышение эффективности использования компьютера путем рационального управления его ресурсами.

*ОС как расширенная машина*

Использование большинства компьютеров на уровне машинного языка затруднительно, особенно это касается ввода-вывода. При работе с диском программисту-пользователю достаточно представлять его в виде некоторого набора файлов, каждый из которых имеет имя. Работа с файлом заключается в его открытии, выполнении чтения или записи, а затем в закрытии файла. Вопросы подобные таким, как какой вид модуляции следует использовать при записи или в каком состоянии сейчас находится двигатель механизма перемещения считывающих головок, не должны волновать пользователя. Программа, которая скрывает от программиста все реалии аппаратуры и предоставляет возможность простого, удобного просмотра указанных файлов, чтения или записи – это, конечно, операционная система. Точно так же, как ОС ограждает программистов от аппарату-

ры дискового накопителя и предоставляет ему простой файловый интерфейс, операционная система берет на себя все малоприятные дела, связанные с обработкой прерываний, управлением таймерами и оперативной памятью, а также другие низкоуровневые проблемы. В каждом случае та абстрактная, воображаемая машина, с которой, благодаря операционной системе, теперь может иметь дело пользователь, гораздо проще и удобнее в обращении, чем реальная аппаратура, лежащая в основе этой абстрактной машины.

С этой точки зрения функцией ОС является предоставление пользователю некоторой расширенной или виртуальной машины, которую легче программировать и с которой легче работать, чем непосредственно с аппаратурой, составляющей реальную машину.

*ОС как система управления ресурсами*

Идея о том, что ОС прежде всего система, обеспечивающая удобный интерфейс пользователям, соответствует рассмотрению сверху вниз. Другой взгляд, снизу вверх, дает представление об ОС как о некотором механизме, управляющем всеми частями сложной системы. Современные вычислительные системы состоят из процессоров, памяти, таймеров, дисков, накопителей на магнитных лентах, сетевых коммуникационной аппаратуры, принтеров и других устройств. В соответствии со вторым подходом функцией ОС является распределение процессоров, памяти, устройств и данных между процессами, конкурирующими за эти ресурсы. ОС должна управлять всеми ресурсами вычислительной машины таким образом, чтобы обеспечить максимальную эффективность ее функционирования. Критерием эффективности может быть, например, пропускная способность или реактивность системы. Управление ресурсами включает решение двух общих, не зависящих от типа ресурса, задач:

*планирование ресурса* – то есть определение, кому, когда, а для делимых ресурсов и в каком количестве, необходимо выделить данный ресурс;

*отслеживание состояния ресурса* – то есть поддержание оперативной информации о том, занят или не занят ресурс, а для делимых ресурсов – какое количество ресурса уже распределено, а какое свободно.

Для решения этих общих задач управления ресурсами разные ОС используют различные алгоритмы, что, в конечном счете, и определяет их облик в целом, включая характеристики производительности, область применения и даже пользовательский интерфейс.

*Целями обучения* по дисциплине “Операционные системы” являются:

1) изучение и практическое освоение теории, средств и методов построения операционных систем и системного программного обеспечения ЭВМ АСОИУ;

2) приобретение практических навыков и умений эффективной и технически грамотной их эксплуатации в практической деятельности.

В результате изучения дисциплины *курсант должен*:

ИМЕТЬ ПРЕДСТАВЛЕНИЕ:

1) о проблемах, перспективах и тенденциях развития операционных систем и системных программных средств ЭВМ АСОИУ;

2) об ОС перспективных образцов АСУ войсковой ПВО.

ЗНАТЬ:

1) стандарты разработки и реализации ОС и системного программного обеспечения (СПО);

2) методические и нормативные материалы по проектированию, производству и сопровождению ОС и системного программного обеспечения;

3) принципы построения и организацию современных операционных систем и СПО, включая ЭВМ АСУ войсковой ПВО;

4) современные программные средства, модели и структуры ОС и СПО, их классификацию;

5) системное программное обеспечение ЭВМ, применяемых в АСОИУ, включая ЭВМ АСУ войсковой ПВО;

6) методы исследования ОС и СПО, включая методы анализа функциональных и эксплуатационных характеристик ОС и СПО, а также оценки их надежности и качества;

7) инструментальные средства разработки программного обеспечения в современных операционных средах.

УМЕТЬ ИСПОЛЬЗОВАТЬ:

1) системные программные средства, операционные системы и оболочки, обслуживающие сервисные программы, в том числе, применяемые в ЭВМ АСОИУ;

2) методические материалы и программные средства по инсталляции, настройке и обслуживанию ОС и СПО;

3) методы и инструментальные средства анализа функциональных и эксплуатационных характеристик операционных систем и системного программного обеспечения, а также оценки их надежности и качества;

4) методические и инструментальные средства по программированию в современных операционных средах.

*Дисциплина формирует* знания принципов построения операционных систем и системного программного обеспечения, обслуживающих и сервисных программ, применяемых в ЭВМ АСОИУ, а также навыки их использования в процессе эксплуатации систем вооружения.

*Научной основой* дисциплиной являются положения теории систем и системного анализа, теории множеств, теории графов и матричного исчисления, а также методы структурного и визуального программирования.

Обучение по дисциплине базируется на знаниях, полученных курсантами при изучении дисциплин: “Информатика”, “Математика”, “Организация ЭВМ и систем”, “Программирование на языке высокого уровня”.

*Дисциплина обеспечивает* изучение следующих дисциплин: “Методы и средства защиты компьютерной информации”, “Сети ЭВМ и телекоммуникации”, “Технологии в АСОИУ”, а также дипломное проектирование.

# ГЛАВА 1

# ОБЩАЯ ХАРАКТЕРИСТИКА ОПЕРАЦИОННЫХ СИСТЕМ

## 1. Назначение, состав и основные функции операционных систем и системного программного обеспечения

### 1.1. Назначение и состав системного программного обеспечения

Под *программой* понимают описание, воспринимаемое ЭВМ и достаточное для решения на ней определенной задачи. Для составления программ используют искусственные языки, называемые языками программирования.

Под программным обеспечением (ПО) в узком смысле понимается просто совокупность программ. В широком смысле в ПО (наряду с программами) включают различные языки, процедуры, правила и документацию, необходимые для использования и эксплуатации программных продуктов.

ПО ЭВМ по функциональному признаку традиционно делится на системное и прикладное.

*Системное ПО* – это совокупность программ, описаний и инструкций, предназначенных для автоматизации трудоемких этапов разработки алгоритмов и программ, а также для организации и контроля вычислительного процесса на машине во время ее функционирования. Оно является необходимым дополнением к техническим средствам ЭВМ. Без системного ПО (СПО) машина по сути безжизненна.

*Прикладное ПО* представляет собой совокупность программ решения конкретных задач из различных сфер применения.

В свою очередь СПО ЭВМ по функциональному признаку подразделяется (рис. 1.1) на:

операционные системы (ОС);
сервисные системы;
инструментальные системы;
системы контроля и диагностики.

Рис. 1.1. Структура СПО

*Операционные системы* являются неотъемлемым обязательным дополнением ПЭВМ. Операционная система – это комплекс управляющих и обрабатывающих программ, который, с одной стороны, выступает как интерфейс между аппаратурой компьютера и пользователем с его задачами, а с другой – предназначен для наиболее эффективного использования ресурсов вычислительной системы и организации надежных вычислений. Любой из компонентов прикладного программного обеспечения обязательно работает под управлением операционной системы.

*Сервисные системы* расширяют возможности ОС, предоставляя пользователю, а также выполняемым программам набор дополнительных услуг.

По функциональному назначению сервисные системы делят на:

1) *интерфейсные системы (interface)*, в основном, графического типа, модифицирующие как пользовательский, так и программный интерфейс ОС, а также иногда реализующие и дополнительные возможности по распределению ресурсов ЭВМ; вследствие этого они считаются естественным “продолжением” ОС;

2) *оболочки (shell) ОС*, модифицирующие только пользовательский интерфейс, повышая уровень (главным образом за счет “меню” и использования функциональных клавиш), а также предоставляя дополнительные возможности;

3) *утилиты (utility)* – специальные системные программы, с помощью которых можно как обслуживать саму операционную систему, так и подготавливать носители данных, выполнять перекодирование данных, осуществлять оптимизацию размещения данных на носителе и производить некоторые другие работы, связанные с обслуживанием вычислительной системы.

*Инструментальные системы* предназначены для разработки ПО, хотя часть из них может применяться и для решения прикладных задач. Инструментальные системы часто называют *системами программирования.* Но системы программирования обладают универсальностью, т.е. с их помощью можно запрограммировать и решить любую задачу, допускающую алгоритмическое решение. Инструментальные же системы часто являются специализированными в том смысле, что они служат для создания ПО определенного функционального назначения.

*Системы контроля и диагностики* предназначены для облегчения тестирования оборудования и поиска неисправностей. Система контроля и диагностики может быть разделена на три взаимосвязанные подсистемы: подсистему контроля, подсистему диагностики и подсистему восстановления.

*Подсистема контроля* предназначена для контроля правильности функционирования самой ЭВМ и связанных с ней периферийных устройств или элементов автоматизированной системы управления в период проведения функционального контроля.

*Подсистема диагностики* служит для поиска места отказа в аппаратуре ЭВМ. Она включается в работу операционной системой сразу после обнаружения факта отказа аппаратуры подсистемой контроля или по командам оператора. Подсистема диагностики обнаруживает место отказа с точностью до типового элемента замены и выдает необходимые сведения об этом подсистеме восстановления работоспособности ЭВМ.

*Подсистема восстановления работоспособности ЭВМ* производит автоматическое отключение неисправного устройства и передает его функции. Если возможно, другому устройству.

В ряде случаев подсистема контроля и диагностики не выделяется в отдельную систему, а входит в состав операционной системы.

## 1.2. Назначение и основные функции операционных систем

По определению ГОСТ 19781-83 под *операционной системой* понимают систему программ, предназначенную для обеспечения определенного уровня эффективности вычислительной системы за счет автоматизированного управления ее работой и предоставляемых пользователям определенного рода услуг. Одну из особых категорий пользователей составляют специалисты, которых интересуют не только услуги, предоставляемые

ОС, но и то, каким образом эти услуги реализуются. Такого рода знания необходимы для успешного проектирования и эксплуатации как ЭВМ, так и автоматизированных систем.

Проведенный анализ эволюционного развития ОС дает основание предложить следующую трактовку определения ОС.

*Операционная система* – это упорядоченная последовательность системных управляющих программ совместно с необходимыми информационными массивами, предназначенная для планирования исполнения пользовательских программ и управления всеми ресурсами вычислительной машины (программами, данными, аппаратурой, оператором и другими распределяемыми и управляемыми объектами) с целью предоставления возможности пользователям эффективно (в некотором смысле) решать задачи, сформулированные в терминах вычислительной системы.

Это определение не противоречит данному в ГОСТ.

По современным представлениям *ОС должна уметь делать следующее*:

1. Обеспечивать загрузку пользовательских программ в оперативную память и их исполнение (этот пункт не относится к ОС, предназначенным для прошивки в ПЗУ).

2. Обеспечивать управление памятью. В простейшем случае это указание единственной загруженной программе адреса, на котором кончается память, доступная для использования, и начинается память, занятая системой. В многопроцессных системах это сложная задача управления системными ресурсами.

3. Обеспечивать работу с устройствами долговременной памяти, такими как магнитные диски, ленты, оптические диски, флэш-память и т. д. Как правило, ОС управляет свободным пространством на этих носителях и структурирует пользовательские данные в виде файловых систем.

4. Предоставлять более или менее стандартизованный доступ к различным периферийным устройствам, таким как терминалы, модемы, печатающие устройства или двигатели, поворачивающие антенны РЛС.

5. Предоставлять некоторый пользовательский интерфейс: часть систем ограничивается командной строкой, в то время как другие на 90% состоят из интерфейсной подсистемы. Встраиваемые системы часто не имеют никакого пользовательского интерфейса.

Существуют ОС, функции которых этим и исчерпываются. Одна из хорошо известных систем такого типа – *дисковая операционная система* MS DOS.

Более развитые ОС предоставляют также следующие возможности:

6. Параллельное (или псевдопараллельное, если машина имеет только один процессор) исполнение нескольких задач.

7. Организацию взаимодействия задач друг с другом.

8. Организацию межмашинного взаимодействия и разделения ресурсов.

9. Защиту системных ресурсов, данных и программ пользователя, исполняющихся процессов и самой себя от ошибочных и зловредных действий пользователей и их программ.

10. Аутентификацию (проверку того, что пользователь является тем, за кого он себя выдает), авторизацию (проверка, что тот, за кого себя выдает пользователь, имеет право выполнять ту или иную операцию) и другие средства обеспечения безопасности.

## 1.3. Основные понятия операционных систем

Операционная система выполняет функции управления вычислительными процессами в вычислительной системе, распределяет ресурсы вычислительной системы между различными вычислительными процессами и образует программную среду, в которой выполняются прикладные программы пользователей. Такая среда называется операционной. В общем случае *операционная среда* означает интерфейс, необходимый программам для обращения к операционной системе с целью получить определенный сервис – выполнить операцию ввода-вывода, получить или освободить участок памяти и т. д.

Операционная среда может включать несколько интерфейсов: пользовательские и программные. Если говорить о пользовательских, то, например, система Linux имеет для пользователя как интерфейсы командной строки (можно использовать различные “оболочки” – shell), интерфейс наподобие Norton Commander – Midnight Commander, так и графические интерфейсы – X-Window с различными менеджерами окон – KDE, Gnome и т. д. Если же говорить о программных интерфейсах, то в той же ОС Linux программы могут обращаться как к операционной системе за соответствующими сервисами и функциями, так и к графической подсистеме (если она используется). С точки зрения архитектуры процессора (и всего ПК в целом) двоичная программа, созданная для работы в среде Linux, использует те же команды и форматы данных, что и программа, созданная для работы в среде Windows NT. Однако в первом случае мы имеем обращение к одной операционной среде, а во втором – к другой. И программа, созданная для Windows непосредственно, не будет выполняться в Linux; однако если ОС Linux организовать полноценную операционную среду Windows, то наша Windows-программа сможет быть выполнена. Можно сказать, что операционная среда – это то системное программное окружение, в котором могут выполняться программы, созданные по правилам работы этой среды.

Понятие “*вычислительный процесс*” (или просто – “процесс”) является одним из основных при рассмотрении операционных систем. Как понятие, процесс является определенным видом абстракции, и мы будем придерживаться следующего неформального определения. Последовательный

процесс (иногда называемый “задачей”) – это выполнение отдельной программы с ее данными на последовательном процессоре. Концептуально процессор рассматривается в двух аспектах: во-первых, он является носителем данных и, во-вторых, он (одновременно) выполняет операции, связанные с их обработкой.

Определение концепции процесса преследует цель выработать механизмы распределения и управления ресурсами. Понятие *ресурса*, так же как и понятие процесса, является, пожалуй, основным при рассмотрении операционных систем. Термин ресурс обычно применяется по отношению к повторно используемым, относительно стабильным и часто недостающим объектам, которые запрашиваются, используются и освобождаются процессами в период их активности. Другими словами, ресурсом называется всякий объект, который может распределяться внутри системы.

Далее рассмотрим понятия, с которыми часто придется сталкиваться при изучении операционных систем.

*Расслоение памяти* обеспечивает возможность одновременного доступа к последовательным ячейкам основной памяти, поскольку ячейки с соседними адресами размещаются в различных модулях памяти.

*Механизм прерываний* играет важную роль для режимов работы, при которых много операций могут выполняться асинхронно, но в определенных случаях требуют синхронизации.

*Буферизация* с несколькими буферами позволяет эффективно совмещать операции ввода-вывода с вычислениями.

*Спулинг* (ввод-вывод с буферизацией) позволяет отделить работающую программу от низкоскоростных устройств ввода-вывода, таких, как принтеры и устройства ввода данных с перфокарт. При спулинге ввод-вывод данных для программы осуществляется при посредстве высокоскоростного внешнего запоминающего устройства, например, накопителя на магнитных дисках, а фактическое чтение или распечатка данных производится в то время, когда устройства ввода перфокарт и принтеры свободны.

Для изоляции пользователей друг от друга в многоабонентских системах необходимо предусматривать *защиту памяти*; защиту можно реализовывать несколькими различными способами, в том числе при помощи граничных регистров или ключей защиты.

Применение *стандартного интерфейса ввода-вывода* существенно упрощает подключение к машине новых внешних устройств. Внешние устройства могут работать под непосредственным управлением центрального процессора в *режиме он-лайн*, или *автономно* (*оф-лайн*), под управлением отдельных контроллеров, независимых от процессора. Функционально законченные вычислительные машины, которые выполняют операции ввода данных с перфокарт на магнитную ленту, вывода данных с магнитной ленты на печать и т. д. для более крупных машин, называются процессорами ввода-вывода, или компьютерами-сателлитами.

*Канал* – специализированная вычислительная машина для выполнения операций ввода-вывода без участия центрального процессора. Для координации взаимодействия между центральным процессором и каналом применяется, как правило, способ регулярного опроса или механизм прерываний. Наиболее известные типы каналов – это селекторные, байт-мультиплексные и блок-мультиплексные каналы.

*Система управления вводом-выводом* (*IOCS*) – это пакет программ, назначение которого заключается в том, чтобы освободить пользователя от необходимости детального управления вводом-выводом. Пакеты IOCS являются важной частью современных операционных систем.

*Метод относительной адресации* дает возможность работать с очень большим адресным пространством без необходимости увеличивать размер машинного слова; во время выполнения программы все адреса формируются путем прибавления смещения к содержимому базового регистра. Благодаря этому упрощается также перемещение программ по памяти.

Наличие в машине нескольких режимов работы обеспечивает защиту программ и данных. В режиме *супервизора* могут выполняться любые команды (включая привилегированные), а в режиме задачи – только непривилегированные. Эти режимы определяют границу между возможностями пользователя и возможностями операционной системы. В некоторых машинах предусматриваются более двух подобных режимов работы.

*Системы виртуальной памяти* в обычном случае дают возможность программам работать с гораздо более широким диапазоном адресов, чем адресное пространство имеющейся основной памяти. Это позволяет освободить программиста от ограничений, связанных с емкостью основной памяти.

*Метод прямого доступа к памяти* исключает необходимость прерывать работу центрального процессора при передаче каждого байта блока данных во время выполнения операций ввода-вывода – в этом случае требуется только один сигнал прерывания, вырабатываемый при завершении передачи всего блока. Каждый символ записывается в основную память и читается из нее с захватом цикла памяти – здесь каналу предоставляется приоритет, поскольку центральный процессор может и подождать.

В архитектурах компьютеров высокого быстродействия применяются *конвейеры*, которые позволяют совмещать работу нескольких команд, в каждый конкретный момент времени находящихся на различных стадиях выполнения.

В современных вычислительных машинах реализуется *иерархическая структура памяти*, включающая кэш-память, основную (первичную, оперативную) память и внешнюю (вторичную, массовую) память; при переходе с уровня на уровень иерархии в указанном порядке емкости памяти увеличиваются, а стоимость в расчете на байт уменьшается.

*Программное обеспечение* – это программы, состоящие из команд, которые интерпретируются аппаратными средствами; команды определяют

алгоритмы, обеспечивающие решение задач. Программы для компьютера можно писать на машинном языке, языке ассемблера или языках высокого уровня. Программисты редко работают непосредственно на машинных языках – программы в машинных кодах генерируются при помощи ассемблеров и компиляторов. *Макропроцессоры* дают возможность программистам, работающим на языке ассемблера, писать макрокоманды, которые порождают много команд на языке ассемблера. *Компиляторы* обеспечивают трансляцию программ, написанных на языках высокого уровня, на машинный язык. *Интерпретаторы* непосредственно выполняют исходные программы, не генерируя при этом объектные модули.

*Процедурно-ориентированные* языки являются универсальными, а *проблемно-ориентированные* языки специализируются для эффективного решения задач определенных классов. *Компиляторы без оптимизации* работают быстро, однако генерируют относительно неэффективные коды; *оптимизирующие компиляторы* позволяют получать эффективные коды, но работают гораздо медленнее, чем без оптимизации.

*Абсолютные загрузчики* осуществляют загрузку программ в конкретные ячейки, адреса которых указываются при компиляции; перемещающие загрузчики могут размещать программы в различных свободных участках памяти. Привязка программы к памяти по абсолютным адресам осуществляется во время трансляции, а привязка перемещаемых программ – во время загрузки или даже во время выполнения.

*Связывающие загрузчики* объединяют отдельные блоки программы, создавая единый модуль, готовый к выполнению; этот выполняемый модуль размещается в основной памяти. *Редакторы связей* также осуществляют объединение программ, однако сформированный ими готовый к выполнению модуль записывается во внешнюю память для последующего использования.

*Микропрограммирование* – это написание программ, которые управляют элементарными операциями аппаратуры; микропрограммирование играет исключительно важную роль в современных архитектурах компьютеров и операционных системах. *Динамическое микропрограммирование* предусматривает возможность простой загрузки новых микропрограмм в управляющую память для непосредственного выполнения.

*Команды вертикального микрокода* очень похожи на команды машинного языка; типичная вертикальная микрокоманда указывает, что необходимо выполнить определенную операцию с определенными данными. *Горизонтальный микрокод* содержит команды с более широкими возможностями – они позволяют задавать одновременное выполнение многих операций над многими элементами данных.

Микропрограммирование зачастую применяется, при *эмуляции*, позволяя сделать один компьютер функциональным эквивалентом другого компьютера. Эмуляция особенно необходима и полезна в тех случаях, когда

пользователям приходится переводить свои программы с машины на машину.

Микропрограммирование позволяет реализовать *микродиагностику*, т. е. контроль ошибок с гораздо большей степенью детализации, чем это возможно для команд машинного языка.

Благодаря микропрограммированию можно специализировать компьютер применительно к требованиям конкретных пользователей. Поставщики компьютеров обычно предлагают факультативные средства микропрограммной поддержки, позволяющие повысить скоростные характеристики; реализация часто выполняемых последовательностей команд при помощи микрокода сокращает время их выполнения. Многие функции операционных систем в современных вычислительных машинах реализуются не обычными программами, а микропрограммами, и благодаря этому, как правило, обеспечивается более высокая скорость выполнения и более надежная защита.

## 2. Классификация операционных систем

### 2.1. Способы классификации операционных систем

ОС классифицируются по следующим признакам:

1) по *количеству пользователей,* одновременно обслуживаемых системой;

2) по *числу процессов,* которые могут одновременно выполняться под управлением ОС;

3) по *типу доступа* пользователя к ЭВМ;

4) по *типу средств вычислительной техники,* для управления ресурсами которых система предназначена;

5) по *поддерживаемым режимам работы* ЭВМ;

6) по *функциональным возможностям*.

В соответствии с *первым признаком* различают *однопользовательские* и *многопользовательские ОС.* Многопользовательские системы поддерживают одновременную работу на ЭВМ нескольких пользователей (конечно, за различными терминалами).

*Второй признак* делит ОС на *однозадачные* и *многозадачные.* Заметим, что если система многопользовательская, то обычно она и многозадачная, но не наоборот.

В соответствии с *третьим* признаком ОС делятся на:

*системы с пакетной обработкой,* когда из программ, подлежащих выполнению, формируется пакет, который предъявляется ЭВМ. В этом случае пользователи непосредственно с ОС не взаимодействуют. Данный тип

ОС предназначен для наиболее эффективного использования ресурсов ЭВМ;

*системы разделения времени,* обеспечивающие одновременный диалоговый (интерактивный) доступ к ЭВМ нескольких пользователей через терминалы. Ресурсы ЭВМ выделяются при этом каждому пользователю “по очереди” в соответствии с той или иной дисциплиной обслуживания. Этот тип ОС предназначен для обеспечения удобства работы группы пользователей;

*системы реального времени,* которые должны обеспечивать гарантированное время ответа на внешние события. Такие ОС служат для управления внешними по отношению к ЭВМ процессами и объектами.

По *четвертому* признаку ОС делятся на *однопроцессорные, многопроцессорные, сетевые* и *распределенные.*

ОС не могут, как правило, предоставить пользователям возможности, которыми не обладает ЭВМ. Они в состоянии только эффективно использовать аппаратные средства компьютера. Поэтому мы сначала перечислим возможные режимы работы ЭВМ, чтобы понять, какими типами ОС они могут комплектоваться.

В настоящее время ЭВМ поддерживают широкий спектр *режимов работы,* среди которых:

1) однопрограммный режим;

2) однопользовательский многопрограммный, или просто многопрограммный режим;

3) многопользовательский многопрограммный или просто многопрограммный режим;

4) система виртуальных машин (дальнейшее развитие мультипрограммирования, основным признаком которого является возможность одновременной работы нескольких ОС.

С точки зрения микропроцессора режимы 2 и 3 близки друг к другу, но для обеспечения последнего необходимо наличие нескольких терминалов (дисплеев и клавиатур). Многопрограммные режимы могут реализовываться как на одно-, так и на многопроцессорных ЭВМ. Для поддержки перечисленных режимов работы ЭВМ существуют следующие *типы* ОС

1) однопользовательские однозадачные, или просто однозадачные;

2) однопользовательские многозадачные, или просто многозадачные;

3) многопользовательские многозадачные, или просто многопользовательские.

Для обеспечения работы ЭВМ в режиме системы виртуальных машин необходим монитор виртуальных машин.

При рассмотрении режимов работы ЭВМ и ОС не случайно использовались различные термины – соответственно “программа” и “задача”. Здесь необходимо сделать несколько дополнительных пояснений.

На аппаратном уровне случаи одновременного выполнения последовательностей команд нескольких программ или одной программы неразли-

чимы. Понятие же “задача” вообще не вводится, а посему можно использовать лишь термин “программа”, понимая под многопрограммностью способность одновременного (при наличии одного процессора – только попеременного) выполнения нескольких последовательностей команд.

На уровне же ОС дело обстоит несколько иначе: считается, что система организует выполнение задачи, формируемой из целой программы или из логически законченного фрагмента программ. Поэтому в данном случае правомерно говорить об одно- или многозадачности.

Для многопользовательских и многозадачных ОС важным показателем является *дисциплина обслуживания.* В соответствии с этим различают вытесняющий и согласующий режимы многозадачной работы.

При *вытесняющей* организации выделением задачам процессорного времени занимается исключительно ОС. Примерами такого режима являются *квантование,* когда каждой задаче процессорное время выделяется по очереди, причем на фиксированный промежуток времени, и *приоритетное обслуживание.*

В случае *согласующей* организации каждая задача, получившая управление, сама определяет, когда ей отдать процессор другой задаче. Иначе говоря, здесь инициатива исходит не от ОС, а главным образом от самой задачи. В общем случае согласование эффективнее и надежнее вытеснения, так как позволяет программе самой выбирать удобный и безопасный момент своего прерывания. Однако при этом ни одна из программ не должна узурпировать процессор, добровольно отказываясь от монопольного его использования.

## 2.2. Классификация операционных систем по функциональным возможностям

В п. 1.2 были определены функции, которые должна выполнять операционная система. По тому, какие из перечисленных функций реализованы и каким было уделено больше внимания, а каким меньше, системы можно разделить на несколько классов (рис. 1.2).

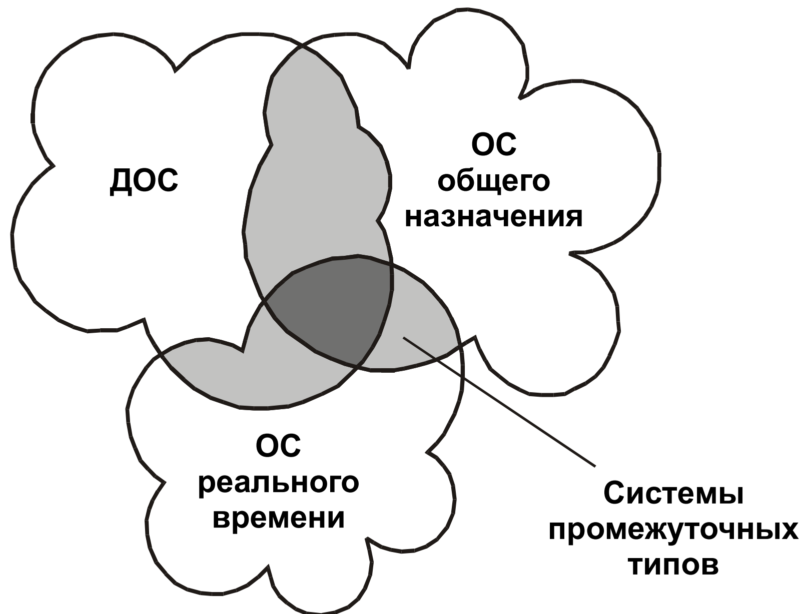



Рис. 1.2. Классификация операционных систем

### 2.2.1. ДОС (Дисковые Операционные Системы)

Это системы, берущие на себя выполнение только первых четырех функций. Как правило, они представляют собой некий резидентный набор подпрограмм, не более того. ДОС загружает пользовательскую программу в память и передает ей управление, после чего программа делает с системой все, что ей заблагорассудится. При завершении программы считается хорошим тоном оставлять машину в таком состоянии, чтобы ДОС могла продолжить работу. Если же программа приводит машину в какое-то другое состояние, что ж, ДОС ничем ей в этом не может помешать.

Характерный пример – различные загрузочные мониторы для машин класса Spectrum. Как правило, такие системы работают одновременно только с одной программой. Дисковая операционная система MS DOS для IBM PC-совместимых машин является прямым наследником одного из таких резидентных мониторов.

Существование систем этого класса обусловлено их простотой и тем, что они потребляют мало ресурсов. Еще одна причина, по которой такие системы могут использоваться даже на довольно мощных машинах – требование программной совместимости с ранними моделями того же семейства компьютеров.

### 2.2.2. ОС общего назначения

К этому классу относятся системы, берущие на себя выполнение всех перечисленных функций. Разделение на ОС общего назначения и ДОС

идет, по-видимому, от систем IBM DOS/360 и OS/360 для больших компьютеров этой фирмы, клоны которых известны у нас в стране под названием ЕС ЭВМ серии 10XX (у IBM была еще TOS/360, Tape Operating System – Ленточная Операционная Система).

Далее под ОС будут подразумеваться системы “общего назначения”, т. е. рассчитанные на интерактивную работу одного или нескольких пользователей в режиме разделения времени, при не очень жестких требованиях ко времени реакции системы на внешние события. Как правило, в таких системах уделяется большое внимание защите самой системы, программного обеспечения и пользовательских данных от ошибочных и злонамеренных программ и пользователей. Обычно подобные системы используют встроенные в архитектуру процессора средства защиты и виртуализации памяти. К этому классу относятся такие широко распространенные системы, как Windows 2000, XP, системы семейства Unix.

### 2.2.3. Системы реального времени

Это системы, предназначенные для облегчения разработки и управления так называемыми приложениями *реального времени* (РВ) – программами, управляющих некомпьютерным оборудованием, часто с очень жесткими ограничениями по времени. Примером таких систем являются операционные системы бортовых ЭВМ образцов вооружения. Подобные системы обязаны поддерживать многопоточность, гарантированное время реакции на внешнее событие, простой доступ к таймеру и внешним устройствам.

Способность гарантировать время реакции является отличительным признаком систем РВ**.** Важно учитывать различие между гарантированностью и просто высокой производительностью. Далеко не все алгоритмы и технические решения, даже и обеспечивающие отличное среднее время реакции, годятся для приложений и операционных систем РВ**.**

По другим признакам эти системы могут относиться как к классу ДОС (RT-11), так и к ОС (OS-9, QNX).

Так называемое “мягкое реальное время”, предоставляемое современными Win32 платформами, не является реальным временем вообще. Система “мягкого РВ” обеспечивает не гарантированное, а всего лишь среднее время реакции. Для мультимедийных приложений и игр различие между “средним” и “гарантированным” не очень критично. Но для промышленных приложений, где необходимо настоящее реальное время, это обычно неприемлемо.

### 2.2.4. Средства кросс-разработки

Это системы, предназначенные для разработки программ в двухмашинной конфигурации, когда редактирование, компиляция, а зачастую и отладка кода производятся на инструментальной машине, а потом скомпилированный код загружается в целевую систему. Чаще всего они используются для написания и отладки программ, позднее прошиваемых в ПЗУ. Примерами таких ОС являются системы программирования микроконтроллеров Intel, Atmel, PIC и др., системы VxWorks, Windows CE, Palm OS и т. д. Для разработки программного обеспечения для вычислительных систем комплексов вооружения войсковой ПВО используется инструментальная среда Tornado, работающая под управлением ОС реального времени VxWorks.

Такие системы, как правило, включают в себя:

набор компиляторов и ассемблеров, работающих на инструментальной машине с “нормальной” ОС;

библиотеки, выполняющие большую часть функций ОС при работе программы (но не загрузку этой программы!);

средства отладки.

### 2.2.5. Системы виртуальных машин

Такие системы стоят несколько особняком. Система виртуальных машин (СВМ) – это ОС, допускающая одновременную работу нескольких программ, но создающая при этом для каждой программы иллюзию того, что машина находится в полном ее распоряжении, как при работе под управлением ДОС. Зачастую, “программой” оказывается полноценная операционная система – примерами таких систем являются VMWare для машин с архитектурой x86 или VM для System/370 и ее потомков.

Виртуальные машины являются ценным средством при разработке и тестировании кросс-платформенных приложений. Реже они используются для отладки модулей ядра или самой операционной системы.

Такие системы отличаются высокими накладными расходами и сравнительно низкой надежностью, поэтому относительно редко находят промышленное применение.

Часто СВМ являются подсистемой ОС общего назначения: MS DOS и MS Windows-эмуляторы для UNIX и OS/2, подсистема WoW в Windows NT/2000/XP, сессия DOS в Windows 3.x/95/98/ME, эмулятор RT-11 в VAX/VMS.

### 2.2.6. Системы промежуточных типов

Существуют системы, которые нельзя отнести к одному из вышеперечисленных классов. Такова, например, система RT-11, которая, по сути своей, является ДОС, но позволяет одновременное исполнение нескольких программ с довольно богатыми средствами взаимодействия и синхронизации. Другим примером промежуточной системы является Windows 95, которые, как ОС, используют аппаратные средства процессора для защиты и виртуализации памяти и даже могут обеспечивать некоторое подобие многозадачности, но не защищают себя и программы от ошибок других программ, подобно ДОС.

Некоторые системы реального времени, например QNX, могут использоваться как в качестве самостоятельной ОС, загружаемой с жесткого диска в оперативную память, так и будучи прошиты в ПЗУ. Эти системы могут быть отнесены одновременно и к ОС общего назначения, и к системам кросс-разработки.

Таких примеров “гибридизации” можно привести множество, поэтому к вышеприведенной классификации следует относиться с определенной осторожностью.

## 2.3. Семейства операционных систем

Часто можно проследить преемственность между различными ОС, необязательно разработанными одной компанией. Отчасти такая преемственность обусловлена требованиями совместимости или хотя бы переносимости прикладного программного обеспечения, отчасти – заимствованием отдельных удачных концепций.

На основании такой преемственности можно выстроить “генеалогические деревья” операционных систем и – с той или иной обоснованностью – объединять их в семейства. Следует заметить, что граф родства ОС не является деревом и нередко содержит циклы, поэтому бесспорной многоуровневой классификации, охватывающей всю техносферу, выстроить не удается.

Тем не менее, с достаточно большой уверенностью можно выделить минимум три семейства ныне эксплуатирующихся ОС и еще несколько – вымерших или близких к тому. Три ныне процветающих семейства:

1. Операционные системы для больших компьютеров фирмы IBM – OS/390, z/OS и IBM VM.

2. Обширное, бурно развивающееся и имеющее трудно определимые границы семейство Unix. Прежде всего в него входят ОС трех основных родов:

*Unix System V Release 4.x*: SunSoft Solaris, SCO UnixWare;
*Berkeley Software Distribution Unix*: BSDI, FreeBSD;
*Linux*.

3. Семейство прямых и косвенных потомков Control Program/Monitor (CP/M) фирмы Digital Research. В этом семействе можно выделить также весьма широко известное подсемейство Win32-платформ.

Еще одно практически вымершее к настоящему моменту, но оставившее в наследство ряд важных и интересных концепций семейство – это операционные системы для мини- и микрокомпьютеров фирмы DEC: RT-11, RSX-11 и VAX/VMS.

Ряд систем, в том числе и коммерчески успешных, не могут быть с уверенностью отнесены ни к одному из перечисленных семейств, поэтому к данной классификации надо относиться с осторожностью.

Выбор типа операционной системы часто представляет собой нетривиальную задачу. Некоторые приложения накладывают жесткие требования, которым удовлетворяет только небольшое количество систем. Например, задачи управления промышленным или исследовательским оборудованием в режиме жесткого реального времени вынуждают делать выбор между специализированными ОС реального времени и некоторыми ОС общего назначения, такими как Unix System V Release 4 (хотя Unix SVR4 теоретически способна обеспечивать гарантированное время реакции, системы этого семейства имеют ряд недостатков с точки зрения задач РВ, поэтому чаще всего предпочтительными оказываются специализированные ОС – QNX, VxWorks, OS-9 и т. д.). Другие приложения, например серверы баз данных, просто требуют высокой надежности и производительности, что отсекает системы класса ДОС и MS Windows.

Наконец, некоторые задачи, такие как автоматизация офисной работы в небольших организациях, не предъявляют высоких требований к надежности, производительности и времени реакции системы, что предоставляет широкий выбор между различными ДОС, MS Windows, Mac OS и многими системами общего назначения. При этом технические параметры системы перестают играть роль, и в игру вступают другие факторы. На заре развития персональной техники таким фактором была стоимость аппаратного обеспечения, вынуждавшая делать выбор в пользу ДОС и, позднее, MS Windows.

Нужно отметить, впрочем, что современные версии Windows, несмотря на низкую надежность, сложность конфигурации и поддержки и ряд функциональных недостатков, вполне адекватны большинству задач офисной автоматизации. Основная проблема MS Windows состоит в том, что она не обеспечивает путей плавного и безболезненного перехода к другим платформам, даже если возникнет необходимость такого перехода. Строго говоря, тот же недостаток свойственен многим другим закрытым (closed) платформам, поставляемым одной фирмой и использующим нестандартные “фирменные” интерфейсы. Пока “закрытое” решение соответствует

предъявляемым требованиям, все хорошо, но когда требования выходят за пределы технологических возможностей данного решения, проблема заходит в тупик.

Альтернативой закрытым решениям является концепция открытых систем. Идея открытых систем исходит из того, что для разных задач необходимы разные системы – как специализированные, так и системы общего назначения, просто по-разному настроенные и сбалансированные. Сложность состоит в том, чтобы обеспечить:

*взаимодействие* разнородных систем в гетерогенной сети;

*обмен данными* между различными приложениями на разных платформах;

*переносимость* прикладного ПО с одной платформы на другую, хотя бы путем перекомпиляции исходных текстов;,

по возможности *однородный пользовательский интерфейс*.

Эти задачи предполагается решать при помощи открытых стандартов – стандартных сетевых протоколов, стандартных форматов данных, стандартизации программных интерфейсов – *API (Application Program Interface,* интерфейс прикладных программ) и, наконец, стандартизации пользовательского интерфейса.

В качестве стандартного сетевого протокола предлагалась семиуровневая модель OSI**,** но прежде, чем на основе этой модели было разработано, что-то полезное, получило широкое распространение семейство протоколов TCP/IP. Документация по протоколам этого семейства имеет статус *public domain (общественная собственность)*; кроме того, есть, по крайней мере, одна программная реализация этого протокола, также имеющая статус public domain – сетевое ПО системы BSD Unix, это стало вполне приемлемым основанием для применения TCP/IP в открытых системах.

Для того чтобы как-то обеспечить переносимость программ между системами различных типов, принимались различные стандарты интерфейса между пользовательской программой и ОС. Одним из первых таких стандартов был стандарт библиотек ANSI C. Он основан на системных вызовах ОС Unix, но функции MS DOS для работы с файлами тоже достаточно близки к этому стандарту.

Позднее делалось еще несколько попыток стандартизировать интерфейс системных вызовов. Одной из относительно удачных попыток такого рода был POSIX (Portable Operating System Interface [based on] uniX – переносимый интерфейс операционной системы, основанный на Unix), который в той или иной форме поддерживается всеми системами семейства Unix и некоторыми ОС, не входящими в это семейство, например Windows NT. Но наибольший успех имела деятельность консорциума X/Open, который в 1998 году сертифицировал операционную систему OS/390 фирмы IBM как соответствующую спецификациям Unix/95 и, таким образом, дал представителю самого древнего из современных родов операционных систем право называться UNIX ™.

# 3. Принципы построения операционных систем

## 3.1. Основные принципы построения ОС

Операционные системы различаются по назначению, выполняемым функциям, формам реализации. В этом смысле каждая ОС – это уникальная сложная программная система. Одновременно можно говорить о тождественном равенстве ОС в смысле использования некоторых общих принципов, которые положены в основу их разработки.

*Принцип модульности.* Этот принцип в равной степени отражает технологические и эксплуатационные свойства. Причем наибольший эффект от его использования достижим в случае, когда принцип распространен одновременно на операционную систему, прикладные программы и аппаратуру. Под модулем в общем случае понимают функциональный элемент рассматриваемой системы, имеющий оформление, законченное и выполненное в пределах требований системы, и средства сопряжения с подобными элементами или элементами более высокого уровня данной или другой системы. По своему определению модуль предполагает легкий способ его замены на другой при наличии заданных интерфейсов. Способы обособления составных частей ОС в отдельные модули могут быть существенно различными. Чаще всего разделение происходит по функциональному признаку. В значительной степени разделение системы на модули определяется используемым методом проектирования ОС.

Как правило, модули строят с учетом характера их будущего использования. Модуль может быть построен так, что после очередного своего исполнения становится непригоден для последующего своего исполнения. Это происходит, например, если при исполнении были изменены и не восстановлены в исходное состояние либо совсем исчезли некоторые команды или данные модуля. Так, при работе двухпроходного транслятора его часть, реализующая первый проход, вызывает и загружает на место своего расположения в оперативной памяти другую часть транслятора, реализующую второй проход. Очевидно, что если такой транслятор кончил работать и возникла необходимость повторной работы с ним, то потребуется повторно сделать копирование исходного текста транслятора в оперативную память. Модули, которые могут испортить сами себя после окончания работы и не восстанавливаются в исходное состояние, называют однократными. Если же модули могут испортить себя, но в конце работы восстанавливаются, то их называют многократными.

Особо важное значение при построении ОС имеют модули, называемые параллельно используемыми или реентерабельными. Каждый реентерабельный модуль можно использовать параллельно (одновременно) не-

сколькими программами при их исполнении. Пусть две программы исполняются одновременно на разных процессорах в мультипроцессорной системе с обобщенной оперативной памятью. В каждой программе есть обращение к некоторой подпрограмме *P,* выполненной в форме реентерабельного модуля. Тогда достаточно в оперативной памяти иметь единственную копию подпрограммы *P,* к которой допустим одновременный доступ для исполнения двух названных программ.

*Принцип функциональной избирательности.* Этот принцип является логическим продолжением модульного принципа. В ОС выделяется некоторая часть важных модулей, которые должны быть постоянно “под рукой” для эффективной организации вычислительного процесса. Эту часть в ОС называют ядром, так как это действительно основа системы. При формировании состава ядра требуется удовлетворить двум противоречивым требованиям. В состав ядра должны войти используемые наиболее часто системные модули. Количество модулей должно быть таковым, чтобы объем памяти, занимаемый ядром, был бы не слишком большим. В его состав, как правило, входят модули по управлению системой прерываний, средства по переводу программ из состояния счета в состояние ожидания, готовности и обратно, средства по распределению таких основных ресурсов, как оперативная память и процессор. Программы, входящие в состав ядра, помещают перед работой ОС в оперативную память, где они постоянно хранятся и доступны для использования по мере функционирования системы. Такие программы называют резидентными. Помимо них существуют транзитные системные программы, которые постоянно хранятся в памяти на магнитных дисках, загружаются в оперативную память только при необходимости и в случае отсутствия свободного пространства могут перекрывать друг друга.

*Принцип генерируемости.* Основное положение этого принципа определяет такой способ исходного представления системной программы ОС, который позволял бы настраивать эту системную программу исходя из конкретной конфигурации конкретной машины и круга решаемых проблем. Эта процедура проводится редко перед достаточно протяженным периодом эксплуатации ОС. Процесс генерации осуществляется с помощью специальной программы-генератора и входного языка для этой программы, позволяющего описывать программные возможности системы и конфигурацию машины. В результате генерации получается полная версия ОС. Она чаще всего представлена на магнитной ленте, которую в данном случае называют дистрибутивной. Сгенерированная версия ОС представляет собой совокупность системных наборов модулей и данных.

Рассмотренный принцип модульности положительно проявляется при генерации ОС. Он существенно упрощает настройку ОС на требуемую конфигурацию вычислительной системы. В наши дни при использовании персональных компьютеров с принципом генерируемости ОС можно столкнуться разве что только при работе с Linux. В этой UNIX-системе

имеется возможность не только использовать какое-то готовое ядро ОС, но и самому сгенерировать такое ядро, которое будет оптимальным для данного конкретного персонального компьютера и решаемых на нем задач. Кроме генерации ядра в Linux имеется возможность указать и набор подгружаемых драйверов и служб, то есть часть функций может реализовываться модулями, непосредственно входящими в ядро системы, а часть модулями, имеющими статус подгружаемых, транзитных.

В остальных современных распространенных ОС для персональных компьютеров конфигурирование ОС под соответствующий состав оборудования осуществляется на этапе инсталляции, а потом состав драйверов и изменение некоторых параметров ОС может быть осуществлено посредством редактирования конфигурационного файла.

*Принцип функциональной избыточности.* Этот принцип учитывает возможность проведения одной и той же работы различными средствами. В состав ОС может входить несколько типов мониторов. *Монитор* – это управляющая программа ОС. Каждый монитор предполагает свою организацию обработки пользовательских программ и альтернативен другим. Наличие нескольких типов мониторов позволяет пользователям быстро адаптировать ОС к определенной конфигурации машины, обеспечить максимально эффективную загрузку технических средств при решении конкретного класса задач.

*Принцип виртуализации.* Этот принцип позволяет представить структуру системы в виде определенного набора планировщиков процессов и распределителей ресурсов (мониторов) и использовать единую централизованную схему распределения ресурсов.

Наиболее естественным и законченным проявлением концепции виртуальности является понятие *виртуальной* машины. По сути, любая операционная система, являясь средством распределения ресурсов и организуя по определенным правилам управление процессами, скрывает от пользователя и его приложений реальные аппаратные и иные ресурсы, заменяя их некоторой абстракцией. В результате пользователи видят и используют виртуальную машину как некое устройство, способное воспринимать их программы, написанные на определенном языке программирования, выполнять их и выдавать результаты. При таком языковом представлении пользователя совершенно не интересует реальная конфигурация вычислительной системы, способы эффективного использования ее компонентов и подсистем. Он мыслит и работает с машиной в терминах используемого им языка и тех ресурсов, которые ему предоставляются в рамках виртуальной машины.

*Принцип независимости программ от внешних устройств.* Этот принцип позволяет одинаково осуществлять операции управления внешними устройствами независимо от их конкретных физических характеристик. Принцип заключается в том, что связь программ с конкретными устройствами производится не на уровне трансляции программы, а в период пла-

нирования ее исполнения. Перекомпиляция при работе программы с новым устройством, на котором располагаются данные, не требуется. Например, программе, содержащей операции обработки последовательного набора данных, безразлично на каком носителе эти данные будут располагаться. Смена носителя и данных, размещаемых на них (при неизменности структурных характеристик данных), не принесет каких-либо изменений в программу, если в системе реализован принцип независимости. Наиболее последовательно данный принцип реализован в ОС UNIX.

*Принцип совместимости.* Одним из аспектов совместимости является способность ОС выполнять программы, написанные для других ОС или для более ранних версий данной операционной системы, а также для другой аппаратной платформы.

Одним из средств обеспечения совместимости программных и пользовательских интерфейсов является соответствие стандартам POSIX. Использование стандарта POSIX позволяет создавать программы в стиле UNIX, которые впоследствии могут легко переноситься из одной системы в другую.

*Принцип открытой и наращиваемой ОС.* Открытая ОС доступна для анализа как пользователям, так и системным специалистам, обслуживающим ЭВМ. Наращиваемая (модифицируемая, развиваемая) ОС позволяет не только использовать возможности генерации, но и вводить в ее состав новые модули, совершенствовать существующие и т. д. К открытым ОС, прежде всего, следует отнести UNIX-системы и, естественно, ОС Linux.

*Принцип мобильности (переносимости).* Операционная система относительно легко должна переноситься с процессора одного типа на процессор другого типа и с аппаратной платформы (которая включает наряду с типом процессора и способ организации всей аппаратуры компьютера, иначе говоря, архитектуру вычислительной системы) одного типа на аппаратную платформу другого типа. Принцип переносимости очень близок принципу совместимости, хотя это и не одно и то же.

Введение стандартов POSIX преследовало цель обеспечить переносимость создаваемого программного обеспечения.

*Принцип обеспечения безопасности вычислений.* Этот принцип определяет необходимость разработки мер, ограждающих программы и данные пользователей от искажений или нежелательных влияний друг на друга, а также пользователей на ОС и наоборот. Программы должны быть гарантированно защищены как при своем исполнении, так и в режиме хранения. Способов влияния, как умышленных, так и неумышленных, чрезвычайно много. Не все их можно предотвратить чисто техническими средствами. Особенно трудно обеспечить защиту, когда используется разделение ресурсов. Несмотря на сложность используемых механизмов, принцип защиты реализуется в той или иной форме в каждой мультипрограммной ОС.

## 3.2. Функциональные компоненты операционной системы

Функции операционной системы автономного компьютера обычно группируются либо в соответствии с типами локальных ресурсов, которыми управляет ОС, либо в соответствии со специфическими задачами, применимыми ко всем ресурсам. Иногда такие группы функций называют подсистемами. Наиболее важными подсистемами управления ресурсами являются подсистемы управления процессами, памятью, файлами и внешними устройствами, а подсистемами, общими для всех ресурсов, являются подсистемы пользовательского интерфейса, защиты данных и администрирования.

### 3.2.1. Управление процессами

Важнейшей частью операционной системы, непосредственно влияющей на функционирование вычислительной машины, является подсистема управления процессами.

Для каждого вновь создаваемого процесса ОС генерирует системные информационные структуры, которые содержат данные о потребностях процесса в ресурсах вычислительной системы, а также о фактически выделенных ему ресурсах. Таким образом, процесс можно также определить как некоторую заявку на потребление системных ресурсов.

Чтобы процесс мог быть выполнен, операционная система должна назначить ему область оперативной памяти, в которой будут размещены коды и данные процесса, а также предоставить ему необходимое количество процессорного времени. Кроме того, процессу может понадобиться доступ к таким ресурсам, как файлы и устройства ввода-вывода.

В информационные структуры процесса часто включаются вспомогательные данные, характеризующие историю пребывания процесса в системе (например, какую долю времени процесс потратил на операции ввода-вывода, а какую на вычисления), его текущее состояние (активное или заблокированное), степень привилегированности процесса (значение приоритета). Данные такого рода могут учитываться операционной системой при принятии решения о предоставлении ресурсов процессу.

В мультипрограммной операционной системе одновременно может существовать несколько процессов. Часть процессов порождается по инициативе пользователей и их приложений, такие процессы обычно называют пользовательскими. Другие процессы, называемые системными, инициализируются самой операционной системой для выполнения своих функций.

Поскольку процессы часто одновременно претендуют на одни и те же ресурсы, то в обязанности ОС входит поддержание очередей заявок про-

цессов на ресурсы, например очереди к процессору, к принтеру, к последовательному порту.

Важной задачей операционной системы является защита ресурсов, выделенных данному процессу, от остальных процессов. Одним из наиболее тщательно защищаемых ресурсов процесса являются области оперативной памяти, в которой хранятся коды и данные процесса. Совокупность всех областей оперативной памяти, выделенных операционной системой процессу, называется его адресным пространством. Говорят, что каждый процесс работает в своем адресном пространстве, имея в виду защиту адресных пространств, осуществляемую ОС. Защищаются и другие типы ресурсов, такие как файлы, внешние устройства и т. д. Операционная система может не только защищать ресурсы, выделенные одному процессу, но и организовывать их совместное использование, например разрешать доступ к некоторой области памяти нескольким процессам.

На протяжении периода существования процесса его выполнение может быть многократно прервано и продолжено. Для того чтобы возобновить выполнение процесса, необходимо восстановить состояние его операционной среды. Состояние операционной среды идентифицируется состоянием регистров и программного счетчика, режимом работы процессора, указателями на открытые файлы, информацией о незавершенных операциях ввода-вывода, кодами ошибок выполняемых данным процессом системных вызовов и т. д. Эта информация называется контекстом процесса. Говорят, что при смене процесса происходит переключение контекстов.

Операционная система берет на себя также функции синхронизации процессов, позволяющие процессу приостанавливать свое выполнение до наступления какого-либо события в системе, например завершения операции ввода-вывода, осуществляемой по его запросу операционной системой.

В операционной системе нет однозначного соответствия между процессами и программами. Один и тот же программный файл может породить несколько параллельно выполняемых процессов, а процесс может в ходе своего выполнения сменить программный файл и начать выполнять другую программу.

Для реализации сложных программных комплексов полезно бывает организовать их работу в виде нескольких параллельных процессов, которые периодически взаимодействуют друг с другом и обмениваются некоторыми данными. Так как операционная система защищает ресурсы процессов и не позволяет одному процессу писать или читать из памяти другого процесса, то для оперативного взаимодействия процессов ОС должна предоставлять особые средства, которые называют средствами межпроцессного взаимодействия.

Таким образом, подсистема управления процессами планирует выполнение процессов, то есть распределяет процессорное время между не-

сколькими одновременно существующими в системе процессами, занимается созданием и уничтожением процессов, обеспечивает процессы необходимыми системными ресурсами, поддерживает синхронизацию процессов, а также обеспечивает взаимодействие между процессами.

### 3.2.2. Управление памятью

Память является для процесса таким же важным ресурсом, как и процессор, так как процесс может выполняться процессором только в том случае, если его коды и данные (не обязательно все) находятся в оперативной памяти.

Управление памятью включает распределение имеющейся физической памяти между всеми существующими в системе в данный момент процессами, загрузку кодов и данных процессов в отведенные им области памяти, настройку адресно-зависимых частей кодов процесса на физические адреса выделенной области, а также защиту областей памяти каждого процесса.

Существует большое разнообразие алгоритмов распределения памяти. Они могут отличаться, например, количеством выделяемых процессу областей памяти, степенью свободы границы областей. В некоторых системах распределение памяти выполняется страницами фиксированного размера, а в других – сегментами переменной длины.

Одним из наиболее популярных способов управления памятью в современных операционных системах является так называемая виртуальная память. Наличие в ОС механизма виртуальной памяти позволяет программисту писать программу так, как будто в его распоряжении имеется однородная оперативная память большого объема, часто существенно превышающего объем имеющейся физической памяти. В действительности все данные, используемые программой, хранятся на диске и при необходимости частями (сегментами или страницами) отображаются в физическую память. При перемещении кодов и данных между оперативной памятью и диском подсистема виртуальной памяти выполняет трансляцию виртуальных адресов, полученных в результате компиляции и компоновки программы, в физические адреса ячеек оперативной памяти. Очень важно, что все операции по перемещению кодов и данных между оперативной памятью и дисками, а также трансляция адресов выполняются ОС прозрачно для программиста.

Защита памяти – это избирательная способность предохранять выполняемую задачу от записи или чтения памяти, назначенной другой задаче. Правильно написанные программы не пытаются обращаться к памяти, назначенной другим. Однако реальные программы часто содержат ошибки, в результате которых такие попытки иногда предпринимаются. Средства

защиты памяти, реализованные в операционной системе, должны пресекать несанкционированный доступ процессов к чужим областям памяти.

Таким образом, функциями ОС по управлению памятью являются отслеживание свободной и занятой памяти; выделение памяти процессам и освобождение памяти при завершении процессов; защита памяти; вытеснение процессов из оперативной памяти на диск, когда размеры основной памяти недостаточны для размещения в ней всех процессов, и возвращение их в оперативную память, когда в ней освобождается место, а также настройка адресов программы на конкретную область физической памяти.

### 3.2.3. Управление файлами и внешними устройствами

Способность ОС к “экранированию” сложностей реальной аппаратуры очень ярко проявляется в одной из основных подсистем ОС – файловой системе. Операционная система виртуализирует отдельный набор данных, хранящихся на внешнем накопителе, в виде файла – простой неструктурированной последовательности байтов, имеющей символьное имя. Для удобства работы с данными файлы группируются в каталоги, которые, в свою очередь, образуют группы – каталоги более высокого уровня. Пользователь может с помощью ОС выполнять над файлами и каталогами такие действия, как поиск по имени, удаление, вывод содержимого на внешнее устройство (например, на дисплей), изменение и сохранение содержимого.

Чтобы представить большое количество наборов данных, разбросанных случайным образом по цилиндрам и поверхностям дисков различных типов, в виде хорошо всем знакомой и удобной иерархической структуры файлов и каталогов, операционная система должна решить множество задач. Файловая система ОС выполняет преобразование символьных имен файлов, с которыми работает пользователь или прикладной программист, в физические адреса данных на диске, организует совместный доступ к файлам, защищает их от несанкционированного доступа.

При выполнении своих функций файловая система тесно взаимодействует с подсистемой управления внешними устройствами, которая по запросам файловой системы осуществляет передачу данных между дисками и оперативной памятью.

Подсистема управления внешними устройствами, называемая также подсистемой ввода-вывода, исполняет роль интерфейса ко всем устройствам, подключенным к компьютеру. Спектр этих устройств очень обширен. Номенклатура выпускаемых накопителей на жестких, гибких и оптических дисках, принтеров, сканеров, мониторов, плоттеров, модемов, сетевых адаптеров и более специальных устройств ввода-вывода, таких как, например, аналого-цифровые преобразователи, может насчитывать сотни моделей. Эти модели могут существенно отличаться набором и последо-

вательностью команд, с помощью которых осуществляется обмен информацией с процессором и памятью компьютера, скоростью работы, кодировкой передаваемых данных, возможностью совместного использования и множеством других деталей.

Программа, управляющая конкретной моделью внешнего устройства и учитывающая все его особенности, обычно называется *драйвером* этого устройства (от английского drive – управлять, вести). Драйвер может управлять единственной моделью устройства, или же группой устройств определенного типа. Для пользователя очень важно, чтобы операционная система включала как можно больше разнообразных драйверов, так как это гарантирует возможность подключения к компьютеру большого числа внешних устройств различных производителей.

Созданием драйверов устройств занимаются как разработчики конкретной ОС, так и специалисты компаний, выпускающих внешние устройства. Операционная система должна поддерживать хорошо определенный интерфейс между драйверами и остальной частью ОС, чтобы разработчики из компаний-производителей устройств ввода-вывода могли поставлять вместе со своими устройствами драйверы для данной операционной системы.

Поддержание высокоуровневого унифицированного интерфейса прикладного программирования к разнородным устройствам ввода-вывода является одной из наиболее важных задач ОС. Со времени появления ОС UNIX такой унифицированный интерфейс в большинстве операционных систем строится на основе концепции файлового доступа. Эта концепция заключается в том, что обмен с любым внешним устройством выглядит как обмен с файлом, имеющим имя и представляющим собой неструктурированную последовательность байтов. В качестве файла может выступать как реальный файл на диске, так и алфавитно-цифровой терминал, печатающее устройство или сетевой адаптер.

### 3.2.4. Защита данных и администрирование

Безопасность данных вычислительной системы обеспечивается средствами отказоустойчивости ОС, направленными на защиту от сбоев и отказов аппаратуры и ошибок программного обеспечения, а также средствами защиты от несанкционированного доступа. В последнем случае ОС защищает данные от ошибочного или злонамеренного поведения пользователей системы.

Первым рубежом обороны при защите данных от несанкционированного доступа является процедура логического входа. Операционная система должна убедиться, что в систему пытается войти пользователь, вход которого разрешен администратором. Функции защиты ОС вообще очень тесно связаны с функциями администрирования, так как именно администра-

тор определяет права пользователей при их обращении к разным ресурсам системы – файлам, каталогам, принтерам, сканерам и т. п. Кроме того, администратор ограничивает возможности пользователей в выполнении тех или иных системных действий. Например, пользователю может быть запрещено выполнять процедуру завершения работы ОС, устанавливать системное время, завершать чужие процессы, создавать учетные записи пользователей, изменять права доступа к некоторым каталогам и файлам. Администратор может также урезать возможности пользовательского интерфейса, убрав, например, некоторые пункты из меню операционной системы, выводимого на дисплей пользователя.

Важным средством защиты данных являются функции аудита ОС, заключающиеся в фиксации всех событий, от которых зависит безопасность системы. Например, попытки удачного и неудачного логического входа в систему, операции доступа к некоторым каталогам и файлам, использование принтеров и т. п. Список событий, которые необходимо отслеживать, определяет администратор ОС.

Поддержка отказоустойчивости реализуется операционной системой, как правило, на основе резервирования. Чаще всего в функции ОС входит поддержание нескольких копий данных на разных дисках или разных дисковых накопителях. Резервируются также принтеры и другие устройства ввода-вывода. При отказе одного из избыточных устройств операционная система должна быстро и прозрачным для пользователя образом произвести реконфигурацию системы и продолжить работу с резервным устройством. Особым случаем обеспечения отказоустойчивости является использование нескольких процессоров, то есть мультипроцессирование, когда система продолжает работу при отказе одного из процессоров, хотя и с меньшей производительностью. В вычислительных системах комплексов вооружения обычно используется несколько бортовых ЭВМ (БЭВМ), одна из которых находится в режиме “горячего резерва”, то есть при отказе основной ЭВМ управление автоматически передается резервной.

### 3.2.5. Интерфейс прикладного программирования

Прикладные программисты используют в своих приложениях обращения к ОС, когда для выполнения тех или иных действий им требуется особый статус, которым обладает только операционная система. Например, в большинстве современных ОС все действия, связанные с управлением аппаратными средствами компьютера, может выполнять только ОС. Помимо этих функций прикладной программист может воспользоваться набором сервисных функций ОС, которые упрощают написание приложений. Функции такого типа реализуют универсальные действия, часто требующиеся в различных приложениях, такие, например, как обработка текстовых строк. Эти функции могли бы быть выполнены и самим приложением,

однако гораздо проще использовать уже готовые, отлаженные процедуры, включенные в состав операционной системы. В то же время даже при наличии в ОС соответствующей функции программист может реализовать ее самостоятельно в рамках приложения, если предложенный операционной системой вариант его не вполне устраивает.

Возможности операционной системы доступны прикладному программисту в виде набора функций, называющегося *интерфейсом прикладного программирования* (*Application Programming Interface*, *API*). От конечного пользователя эти функции скрыты за оболочкой алфавитно-цифрового или графического пользовательского интерфейса.

Для разработчиков приложений все особенности конкретной операционной системы представлены особенностями ее API. Поэтому операционные системы с различной внутренней организацией, но с одинаковым набором функций API кажутся им одной и той же ОС, что упрощает стандартизацию операционных систем и обеспечивает переносимость приложений между внутренне различными ОС, соответствующими определенному стандарту на API. Например, следование общим стандартам API UNIX, одним из которых является стандарт POSIX, позволяет говорить о некоторой обобщенной операционной системе UNIX, хотя многочисленные версии этой ОС от разных производителей иногда существенно отличаются внутренней организацией.

Приложения выполняют обращения к функциям API с помощью системных вызовов. Способ реализации системных вызовов зависит от структурной организации ОС, которая, в свою очередь, тесно связана с особенностями аппаратной платформы. Кроме того, он зависит от языка программирования. При использовании ассемблера программист устанавливает значения регистров и/или областей памяти, а затем выполняет специальную инструкцию вызова сервиса или программного прерывания для обращения к некоторой функции ОС. При использовании языков высокого уровня функции ОС вызываются тем же способом, что и написанные пользователем подпрограммы, требуя задания определенных аргументов в определенном порядке.

### 3.2.6. Пользовательский интерфейс

Операционная система должна обеспечивать удобный интерфейс не только для прикладных программ, но и для человека, работающего за терминалом. Этот человек может быть конечным пользователем, администратором ОС или программистом.

В ранних операционных системах пакетного режима функции пользовательского интерфейса были сведены к минимуму и не требовали наличия терминала. Команды языка управления заданиями набивались на перфокарты, а результаты выводились на печатающее устройство.

Современные ОС поддерживают развитые функции пользовательского интерфейса для интерактивной работы за терминалами двух типов: алфавитно-цифровыми и графическими.

При работе за алфавитно-цифровым терминалом пользователь имеет в своем распоряжении систему команд, мощность которой отражает функциональные возможности данной ОС. Обычно командный язык ОС позволяет запускать и останавливать приложения, выполнять различные операции с файлами и каталогами, получать информацию о состоянии ОС (количество работающих процессов, объем свободного пространства на дисках и т. п.), администрировать систему. Команды могут вводиться не только в интерактивном режиме с терминала, но и считываться из так называемого командного файла, содержащего некоторую последовательность команд.

Программный модуль ОС, ответственный за чтение отдельных команд или же последовательности команд из командного файла, иногда называют командным интерпретатором.

Ввод команды может быть упрощен, если операционная система поддерживает графический пользовательский интерфейс. В этом случае пользователь для выполнения нужного действия с помощью мыши выбирает на экране нужный пункт меню или графический символ.

## 3.3. ОС Windows

В качестве примера операционной системы рассмотрим ОС Windows NT. В основу Windows NT положена модульная клиент-серверная модель. Чтобы понять, как взаимодействуют между собой отдельные компоненты операционной системы, прежде всего рассмотрим режимы работы процессора.

### 3.3.1. Привилегированный и пользовательский режимы

В Windows NT предусмотрено два режима работы процессора – *привилегированный* и *пользовательский.* В привилегированном режиме исполняемая программа получает полный доступ к аппаратным средствам и системным данным. Пользовательский режим является непривилегированным. В нем программа имеет ограниченный доступ к данным системы, а к аппаратным средствам обращается через посредников, роль которых играют различные службы ОС. Некоторые компоненты Windows NT (рис. 1.3) работают в привилегированном режиме, другие – в пользовательском.

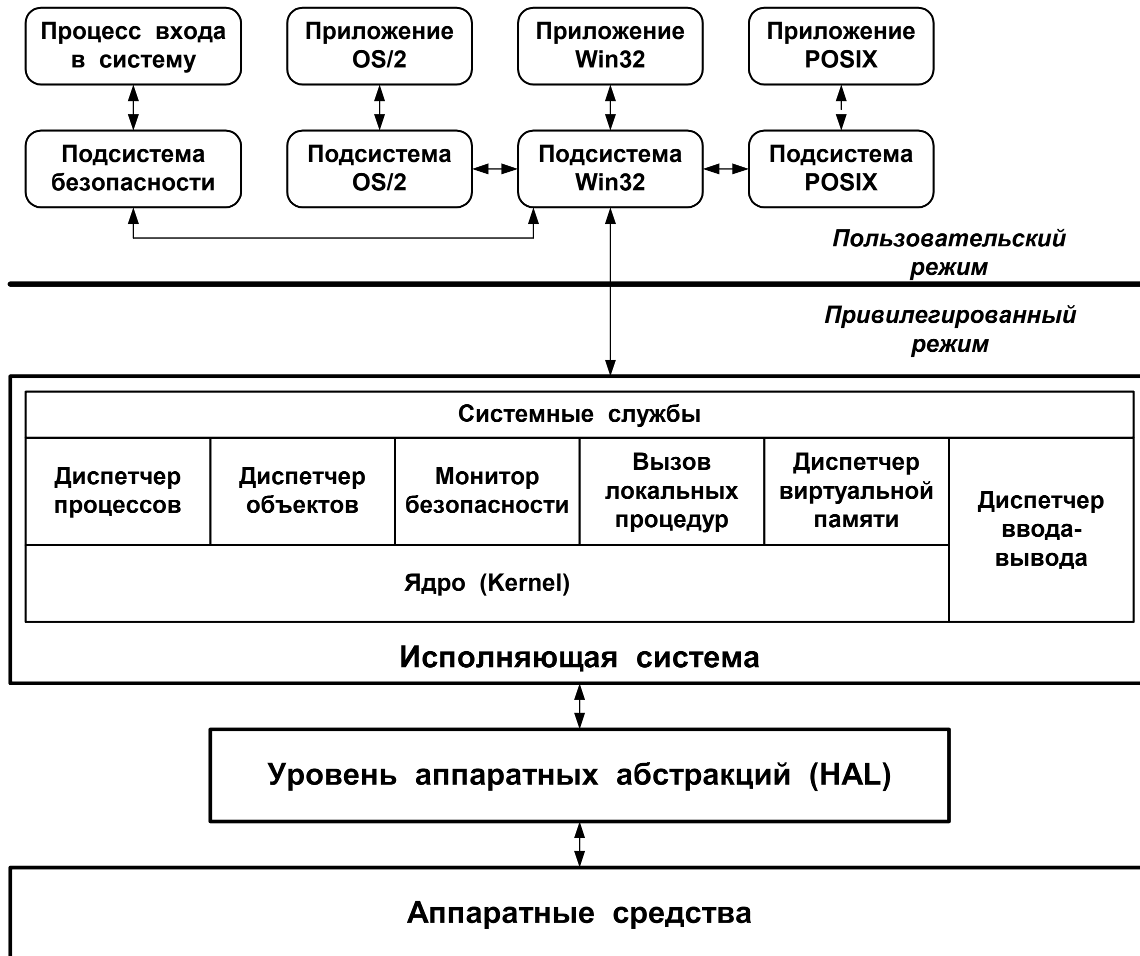



Рис. 1.3. Архитектура Windows NT

Управление объектами, безопасность, вызовы локальных процедур, управление процессами, управление памятью, ввод-вывод – все эти низкоуровневые компоненты составляют *исполняющую систему* Windows NT, которая работает в привилегированном режиме.

В привилегированном режиме функционирует и *уровень абстрагирования от аппаратных средств* (HAL, Hardware Abstraction Layer). Он представляет собой низкоуровневый интерфейс, лежащий между исполняющей системой и аппаратными средствами компьютера.

*Защищенные подсистемы* Windows NT, в отличие от исполняющей системы и уровня абстрагирования от аппаратных средств, работают в пользовательском режиме, что закрывает им низкоуровневый доступ к аппаратным средствам и данным компьютера. Но операционная система от этого становится более надежной и безопасной, ведь подсистемы могут обращаться к компьютеру только через посредника – исполнительную систему, которая контролирует работу всей машины.

Для лучшего понимания значения такого разделения функций подробнее рассмотрим компоненты операционной системы и их взаимодействие.

### 3.3.2. Уровень абстрагирования от аппаратных средств

Уровень абстрагирования от аппаратных средств (HAL) занимает низшую ступень операционной системы Windows NT. О его функциях говорит само название: он отделяет (абстрагирует) аппаратную часть компьютера от компонентов исполняющей системы. HAL разрабатывается под конкретную аппаратную платформу и содержит код, обеспечивающий связь с ее техническими средствами и управление ими. В результате такие аппаратно-зависимые элементы, как интерфейс ввода-вывода, контроллеры прерываний и средства межпроцессорной связи, оказываются скрытыми глубоко в недрах HAL и надежно изолированными от других частей операционной системы. Если сравнить, скажем, версии Windows NT для платформ MIPS и Intel, то окажется, что различия между ними заключаются главным образом в уровне абстрагирования от аппаратных средств.

HAL обеспечивает связь исполнительных компонентов с аппаратными средствами компьютера, то есть играет роль посредника между процессором и остальными элементами операционной системы. Защищенные подсистемы и приложения обращаются к аппаратным средствам через исполняющую систему, которая, в свою очередь, прибегает к услугам уровня абстрагирования от аппаратных средств. Таким образом, именно HAL контролирует весь доступ к аппаратным средствам.

### 3.3.3. Исполняющая система

Исполняющая система представляет собой набор компонентов, создающих ядро Windows NT. Ее составляющие можно разделить на две группы – *системные службы* и *внутренние подпрограммы*. К системным службам обращаются подсистемы и другие исполнительные компоненты, вызов же внутренних подпрограмм могут осуществлять только прочие исполнительные объекты.

Каждый исполнительный компонент снабжен API-подобным интерфейсом, который обеспечивает связь с другими компонентами, подсистемами Windows NT и приложениями. При этом исполнительные компоненты выполняют функции серверов, а подсистемы – клиентов. Когда же компоненты вступают во взаимодействие с другими исполнительными объектами, то им отводится уже роль клиентов.

Любой исполнительный компонент совершенно независим от других исполнительных объектов. Благодаря этому появляется возможность добавлять в операционную систему новые функции путем простой модернизации или замены одних компонентов, не затрагивая при этом другие. Если новая версия оснащена службами и интерфейсами, обеспечивающими ее взаимодействие с другими компонентами и подсистемами, работо-

способность операционной системы полностью сохраняется. Так, например, создатели Windows NT смогли установить в своем продукте новый Монитор безопасности, не внося изменений в другие компоненты.

### 3.3.4. Защищенные подсистемы

К компонентам Windows NT, работающим в пользовательском режиме, относятся так называемые *защищенные подсистемы*. Это название они получили потому, что каждая такая подсистема представляет собой независимый процесс, протекающий в собственном адресном пространстве и защищенный от других подсистем (и зависших приложений) диспетчером виртуальной памяти Windows NT. Если, например, клиент Win32 каким-либо образом дестабилизирует работу подсистемы Win32, это совершенно не скажется на работоспособности подсистемы безопасности.

Защищенные подсистемы Windows NT снабжены собственными интерфейсами прикладного программирования, обеспечивающими их связь как между собой, так и с приложениями. В клиент-серверной модели такие подсистемы выполняют функции сервера. Приложения или другие подсистемы направляют API-вызовы на исполняющую систему, откуда средства вызова локальных процедур (LPC) пересылают их на сервер. Сервер отвечает на запрос, и его сообщение передается через LPC корреспонденту, генерировавшему запрос.

Защищенные подсистемы делятся на два типа – *интегральные подсистемы* и *подсистемы среды,* которые создают среду, необходимую для работы конкретной программы. Так, работу приложений Windows в среде Windows NT обеспечивает подсистема Win32. Другие подсистемы среды обслуживают работу приложений под Winl6, POSIX и MS-DOS.

К интегральным подсистемам относятся те, которые обслуживают деятельность всей системы. В качестве примера можно привести подсистему безопасности – она производит аутентификацию при входе в систему и предоставлении доступа к файлам.

Как уже упоминалось, в число защищенных подсистем Windows NT входят и подсистемы среды, которые, как следует из их названия, создают среду для выполнения приложений. Главной из них является подсистема Win32. Именно она обслуживает запросы других подсистем среды на проведение операций ввода-вывода. Кроме Win32 в Windows NT имеются также подсистемы среды Winl6, POSIX и MS-DOS.

Подсистемы среды функционируют как независимые процессы с собственным адресным пространством. Здесь уместна аналогия с Windows 3.x, которая представляет собой DOS-приложение, создающее особую среду, необходимую для работы Windows-программ. Подсистемы среды также устанавливаются поверх ядра Windows NT и создают среду для работы приложений. Если программе конечного пользователя или самой

подсистеме требуется какая-либо функция ядра операционной системы, она передает управление исполняющей системе, которая вновь возвращает его подсистеме после выполнения запроса. Исполняющая система планирует потоки подсистем среды так же, как и потоки других приложений, что обеспечивает многозадачность всех подсистем среды.

Предоставляя для исполнения приложений замкнутые среды, Windows NT создает условия, при которых сбой одной программы не оказывает никакого влияния на другие. Так, аварийный останов приложения Winl6 совершенно не скажется на работающем приложении Win32.

Отказавшись от встраивания всех функций в ядро и сделав подсистемы среды независимыми пользовательскими процессами, разработчики Windows NT значительно повысили надежность и наращиваемость операционной системы. Во-первых, разделение подсистем позволило создать предельно стабильную основу операционной системы (ее ядро) и при этом обеспечить поддержку множества существующих интерфейсов прикладного программирования. Во-вторых, появилась возможность вносить изменения в API и расширять их функциональность без вмешательства в ядро. И, в-третьих, открылись перспективы совершенствования операционной системы простым включением в нее новых интерфейсов прикладного программирования.

# ГЛАВА 2

# ОРГАНИЗАЦИЯ ОПЕРАЦИОННЫХ СИСТЕМ

## 1. Архитектура операционной системы

Любая сложная система должна иметь понятную и рациональную структуру, то есть разделяться на части – модули, имеющие вполне законченное функциональное назначение с четко оговоренными правилами взаимодействия. Ясное понимание роли каждого отдельного модуля существенно упрощает работу по модификации и развитию системы.

Под *архитектурой ОС* понимают структурную организацию ОС на основе различных программных модулей. Обычно в состав ОС входят исполняемые и объектные модули стандартных для данной ОС форматов, библиотеки разных типов, модули исходного текста программ, программные модули специального формата (например, загрузчик ОС, драйверы ввода-вывода), конфигурационные файлы, файлы документации, модули справочной системы и т. д.

Большинство современных операционных систем представляют собой хорошо структурированные модульные системы, способные к развитию, расширению и переносу на новые платформы. Какой-либо единой архитектуры операционных систем не существует, но существуют универсальные подходы к структурированию ОС.

### 1.1. Ядро и вспомогательные модули ОС

#### 1.1.1. Назначение ядра и вспомогательных модулей ОС

Наиболее общим подходом к структуризации операционной системы является разделение всех ее модулей на две группы:

1) ядро – модули, выполняющие основные функции ОС;

2) модули, выполняющие вспомогательные функции ОС.

*Модули ядра* выполняют такие базовые функции ОС, как управление процессами, памятью, устройствами ввода-вывода и т. п. Ядро составляет

сердцевину операционной системы, без него ОС является полностью неработоспособной и не сможет выполнить ни одну из своих функций.

В состав ядра входят функции, решающие внутрисистемные задачи организации вычислительного процесса, такие, как переключение контекстов, загрузка/выгрузка станиц, обработка прерываний. Эти функции недоступны для приложений. Другой класс функций ядра служит для поддержки приложений, создавая для них так называемую прикладную программную среду. Приложения могут обращаться к ядру с запросами – системными вызовами – для выполнения тех или иных действий, например для открытия и чтения файла, вывода графической информации на дисплей, получения системного времени и т. д. Функции ядра, которые могут вызываться приложениями, образуют интерфейс прикладного программирования – API.

Функции, выполняемые модулями ядра, являются наиболее часто используемыми функциями операционной системы, поэтому скорость их выполнения определяет производительность всей системы в целом. Для обеспечения высокой скорости работы ОС все модули ядра или большая их часть постоянно находятся в оперативной памяти, то есть являются резидентными.

Ядро является движущей силой всех вычислительных процессов в компьютерной системе, и крах ядра равносилен краху всей системы. Поэтому разработчики операционной системы уделяют особое внимание надежности кодов ядра, в результате процесс их отладки может растягиваться на многие месяцы.

Обычно ядро оформляется в виде программного модуля некоторого специального формата, отличающегося от формата пользовательских приложений.

Остальные модули ОС выполняют весьма полезные, но менее обязательные функции. Например, к таким вспомогательным модулям могут быть отнесены программы архивирования данных на магнитной ленте, дефрагментации диска, текстового редактора. Вспомогательные модули ОС оформляются либо в виде приложений, либо в виде библиотек процедур.

Поскольку некоторые компоненты ОС оформлены как обычные приложения, то есть в виде исполняемых модулей стандартного для данной ОС формата, то часто бывает очень сложно провести четкую грань между операционной системой и приложениями (рис.2.1).

*Вспомогательные модули* ОС обычно подразделяются на следующие группы:

*утилиты* – программы, решающие отдельные задачи управления и сопровождения компьютерной системы, такие, например, как программы сжатия дисков, архивирования данных на магнитную ленту;

*системные обрабатывающие программы* – текстовые или графические редакторы, компиляторы, компоновщики, отладчики;

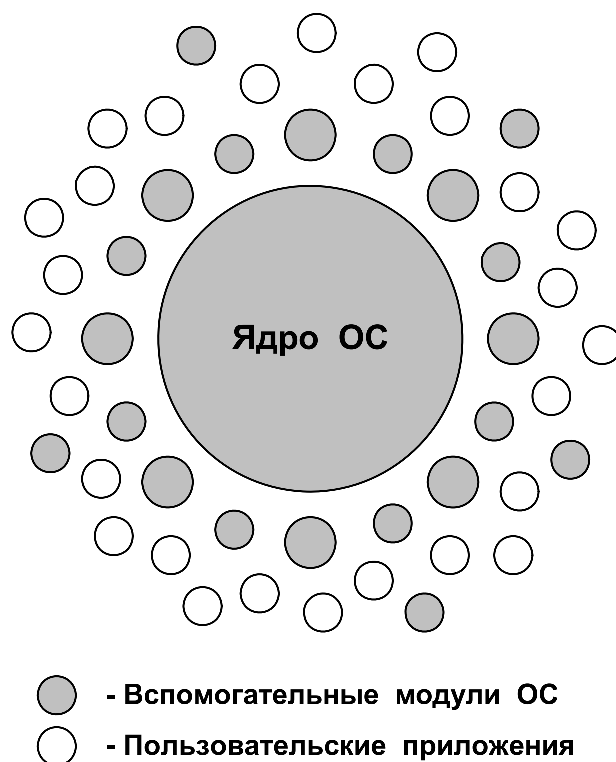



Рис. 2.1. Нечеткость границы между ОС и приложениями

*программы предоставления пользователю дополнительных услуг* – специальный вариант пользовательского интерфейса, калькулятор и даже игры;

*библиотеки процедур* различного назначения, упрощающие разработку приложений, например библиотека математических функций, функций ввода-вывода и т. д.

Как и обычные приложения, для выполнения своих задач утилиты, обрабатывающие программы и библиотеки ОС, обращаются к функциям ядра посредством системных вызовов (рис. 2.2).

Разделение операционной системы на ядро и модули-приложения обеспечивает легкую расширяемость ОС. Чтобы добавить новую высокоуровневую функцию, достаточно разработать новое приложение, и при этом не требуется модифицировать ответственные функции, образующие ядро системы. Однако внесение изменений в функции ядра может оказаться гораздо сложнее, и сложность эта зависит от структурной организации самого ядра. В некоторых случаях каждое исправление ядра может потребовать его полной перекомпиляции.

Модули ОС, оформленные в виде утилит, системных обрабатывающих программ и библиотек, обычно загружаются в оперативную память только на время выполнения своих функций, то есть являются транзитными. Постоянно в оперативной памяти располагаются только самые необходимые коды ОС, составляющие ее ядро. Такая организация ОС экономит оперативную память компьютера.

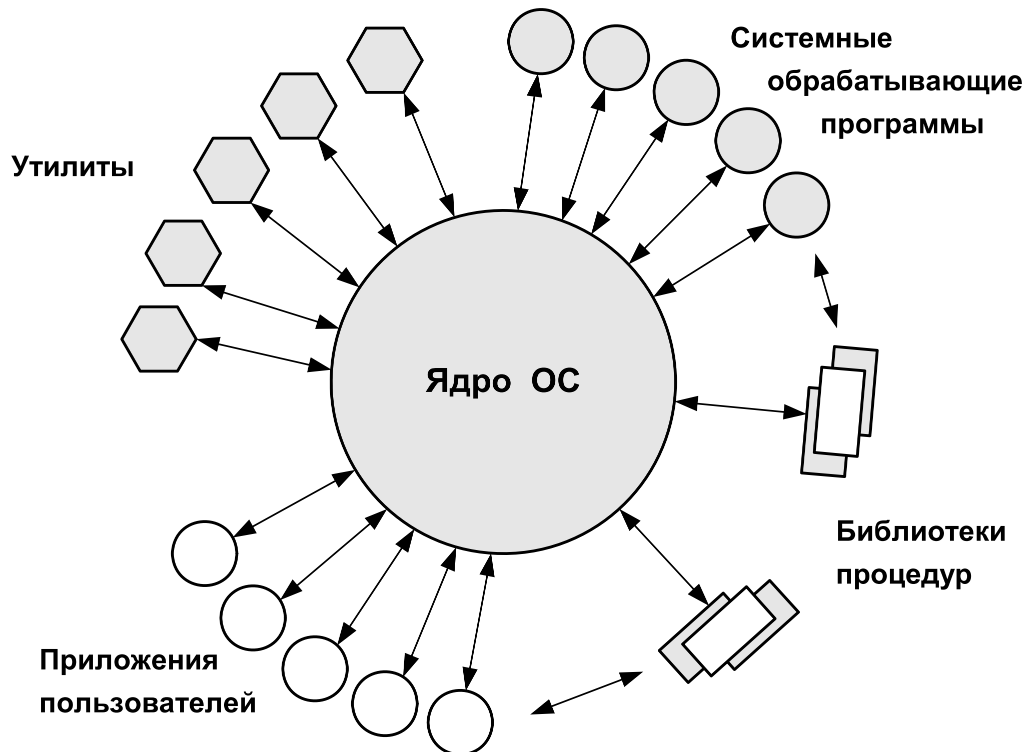



Рис. 2.2. Взаимодействие между ядром и вспомогательными модулями ОС

Важным свойством архитектуры ОС, основанной на ядре, является возможность защиты кодов и данных операционной системы за счет выполнения функций ядра в привилегированном режиме.

### 1.1.2. Ядро в привилегированном режиме

Для надежного управления ходом выполнения приложений операционная система должна иметь по отношению к приложениям определенные привилегии. Иначе некорректно работающее приложение может вмешаться в работу ОС и, например, разрушить часть ее кодов. Все усилия разработчиков операционной системы окажутся напрасными, если их решения воплощены в незащищенные от приложений модули системы, какими бы элегантными и эффективными эти решения ни были. Операционная система должна обладать исключительными полномочиями также для того, чтобы играть роль арбитра в споре приложений за ресурсы компьютера в мультипрограммном режиме. Ни одно приложение не должно иметь возможности без ведома ОС получать дополнительную область памяти, занимать процессор дольше разрешенного операционной системой периода времени, непосредственно управлять совместно используемыми внешними устройствами.

Обеспечить привилегии операционной системе невозможно без специальных средств аппаратной поддержки. Аппаратура компьютера должна поддерживать как минимум два режима работы – *пользовательский ре-*

*жим* (user mode) и *привилегированный режим*, который также называют режимом ядра (kernel mode), или режимом супервизора (supervisor mode). Подразумевается, что операционная система или некоторые ее части работают в привилегированном режиме, а приложения – в пользовательском режиме.

Так как ядро выполняет все основные функции ОС, то чаще всего именно ядро становится той частью ОС, которая работает в привилегированном режиме (рис. 2.3). Иногда это свойство – работа в привилегированном режиме – служит основным определением понятия “ядро”.

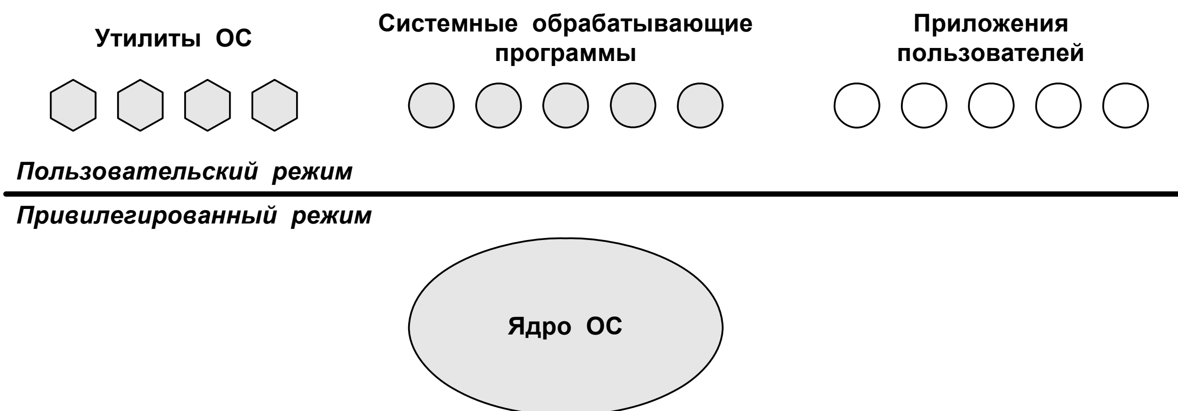



Рис. 2.3. Архитектура ОС с ядром в привилегированном режиме

Приложения ставятся в подчиненное положение за счет запрета выполнения в пользовательском режиме некоторых критичных команд, связанных с переключением процессора с задачи на задачу, управлением устройствами ввода-вывода, доступом к механизмам распределения и защиты памяти. Выполнение некоторых инструкций в пользовательском режиме запрещается безусловно (очевидно, что к таким инструкциям относится инструкция перехода в привилегированный режим), тогда как другие запрещается выполнять только при определенных условиях. Например, инструкции ввода-вывода могут быть запрещены приложениям при доступе к контроллеру жесткого диска, который хранит данные, общие для ОС и всех приложений, но разрешены при доступе к последовательному порту, который выделен в монопольное владение для определенного приложения.

Очень важно, что механизмы защиты памяти используются операционной системой не только для защиты своих областей памяти от приложений, но и для защиты областей памяти, выделенных ОС какому-либо приложению, от остальных приложений. Говорят, что каждое приложение работает в своем адресном пространстве. Это свойство позволяет локализовать некорректно работающее приложение в собственной области памяти, так что его ошибки не оказывают влияния на остальные приложения и операционную систему.

Между количеством уровней привилегий, реализуемых аппаратно, и количеством уровней привилегий, поддерживаемых ОС, нет прямого соответствия. Так, на базе четырех уровней, обеспечиваемых процессорами компании Intel, операционная система OS/2 строит трехуровневую систему привилегий, а операционные системы Windows NT, UNIX и некоторые другие ограничиваются двухуровневой системой.

С другой стороны, если аппаратура поддерживает хотя бы два уровня привилегий, то ОС может на этой основе создать программным способом сколь угодно развитую систему защиты.

Эта система может, например, поддерживать несколько уровней привилегий, образующих иерархию. Наличие нескольких уровней привилегий позволяет более тонко распределять полномочия как между модулями операционной системы, так и между самими приложениями. Появление внутри операционной системы более привилегированных и менее привилегированных частей позволяет повысить устойчивость ОС к внутренним ошибкам программных кодов, так как такие ошибки будут распространяться только внутри модулей с определенным уровнем привилегий. Дифференциация привилегий в среде прикладных модулей позволяет строить сложные прикладные комплексы, в которых часть более привилегированных модулей может, например, получать доступ к данным менее привилегированных модулей и управлять их выполнением.

На основе двух режимов привилегий процессора ОС может построить сложную систему индивидуальной защиты ресурсов, примером которой является типичная система защиты файлов и каталогов. Такая система позволяет задать для любого пользователя определенные права доступа к каждому из файлов и каталогов.

Повышение устойчивости операционной системы, обеспечиваемое переходом ядра в привилегированный режим, достигается за счет некоторого замедления выполнения системных вызовов. Системный вызов привилегированного ядра инициирует переключение процессора из пользовательского режима в привилегированный, а при возврате к приложению – переключение из привилегированного режима в пользовательский (рис. 2.4). Во всех типах процессоров из-за дополнительной двукратной задержки переключения переход на процедуру со сменой режима выполняется медленнее, чем вызов процедуры без смены режима.

Архитектура ОС, основанная на привилегированном ядре и приложениях пользовательского режима, стала, по существу, классической. Ее используют многие популярные операционные системы, в том числе многочисленные версии UNIX, VAX VMS, IBM OS/390, OS/2, и с определенными модификациями – Windows NT.

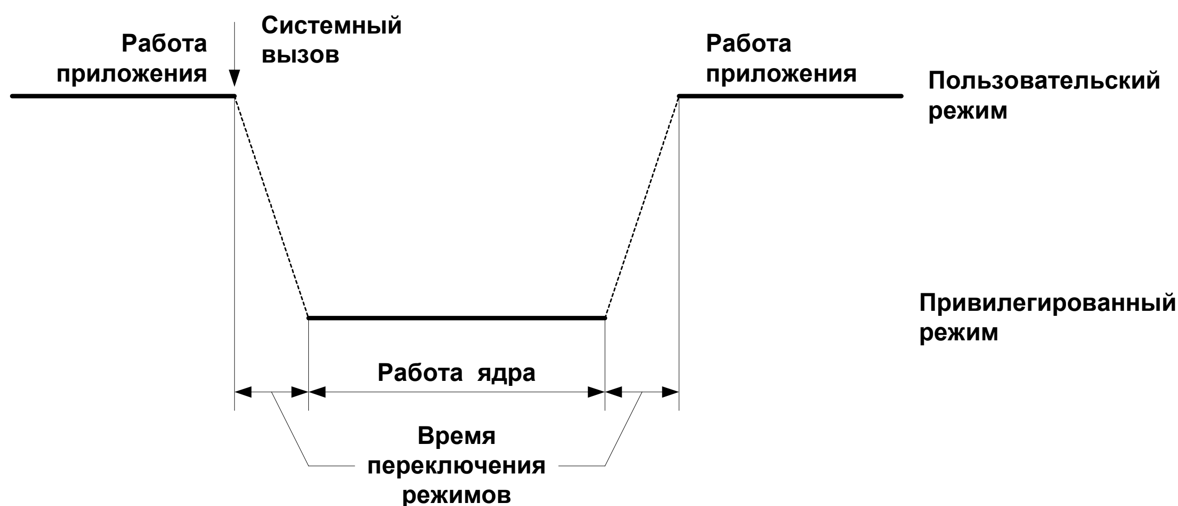



Рис. 2.4. Смена режимов при выполнении системного вызова к привилегированному ядру

### 1.1.3. Многослойная структура ОС

Вычислительную систему, работающую под управлением ОС на основе ядра, можно рассматривать как систему, состоящую из трех иерархически расположенных слоев: нижний слой образует аппаратура, промежуточный – ядро, а утилиты, обрабатывающие программы и приложения, составляют верхний слой системы (рис. 2.5). Слоистую структуру вычислительной системы принято изображать в виде системы концентрических окружностей, иллюстрируя тот факт, что каждый слой может взаимодействовать только со смежными слоями. Действительно, при такой организации ОС приложения не могут непосредственно взаимодействовать с аппаратурой, а только через слой ядра.

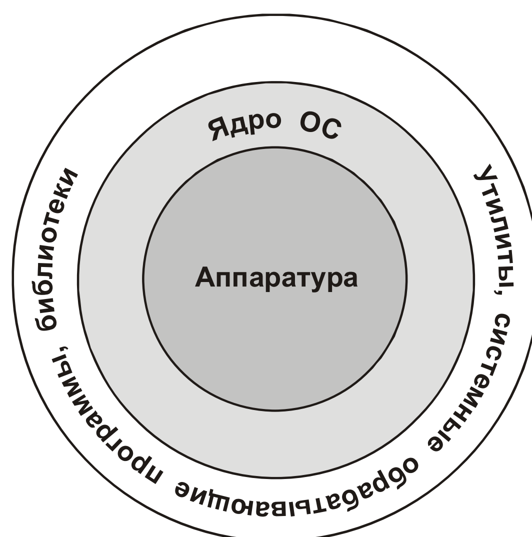



Рис. 2.5. Трехслойная схема вычислительной системы

Многослойный подход является универсальным и эффективным способом декомпозиции сложных систем любого типа, в том числе и программных. В соответствии с этим подходом система состоит из иерархии слоев. Каждый слой обслуживает вышележащий слой, выполняя для него некоторый набор функций, которые образуют межслойный интерфейс. На основе функций нижележащего слоя следующий (вверх по иерархии) слой строит свои функции – более сложные и более мощные, которые, в свою очередь, оказываются примитивами для создания еще более мощных функций вышележащего слоя. Строгие правила касаются только взаимодействия между слоями системы, а между модулями внутри слоя связи могут быть произвольными. Отдельный модуль может выполнить свою работу либо самостоятельно, либо обратиться к другому модулю своего слоя, либо обратиться за помощью к нижележащему слою через межслойный интерфейс.

Такая организация системы имеет следующие достоинства:

1. Существенно упрощается разработка системы, так как сначала определяются “сверху вниз” функции слоев и межслойные интерфейсы, а затем при детальной реализации постепенно наращивается мощность функций слоев, двигаясь “снизу вверх”.

2. При модернизации системы можно изменять модули внутри слоя без необходимости производить какие-либо изменения в остальных слоях, если при этих внутренних изменениях межслойные интерфейсы остаются в силе.

Поскольку ядро представляет собой сложный многофункциональный комплекс, то многослойный подход обычно распространяется и на структуру ядра.

Ядро может состоять из следующих слоев (рис. 2.6):

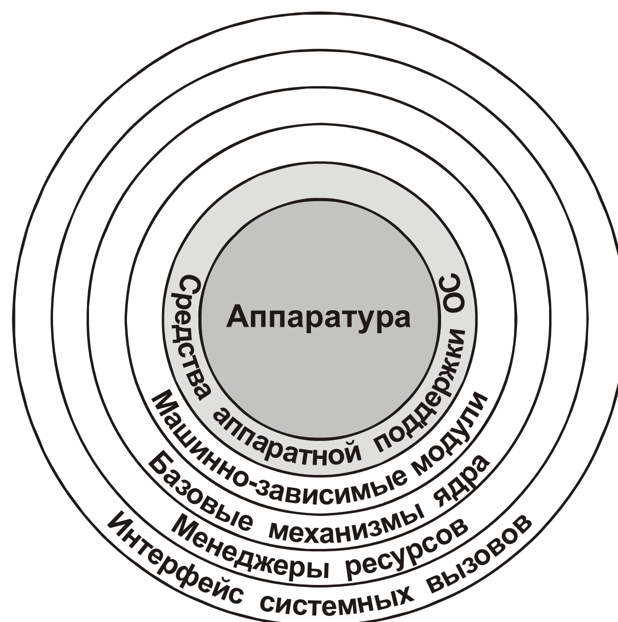



Рис. 2.6. Многослойная структура ядра ОС

*Средства аппаратной поддержки ОС*. Обычно под операционной системой понимают комплекс программ, однако часть функций ОС может выполняться и аппаратными средствами. К операционной системе относят, естественно, не все аппаратные устройства компьютера, а только средства аппаратной поддержки ОС, то есть те, которые прямо участвуют в организации вычислительных процессов: средства поддержки привилегированного режима, систему прерываний, средства переключения контекстов процессов, средства защиты областей памяти и т. п.

*Машинно-зависимые компоненты ОС*. Этот слой образуют программные модули, в которых отражается специфика аппаратной платформы компьютера. В идеале этот слой полностью экранирует вышележащие слои ядра от особенностей аппаратуры. Это позволяет разрабатывать вышележащие слои на основе машинно-независимых модулей, существующих в единственном экземпляре для всех типов аппаратных платформ, поддерживаемых данной ОС. Примером экранирующего слоя может служить слой HAL операционной системы Windows NT.

*Базовые механизмы ядра*. Этот слой выполняет наиболее примитивные операции ядра, такие как программное переключение контекстов процессов, диспетчеризацию прерываний, перемещение страниц из памяти на диск и обратно и т. п. Модули данного слоя не принимают решений о распределении ресурсов – они только отрабатывают принятые “наверху” решения, что и дает повод называть их исполнительными механизмами для модулей верхних слоев.

*Менеджеры ресурсов*. Этот слой состоит из мощных функциональных модулей, реализующих стратегические задачи по управлению основными ресурсами вычислительной системы. Обычно на данном слое работают менеджеры (называемые также диспетчерами) процессов, ввода-вывода, файловой системы и оперативной памяти. Каждый из менеджеров ведет учет свободных и используемых ресурсов определенного типа и планирует их распределение в соответствии с запросами приложений. Например, менеджер виртуальной памяти управляет перемещением страниц из оперативной памяти на диск и обратно. Для исполнения принятых решений менеджер обращается к нижележащему слою базовых механизмов с запросами о загрузке (выгрузке) конкретных страниц. Внутри слоя менеджеров существуют тесные взаимные связи, отражающие тот факт, что для выполнения процессу нужен доступ одновременно к нескольким ресурсам – процессору, области памяти, возможно, к определенному файлу или устройству ввода-вывода. Например, при создании процесса менеджер процессов обращается к менеджеру памяти, который должен выделить процессу определенную область памяти для его кодов и данных.

*Интерфейс системных вызовов*. Этот слой является самым верхним слоем ядра и взаимодействует непосредственно с приложениями и системными утилитами, образуя прикладной программный интерфейс операционной системы. Функции API, обслуживающие системные вызовы, пре-

доставляют доступ к ресурсам системы в удобной и компактной форме, без указания деталей их физического расположения.

Приведенное разбиение ядра ОС на слои является достаточно условным. В реальной системе количество слоев и распределение функций между ними может быть и иным. В системах, предназначенных для аппаратных платформ одного типа, например ОС NetWare, слой машинно-зависимых модулей обычно не выделяется, сливаясь со слоем базовых механизмов и, частично, со слоем менеджеров ресурсов. Не всегда оформляются в отдельный слой базовые механизмы – в этом случае менеджеры ресурсов не только планируют использование ресурсов, но и самостоятельно реализуют свои планы.

Возможна и противоположная картина, когда ядро состоит из большего количества слоев. Например, менеджеры ресурсов, составляя определенный слой ядра, в свою очередь, могут обладать многослойной структурой. Прежде всего это относится к менеджеру ввода-вывода, нижний слой которого составляют драйверы устройств, например драйвер жесткого диска или драйвер сетевого адаптера, а верхние слои – драйверы файловых систем или протоколов сетевых служб, имеющие дело с логической организацией информации.

Выбор количества слоев ядра является ответственным и сложным делом: увеличение числа слоев ведет к некоторому замедлению работы ядра за счет дополнительных накладных расходов на межслойное взаимодействие, а уменьшение числа слоев ухудшает расширяемость и логичность системы. Обычно операционные системы, прошедшие долгий путь эволюционного развития, например многие версии UNIX, имеют неупорядоченное ядро с небольшим числом четко выделенных слоев, а у сравнительно “молодых” операционных систем, таких как Windows NT, ядро разделено на большее число слоев и их взаимодействие формализовано в гораздо большей степени.

## 1.2. Микроядерная архитектура

### 1.2.1. Концепция

Микроядерная архитектура является альтернативой классическому способу построения операционной системы. Под классической архитектурой в данном случае понимается рассмотренная выше структурная организация ОС, в соответствии с которой все основные функции операционной системы, составляющие многослойное ядро, выполняются в привилегированном режиме. При этом некоторые вспомогательные функции ОС оформляются в виде приложений и выполняются в пользовательском режиме наряду с обычными пользовательскими программами (становясь

системными утилитами или обрабатывающими программами). Каждое приложение пользовательского режима работает в собственном адресном пространстве и защищено тем самым от какого-либо вмешательства других приложений. Код ядра, выполняемый в привилегированном режиме, имеет доступ к областям памяти всех приложений, но сам полностью от них защищен. Приложения обращаются к ядру с запросами на выполнение системных функций.

Суть микроядерной архитектуры состоит в следующем. В привилегированном режиме остается работать только очень небольшая часть ОС, называемая микроядром (рис. 2.7).

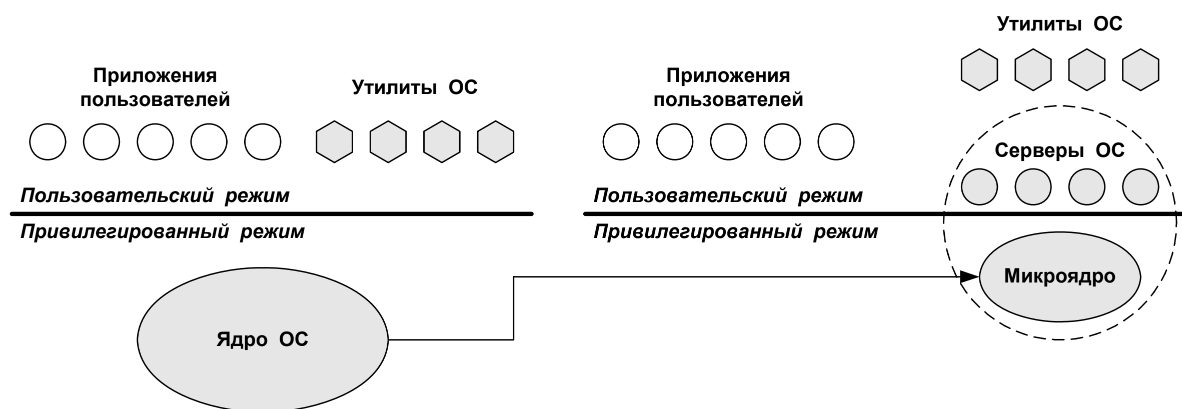



Рис. 2.7. Перенос основного объема функций ядра в пользовательское пространство

Микроядро защищено от остальных частей ОС и приложений. В состав микроядра обычно входят машинно-зависимые модули, а также модули, выполняющие базовые (но не все!) функции ядра по управлению процессами, обработке прерываний, управлению виртуальной памятью, пересылке сообщений и управлению устройствами ввода-вывода, связанные с загрузкой или чтением регистров устройств. Набор функций микроядра обычно соответствует функциям слоя базовых механизмов обычного ядра. Такие функции операционной системы трудно, если не невозможно, выполнить в пространстве пользователя.

Все остальные более высокоуровневые функции ядра оформляются в виде приложений, работающих в пользовательском режиме. Однозначного решения о том, какие из системных функций нужно оставить в привилегированном режиме, а какие перенести в пользовательский, не существует. В общем случае многие менеджеры ресурсов, являющиеся неотъемлемыми частями обычного ядра – файловая система, подсистемы управления виртуальной памятью и процессами, менеджер безопасности и т. п., – становятся “периферийными” модулями, работающими в пользовательском режиме.

Работающие в пользовательском режиме менеджеры ресурсов имеют принципиальные отличия от традиционных утилит и обрабатывающих

программ операционной системы, хотя при микроядерной архитектуре все эти программные компоненты также оформлены в виде приложений. Утилиты и обрабатывающие программы вызываются в основном пользователями. Ситуации, когда одному приложению требуется выполнение функции (процедуры) другого приложения, возникают крайне редко. Поэтому в операционных системах с классической архитектурой отсутствует механизм, с помощью которого одно приложение могло бы вызвать функции другого.

Совсем другая ситуация возникает, когда в форме приложения оформляется часть операционной системы. По определению, основным назначением такого приложения является обслуживание запросов других приложений, например, создание процесса, выделение памяти, проверка прав доступа к ресурсу и т. д. Именно поэтому менеджеры ресурсов, вынесенные в пользовательский режим, называются серверами ОС, то есть модулями, основным назначением которых является обслуживание запросов локальных приложений и других модулей ОС. Очевидно, что для реализации микроядерной архитектуры необходимым условием является наличие в операционной системе удобного и эффективного способа вызова процедур одного процесса из другого. Поддержка такого механизма и является одной из главных задач микроядра.

Схематично механизм обращения к функциям ОС, оформленным в виде серверов, выглядит следующим образом (рис. 2.8).

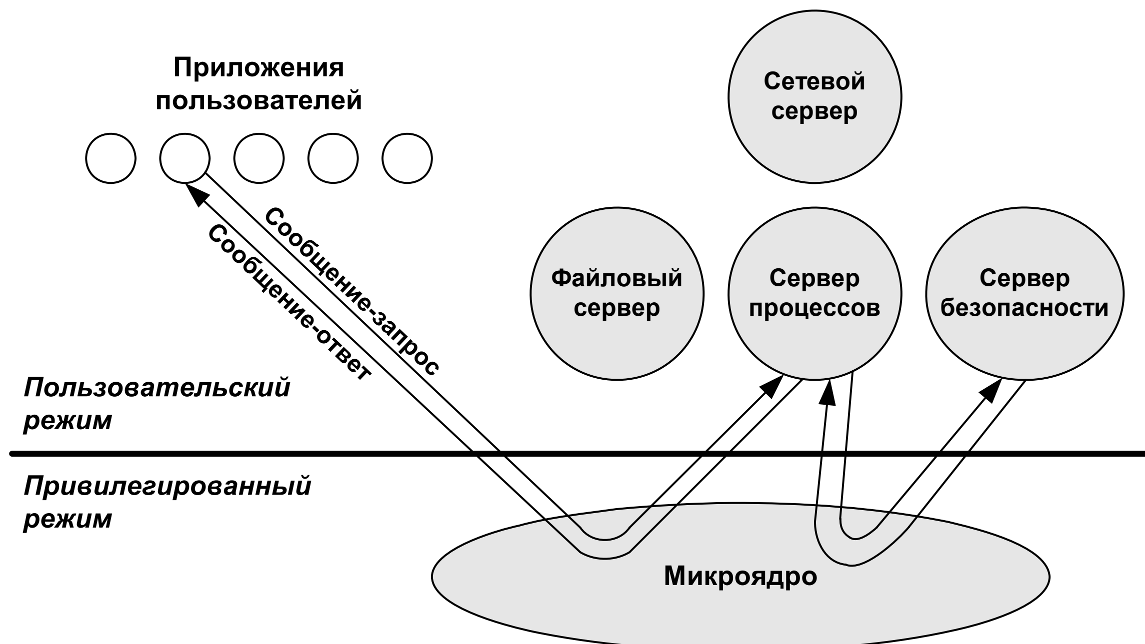



Рис. 2.8. Реализация системного вызова в микроядерной архитектуре

Клиент, которым может быть либо прикладная программа, либо другой компонент ОС, запрашивает выполнение некоторой функции у соответствующего сервера, посылая ему сообщение. Непосредственная передача

сообщений между приложениями невозможна, так как их адресные пространства изолированы друг от друга. Микроядро, выполняющееся в привилегированном режиме, имеет доступ к адресным пространствам каждого из этих приложений и поэтому может работать в качестве посредника. Микроядро сначала передает сообщение, содержащее имя и параметры вызываемой процедуры нужному серверу, затем сервер выполняет запрошенную операцию, после чего ядро возвращает результаты клиенту с помощью другого сообщения.

Таким образом, работа микроядерной операционной системы соответствует известной модели клиент-сервер, в которой роль транспортных средств выполняет микроядро.

### 1.2.2. Преимущества и недостатки микроядерной архитектуры

Операционные системы, основанные на концепции микроядра, в высокой степени удовлетворяют большинству требований, предъявляемых к современным ОС, обладая переносимостью, расширяемостью, надежностью и создавая хорошие предпосылки для поддержки распределенных приложений. За эти достоинства приходится платить снижением производительности, и это является основным недостатком микроядерной архитектуры.

Высокая степень *переносимости* обусловлена тем, что весь машинно-зависимый код изолирован в микроядре, поэтому для переноса системы на новый процессор требуется меньше изменений и все они логически сгруппированы вместе.

*Расширяемость* присуща микроядерной ОС в очень высокой степени. В традиционных системах даже при наличии многослойной структуры нелегко удалить один слой и поменять его на другой по причине множественности и размытости интерфейсов между слоями. Добавление новых функций и изменение существующих требует хорошего знания операционной системы и больших затрат времени. В то же время ограниченный набор четко определенных интерфейсов микроядра открывает путь к упорядоченному росту и эволюции ОС. Добавление новой подсистемы требует разработки нового приложения, что никак не затрагивает целостность микроядра. Микроядерная структура позволяет не только добавлять, но и сокращать число компонентов операционной системы, что также бывает очень полезно. При микроядерном подходе *конфигурируемость* ОС не вызывает никаких проблем и не требует особых мер – достаточно изменить файл с настройками начальной конфигурации системы или же остановить не нужные больше серверы в ходе работы обычными для остановки приложений средствами.

Использование микроядерной модели повышает *надежность* ОС. Каждый сервер выполняется в виде отдельного процесса в своей собствен-

ной области памяти и таким образом защищен от других серверов операционной системы, что не наблюдается в традиционной ОС, где все модули ядра могут влиять друг на друга. И если отдельный сервер терпит крах, то он может быть перезапущен без останова или повреждения остальных серверов ОС. Более того, поскольку серверы выполняются в пользовательском режиме, они не имеют непосредственного доступа к аппаратуре и не могут модифицировать память, в которой хранится и работает микроядро. Другим потенциальным источником повышения надежности ОС является уменьшенный объем кода микроядра по сравнению с традиционным ядром – это снижает вероятность появления ошибок программирования.

Модель с микроядром хорошо подходит для поддержки *распределенных вычислений*, так как использует механизмы, аналогичные сетевым: взаимодействие клиентов и серверов путем обмена сообщениями. Серверы микроядерной ОС могут работать как на одном, так и на разных компьютерах. В этом случае при получении сообщения от приложения микроядро может обработать его самостоятельно и передать локальному серверу или же переслать по сети микроядру, работающему на другом компьютере. Переход к распределенной обработке требует минимальных изменений в работе операционной системы – просто локальный транспорт заменяется на сетевой.

*Производительность*. При классической организации ОС (рис. 2.9, а) выполнение системного вызова сопровождается двумя переключениями режимов, а при микроядерной организации (рис. 2.9, б) – четырьмя. Таким образом, операционная система на основе микроядра при прочих равных условиях всегда будет менее производительной, чем ОС с классическим ядром. Именно по этой причине микроядерный подход не получил такого широкого распространения, которое ему предрекали.

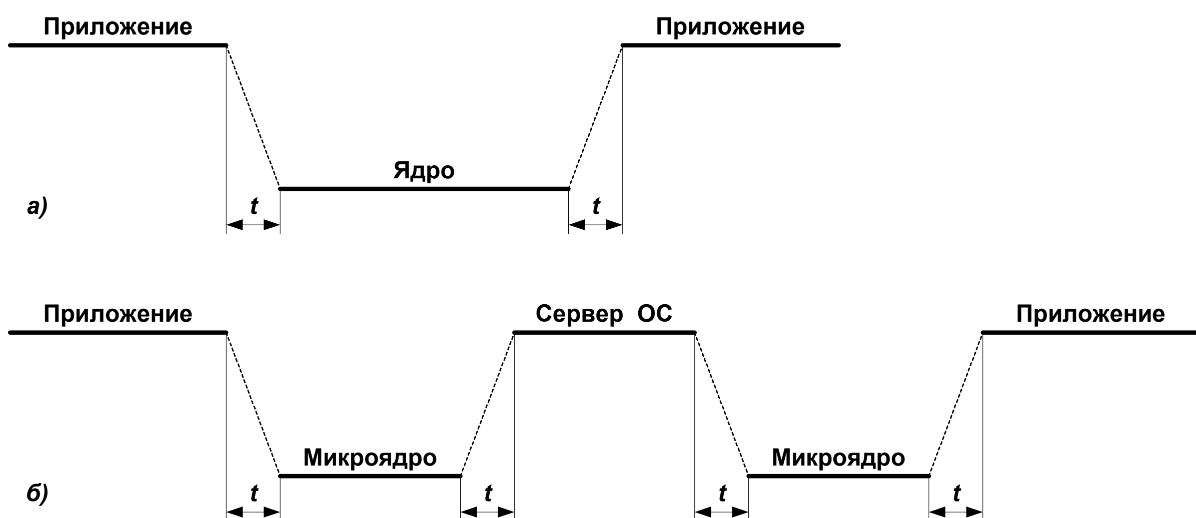



Рис. 2.9. Смена режимов при выполнении системного вызова

Главной проблемой, с которой сталкиваются разработчики операционной системы, решившие применить микроядерный подход, является вопрос о том, что включать в микроядро, а что выносить в пользовательское пространство. В идеальном случае микроядро может состоять только из средств передачи сообщений, средств взаимодействия с аппаратурой, в том числе средств доступа к механизмам привилегированной защиты. Однако многие разработчики не всегда жестко придерживаются принципа минимизации функций ядра, часто жертвуя этим ради повышения производительности. В результате реализации ОС образуют некоторый спектр, на одном краю которого находятся системы с минимально возможным микроядром, а на другом – системы, подобные Windows NT, в которых микроядро выполняет достаточно большой объем функций.

## 1.3. Аппаратная зависимость и переносимость ОС

Многие операционные системы успешно работают на различных аппаратных платформах без существенных изменений в своем составе. Во многом это объясняется тем, что, несмотря на различия в деталях, средства аппаратной поддержки ОС большинства компьютеров приобрели сегодня много типовых черт, а именно эти средства в первую очередь влияют на работу компонентов операционной системы. В результате в ОС можно выделить достаточно компактный слой машинно-зависимых компонентов ядра и сделать остальные слои ОС общими для разных аппаратных платформ.

### 1.3.1. Типовые средства аппаратной поддержки ОС

Четкой границы между программной и аппаратной реализацией функций ОС не существует – решение о том, какие функции ОС будут выполняться программно, а какие аппаратно, принимается разработчиками аппаратного и программного обеспечения компьютера. Тем не менее, практически все современные аппаратные платформы имеют некоторый типичный набор средств аппаратной поддержки ОС, в который входят следующие компоненты:

1) средства поддержки привилегированного режима;
2) средства трансляции адресов;
3) средства переключения процессов;
4) система прерываний;
5) системный таймер;
6) средства защиты областей памяти.

*Средства поддержки привилегированного режима* обычно основаны на системном регистре процессора, часто называемом “словом состояния”

машины или процессора. Этот регистр содержит некоторые признаки, определяющие режимы работы процессора, в том числе и признак текущего режима привилегий. Смена режима привилегий выполняется за счет изменения слова состояния машины в результате прерывания или выполнения привилегированной команды. Число градаций привилегированности может быть разным у разных типов процессоров, наиболее часто используются два уровня (ядро-пользователь) или четыре (например, ядро-супервизор-выполнение-пользователь у платформы VAX или 0-1-2-3 у процессоров Intel x86/Pentium). В обязанности средств поддержки привилегированного режима входит выполнение проверки допустимости выполнения активной программой инструкций процессора при текущем уровне привилегированности.

*Средства трансляции адресов* выполняют операции преобразования виртуальных адресов, которые содержатся в кодах процесса, в адреса физической памяти. Таблицы, предназначенные при трансляции адресов, обычно имеют большой объем, поэтому для их хранения используются области оперативной памяти, а аппаратура процессора содержит только указатели на эти области. Средства трансляции адресов используют данные указатели для доступа к элементам таблиц и аппаратного выполнения алгоритма преобразования адреса, что значительно ускоряет процедуру трансляции по сравнению с ее чисто программной реализацией.

*Средства переключения процессов* предназначены для быстрого сохранения контекста приостанавливаемого процесса и восстановления контекста процесса, который становится активным. Содержимое контекста обычно включает содержимое всех регистров общего назначения процессора, регистра флагов операции (то есть флагов нуля, переноса, переполнения и т. п.), а также тех системных регистров и указателей, которые связаны с отдельным процессом, а не операционной системой, например указателя на таблицу трансляции адресов процесса. Для хранения контекстов приостановленных процессов обычно используются области оперативной памяти, которые поддерживаются указателями процессора.

*Система прерываний* позволяет компьютеру реагировать на внешние события, синхронизировать выполнение процессов и работу устройств ввода-вывода, быстро переходить с одной программы на другую. Механизм прерываний нужен для того, чтобы оповестить процессор о возникновении в вычислительной системе некоторого непредсказуемого события или события, которое не синхронизировано с циклом работы процессора. Примерами таких событий могут служить завершение операции ввода-вывода внешним устройством (например, запись блока данных контроллером диска), некорректное завершение арифметической операции (например, переполнение регистра), истечение интервала астрономического времени.

*Системный таймер*, часто реализуемый в виде быстродействующего регистра-счетчика, необходим операционной системе для выдержки ин-

тервалов времени. Для этого в регистр таймера программно загружается значение требуемого интервала в условных единицах, из которого затем автоматически с определенной частотой начинает вычитаться по единице. Частота “тиков” таймера, как правило, тесно связана с частотой тактового генератора процессора. Не следует путать таймер ни с тактовым генератором, который вырабатывает сигналы, синхронизирующие все операции в компьютере, ни с системными часами – работающей на батареях электронной схеме, – которые ведут независимый отсчет времени и календарной даты. При достижении нулевого значения счетчика таймер инициирует прерывание, которое обрабатывается процедурой операционной системы. Прерывания от системного таймера используются ОС в первую очередь для слежения за тем, как отдельные процессы расходуют время процессора. Например, в системе разделения времени при обработке очередного прерывания от таймера планировщик процессов может принудительно передать управление другому процессу, если данный процесс исчерпал выделенный ему квант времени.

*Средства защиты областей памяти* обеспечивают на аппаратном уровне проверку возможности программного кода осуществлять с данными определенной области памяти такие операции, как чтение, запись или выполнение (при передачах управления). Если аппаратура компьютера поддерживает механизм трансляции адресов, то средства защиты областей памяти встраиваются в этот механизм. Функции аппаратуры по защите памяти обычно состоят в сравнении уровней привилегий текущего кода процессора и сегмента памяти, к которому производится обращение.

### 1.3.2. Машинно-зависимые компоненты ОС

Одна и та же операционная система не может без каких-либо изменений устанавливаться на компьютерах, отличающихся типом процессора или/и способом организации всей аппаратуры. В модулях ядра ОС не могут не отразиться такие особенности аппаратной платформы, как количество типов прерываний и формат таблицы ссылок на процедуры обработки прерываний, состав регистров общего назначения и системных регистров, состояние которых нужно сохранять в контексте процесса, особенности подключения внешних устройств и многие другие.

Однако опыт разработки операционных систем показывает: ядро можно спроектировать таким образом, что только часть модулей будут машинно-зависимыми, а остальные не будут зависеть от особенностей аппаратной платформы. В хорошо структурированном ядре машинно-зависимые модули локализованы и образуют программный слой, естественно примыкающий к слою аппаратуры (рис. 2.6). Такая локализация

машинно-зависимых модулей существенно упрощает перенос операционной системы на другую аппаратную платформу.

Для уменьшения количества машинно-зависимых модулей производители операционных систем обычно ограничивают универсальность машинно-независимых модулей. Это означает, что их независимость носит условный характер и распространяется только на несколько типов процессоров и созданных на основе этих процессоров аппаратных платформ. По этому пути пошли, например, разработчики ОС Windows NT, ограничив количество типов процессоров для своей системы четырьмя и поставляя различные варианты кодов ядра для однопроцессорных и многопроцессорных компьютеров.

Особое место среди модулей ядра занимают низкоуровневые драйверы внешних устройств. С одной стороны эти драйверы, как и высокоуровневые драйверы, входят в состав менеджера ввода-вывода, то есть принадлежат слою ядра, занимающему достаточно высокое место в иерархии слоев. С другой стороны, низкоуровневые драйверы отражают все особенности управляемых внешних устройств, поэтому их можно отнести и к слою машинно-зависимых модулей. Такая двойственность низкоуровневых драйверов еще раз подтверждает схематичность модели ядра со строгой иерархией слоев.

Для компьютеров на основе процессоров Intel x86/Pentium разработка экранирующего машинно-зависимого слоя ОС несколько упрощается за счет встроенной в постоянную память компьютера базовой системы ввода-вывода – BIOS. BIOS содержит драйверы для всех устройств, входящих в базовую конфигурацию компьютера: жестких и гибких дисков, клавиатуры, дисплея и т. д. Эти драйверы выполняют весьма примитивные операции с управляемыми устройствами, например чтение группы секторов данных с определенной дорожки диска, но за счет этих операций экранируются различия аппаратных платформ персональных компьютеров и серверов на процессорах Intel разных производителей. Разработчики операционной системы могут пользоваться слоем драйверов BIOS как частью машинно-зависимого слоя ОС, а могут и заменить все или часть драйверов BIOS компонентами ОС.

### 1.3.3. Переносимость операционной системы

Если код операционной системы может быть сравнительно легко перенесен с процессора одного типа на процессор другого типа и с аппаратной платформы одного типа на аппаратную платформу другого типа, то такую ОС называют *переносимой* (portable), или *мобильной*.

Хотя ОС часто описываются либо как переносимые, либо как непереносимые, мобильность – это не бинарное состояние, а понятие степени. Вопрос не в том, может ли быть система перенесена, а в том, насколько

легко можно это сделать. Для того чтобы обеспечить свойство мобильности ОС, разработчики должны следовать следующим правилам.

1. Большая часть кода должна быть написана на языке, трансляторы которого имеются на всех машинах, куда предполагается переносить систему. Такими языками являются стандартизованные языки высокого уровня. Большинство переносимых ОС написано на языке С, который имеет много особенностей, полезных для разработки кодов операционной системы, и компиляторы которого широко доступны.

2. Объем машинно-зависимых частей кода, которые непосредственно взаимодействуют с аппаратными средствами, должен быть по возможности минимизирован. Так, например, следует всячески избегать прямого манипулирования регистрами и другими аппаратными средствами процессора.

3. Аппаратно-зависимый код должен быть надежно изолирован в нескольких модулях, а не быть распределен по всей системе. Изоляции подлежат все части ОС, которые отражают специфику как процессора, так и аппаратной платформы в целом. Низкоуровневые компоненты ОС, имеющие доступ к процессорно-зависимым структурам данных и регистрам, должны быть оформлены в виде компактных модулей, которые могут быть заменены аналогичными модулями для других процессоров.

В идеале слой машинно-зависимых компонентов ядра полностью экранирует остальную часть ОС от конкретных деталей аппаратной платформы (кэши, контроллеры прерываний ввода-вывода и т. п.), по крайней мере для того набора платформ, который поддерживает данная ОС. В результате происходит подмена реальной аппаратуры некой унифицированной виртуальной машиной, одинаковой для всех вариантов аппаратной платформы. Все слои операционной системы, которые лежат выше слоя машинно-зависимых компонентов, могут быть написаны для управления именно этой виртуальной аппаратурой. Таким образом, у разработчиков появляется возможность создавать один вариант машинно-независимой части ОС (включая компоненты ядра, утилиты, системные обрабатывающие программы) для всего набора поддерживаемых платформ.

## 2. Ресурсы операционной системы

Понятие “вычислительный процесс” (или просто – “процесс”) является одним из основных при рассмотрении операционных систем. Как понятие, процесс является определенным видом абстракции, поэтому будем придерживаться следующего определения.

*Последовательный процесс* (иногда называемый “задачей”) – это выполнение отдельной программы с ее данными на последовательном процессоре.

Концептуально процессор рассматривается в двух аспектах: во-первых, он является носителем данных и, во-вторых, он (одновременно) выполняет операции, связанные с их обработкой.

В качестве примеров можно назвать следующие процессы (задачи): выполнение прикладных программ пользователей, утилит и других системных обрабатывающих программ. Процессами могут быть редактирование какого-либо текста, трансляция исходной программы, ее компоновка, исполнение. Причем трансляция какой-нибудь исходной программы является одним процессом, а трансляция следующей исходной программы – другим процессом, поскольку, хотя транслятор как объединение программных модулей здесь выступает как одна и та же программа, но данные, которые он обрабатывает, являются разными.

Определение концепции процесса преследует цель выработать механизмы распределения и управления ресурсами. Понятие ресурса, так же как и понятие процесса, является, пожалуй, основным при рассмотрении операционных систем. Термин ресурс обычно применяется по отношению к повторно используемым, относительно стабильным и часто недостающим объектам, которые запрашиваются, используются и освобождаются процессами в период их активности. Другими словами, *ресурсом* называется всякий объект, который может распределяться внутри системы.

## 2.1. Характеристика ресурсов и способов их использования

Ресурсы вычислительной машины делятся на аппаратные и программные (рис. 2.10). *Аппаратные ресурсы* машины – это процессоры, запоминающие устройства, каналы ввода-вывода, периферийные устройства и т. д. *Программными ресурсами* являются программы и данные, которые могут быть предоставлены пользователям (например, трансляторы, пакеты прикладных программ, библиотеки стандартных программ).

Требования к организации использования ресурсов определяются конкретными условиями эксплуатации ЭВМ. При этом практически всегда остаются актуальными следующие два требования: повышение эффективности использования ресурсов и повышения качества обслуживания пользователей. Эти требования противоречивы, преодоление их противоречивости достигается путем разумного компромисса. Причем в зависимости от назначения ЭВМ предпочтение может отдаваться либо первому, либо второму требованию. Выполнение этих требований зависит от множества факторов, определяемых как самими ресурсами машины, так и условиями их использования.

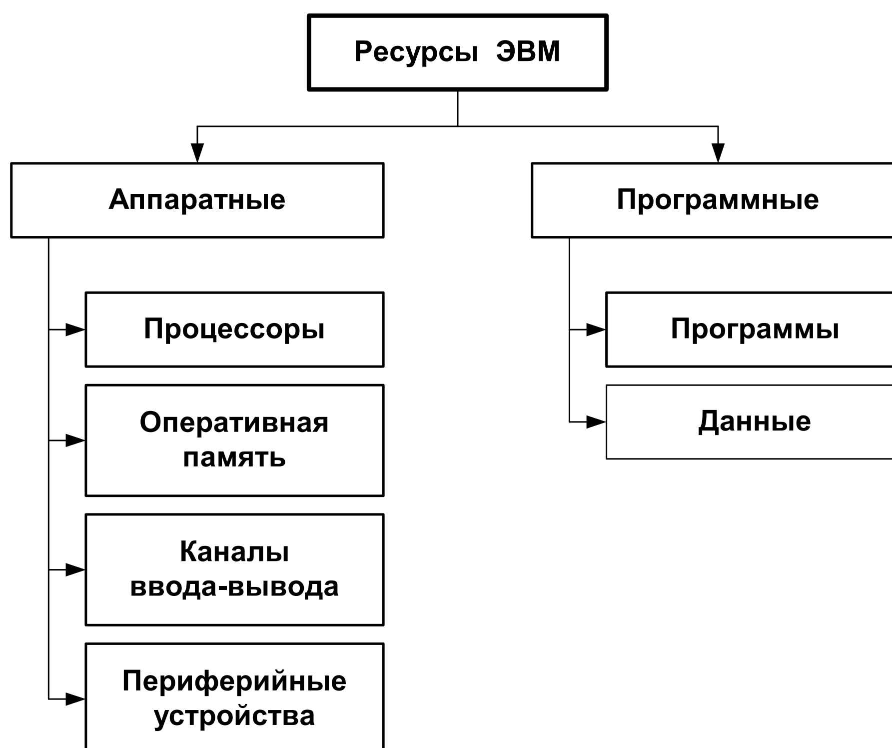



Рис. 2.10. Классификация ресурсов ЭВМ

### 2.1.1. Аппаратные ресурсы

В современных мультипрограммных вычислительных машинах совместно выполняется несколько заданий, в связи с чем должно быть обеспечено совместное использование или разделение ими имеющихся аппаратных ресурсов.

В настоящее время существуют три основных способа использования аппаратных ресурсов (рис.2.11): монопольный, совместный и виртуальный.

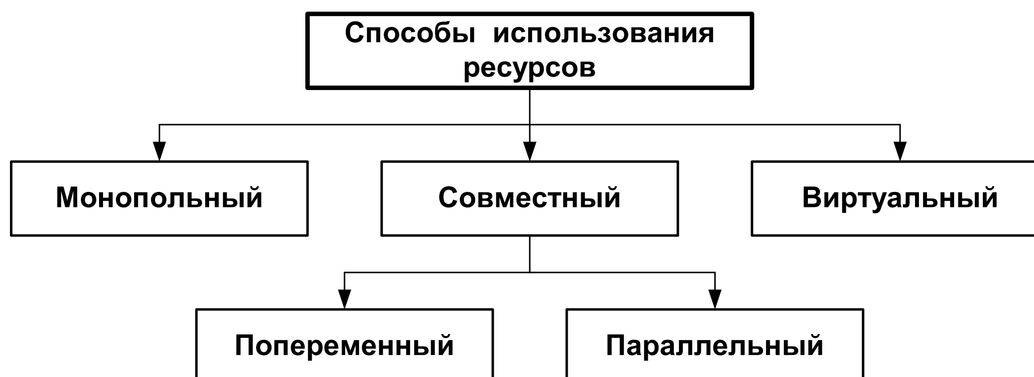



Рис.2.11. Классификация способов использования аппаратных ресурсов

При *монопольном использовании* ресурсы закрепляются за одним процессом на все время его выполнения и становятся недоступными для других процессов независимо от степени их использования. Существуют ресурсы, для которых возможна только такая форма использования. Такие ресурсы называются неделимыми. К таким неделимым ресурсам, допускающим только монопольное использование, можно отнести, например, печатающее устройство (использование его попеременно различными заданиями привело бы к “перемешиванию” на печати записей, относящихся к различным заданиям). К неделимым ресурсам относятся также лентопротяжные устройства, перфокарточные устройства ввода и вывода. Метод монопольного использования не предъявляет никаких дополнительных требований ни к ресурсам, ни к организации управления ими, но не всегда обеспечивает их эффективное использование.

*Совместное использование* ресурсов является средством, обеспечивающим выполнение всех присутствующих в машине заданий в условиях дефицита ресурсов, каждый процесс может владеть выделенными ему ресурсами лишь ограниченное время. Принято говорить, что несколько процессов разделяют ресурс, если этот ресурс в течение некоторого времени находится в распоряжении одного процесса, затем он может быть отобран у данного процесса и передан другому процессу. Иначе говоря, разделяемый ресурс используется несколькими процессами. Совместное использование разделяемого ресурса организуется по одному из следующих принципов:

по принципу попеременного использования;

по принципу параллельного использования.

При *попеременном использовании* разделяемый ресурс совместно используется несколькими процессами, попеременно переходя от процесса к процессу. Например, в системе с разделением времени пять процессов могут разделять (т. е. совместно использовать) один процессор, получая каждый по одной секунде процессорного времени из каждых пяти секунд работы процессора. Практически таким образом происходит и разделение других аппаратных ресурсов, различны лишь интервалы времени, на которые разные ресурсы предоставляются процессам.

При *параллельном использовании* разделяемым ресурсом несколько процессов могут пользоваться одновременно. Примером общедоступного ресурса, которым могут пользоваться сразу несколько процессов, может служить память.

Способ совместного использования ресурсов требует наличия специальных средств преодоления конфликтных ситуаций и управления очередями запросов, поступающих к совместно используемому ресурсу.

Монопольно используемые ресурсы называются *критическими*, поскольку использование их попеременно или параллельно несколькими процессами должно быть исключено, так как это может привести к неправильным результатам или к нежелательной форме представления резуль-

татов, например, как при попеременном использовании печатающего устройства.

С попеременно используемыми ресурсами в общем случае также необходимо обращаться как с критическими, в смысле исключения возможности их параллельного использования несколькими процессами. Вообще говоря, критический ресурс – это ресурс, который допускает обслуживание только одного процесса за один раз.

Возможный характер использования некоторых ресурсов зависит от характера операций обращения к ним. Так, например, несколько процессов, ограничив свои обращения к области данных в оперативной памяти только операциями чтения, могут работать с этим ресурсом одновременно, по отношению же к операции записи этот ресурс в общем случае является неделимым.

Способ *виртуального использования* ресурсов заключается в программной имитации такой среды для выполнения процессов, в которой ограниченные в количественном отношении реальные аппаратные ресурсы как бы расширяются. Ресурс, имитируемый программными средствами, называют виртуальным. Так, в системе с разделением времени с одним процессором имитируется среда, в которой из нескольких процессов каждый обладает виртуальным процессором (“работающим ” несколько медленнее реального физического процессора).

### 2.1.2. Программные ресурсы

Трансляторы, библиотеки программ пользователей, прикладные программы представляют собой программные ресурсы, которые подобно аппаратным ресурсам, подлежат распределению. Эти программы обычно хранятся во внешней памяти и загружаются в оперативную память только тогда, когда в них возникает необходимость. Программы в зависимости от возможностей использования их как ресурсов делятся на три типа:

1. Программы, которые изменяют себя во время выполнения. При каждом обращении к такой однократно используемой программе ее копию необходимо вызывать из внешней памяти.

2. Программы, которые также изменяют себя во время выполнения, однако обладают свойством самовосстанавливаемости: перед повторным использованием программы измененная во время выполнения часть этой программы восстанавливается в первоначальном виде. Одна и та же находящаяся в оперативной памяти копия такой повторно используемой программы может использоваться многократно, но только последовательно: в каждый момент времени она может находиться в распоряжении только одного процесса. Таким образом, такие программы ведут себя как критические ресурсы.

3. Программы, которые не изменяют себя во время выполнения, в связи с чем их называют чистыми процедурами. При выполнении программы этого типа из принадлежащей ей области оперативной памяти осуществляется только считывание. Результаты, направляемые в оперативную память, записываются в область данных, принадлежащую процессу, управляемому данной программой.

Чистые процедуры обеспечивают возможность их параллельного исполнения несколькими процессами. Для каждого процесса выделяется его собственная область данных, а с единственным экземпляром "рабочей" части программы все процессы могут работать как с общедоступным ресурсом. Процесс, управляемый такой программой, может быть прерван до своего завершения. В ту же программу может быть осуществлен повторный вход другим процессом, и такой повторный вход никак не влияет на выполнение первого процесса, который будет использовать эту программу позднее. Поэтому программы данного типа называют реентерабельными.

Использование чистых процедур дает значительную экономию памяти, особенно в системах с одновременным обслуживанием многих пользователей. В современных вычислительных машинах все системные обрабатывающие программы (трансляторы, редакторы связей и т. п.) оформляются как чистые процедуры.

К программным ресурсам, кроме программ, относятся и личные данные пользователей. ОС обеспечивает их сохранность и регулирует к ним доступ.

## 2.2. Понятие и задачи управления ресурсами

Организация эффективного управления ресурсами – главная задача ОС. Чтобы решить эту задачу, ОС должна полностью контролировать распределение ресурсов вычислительной машины. Достигается это обычно тем, что с каждым из основных ресурсов связывается отдельная программа управления данным ресурсом, которую называют диспетчером (супервизором или распорядителем) этого ресурса. В функции каждого диспетчера входит:

1) отслеживание состояния ресурса;
2) определение получателя ресурса;
3) выделение ресурса;
4) освобождение ресурса.

Когда процессу требуется ресурс, он запрашивает его у диспетчера данного ресурса.

Распределение ресурсов ЭВМ представляет собой весьма сложную задачу. Сложность ее состоит в том, что в общем случае требование хорошего обслуживания заданий и требование использования ресурсов с максимальной эффективностью являются противоречивыми. Решение задачи

распределения ресурсов в каждой конкретной обстановке представляет собой некоторый компромисс между этими двумя требованиями. Для разделения неделимых ресурсов и ресурсов, допускающих разделение, используются, как правило, разные стратегии.

Освобождение ресурса может происходить:

1) по указанию владевшего этим ресурсом процесса (например, освобождение затребованной ранее им области оперативной памяти);

2) в результате действий ОС, отбирающей данный попеременно используемый ресурс у процесса (например, передача процессора от одного процесса другому по принципу квантования времени);

3) после нормального или аварийного завершения процесса.

Особый случай представляет освобождение процессора процессом, переходящим в состояние ожидания некоторого события.

Подход к распределению ресурсов влияет не только на эффективность, но и вообще на правильность работы ЭВМ. Игнорирование последнего обстоятельства таит в себе опасность взаимной блокировки нескольких процессов, выполняемых вычислительной машиной. Возникновение взаимной блокировки, как следствие неправильного распределения ресурсов, можно проиллюстрировать простым примером. Предположим, что ЭВМ с одним фотосчитывающим механизмом и одним печатающим устройством выполняются два процесса – *A* и *B*, причем по их запросам процессу *A* было выделено читающее устройство, а процессу *B* – печатающее. В какой-то момент времени процесс *A* запрашивает печатающее устройство. Запрос не удовлетворяется, так как печатающее устройство занято процессом *B*, поэтому выполнение процесса *A* прерывается. Позже процесс *B* выдаст запрос на читающее устройство, который тоже не удовлетворяется ввиду занятости устройства. Процесс *B* также прерывается. Оба процесса могут бесконечно долго ожидать освобождения требуемого ресурса. Ситуацию такого рода называют тупиком.

Используются два подхода к решению проблем тупиков. Первый подход состоит в таком распределении ресурсов, которое предотвращало бы возникновение тупиков. Этот подход может приводить к издержкам из-за того, что при предварительном “безопасном” распределении ресурсов часть закрепляемых за процессом ресурсов фактически может ему не потребоваться и становится, таким образом, “данью за боязнь” возникновения тупика (эти ресурсы могли быть использованы другими процессами).

В некоторых системах идут на риск и допускают возникновение тупиков, применяя те или иные методы их обнаружения. Такой подход может быть использован в системах пакетной обработки, где конфликт двух процессов может быть разрешен в пользу одного из них, а второй процесс может быть запущен повторно. Ясно, что в машинах, работающих в режиме реального времени, такой подход недопустим.

## 2.3. Дисциплины распределения ресурсов, используемые в операционных системах

Сама идея мультипрограммирования непосредственно связана с наличием очередей процессов. Так, процессор – основной ресурс многопрограммной ЭВМ по очереди предоставляется процессам. Подобные очереди неизбежно имеют место при обращении к каналам, к широко используемым внешним устройствам (например, к устройствам печати), наборам данных, модулям операционной системы.

Использование многими процессами того или иного ресурса, который в каждый момент времени может обслуживать лишь один процесс, осуществляется с помощью дисциплин распределения ресурса. Их основой являются:

1) дисциплины формирования очередей на ресурсы или совокупность правил, определяющих размещение процессов в очереди;

2) дисциплины обслуживания очереди или совокупность правил извлечения одного из процессов очереди с последующим представлением выбранному процессу ресурса для использования.

Основным конструктивным, согласующим элементом при реализации той или иной дисциплины диспетчеризации является очередь, в которую по определенным правилам заносятся и извлекаются запросы.

Определяющее влияние на сущность (содержание) дисциплины формирования очередей оказывают:

1) информация о классах и приоритетах заданий и шагов заданий, информация о необходимости обращения к тем или иным устройствам, массивам данных, зафиксированных в операторах языка управления заданиями;

2) соглашения о приоритете уровней запросов прерывания и прерывающих программ, принимаемых при проектировании, разработке ЭВМ;

3) наборы соглашений, принимаемых в вычислительном центре;

4) используемая дисциплина обслуживания очередей, которая зачастую определяет и дисциплину формирования очереди.

Дисциплины формирования очередей разделяются на два класса:

*статический*, где приоритеты назначаются до выполнения пакета заданий; большинство рассмотренных фактов определяет содержание статических дисциплин;

*динамический*, при котором приоритеты определяются в процессе выполнения пакета.

Оба класса широко используются в практике организации вычислительного процесса в ЭВМ.

В ЭВМ не только многопрограммных, но и однопрограммных, однопроцессорных, многопроцессорных широко используется ряд дисциплин обслуживания очередей, ставших классическими. Эти дисциплины рас-

пределения ресурсов часто называют базовыми. Рассмотрим наиболее часто встречающиеся на практике дисциплины обслуживания.

Дисциплина обслуживания в порядке поступления (рис. 2.12.*а*). "Первый пришел – первый обслуживается". В литературе эта дисциплина обозначается как FIFO (First in – First out). Все заявки поступают в конец очереди. Первыми обслуживаются заявки, находящиеся в начале очереди. Самая простая и широко используемая на практике.

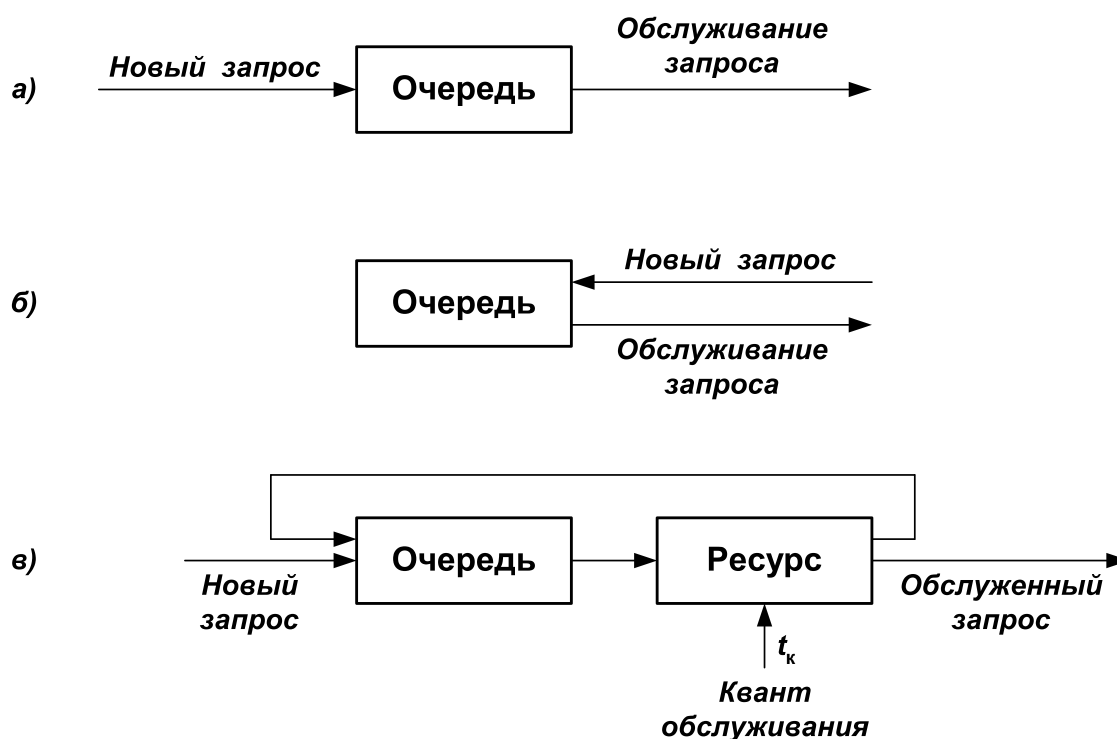



Рис. 2.12. Схемы базовых дисциплин обслуживания процессов:
*а* – первый пришел – первый обслуживается;
*б* – последний пришел – первый обслуживается;
*в* – круговой циклический алгоритм

Дисциплина обслуживания в порядке, обратном порядку поступления (рис. 2.12.*б*). "Последний пришел – первый обслуживается". Обозначается LIFO (Last in – First out). Так же, как и FIFO, проста в реализации и широко используется на практике. Данная дисциплина является основой построения стековой памяти.

Общим для названных дисциплин является простота их реализации и определенная "справедливость" в обслуживании всего потока запросов, поступающих в систему. Среднее время ожидания запросов в очереди при некотором установившемся темпе обслуживания и темпе поступления является одинаковым независимо от характеристик процессов-пользователей. Например, если некоторые процессы предполагают длительное использование ресурсов (отрабатываются "длинные" запросы), а другие, наоборот, – непродолжительное (отрабатываются "короткие" за-

просы), то и “длинные” и “короткие” запросы будут ожидать в очереди в среднем одинаково. Дисциплина FIFO помимо функционального отличия обеспечивает минимизацию дисперсии времени ожидания.

*Круговой циклический алгоритм* (рис. 2.12.*в*). В основе данной дисциплины лежит дисциплина FIFO. Но время обслуживания каждого процесса ограничено и определяется так называемым квантом времени $t_к$. Если запрос на использование ресурса из начала очереди обслуживается до конца за время $t_к$ (например, программа процесса за время $t_к$ полностью выполнена на процессоре), то он покидает очередь. Если этот запрос не успевает обслужиться до конца, то его обслуживание прерывается и он поступает в конец очереди. Дисциплина широко используется на практике, в частности при реализации режима разделения времени.

Хотя в данной дисциплине нет явных приоритетов, здесь в наиболее благоприятных условиях оказываются короткие запросы, т. е. запросы от процессов, которым требуется меньшее время использования ресурсов. Короткие запросы обслуживаются быстрее, т. е. имеют меньшие средние времена ожидания в системе, чем длинные запросы. Степень благоприятствования коротким запросам тем больше, чем меньше длительность кванта мультиплексирования, чем ближе она к длительности интервала номинального использования ресурса процессом. Однако уменьшение длительности кванта ведет к увеличению накладных расходов, необходимых для отработки прерываний и перераспределения ресурса. Это происходит из-за возрастания частоты прерываний, что особенно неблагоприятно может сказаться на отработке “длинных” запросов. Поэтому на практике используют различные модификации данного алгоритма.

Все рассмотренные дисциплины являются одноочередными. В операционных системах ЭВМ широко используются *многоочередные дисциплины* (рис. 2.13). Здесь организуется $N$ очередей. Все новые запросы поступают в конец первой очереди. Первый запрос из очереди $i$ ($1 \le i \le N$) поступает на обслуживание лишь тогда, когда все очереди от 1 до ($i - 1$)-й пустые. На обслуживание выделяется квант времени $t_к$. Если за это время обслуживание запроса завершается полностью, то он покидает систему. В противном случае недообслуженный запрос поступает в конец очереди с номером $i + 1$.

После обслуживания из очереди $i$ система выбирает для обслуживания запрос из непустой очереди с самым младшим номером. Таким запросом может быть следующий запрос из очереди $i$ или из очереди $i + 1$ (при условии, что после обслуживания запроса из очереди $i$ последняя оказалась пустой). Новый запрос поступает в первую очередь ($i = 1$). В такой ситуации после окончания времени $t_к$, выделенного для обслуживания запроса из очереди $i$, будет начато обслуживание запроса 1-й очереди.

Если система выходит на обслуживание заявок из очереди $N$, то они обслуживаются либо по дисциплине FIFO (каждая заявка обслуживается до конца), либо по круговому циклическому алгоритму.

Рис. 2.13. Схема многоочередной дисциплины обслуживания

Данная система наиболее быстро обслуживает все короткие по времени обслуживания запросы. Недостаток системы заключается в непроизводительных затратах времени на перемещение запросов из одной очереди в другую.

Все рассмотренные дисциплины обеспечивают не только обслуживание очередей, но и их формирование. Для всех таких дисциплин характерно отсутствие приоритетности запросов при их поступлении в систему. Но после поступления каждый запрос приобретает приоритет, который изменяется в процессе обслуживания одной очереди или многих очередей.

Существуют многоочередные дисциплины, в которые поступают запросы, имеющие определенный приоритет на обслуживание тем или иным ресурсом (рис. 2.14). Основу этой дисциплины представляет ранее рассмотренная многоочередная дисциплина. Поступающие в систему новые запросы попадают в очередь в соответствии с имеющимися приоритетами.

В многоочередных дисциплинах возможны различные стратегии поведения системы по отношению к новым запросам, поступающим в систему. Эти стратегии определяются дисциплинами обслуживания запросов с абсолютными и относительными приоритетами.

*Обслуживание с абсолютным приоритетом*. В многоочередной дисциплине, где вновь поступающий запрос имеет определенный приоритет, используется обслуживание с абсолютным приоритетом.

Рис. 2.14. Схема приоритетной многоочередной
дисциплины обслуживания

Здесь приоритет определяется очередью (ее номером), и первыми обслуживаются запросы, обладающие наивысшим приоритетом (из очереди с меньшим номером). Допустим, система обслуживает запрос из очереди с номером *i,* где $1 < i \le N$ и в систему поступает более приоритетный запрос в очередь, например, с номером *i*-1. В таких условиях обслуживание *i*-го уровня прерывается и система начинает обслуживать запрос *i*-1 уровня. После окончания его обслуживания происходит дообслуживание прерванного запроса *i*-го уровня.

В данной дисциплине еще более увеличивается степень дискриминации по среднему времени ожидания в очереди между высоко- и низкоприоритетными запросами. Время ожидания высокоприоритетных заявок сокращается, но снова за счет большего ухудшения в обслуживании низкоприоритетных заявок.

Цена за увеличение степени дискриминации – усложнение логики системы, а следовательно, и ее реализации. Кроме того, появляется проблема прерывания. Во-первых, появляются накладные расходы, необходимые для отработки прерывание процесса выполнения и перераспределения ресурса. Эти расходы при достаточной интенсивности прерываний могут стать весьма ощутимыми. Во-вторых, необходимо выбрать наиболее правомочное правило о дооб-служивании прерываемых процессов – когда выделять им вновь ресурс, учитывать или нет, что ресурс уже использовался до прерывания, и т. д.

*Обслуживание с относительным приоритетом*. При данной дисциплине заявка, входящая в систему, не вызывает прерывания об-

служиваемой заявки, даже если последняя и менее приоритетная. В данном случае только после окончания обслуживания менее приоритетной заявки начнется обслуживание более приоритетной.

Все изложенное касается дисциплины распределения ресурсов без учета взаимосвязи процессов. Процессы, как правило, используют различные ресурсы и этот факт оказывает влияние на рассмотренные дисциплины распределения. На практике должна быть построена стратегия распределения, удовлетворительная не только в отношении некоторого конкретного ресурса, но и согласованная со стратегиями распределения других ресурсов.

Реализация дисциплин распределения на практике также усложнится из-за необходимости анализа возможности возникновения тупиковых ситуаций, проверки анализа полномочий на использование каждым процессом распределяемого ресурса.

Наряду с рассмотренными дисциплинами распределения ресурсов существуют и другие, содержание которых определяется спецификой распределяемого ресурса. Много таких оригинальных дисциплин имеет место при статическом и динамическом распределении между процессами оперативной памяти.

## 3. Принципы построения интерфейсов операционных систем

ОС всегда выступает как интерфейс между аппаратурой компьютера и пользователем с его задачами. Под интерфейсами операционных систем здесь и далее следует понимать специальные интерфейсы системного и прикладного программирования, предназначенные для выполнения следующих задач:

1. *Управление процессами*, которое включает в себя следующий набор основных функций:

запуск, приостанов и снятие задачи с выполнения;

задание или изменение приоритета задачи;

взаимодействие задач между собой (механизмы сигналов, семафорные примитивы, очереди, конвейеры, почтовые ящики);

RPC (remote procedure call) – удаленный вызов подпрограмм.

2. *Управление памятью*:

запрос на выделение блока памяти;

освобождение памяти;

изменение параметров блока памяти (например, память может быть заблокирована процессом либо предоставлена в общий доступ);

отображение файлов на память (имеется не во всех системах).

3. *Управление вводом/выводом*:

запрос на управление виртуальными устройствами (напомним, что управление вводом/выводом является привилегированной функцией са-

мой ОС, и никакая из пользовательских задач не должна иметь возможности непосредственно управлять устройствами);

файловые операции (запросы к системе управления файлами на создание, изменение и удаление данных, организованных в файлы).

Здесь перечислены основные наборы функций, которые выполняются ОС по соответствующим запросам от задач. Что касается пользовательского интерфейса операционной системы, то он реализуется с помощью специальных программных модулей, которые принимают его команды на соответствующем языке (возможно, с использованием графического интерфейса) и транслируют их в обычные вызовы в соответствии с основным интерфейсом системы. Обычно эти модули называют интерпретатором команд. Получив от пользователя команду, такой модуль после лексического и синтаксического анализа либо сам выполняет действие, либо, что случается чаще, обращается к другим модулям ОС, используя механизм API. Надо заметить, что в последние годы большую популярность получили графические интерфейсы (GUI), в которых задействованы соответствующие манипуляторы типа “мышь” или “трекбол”. Указание курсором на объекты и щелчок (клик) или двойной щелчок по соответствующим клавишам приводит к каким-либо действиям – запуску программы, ассоциированной с указываемым объектом, выбору и/или активизации пунктов меню и т. д. Можно сказать, что такая интерфейсная подсистема транслирует “команды” пользователя в обращения к ОС.

## 3.1. Интерфейс прикладного программирования

### 3.1.1. Понятие интерфейса прикладного программирования

Прежде всего необходимо однозначно разделить общий термин API (application programming interface, интерфейс прикладного программирования) на следующие направления:

1) API как интерфейс высокого уровня, принадлежащий к библиотекам RTL;

2) API прикладных и системных программ, входящих в поставку операционной системы;

3) прочие API.

*Интерфейс прикладного программирования*, как это и следует из названия, *предназначен* для использования прикладными программами системных ресурсов ОС и реализуемых ею функций. API описывает совокупность функций и процедур, принадлежащих ядру или надстройкам ОС.

Итак, API представляет собой набор функций, предоставляемых системой программирования разработчику прикладной программы и ориентированных на организацию взаимодействия результирующей прикладной

программы с целевой вычислительной системой. Целевая вычислительная система представляет собой совокупность программных и аппаратных средств, в окружении которых выполняется результирующая программа. Сама результирующая программа порождается системой программирования на основании кода исходной программы, созданного разработчиком, а также объектных модулей и библиотек, входящих в состав системы программирования.

В принципе API используется не только прикладными, но и многими системными программами как в составе ОС, так и в составе системы программирования.

Но дальше речь пойдет только о функциях API с точки зрения разработчика прикладной программы. Для системной программы существуют некоторые дополнительные ограничения на возможные реализации API.

Функции API позволяют разработчику строить результирующую прикладную программу так, чтобы использовать средства целевой вычислительной системы для выполнения типовых операций. При этом разработчик программы избавлен от необходимости создавать исходный код для выполнения этих операций.

Программный интерфейс API включает в себя не только сами функции, но и соглашения об их использовании, которые регламентируются операционной системой (ОС), архитектурой целевой вычислительной системы и системой программирования.

Существует несколько вариантов реализации API:

1) реализация на уровне ОС;

2) реализация на уровне системы программирования;

3) реализация на уровне внешней библиотеки процедур и функций.

Система программирования в каждом из этих вариантов предоставляет разработчику средства для подключения функций API к исходному коду программы и организации их вызовов. Объектный код функций API подключается к результирующей программе компоновщиком при необходимости.

Возможности API можно оценивать со следующих позиций:

1) эффективность выполнения функций API – включает в себя скорость выполнения функций и объем вычислительных ресурсов, потребных для их выполнения;

2) широта предоставляемых возможностей;

3) зависимость прикладной программы от архитектуры целевой вычислительной системы.

В идеале хотелось бы иметь набор функций API, выполняющихся с наивысшей эффективностью, предоставляющих пользователю все возможности современных ОС и имеющих минимальную зависимость от архитектуры вычислительной системы (еще лучше – лишенных такой зависимости).

Добиться наивысшей эффективности выполнения функций API практически трудно по тем же причинам, по которым невозможно добиться наивысшей эффективности выполнения для любой результирующей программы. Поэтому об эффективности API можно говорить только в сравнении его характеристик с другим API.

Что касается двух других показателей, то в принципе нет никаких технических ограничений на их реализацию.

### 3.1.2. Реализация функций API на уровне ОС

При реализации функций API на уровне ОС за их выполнение ответственность несет ОС. Объектный код, выполняющий функции, либо непосредственно входит в состав ОС (или даже ядра ОС), либо поставляется в составе динамически загружаемых библиотек, разработанных для данной ОС. Система программирования ответственна только за то, чтобы организовать интерфейс для вызова этого кода.

В таком варианте результирующая программа обращается непосредственно к ОС. Поэтому достигается наибольшая эффективность выполнения функций API по сравнению со всеми другими вариантами реализации API.

Недостатком организации API по такой схеме является практически полное отсутствие переносимости не только кода результирующей программы, но и кода исходной программы. Программа, созданная для одной архитектуры вычислительной системы, не сможет исполняться на вычислительной системе другой архитектуры даже после того, как ее объектный код будет полностью перестроен. Чаше всего система программирования не сможет выполнить перестроение исходного кода для новой архитектуры вычислительной системы, поскольку многие функции API, ориентированные на определенную ОС, будут в новой архитектуре просто отсутствовать.

Таким образом, в данной схеме для переноса прикладной программы с одной целевой вычислительной системы на другую будет требоваться изменение исходного кода программы.

Переносимости можно было бы добиться, если унифицировать функции API в различных ОС. С учетом корпоративных интересов производителей ОС такое направление их развития представляется практически невозможным. В лучшем случае при приложении определенных организационных усилий удается добиться стандартизации смыслового (семантического) наполнения основных функций API, но не их программного интерфейса.

Хорошо известным примером API такого рода может служить набор функций, предоставляемых пользователю со стороны ОС типа Microsoft Windows – WinAPI (Windows API).

### 3.1.3. Реализация функций API на уровне системы программирования

Если функции API реализуются на уровне системы программирования, они предоставляются пользователю в виде библиотеки функций соответствующего языка программирования. Обычно речь идет о библиотеке времени исполнения – RTL (run time library). Система программирования предоставляет пользователю библиотеку соответствующего языка программирования и обеспечивает подключение к результирующей программе объектного кода, ответственного за выполнение этих функций.

Очевидно, что эффективность функций API в таком варианте будет несколько ниже, чем при непосредственном обращении к функциям ОС. Так происходит, поскольку для выполнения многих функций API библиотека RTL языка программирования должна все равно выполнять обращения к функциям ОС. Наличие всех необходимых вызовов и обращений к функциям ОС в объектном коде RTL обеспечивает система программирования.

Однако переносимость исходного кода программы в таком варианте будет самой высокой, поскольку синтаксис и семантика всех функций будут строго регламентированы в стандарте соответствующего языка программирования. Они зависят от языка и не зависят от архитектуры целевой вычислительной системы. Поэтому для выполнения прикладной программы на новой архитектуре вычислительной системы достаточно заново построить код результирующей программы с помощью соответствующей системы программирования.

Единообразное выполнение функций языка обеспечивается системой программирования. При ориентации на различные архитектуры целевой вычислительной системы в системе программирования могут потребоваться различные комбинации вызовов функций ОС для выполнения одних и тех же функций исходного языка. Это должно быть учтено в коде RTL. В общем случае для каждой архитектуры целевой вычислительной системы будет требоваться свой код RTL языка программирования. Выбор того или иного объектного кода RTL для подключения к результирующей программе система программирования обеспечивает автоматически.

Например, рассмотрим функции динамического выделения памяти в языках C и Pascal. В C это функции `malloc`, `realloc` и `free` (в C++ – `new` и `delete`), в Pascal – функции `new` и `dispose`. Если использовать эти функции в исходном тексте программы, то с точки зрения исходной программы они будут действовать одинаковым образом в зависимости только от семантики исходного кода. При этом для разработчика исходной программы не имеет значения, на какую архитектуру ориентирована его программа. Это имеет значение для системы программирования, которая для каждой из этих функций должна подключить к результирующей программе объектный код библиотеки. Этот код будет выполнять обращение к соответствующим функциям ОС. Не исключено даже, что для однотипных по смыслу функций в разных языках (например, `malloc` в C и

`new` в языке Pascal выполняют схожие по смыслу действия) этот код будет выполнять схожие обращения к ОС. Однако для различных вариантов ОС этот код будет различен даже при использовании одного и того же исходного языка.

Проблема, главным образом, заключается в том, что большинство языков программирования предоставляют пользователю не очень широкий набор стандартизованных функций. Поэтому разработчик исходного кода существенно ограничен в выборе доступных функций API. Как правило, набора стандартных функций оказывается недостаточно для создания полноценной прикладной программы. Тогда разработчик может обратиться к функциям других библиотек, имеющихся в составе системы программирования. В этом случае нет гарантии, что функции, включенные в состав данной системы программирования, но не входящие в стандарт языка программирования, будут доступны в другой системе программирования. Особенно если она ориентирована на другую архитектуру целевой вычислительной системы. Такая ситуация уже ближе к третьему варианту реализации API.

Например, те же функции `malloc`, `realloc` и `free` в языке C фактически не входят в стандарт языка. Они входят в состав стандартной библиотеки, которая "де-факто" входит во все системы программирования, построенные на основе языка C. Общепринятые стандарты существуют для многих часто используемых функций языка. Если же взять более специфические функции, такие как функции порождения новых процессов, то для них ни в C, ни в языке Pascal не окажется общепринятого стандарта.

### 3.1.4. Реализация функций API с помощью внешних библиотек

При реализации функций API с помощью внешних библиотек они предоставляются пользователю в виде библиотеки процедур и функций, созданной сторонним разработчиком. Причем разработчиком такой библиотеки может выступать тот же самый производитель.

Система программирования ответственна только за то, чтобы подключить объектный код библиотеки к результирующей программе. Причем внешняя библиотека может быть и динамически загружаемой (загружаемой во время выполнения программы).

С точки зрения эффективности выполнения этот метод реализации API имеет самые низкие результаты, поскольку внешняя библиотека обращается как к функциям ОС, так и к функциям RTL языка программирования. Только при очень высоком качестве внешней библиотеки ее эффективность становится сравнимой с библиотекой RTL.

Если говорить о переносимости исходного кода, то здесь требование только одно – используемая внешняя библиотека должна быть доступна в любой из архитектур вычислительных систем, на которые ориентирована

прикладная программа. Тогда удается достигнуть переносимости. Это возможно, если используемая библиотека удовлетворяет какому-то принятому стандарту, а система программирования поддерживает этот стандарт.

Например, библиотеки, удовлетворяющие стандарту POSIX, доступны в большинстве систем программирования для языка C. И если прикладная программа использует только библиотеки этого стандарта, то ее исходный код будет переносимым. Еще одним примером является широко известная библиотека графического интерфейса XLib, поддерживающая стандарт графической среды X Window.

Для большинства специфических библиотек отдельных разработчиков это не так. Если пользователь использует какую-то библиотеку, то она ориентирована на ограниченный набор доступных архитектур целевой вычислительной системы. Примерами могут служить библиотеки MFC (Microsoft foundation classes) фирмы Microsoft и VCL (visual controls library) фирмы Borland, жестко ориентированные на архитектуру ОС типа Windows.

Тем не менее, многие фирмы-разработчики сейчас стремятся создать библиотеки, которые бы обеспечивали переносимость исходного кода приложений между различными архитектурами целевой вычислительной системы. Пока еще такие библиотеки не получили широкого распространения, но есть несколько попыток их реализации – например, библиотека CLX производства фирмы Borland ориентирована на архитектуру ОС типа Linux и ОС типа Windows.

В целом развитие функций прикладного API идет в направлении попытки создать библиотеки API, обеспечивающие широкую переносимость исходного кода. Однако, учитывая корпоративные интересы различных производителей и сложившуюся ситуацию на рынке системного программного обеспечения, в ближайшее время вряд ли удастся достичь значительных успехов в этом направлении. Разработка широко применимого стандарта API пока еще остается делом будущего.

Поэтому разработчики системных программ вынуждены оставаться в довольно узких рамках ограничений стандартных библиотек языков программирования.

Что касается прикладных программ, то гораздо большую перспективу для них предоставляют технологии, связанные с разработками в рамках архитектуры “клиент-сервер” или трехуровневой архитектуры создания приложений. В этом направлении ведущие производители ОС, СУБД и систем программирования скорее достигнут соглашений, чем в направлении стандартизации API.

## 3.2. Платформенно-независимый интерфейс POSIX

POSIX – платформенно-независимый системный интерфейс для компьютерного окружения. Это стандарт IEEE (Institute of Electrical and Electronical Engineers), описывающий системные интерфейсы для открытых операционных систем, в том числе оболочки, утилиты и инструментарии. Помимо этого, согласно POSIX, стандартизированными являются задачи обеспечения безопасности, задачи реального времени, процессы администрирования, сетевые функции и обработка транзакций. Стандарт базируется на UNIX-системах, но допускает реализацию и в других ОС.

POSIX возник как попытка всемирно известной организации IEEE пропагандировать переносимость приложений в UNIX-средах путем разработки абстрактного, платформенно-независимого стандарта. Однако POSIX не ограничивается только UNIX-системами; существуют различные реализации этого стандарта в системах, которые соответствуют требованиям, предъявляемым стандартом IEEE Standard 1003.1-1990 (POSIX.1). Например, известная ОС реального времени QNX соответствует спецификациям этого стандарта, что облегчает перенос приложений в эту систему, но UNIX-системой не является ни в каком виде, ибо ее архитектура использует абсолютно иные принципы.

Этот стандарт подробно описывает VMS (virtual memory system, систему виртуальной памяти), многозадачность (MPE, multiprocess executing) и технологию переноса операционных систем (CTOS). Таким образом, на самом деле POSIX представляет собой множество стандартов, именуемых POSIX.1 – POSIX.12.

В табл. 2.1 приведены основные направления, описываемые данными стандартами. Следует также особо отметить, что POSIX.1 предполагает язык С как основной язык описания системных функций API.

Таблица 2.1

Семейство стандартов POSIX

| Стандарт | Краткое описание |
|---|---|
| POSIX.0 | Введение в стандарт открытых систем. Данный документ не является стандартом в чистом виде, а представляет собой рекомендации и краткий обзор технологий |
| POSIX.1 | Системный API (язык С) |
| POSIX.2 | Оболочки и утилиты (одобренные IEEE) |
| POSIX.3 | Тестирование и верификация |
| POSIX.4 | Задачи реального времени и потоки |
| POSIX.5 | Использование языка ADA применительно к стандарту POSIX.1 |
| POSIX.6 | Системная безопасность |
| POSIX.7 | Администрирование системы |

Окончание таблицы 2.1

| Стандарт | Краткое описание |
|---|---|
| POSIX.8 | Сети<br>“Прозрачный” доступ к файлам<br>Абстрактные сетевые интерфейсы, не зависящие от физических протоколов<br>RPC (remote procedure calls, вызовы удаленных процедур)<br>Связь системы с протоколо-зависимыми приложениями |
| POSIX.9 | Использование языка FORTRAN применительно к стандарту POSIX.1 |
| POSIX.10 | Super-computing Application Environment Profile (AEP)<br>Профиль прикладной среды организации вычислений на супер-ЭВМ |
| POSIX.11 | Обработка транзакций AEP |
| POSIX.12 | Графический интерфейс пользователя (GUI) |

Таким образом, программы, написанные с соблюдением данных стандартов, будут одинаково выполняться на всех POSIX-совместимых системах. Однако стандарт в некоторых случаях носит лишь рекомендательный характер. Часть стандартов описана очень строго, тогда как другая часть только поверхностно раскрывает основные требования. Нередко программные системы заявляются как POSIX-совместимые, хотя таковыми их назвать нельзя. Причины кроются в формальности подхода к реализации POSIX-интерфейса в различных ОС. На рис. 2.15 изображена типовая схема реализации строго соответствующего POSIX приложения. Для взаимодействия с операционной системой программа использует только библиотеки POSIX.1 и стандартную библиотеку RTL языка С, в которой возможно использование лишь 110 различных функций, также описанных стандартом POSIX.1.

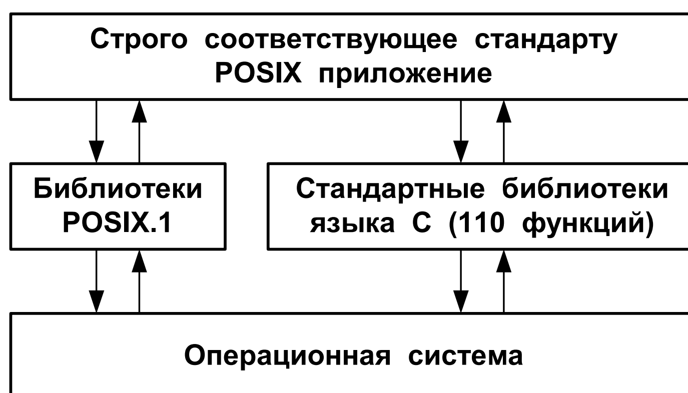



Рис. 2.15. Приложения, строго соответствующие стандарту POSIX

К сожалению, достаточно часто с целью увеличить производительность той или иной подсистемы либо из соображений введения фирменных технологий, которые ограничивают использование приложения соответствующей операционной средой, при программировании используются другие функции, не отвечающие стандарту POSIX.

Реализации POSIX API на уровне операционной системы различны. Если UNIX-системы в своем абсолютном большинстве изначально соответствуют спецификациям IEEE Standard 1003.1-1990, то WinAPI не является POSIX-совместимым. Однако для поддержки данного стандарта в операционной системе MS Windows NT введен специальный модуль поддержки POSIX API, работающий на уровне привилегий пользовательских процессов. Данный модуль обеспечивает конвертацию и передачу вызовов из пользовательской программы к ядру системы и обратно, работая с ядром через WinAPI. Прочие приложения, написанные с использованием WinAPI, могут передавать информацию POSIX-приложениям через стандартные механизмы потоков ввода/вывода.

## 4. Принципы построения и организации ОС АСУ войсковой ПВО

### 4.1. Общие сведения об операционных системах реального времени

Операционные системы (ОС) являются одним из классов программных средств вычислительных систем, как общих, так и встроенных, обеспечивающих решение различных задач (в том числе – в режиме реального времени).

Перечень характеристик операционных систем, определяющих их возможности по решению требуемых задач, а также работе в различных режимах может включать следующие:

аппаратная платформа и скорость работы ОС;

поддерживаемое периферийное оборудование;

возможности для организации сетей;

обеспечение совместимости с другими операционными системами;

переносимость ОС на другие аппаратные платформы;

наличие в составе ОС инструментальных средств для разработки прикладных систем и др.

<u>Операционная система реального времени предназначена</u> для обеспечения разработки прикладного программного обеспечения пользователя, включая трансляцию исходного текста программ, компоновку программ и получение загрузочного модуля, поддержку режима отладки программ пользователя в целевой (инструментальной) ЭВМ, а также для обеспечения функционирования программ пользователя в режиме реального времени и в режиме отладки на целевой ЭВМ.

В настоящее время большинство ОС разрабатывается по двум основным направлениям: в виде единого программного ядра, либо на основе концепции микроядра.

Основной недостаток ОС, выполненных в виде единого программного ядра, обусловлен обязательностью изменения всей системы в целом при необходимости изменения каких-либо отдельных ее функций.

Концепция микроядра свободна от указанного недостатка. Ее преимущество заключается в том, что каждый компонент системы представляет собой самостоятельный процесс, запуск или остановка которого не влияет на работоспособность других процессов.

Все ОС реального времени могут быть классифицированы также по способу и месту проектирования и исполнения прикладных программ.

К первому классу операционных систем, получивших название “self-hosted”, относятся ОС, в которых проектирование прикладного программного обеспечения и его исполнение производится на одной и той же ЭВМ (ОС OS9/9000 и QNX). Операционные системы этого класса принято называть системами “мягкого” реального времени. Их использование осуществляется в случаях, когда конечное применение не требует высокой производительности ОС, максимальной скорости реакции на внешние события и строгой детерминированности ее поведения.

Второй класс составляют ОС, в которых применяется кросс-технология, т.е. проектирование прикладного программного обеспечения ведется на одной машине (host), называемой “инструментальной”, а разработанное программное обеспечение исполняется на другой, “целевой” машине (target). Кросс-технология ОС позволяет применять их в системах “жесткого” реального времени, в которых недостаточная скорость реакции или непредсказуемость поведения влечет за собой “аварию” на объекте управления, и/или невыполнение (боевой) задачи, и/или создает опасность для жизни людей. Кросс-технология обусловливает закрепление за ОС только тех функций, которые необходимы для исполнения целевого программного обеспечения, перенеся функции разработки на “инструментальную” ЭВМ.

Кроме того, встроенные управляющие машины, работающие в режиме “жесткого” реального времени, как правило, не дают возможности реализовать на них средства проектирования в необходимом объеме из-за отсутствия развитого человеко-машинного интерфейса, ограниченного объема оперативной памяти и отсутствия внешних запоминающих устройств. Такие машины, к тому же, глубоко встроены в объект управления, чтобы их можно было использовать в качестве инструментальных.

Система реального времени – многозадачная система, быстродействие которой в значительной степени определяется скоростью переключения задач. Учет этого фактора необходим не только при создании программных комплексов реального времени, но и для любых диалоговых систем.

Медленная реакция системы не только снижает скорость работы, но и в значительной мере раздражает пользователя.

В табл. 2.2 приведены результаты тестирования ряда ОС на различных аппаратных платформах в порядке возрастания времени на переключение. Из таблицы следует, что лучшими характеристиками в отношении минимизации времени на переключение обладают ОС VxWorks, OS 9/9000 и QNX.

Таблица 2.2

Время переключения задач в различных ОС

| ОС | ЭВМ | Время на переключение (микросекунды) |
|---|---|---|
| VxWorks | MC680040-25 | 6 |
| VxWorks | R3000-25 | 24 |
| OS9/9000 | PowerPC601 | 26 |
| QNX 4.2 | ALR Pentium-60 | 28 |
| HP-RT 1.1 | HP-747x-100 | 34 |
| QNX 4.2 | IBM 80486DX2-60 | 44 |
| DEC OSF/1 V1.3 | DEC 21064-150 | 93 |
| SunOS 4.1.3 | SuperSPARCv8-50 | 95 |
| HP-UX 9.x | Snake-60 | 106 |
| SunOS 4.1.3 | Viking-40 | 128 |
| Ultrix 4.3 | Digital MIPS R3000-40 | 132 |
| Linux 0.99. 13p | Gateway 8048DX2-66 | 171 |
| SunOS 4.1.1 | SunSPARC-33 | 198 |
| 386BSD 0.1 | IBM 486DX2-33 | 210 |
| AIX 3.2 | RIOS-50 | 212 |
| SunOS 4.1.3 | SPARCv7-50 | 230 |
| Unicons | Cray Y/MP | 373 |
| QNX 4.2 | IBM 386SX-16 | 525 |
| Solaris 2.3 | MicroSPARCv8-50 | 595 |

Работа системы в реальном масштабе времени предъявляет жесткие требования к производительности файловой системы. При этом в различных операционных системах используемые принципы организации файлов также различны.

Общая проблема всех операционных систем реального времени состоит в том, что каждая такая система снабжена к тому же своим собственным уникальным интерфейсом. Отсутствие совместимости разрабатываемых ОС ограничивает расширение возможностей любой системы реального времени по решению новых задач, использующей в своей работе определенную ОС. Такое положение порождает дилемму: отказаться от

новых возможностей сейчас и ждать пока соответствующие разработчики модернизируют данную ОС или не создадут лучшую, но совместимую со "старой"; либо приобрести и установить новую ОС, наиболее полно отвечающую современным требованиям и запросам по решению задач. Осуществление правильного выбора в этих условиях – решение дополнительной сложной задачи, основанной на многофакторном анализе различных характеристик и возможностей ОС и решаемых задач.

## 4.2. Требования, предъявляемые к ОС реального времени

Как известно, система реального времени (СРВ) должна давать отклик на любые непредсказуемые внешние воздействия в течение предсказуемого интервала времени. Для этого должны быть обеспечены следующие свойства:

*Ограничение времени отклика*, то есть после наступления события реакция на него гарантированно последует до предустановленного крайнего срока. Отсутствие такого ограничения рассматривается как серьезный недостаток программного обеспечения.

*Одновременность обработки*: даже если наступает более одного события одновременно, все временные ограничения для всех событий должны быть выдержаны. Это означает, что системе реального времени должен быть присущ параллелизм. Параллелизм достигается использованием нескольких процессоров в системе и/или многозадачного подхода.

Примерами систем реального времени являются системы управления технологическими процессами, управления вооружением, космической навигации, разведки, управления лабораторными экспериментами, телеметрические системы управления, системы сигнализации – список в принципе бесконечен.

Иногда различают системы "мягкого" и "жесткого" реального времени. Различие между жесткой и мягкой СРВ зависит от требований к системе – система считается жесткой, если "нарушение временных ограничений не допустимо", и мягкой, если "нарушение временных ограничений нежелательно". Термин СРВ часто неправомерно применяют по отношению к быстрым системам.

Часто путают понятия "система реального времени" и "операционная система реального времени", а также неправильно используют атрибуты "мягкая" и "жесткая". Иногда говорят, что та или иная ОСРВ мягкая или жесткая. Нет мягких или жестких ОСРВ. ОСРВ может только служить основой для построения мягкой или жесткой СРВ. Сама по себе ОСРВ не препятствует тому, что ваша СРВ будет мягкой. Например, СРВ, которая должна работать через Ethernet по протоколу TCP/IP не может быть жесткой СРВ, поскольку сама сеть Ethernet в принципе непредсказуема вследствие использования случайного метода доступа к среде передачи данных,

в отличие, например, от IBM Token Ring или ARC-Net, в которых используются детерминированные методы доступа.

Основные требования к ОСРВ.

*Мультипрограммность и многозадачность*

ОС должна быть мультипрограммной и многозадачной (многопоточной – multithreaded) и активно использовать прерывания для диспетчеризации.

Это требование состоит в том, что ОС должна быть многопоточной по принципу абсолютного приоритета (прерываемой). То есть планировщик должен иметь возможность прервать любой поток и предоставить ресурс тому потоку, которому он более необходим. ОС (и аппаратура) должны также обеспечивать прерывания на уровне обработки прерываний.

*Приоритеты задач (потоков)*

В ОС должно существовать понятие приоритета потока.

Проблема в том, чтобы определить, какой задаче ресурс требуется более всего. В идеальной ситуации ОСРВ отдает ресурс потоку или драйверу с ближайшим крайним сроком (это называется управлением временным ограничением, deadline driven OS). Чтобы реализовать это временное ограничение, ОС должна знать, сколько времени требуется каждому из выполняющихся потоков для завершения.

ОС, построенных по этому принципу, практически нет, так как он слишком сложен для реализации. Поэтому разработчики ОС принимают иную точку зрения: вводится понятие уровня приоритета для задачи, и временные ограничения сводят к приоритетам. Так как умозрительные решения чреваты ошибками, показатели СРВ при этом снижаются. Чтобы более эффективно осуществить указанное преобразование ограничений, проектировщик может воспользоваться теорией расписаний или имитационным моделированием, хотя и это может оказаться бесполезным. Тем не менее, так как на сегодняшний день не имеется иного решения, понятие приоритета потока неизбежно.

*Наследование приоритетов*

В ОС должна существовать система наследования приоритетов.

Наследование означает, что блокирующий ресурс поток наследует приоритет потока, который он блокирует (разумеется, это справедливо лишь в том случае, если блокируемый поток имеет более высокий приоритет).

*Синхронизация процессов и задач*

ОС должна обеспечивать мощные, надежные и удобные механизмы синхронизации задач.

Так как задачи разделяют данные (ресурсы) и должны сообщаться друг с другом, представляется логичным, что должны существовать механизмы блокирования и коммуникации. Необходимы механизмы, гарантированно предоставляющие возможность параллельно выполняющимся задачам и процессам оперативно обмениваться сообщениями и синхросигналами.

Эти системные механизмы должны быть всегда доступны процессам, требующим реального времени. Следовательно, системные ресурсы для их функционирования должны быть распределены заранее.

*Предсказуемость*

Поведение ОС должно быть известно и достаточно точно прогнозируемо.

Это означает не то, что ОСРВ должна быть быстрой, а то, что максимальное время выполнения того или иного действия должно быть известно заранее и должно соответствовать требованиям приложения. Времена выполнения системных вызовов и временные характеристики поведения системы в различных обстоятельствах должны быть известны разработчику. Поэтому создатель ОСРВ должен приводить следующие характеристики:

латентную задержку прерывания (то есть время от момента прерывания до момента запуска задачи): она должна быть предсказуема и согласована с требованиями приложения. Эта величина зависит от числа одновременно “висящих” прерываний;

максимальное время выполнения каждого системного вызова. Оно должно быть предсказуемо и не зависимо от числа объектов в системе;

максимальное время маскирования прерываний драйверами и ОС.

## 4.3. Операционные системы образцов АСУ войсковой ПВО

### 4.3.1. Операционная система VxWorks

Операционная система реального времени VxWorks фирмы Wind River Systems предназначена для применения на встроенных вычислительных машинах, работающих в управляющих системах “жесткого” реального времени. ОС VxWorks является системой с кросс-средствами проектирования прикладного программного обеспечения, т.е. проектирование ПО ведется на “инструментальной” ЭВМ (host) для последующего исполнения на “целевой” ЭВМ (target).

ОС VxWorks построена по технологии микроядра, т.е. на нижнем непрерываемом уровне ядра выполняются только базовые функции планирования задач и управления коммуникацией/синхронизацией между задачами. Все остальные функции операционной системы более высокого уровня – уровня управления памятью, вводом/выводом, сетевые средства и т.д. – базируются на простых функциях нижнего уровня.

Такая иерархическая организация “перевернутой пирамиды” (рис. 2.16) позволяет обеспечить быстродействие и детерминированность ядра, а также легко строить необходимую конфигурацию операционной системы (scalable).

Главная особенность ОС реального времени VxWorks – высокая мобильность. VxWorks существует для всех 32-х разрядных микропроцессорных архитектур, которые применяются в целевых встроенных вычислительных машинах: Motorola 68K, Intel i960, SPARC, MIPS 3000 и AMD 29K.

В качестве инструментальной ЭВМ для проектирования прикладного программного обеспечения для VxWorks используются широко распространенные графические станции фирм Sun Microsystems, DEC, Hewllet Packard, IBM, Silicon Graphics, MIPS Computer. Для целевых микропроцессорных архитектур Intel i386/486 и Motorola 68K в качестве инструментальных используется система на PC/AT – совместимых компьютерах.

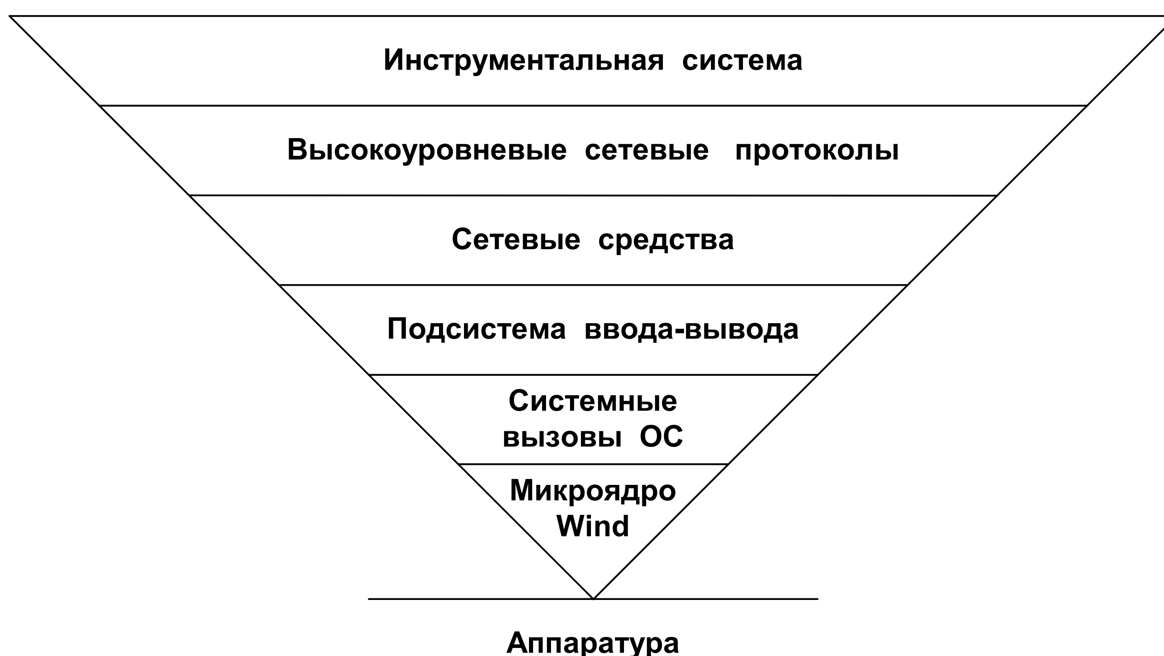



Рис.2.16. Иерархическая структура ОС VxWorks

### 4.3.2. Операционная система ос2000 (ОСРВ “Багет”)

Операционная система реального времени ос2000 является разработкой научно-исследовательского института системных исследований при Российской Академии Наук (НИИСИ РАН). Можно назвать ее первой отечественной операционной системой для встроенных систем современного уровня. Интерфейс и дизайн системы во многом напоминает популярную во всем мире ОСРВ VxWorks компании Wind River System.

Разработка операционной системы реального времени ос2000 базируется на следующих основополагающих принципах:

соответствие международным стандартам,

мобильность,

использование концепции микроядра,

использование объектно-ориентированного подхода,

использование свободно распространяемого программного обеспечения,

распространение ОС вместе со средствами разработки прикладных программ.

Операционная система ос2000 полностью соответствует стандарту POSIX в части, относящейся к реальному времени. Те части стандарта, которые не относятся к системам реального времени (традиционный UNIX) не реализованы.

Для организации (псевдо)параллельной обработки в POSIX используются два понятия – процессы и потоки управления. Процессы в POSIX являются держателями ресурсов (память, таблица открытых файлов и др.) и работают в значительной степени независимо друг от друга. Одной из функций ОС является защита процессов от нежелательного воздействия друг на друга.

С целью повышения мобильности операционная система разбита на три части:

не зависящая от аппаратуры,

зависящая только от типа центрального процессора,

пакет поддержки модуля (платы).

Не зависящая от аппаратуры часть ОС имеет самый большой объем и написана полностью на языке С. Перенос этой части на другие платформы несложен.

Та часть ОС, которая зависит только от типа процессора, написана на языке С или на Ассемблере и имеет сравнительно небольшой объем. Туда входят, например, функции запоминания и восстановления контекста, пролог и эпилог диспетчера прерываний.

Пакет поддержки модуля (ППМ) содержит ту часть ОС, которая зависит от конкретной ЭВМ (платы). ППМ, в частности, содержит драйверы устройств и диспетчер прерываний (за исключением пролога и эпилога).

Граница между этими частями не является жесткой. Например, некоторые не зависящие от аппаратуры функции могут быть переписаны с использованием Ассемблера с целью повышения скорости. В этом случае они станут зависимыми от типа процессора.

ППМ и, в частности, драйверы поставляются вместе с исходными текстами и могут быть изменены пользователем. Внесение изменений в драйверы, а также разработка новых драйверов и включение их в операционную систему производится путем внесения изменений в исходные тексты ППМ. При этом не нужно вносить изменения в ядро операционной системы.

В настоящее время ос2000 содержит пакет поддержки модуля для ЭВМ серии “Багет” с процессором 1В578 (совместимого с процессором MIPS R3000) и для PC-совместимых компьютеров (с процессором Intel).

При использовании ос2000 программное обеспечение системы реального времени состоит из операционной системы и прикладной программы. В системах реального времени граница между операционной системой и прикладной программой не так резко очерчена, как в случае традиционных операционных систем. В частности, в системах реального времени прикладная программа может непосредственно вызывать те функции, которые в случае традиционных ОС могут выполняться только операционной системой (например, запрет или разрешение прерываний).

Для процессоров с архитектурой MIPS прикладная программа выполняется в режиме ядра и виртуальная память не применяется. Это позволяет заметно сократить время обращения к функциям операционной системы, сократить время переключения контекста и увеличить скорость выполнения прикладных программ и сделать ее предсказуемой.

Прикладная программа представляет собой совокупность потоков управления (пользовательских потоков управления). При инициализации системы порождается корневой поток управления прикладной программы, при необходимости другие потоки управления прикладная программа порождает динамически.

Операционная система состоит из ядра и системных потоков управления. Ядро выполняет функции планирования, синхронизации и взаимодействия потоков управления, а также низкоуровневые операции ввода/вывода. Функции ядра выполняются в контексте вызвавшего его потока управления или функции обработки прерывания. Микроядро представляет собой небольшую часть ядра ОС, функциями которой пользуются другие части ОС. Микроядро содержит функции управления потоками нижнего уровня (включая планировщик ) и быстрые средства синхронизации (взаимоисключения). Все другие функции (например, захват и освобождение семафора, низкоуровневые операции ввода/вывода) выполняются вне микроядра, используя его функции.

Системные потоки выполняют более сложные функции операционной системы, такие как ввод/вывод информации по сети или обмен информацией с файловой системой. Использование системных потоков для сложных и протяженных во времени функций ОС позволяет продолжать работу в параллель с выполнением этих функций. В рамках таких потоков можно выполнять часть функций, которые обычно выполняются в рамках обработчиков прерываний драйвера.

### 4.3.3. Операционная система QNX

Операционная система QNX – это ОС стандарта POSIX, позволяющая обеспечить на ЭВМ:

распределенную обработку данных в реальном масштабе времени;

передачу сообщений в качестве основного средства взаимодействия между процессами;

сетевое взаимодействие “каждый с каждым” между любыми узлами сети;

расширение сети простым добавлением узлов, не используя сложных файл-серверов или дополнительного сетевого оборудования.

В общем виде QNX состоит из микроядра, окруженного группой взаимодействующих между собой процессов. QNX содержит:

*администратор процессов* (Process Manager), отвечающий за распределение памяти, запуск и окончание задач в системе;

*администратор периферийных устройств* (Device Manager), управляющий всем периферийным оборудованием: консолью, терминалами, в том числе виртуальными (окнами), модемами, принтерами и т.д. Управление осуществляется на основе взаимодействия рассматриваемого администратора с драйверами этих устройств, являющихся также отдельными задачами. При этом, добавление нового драйвера никак не отразится на работе системы, так как драйвер любого устройства в QNX – обыкновенный процесс для нее.

*администратор файловой системы* (File system Manager).

*администратор сети* (Network Manager), обеспечивающий коммуникации в сети. Его сервис требуется для передачи сообщений между процессами, действующими на различных узлах сети.

Технология микроядра позволяет конструировать необходимую среду верхнего уровня, из которой можно легко получить доступ ко всем функциональным возможностям уровня аппаратного обеспечения. При такой структуре ядро – отправная точка для создания системы.

Ядро ОС QNX обеспечивает поддержку 14 основных системных вызовов для предоставления сервиса по четырем основным направлениям (см. рис. 2.17):

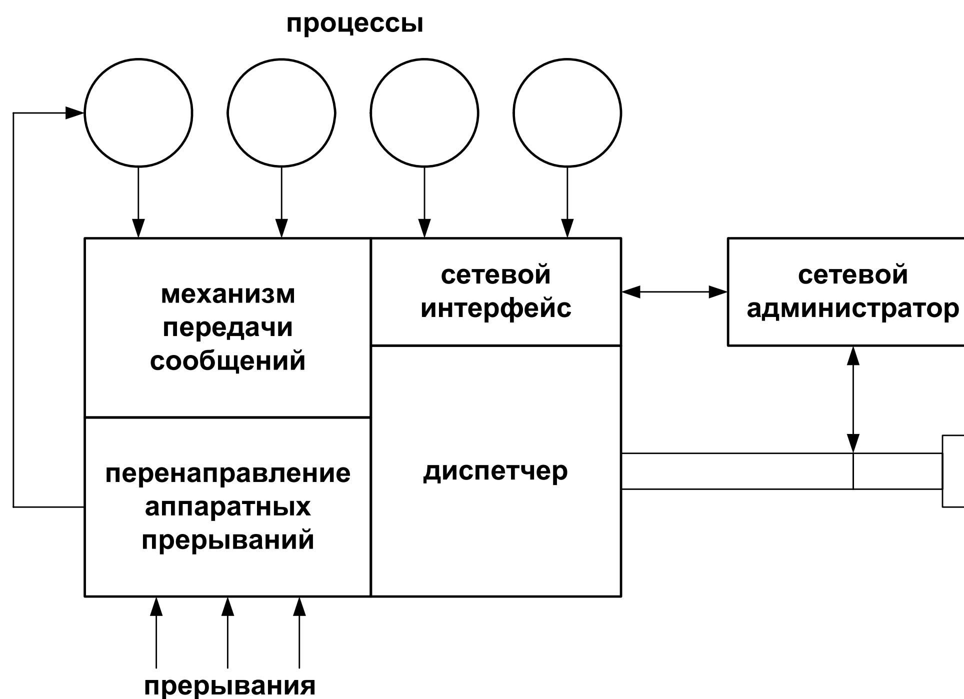



Рис. 2.17. Структура ядра операционной системы QNX

передача сообщений от одного процесса к другому во всей операционной системе;

диспетчеризация процессов при изменении их состояния в результате событий, связанных с сообщениями или прерываниями;

обработка прерываний для перенаправления аппаратных прерываний процессам, осуществляющим передачу данных между ЭВМ и периферийными устройствами на низком уровне. Этот механизм позволяет пользователю осуществлять взаимодействие с аппаратными средствами не напрямую, позволяя, тем самым, избегать конфликтов между различными процессами, работающими с одним и тем же устройством;

перенаправление сообщений по сети.

ОС QNX может осуществлять одновременную передачу данных по трем сетям, сочетающим различное сетевое оборудование: Arcnet, Ethernet и Token Ring. Кроме того, можно создать сеть без использования специальных сетевых плат, ограничившись интерфейсом RS-232. Используя модем возможно осуществление организации полноценной сети с узлами, территориально удаленными на любое расстояние.

Работа системы в реальном масштабе времени предъявляет жесткие требования к производительности файловой системы. Файлы в ОС QNX организованы по принципу набора участков, ссылки на которые находятся в дескрипторах файлов и в отдельных участках дисковой памяти.

ОС QNX поддерживает SCSI – устройства с полным распараллеливанием доступа к различным аппаратным средствам.

В своем составе QNX имеет также различные графические интерфейсы:

*QNX Windows* – полнофункциональная оконная система, выполненная в соответствии со стандартом Open Look.

*Photon* – графический интерфейс для ограниченной в ресурсах встраиваемой системы, поддерживающий стандарт Motif и требующий всего 256 Кбайт оперативной памяти.

*X Window* – графический стандарт для всех открытых систем.

Модульная архитектура QNX также способствует гибкости ОС и может использоваться как в качестве среды разработки, так и в качестве среды исполнения в миниатюрных встраиваемых системах.

Одним из преимуществ рассматриваемой ОС является возможность разработки и отладки программ в той же среде, в которой будет функционировать готовая система. При желании возможно использование ОС UNIX в качестве кросс-среды, что свидетельствует о совместимости программных продуктов для этих систем на уровне исходных текстов.

### 4.3.4. Операционная система Linux (МС ВС)

В 1991 г. Л. Торвальдс, в тот момент — студент университета Хельсинки, приступил к разработке того, что ныне известно как Linux — полноценной операционной системы, основанной на исходных кодах Minix и распространяемой на условиях GPL

В 1992 г. была выпущена первая публичная версия системы. Вышедшее в 1997 г. ядро Linux 2.0 имело вполне приемлемую по стандартам коммерческих ОС надежность и почти все наиболее прогрессивные черты других Unix-систем:

загрузочные модули и разделяемые библиотеки формата ELF;

псевдофайловую систему /proc;

динамическое подключение и отключение своп-файлов;

длинные файлы (64-разрядные — длина файла и смещение в нем);

многопоточность в пределах одного процесса (POSIX thread library);

поддержку симметричной многопроцессорности;

динамическую загрузку и выгрузку модулей ядра;

стек TCP/IP, совместимый с BSD 4.4, с поддержкой IPSec, фильтрации пакетов и др;

бинарную совместимость с UNIX System V на процессорах x86 (iBCS - Intel Binary Compatibility Standard) и, позднее, на SPARC и MIPS;

поддержку задач реального времени (класс планирования реального времени в монолитном Linux невозможен; такие задачи загружаются как модули ядра).

Сам термин “Linux” не вполне определен. Прежде всего, он обозначает собственно ядро - сердце любой версии Linux. В более широком понимании, Linux - любой набор программ, выполняемых в этом ядре и называемый дистрибутивом. Задача ядра - обеспечение базовой среды, в которой могут выполняться программы, в том числе программы базового аппаратного интерфейса и системы управления задачами или выполнением программ.

Если понимать термин “Linux” в широком смысле - как набор программ, выполняемых на ядре Linux, то версий этой операционной системы окажется великое множество. Каждый дистрибутив имеет собственные уникальные характеристики, отличаясь методом установки, набором средств и способом обновления версии. Но поскольку в основе каждого дистрибутива - все тот же Linux, почти любая программа, работающая в текущей версии одного, дистрибутива, будет работать в текущей версии другого.

В техническом представлении операционная система ограничивается ядром, содержащим основные системные функции и необходимым для разработки любой программы.

Особенность ядра Linux, отличающая эту систему от прочих операционных систем для настольных ПК, состоит в том, что это система многозадачная и многопользовательская.

Система Linux поддерживает многопроцессорные компьютеры, наподобие двухпроцессорных систем Pentium III. Эти системы реально выполняют два одновременных действия. Многопроцессорность в сочетании с многозадачностью позволяет значительно увеличить количество программ, одновременно выполняемых на одном компьютере.

Кроме многозадачности, Linux (подобно большинству версий Unix и всем членам ее клона) имеет еще одно важное свойство: это многопользовательская операционная система. Linux допускает одновременную работу нескольких пользователей, что позволяет полностью использовать преимущества многозадачности. Из этого следует огромное достоинство: Linux можно развернуть как сервер приложений. С терминалов или настольных компьютеров пользователи могут входить через ЛВС на сервер Linux и запускать программы на этом сервере, а не на собственных настольных ПК.

Linux перенесен практически на все 32- и 64-разрядные машины, имеющие диспетчер памяти, начиная от Amiga и Atari и заканчивая IBM System/390 и IBM z/90. Бинарные эмуляторы Linux включены в состав Solaris/SPARC и FreeBSD.

# ГЛАВА 3

# УПРАВЛЕНИЕ ПРОЦЕССОМ

## 1. Процессы и потоки

Важнейшей функцией операционной системы является организация рационального использования всех ее аппаратных и информационных ресурсов. К основным ресурсам могут быть отнесены процессоры, память, внешние устройства, данные и программы. Располагающая одними и теми же аппаратными ресурсами, но управляемая различными ОС, вычислительная система может работать с разной степенью эффективности. Поэтому знание внутренних механизмов операционной системы позволяет косвенно судить о ее эксплуатационных возможностях и характеристиках. Хотя и в однопрограммной ОС необходимо решать задачи управления ресурсами (например, распределение памяти между приложением и ОС), главные сложности на этом пути возникают в мультипрограммных ОС, в которых за ресурсы конкурируют сразу несколько приложений. Именно поэтому большая часть всех проблем, рассматриваемых в этой главе, относится к мультипрограммным системам.

### 1.1. Мультипрограммирование

*Мультипрограммирование* или *многозадачность (multitasking),* – это способ организации вычислительного процесса, при котором на одном процессоре попеременно выполняются сразу несколько программ.

Эти программы совместно используют не только процессор, но и другие ресурсы компьютера: оперативную и внешнюю память, устройства ввода-вывода, данные. Мультипрограммирование призвано повысить эффективность использования вычислительной системы, однако эффективность может пониматься по-разному.

Наиболее характерными критериями эффективности вычислительных систем являются:

*пропускная способность* – количество задач, выполняемых вычислительной системой в единицу времени;

*удобство работы пользователей,* заключающееся, в частности, в том, что они имеют возможность интерактивно работать одновременно с несколькими приложениями на одной машине;

*реактивность системы* – способность системы выдерживать заранее заданные (возможно, очень короткие) интервалы времени между запуском программы и получением результата.

В зависимости от выбранного критерия эффективности ОС делятся на системы пакетной обработки, системы разделения времени и системы реального времени. Каждый тип ОС имеет специфические внутренние механизмы и особые области применения. Некоторые операционные системы могут поддерживать одновременно несколько режимов, например, часть задач может выполняться в режиме пакетной обработки, а часть – в режиме реального времени или в режиме разделения времени.

### 1.1.1. Мультипрограммирование в системах пакетной обработки

При использовании мультипрограммирования для повышения пропускной способности компьютера главной целью является минимизация простоев всех устройств компьютера, и, прежде всего, центрального процессора. Такие простои могут возникать из-за приостановки задачи по ее внутренним причинам, связанным, например, с ожиданием ввода данных для обработки. Данные могут храниться на диске или же поступать от пользователя, работающего за терминалом, а также от измерительной аппаратуры, установленной на внешних технических объектах. При возникновении такого рода блокировки выполняемой задачи естественным решением, ведущим к повышению эффективности использования процессора, является переключение процессора на выполнение другой задачи, у которой есть данные для обработки. Такая концепция мультипрограммирования положена в основу так называемых пакетных систем.

Главной целью и критерием эффективности систем пакетной обработки является максимальная пропускная способность, то есть решение максимального числа задач в единицу времени. Для достижения этой цели используется следующая схема функционирования: в начале работы формируется пакет заданий, каждое задание содержит требование к системным ресурсам; из этого пакета заданий формируется мультипрограммная смесь, то есть множество одновременно выполняемых задач. Для одновременного выполнения выбираются задачи, предъявляющие разные требования к ресурсам, так, чтобы обеспечивалась сбалансированная загрузка всех устройств вычислительной машины. Например, в мультипрограммной смеси желательно одновременное присутствие вычислительных задач и задач с интенсивным вводом-выводом. Таким образом, выбор нового задания из пакета заданий зависит от внутренней ситуации, складывающейся в системе, то есть выбирается “выгодное” задание. Следовательно, в

вычислительных системах, работающих под управлением пакетных ОС, невозможно гарантировать выполнение того или иного задания в течение определенного периода времени.

### 1.1.2. Мультипрограммирование в системах разделения времени

Повышение удобства и эффективности работы пользователя является целью другого способа мультипрограммирования – разделения времени. В *системах разделения времени* пользователям (или одному пользователю) предоставляется возможность интерактивной работы сразу с несколькими приложениями. Для этого каждое приложение должно регулярно получать возможность “общения” с пользователем.

В системах разделения времени ОС принудительно периодически приостанавливает приложения, не дожидаясь, когда они добровольно освободят процессор. Всем приложениям попеременно выделяется квант процессорного времени, таким образом пользователи, запустившие программы на выполнение, получают возможность поддерживать с ними диалог.

Системы разделения времени призваны исправить основной недостаток систем пакетной обработки – изоляцию пользователя-программиста от процесса выполнения его задач. Каждому пользователю в этом случае предоставляется терминал, с которого он может вести диалог со своей программой. Так как в системах разделения времени каждой задаче выделяется только квант процессорного времени, ни одна задача не занимает процессор надолго и время ответа оказывается приемлемым. Если квант выбран достаточно небольшим, то у всех пользователей, одновременно работающих на одной и той же машине, складывается впечатление, что каждый из них единолично использует машину.

Ясно, что системы разделения времени обладают меньшей пропускной способностью, чем системы пакетной обработки, так как на выполнение принимается каждая запущенная пользователем задача, а не та, которая “выгодна” системе. Кроме того, производительность системы снижается из-за возросших накладных расходов вычислительной мощности на более частое переключение процессора с задачи на задачу. Это вполне соответствует тому, что критерием эффективности систем разделения времени является не максимальная пропускная способность, а удобство и эффективность работы пользователя. Вместе с тем мультипрограммное выполнение интерактивных приложений повышает и пропускную способность компьютера (пусть и не в такой степени, как пакетные системы). Аппаратура загружается лучше, поскольку в то время, пока одно приложение ждет сообщения пользователя, другие приложения могут обрабатываться процессором.

### 1.1.3. Мультипрограммирование в системах реального времени

Еще одна разновидность мультипрограммирования используется в *системах реального времени,* предназначенных для управления от компьютера различными техническими объектами или технологическими процессами. Во всех этих случаях существует предельно допустимое время, в течение которого должна быть выполнена та или иная управляющая объектом программа. В противном случае может произойти авария. Таким образом, критерием эффективности здесь является способность выдерживать заранее заданные интервалы времени между запуском программы и получением результата (управляющего воздействия). Это время называется временем реакции системы, а соответствующее свойство системы – реактивностью. Требования ко времени реакции зависят от специфики управляемого процесса.

В системах реального времени мультипрограммная смесь представляет собой фиксированный набор заранее разработанных программ, а выбор программы на выполнение осуществляется по прерываниям или в соответствии с расписанием плановых работ.

Способность аппаратуры компьютера и ОС к быстрому ответу зависит в основном от скорости переключения с одной задачи на другую и, в частности, от скорости обработки сигналов прерывания. Если при возникновении прерывания процессор должен опросить сотни потенциальных источников прерывания, то реакция системы будет слишком медленной. Время обработки прерывания в системах реального времени часто определяет требования к классу процессора даже при небольшой его загрузке.

В системах реального времени не стремятся максимально загружать все устройства, наоборот, при проектировании программного управляющего комплекса обычно закладывается некоторый “запас” вычислительной мощности на случай пиковой нагрузки. Статистические аргументы о низкой вероятности возникновения пиковой нагрузки, основанные на том, что вероятность одновременного возникновения большого количества независимых событий очень мала, не применимы ко многим ситуациям в системах управления. Если система реального времени не спроектирована для поддержки пиковой нагрузки, то может случиться так, что система не справится с работой именно тогда, когда она нужна в наибольшей степени.

### 1.1.4. Мультипроцессорная обработка

*Мультипроцессорная обработка* – это способ организации вычислительного процесса в системах с несколькими процессорами, при котором несколько задач (процессов, потоков) могут одновременно выполняться на разных процессорах системы.

Не следует путать *мультипроцессорную* обработку с *мультипрограммной* обработкой. В мультипрограммных системах параллельная работа разных устройств позволяет одновременно вести обработку нескольких программ, но при этом в процессоре в каждый момент времени выполняется только *одна* программа. То есть в этом случае несколько задач выполняются попеременно на одном процессоре, создавая лишь видимость параллельного выполнения. А в мультипроцессорных системах *несколько задач* выполняются действительно *одновременно,* так как имеется несколько обрабатывающих устройств – процессоров. Конечно, мультипроцессирование вовсе не исключает мультипрограммирования: на каждом из процессоров может попеременно выполняться некоторый закрепленный за данным процессором набор задач.

Мультипроцессорная организация системы приводит к усложнению всех алгоритмов управления ресурсами, например требуется планировать процессы не для одного, а для нескольких процессоров, что гораздо сложнее. Сложности заключаются и в возрастании числа конфликтов по обращению к устройствам ввода-вывода, данным, общей памяти и совместно используемым программам. Необходимо предусмотреть эффективные средства блокировки при доступе к разделяемым информационным структурам ядра. Все эти проблемы должна решать операционная система путем синхронизации процессов, ведения очередей и планирования ресурсов. Более того, сама операционная система должна быть спроектирована так, чтобы уменьшить существующие взаимозависимости между собственными компонентами.

В наши дни становится общепринятым введение в ОС функций поддержки мультипроцессорной обработки данных. Такие функции имеются во всех популярных ОС, таких как Sun Solaris 2.x, Santa Crus Operations Open Server 3.x, IBM OS/2, Microsoft Windows NT и Novell NetWare, начиная с 4.1.

Мультипроцессорные системы часто характеризуют либо как симметричные, либо как несимметричные. При этом следует четко определять, к какому аспекту мультипроцессорной системы относится эта характеристика – к типу архитектуры или к способу организации вычислительного процесса.

*Симметричная архитектура* мультипроцессорной системы предполагает однородность всех процессоров и единообразие включения процессоров в общую схему мультипроцессорной системы. Традиционные симметричные мультипроцессорные конфигурации разделяют одну большую память между всеми процессорами.

Масштабируемость, или возможность наращивания числа процессоров, в симметричных системах ограничена вследствие того, что все они пользуются одной и той же оперативной памятью и, следовательно, должны располагаться в одном корпусе. Такая конструкция, называемая *масшта-*

*бируемой по вертикали,* практически ограничивает число процессоров до четырех или восьми.

В симметричных архитектурах все процессы пользуются одной и той же схемой отображения памяти. Они могут очень быстро обмениваться данными, так что обеспечивается достаточно высокая производительность для тех приложений (например, при работе с базами данных), в которых несколько задач должны активно взаимодействовать между собой.

В *асимметричной архитектуре* разные процессоры могут отличаться как своими характеристиками (производительностью, надежностью, системой команд и т. д., вплоть до модели микропроцессора), так и функциональной ролью, которая поручается им в системе. Например, одни процессоры могут предназначаться для работы в качестве основных вычислителей, другие – для управления подсистемой ввода-вывода, третьи – еще для каких-то особых целей.

Функциональная неоднородность в асимметричных архитектурах влечет за собой структурные отличия во фрагментах системы, содержащих разные процессоры системы. Например, они могут отличаться схемами подключения процессоров к системной шине, набором периферийных устройств и способами взаимодействия процессоров с устройствами.

Масштабирование в асимметричной архитектуре реализуется иначе, чем в симметричной. Так как требование единого корпуса отсутствует, система может состоять из нескольких устройств, каждое из которых содержит один или несколько процессоров. Это *масштабирование по горизонтали.* Каждое такое устройство называется *кластером,* а вся мультипроцессорная система – кластерной.

Другим аспектом мультипроцессорных систем, который может характеризоваться симметрией или ее отсутствием, является способ организации вычислительного процесса. Последний, как известно, определяется и реализуется операционной системой.

*Асимметричное мультипроцессирование* является наиболее простым способом организации вычислительного процесса в системах с несколькими процессорами. Этот способ часто называют также “ведущий-ведомый”.

Функционирование системы по принципу “ведущий-ведомый” предполагает выделение одного из процессоров в качестве “ведущего”, на котором работает операционная система и который управляет всеми остальными “ведомыми” процессорами. То есть ведущий процессор берет на себя функции распределения задач и ресурсов, а ведомые процессоры работают только как обрабатывающие устройства и никаких действий по организации работы вычислительной системы не выполняют.

Так как операционная система работает только на одном процессоре и функции управления полностью централизованы, то такая операционная система оказывается не намного сложнее ОС однопроцессорной системы.

Асимметричная организация вычислительного процесса может быть реализована как для симметричной мультипроцессорной архитектуры, в которой все процессоры аппаратно неразличимы, так и для несимметричной, для которой характерна неоднородность процессоров, их специализация на аппаратном уровне.

В архитектурно-асимметричных системах на роль ведущего процессора может быть назначен наиболее надежный и производительный процессор. Если в наборе процессоров имеется специализированный процессор, ориентированный, например, на матричные вычисления, то при планировании процессов операционная система, реализующая асимметричное мультипроцессирование, должна учитывать специфику этого процессора. Такая специализация снижает надежность системы в целом, так как процессоры не являются взаимозаменяемыми.

*Симметричное мультипроцессирование* как способ организации вычислительного процесса может быть реализовано в системах только с симметричной мультипроцессорной архитектурой. Напомним, что в таких системах процессоры работают с общими устройствами и разделяемой основной памятью.

Симметричное мультипроцессирование реализуется общей для всех процессоров операционной системой. При симметричной организации все процессоры равноправно участвуют и в управлении вычислительным процессом, и в выполнении прикладных задач. Например, сигнал прерывания от принтера, который распечатывает данные прикладного процесса, выполняемого на некотором процессоре, может быть обработан совсем другим процессором. Разные процессоры могут в какой-то момент одновременно обслуживать как разные, так и одинаковые модули общей операционной системы. Для этого программы операционной системы должны обладать свойством повторной входимости *(реентерабельностью).*

Операционная система полностью децентрализована. Модули ОС выполняются на любом доступном процессоре. Как только процессор завершает выполнение очередной задачи, он передает управление планировщику задач, который выбирает из общей для всех процессоров системной очереди задачу, которая будет выполняться на данном процессоре следующей. Все ресурсы выделяются для каждой выполняемой задачи по мере возникновения в них потребностей и никак не закрепляются за процессором. При таком подходе все процессоры работают с одной и той же динамически выравниваемой нагрузкой. В решении одной задачи могут участвовать сразу несколько процессоров, если она допускает такое распараллеливание, например, путем представления в виде нескольких потоков.

В случае отказа одного из процессоров симметричные системы, как правило, сравнительно просто реконфигурируются, что является их большим преимуществом перед плохо реконфигурируемыми асимметричными системами.

Симметричная и асимметричная организация вычислительного процесса в мультипроцессорной системе не связана напрямую с симметричной или асимметричной архитектурой, она определяется типом операционной системы. Так, в симметричных архитектурах вычислительный процесс может быть организован как симметричным образом, так и асимметричным. Однако асимметричная архитектура непременно влечет за собой и асимметричный способ организации вычислений.

### 1.2. Понятия процесса и потока

В обширной литературе по тематике операционных систем понятие "процесс" является базовым и одновременно наименее точно определенным. Существует множество определений как формального, так и неформального свойства. Неоднозначность в определении объяснима. Понятие "процесс" является определенным видом абстракции, которую по-разному используют, а следовательно, и истолковывают различные категории лиц. Так, точки зрения прикладных и системных программистов расходятся в деталях, в формах восприятия и реализации этого понятия.

Термин "процесс" впервые начали применять разработчики системы MULTICS в 60-х годах. За прошедшее время термин "процесс", используемый в ряде случаев как синоним "задачи", получил много различных определений. Общепринятого определения пока нет, однако чаще всего под процессом понимается "программа во время выполнения".

Есть все основания утверждать, что архитектура современной многопрограммной ЭВМ многопроцессорная. В самом деле, процессор – это любое устройство в составе ЭВМ, способное автоматически выполнять допустимые для него действия в некотором определенном порядке, т. е. по программе, хранимой в памяти и непосредственно доступной такому активному устройству. Тогда помимо центрального процессора (одного или нескольких) можно назвать процессором канал или устройство, работающее с каналом. В данной трактовке оператор также попадает под определение процессора. Между процессорами в системе существуют информационные и управляющие связи.

Каждый процессор – это такой объект в системе, которым, в общем случае, хотели бы воспользоваться одновременно несколько пользователей для исполнения своей программы на процессоре (напомним, речь идет не обязательно о центральном процессоре). В отношении каждого пользователя, претендующего на исполнение программы на некотором процессоре, и системы, распределяющей этот процессор среди многих пользователей, вводится понятие "процесс". В общем случае процесс – это некоторая деятельность, связанная с исполнением программы на процессоре. Согласно ГОСТ 19.781-83 <u>*процесс*</u> – это система действий, реализующая определенную функцию в вычислительной системе и оформленная так, что

управляющая программа вычислительной системы может перераспределять ресурсы этой системы в целях обеспечения мультипрограммирования.

Деятельность может протекать по-разному. Программе в некоторый момент можно предоставить процессор или изъять его. При исполнении программе могут потребоваться результаты работы других процессоров или какие-либо другие ресурсы. Иначе говоря, деятельностью, т. е. ходом развития, процесса необходимо управлять. Управление процессами как в отношении каждого, так и в отношении их совокупности – это функция ОС.

Необходимо различать системные управляющие процессы, представляющие работу супервизора операционной системы и занимающиеся распределением и управлением ресурсов, от всех других процессов: системных обрабатывающих процессов, которые не входят в ядро операционной системы, и процессов пользователя. Для системных управляющих процессов в большинстве операционных систем ресурсы распределяются изначально и однозначно. Эти процессы управляют ресурсами системы, за использование которых существует конкуренция между всеми остальными процессами. Поэтому исполнение системных управляющих программ не принято называть процессами. Термин задача можно употреблять только по отношению к процессам пользователей и к системным обрабатывающим процессам. Однако это справедливо не для всех ОС. Например, в так называемых "микроядерных" ОС (например, QNX) большинство управляющих программных модулей самой ОС и даже драйверы имеют статус высокоприоритетных процессов, для выполнения которых необходимо выделить соответствующие ресурсы. Аналогично и в UNIX-системах выполнение системных программных модулей тоже имеет статус системных процессов, которые получают ресурсы для своего исполнения.

Если обобщать и рассматривать не только обычные ОС общего назначения, но и, например, ОС реального времени, то можно сказать, что процесс может находиться в активном и пассивном (не активном) состоянии. В активном состоянии процесс может участвовать в конкуренции за использование ресурсов вычислительной системы, а в пассивном – он только известен системе, но в конкуренции не участвует (хотя его существование в системе и сопряжено с предоставлением ему оперативной и/или внешней памяти). В свою очередь, активный процесс может быть в одном из следующих состояний (рис. 3.1):

*выполнения* – все затребованные процессом ресурсы выделены. В этом состоянии в каждый момент времени может находиться только один процесс, если речь идет об однопроцессорной вычислительной системе;

*готовности к выполнению* – ресурсы могут быть предоставлены, тогда процесс перейдет в состояние выполнения;

*блокирования или ожидания* – затребованные ресурсы не могут быть предоставлены, или не завершена операция ввода/вывода.

В большинстве операционных систем последнее состояние, в свою очередь, подразделяется на множество состояний ожидания, соответствующих определенному виду ресурса, из-за отсутствия которого процесс переходит в заблокированное состояние.

В обычных ОС, как правило, процесс появляется при запуске какой-нибудь программы. ОС организует (порождает или выделяет) для нового процесса соответствующий дескриптор процесса, и процесс (задача) начинает развиваться (выполняться). Поэтому пассивного состояния не существует. В ОС реального времени ситуация иная. Обычно при проектировании системы реального времени уже заранее бывает известен состав программ (задач), которые должны будут выполняться. Известны и многие их параметры, которые необходимо учитывать при распределении ресурсов (например, объем памяти, приоритет, средняя длительность выполнения, открываемые файлы, используемые устройства и т. п.). Поэтому для них заранее заводят дескрипторы задач с тем, чтобы впоследствии не тратить драгоценное время на организацию дескриптора и поиск для него необходимых ресурсов. Таким образом, в ОСРВ многие процессы (задачи) могут находиться в состоянии бездействия.

За время своего существования процесс может неоднократно совершать переходы из одного состояния в другое. Это обусловлено обращениями к операционной системе с запросами ресурсов и выполнения системных функций, которые предоставляет операционная система, взаимодействием с другими процессами, появлением сигналов прерывания от таймера, каналов и устройств ввода/вывода, а также других устройств. Возможные переходы процесса из одного состояния в другое отображены в виде графа состояний на рис. 3.1. Рассмотрим эти переходы из одного состояния в другое более подробно.

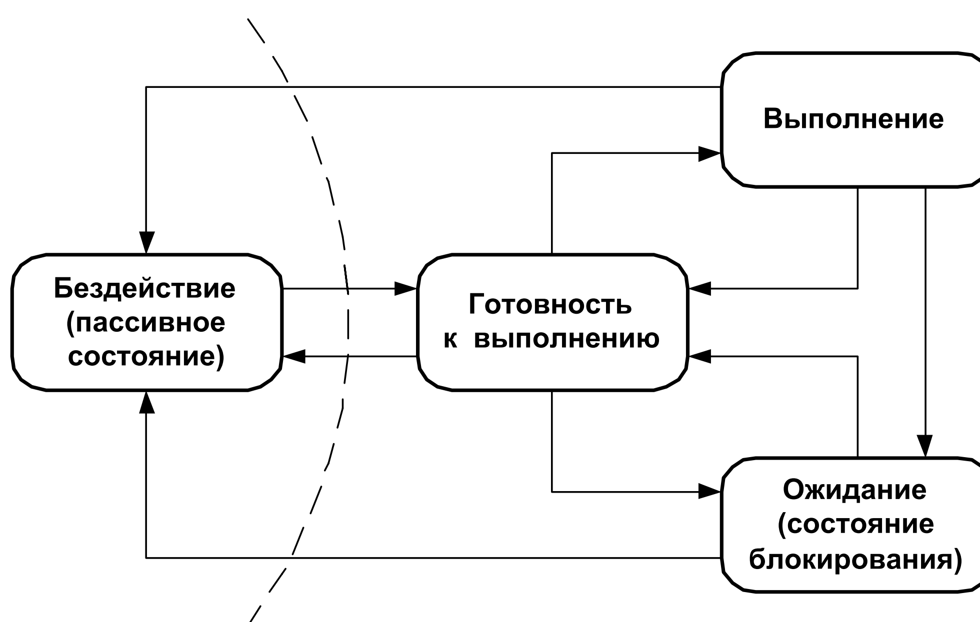



Рис. 3.1. Граф состояний процесса

Процесс из состояния бездействия может перейти в состояние готовности в следующих случаях:

по команде оператора (пользователя). Имеет место в тех диалоговых операционных системах, где программа может иметь статус задачи (и при этом являться пассивной), а не просто быть исполняемым файлом и только на время исполнения получать статус задачи (как это происходит в большинстве современных ОС для ПК);

при выборе из очереди планировщиком (характерно для операционных систем, работающих в пакетном режиме);

по вызову из другой задачи (посредством обращения к супервизору один процесс может создать, инициировать, приостановить, остановить, уничтожить другой процесс);

по прерыванию от внешнего инициативного[1] устройства (сигнал о свершении некоторого события может запускать соответствующую задачу);

при наступлении запланированного времени запуска программы.

Последние два способа запуска задачи, при которых процесс из состояния бездействия переходит в состояние готовности, характерны для операционных систем реального времени.

Процесс, который может исполняться, как только ему будет предоставлен процессор, а для диск-резидентных задач в некоторых системах – и оперативная память, находится в состоянии готовности. Считается, что такому процессу уже выделены все необходимые ресурсы за исключением процессора.

Из состояния выполнения процесс может выйти по одной из следующих причин:

процесс завершается, при этом он посредством обращения к супервизору передает управление операционной системе и сообщает о своем завершении. В результате этих действий супервизор либо переводит его в список бездействующих процессов (процесс переходит в пассивное состояние), либо уничтожает (уничтожается, естественно, не сама программа, а именно задача, которая соответствовала исполнению некоторой программы). В состояние бездействия процесс может быть переведен принудительно: по команде оператора (действие этой и других команд оператора реализуется системным процессом, который “транслирует” команду в запрос к супервизору с требованием перевести указанный процесс в состояние бездействия), или путем обращения к супервизору операционной системы из другой задачи с требованием остановить данный процесс;

процесс переводится супервизором операционной системы в состояние готовности к исполнению в связи с появлением более приоритетной задачи или в связи с окончанием выделенного ему кванта времени;

---

[1] Устройство называется “инициативным”, если по сигналу запроса на прерывание от него должна запускаться некоторая задача

процесс блокируется (переводится в состояние ожидания) либо вследствие запроса операции ввода/вывода (которая должна быть выполнена прежде, чем он сможет продолжить исполнение), либо в силу невозможности предоставить ему ресурс, запрошенный в настоящий момент, а также по команде оператора на приостановку задачи или по требованию через супервизор от другой задачи.

При наступлении соответствующего события (завершилась операция ввода/вывода, освободился затребованный ресурс, в оперативную память загружена необходимая страница виртуальной памяти и т. д.) процесс деблокируется и переводится в состояние готовности к исполнению.

Таким образом, движущей силой, меняющей состояния процессов, являются события. Один из основных видов событий – это прерывания.

*Процессы и потоки.*

Чтобы поддерживать мультипрограммирование, ОС должна определить и оформить для себя те внутренние единицы работы, между которыми будет разделяться процессор и другие ресурсы компьютера. В настоящее время в большинстве операционных систем определены два типа единиц работы. Более крупная единица работы, обычно носящая название *процесса,* или *задачи,* требует для своего выполнения нескольких более мелких работ, для обозначения которых используют термины *"поток"* или *"нить"*.

При использовании этих терминов часто возникают сложности. Это происходит в силу нескольких причин. Во-первых, – специфика различных ОС, когда совпадающие по сути понятия получили разные названия, например задача (task) в OS/2, OS/360 и процесс (process) в UNIX, Windows NT, NetWare. Во-вторых, по мере развития системного программирования и методов организации вычислений некоторые из этих терминов получили новое смысловое значение, особенно это касается понятия "процесс", который уступил многие свои свойства новому понятию "поток". В-третьих, терминологические сложности порождаются наличием нескольких вариантов перевода англоязычных терминов на русский язык. Например, термин "thread" переводится как "нить", "поток", "облегченный процесс", "минизадача" и др. Далее в качестве названия единиц работы ОС будут использоваться термины "процесс" и "поток". В тех же случаях, когда различия между этими понятиями не будут играть существенной роли, они объединяются под обобщенным термином "задача".

Итак, в чем же состоят принципиальные отличия в понятиях "процесс" и "поток"?

Очевидно, что любая работа вычислительной системы заключается в выполнении некоторой программы. Поэтому и с процессом, и с потоком связывается определенный программный код, который для этих целей оформляется в виде исполняемого модуля. Чтобы этот программный код мог быть выполнен, его необходимо загрузить в оперативную память, возможно, выделить некоторое место на диске для хранения данных, пре-

доставить доступ к устройствам ввода-вывода, например к последовательному порту для получения данных по подключенному к этому порту модему, и т. д. В ходе выполнения программе может также понадобиться доступ к информационным ресурсам, например файлам, портам TCP/UPD, семафорам. И, конечно же, невозможно выполнение программы без предоставления ей процессорного времени, то есть времени, в течение которого процессор выполняет коды данной программы.

В операционных системах, где существуют и процессы, и потоки, процесс рассматривается операционной системой как заявка на потребление всех видов ресурсов, кроме одного – процессорного времени. Этот последний важнейший ресурс распределяется операционной системой между другими единицами работы – потоками, которые и получили свое название благодаря тому, что они представляют собой последовательности (потоки выполнения) команд.

В простейшем случае процесс состоит из одного потока, и именно таким образом трактовалось понятие “процесс” до середины 80-х годов (например, в ранних версиях UNIX) и в таком же виде оно сохранилось в некоторых современных ОС. В таких системах понятие “поток” полностью поглощается понятием “процесс”, то есть остается только одна единица работы и потребления ресурсов – процесс. Мультипрограммирование осуществляется в таких ОС на уровне процессов.

Для того чтобы процессы не могли вмешаться в распределение ресурсов, а также не могли повредить коды и данные друг друга, важнейшей задачей ОС является *изоляция* одного процесса от другого. Для этого операционная система обеспечивает каждый процесс отдельным виртуальным адресным пространством, так что ни один процесс не может получить прямого доступа к командам и данным другого процесса.

*Виртуальное адресное пространство процесса* – это совокупность адресов, которыми может манипулировать программный модуль процесса. Операционная система отображает виртуальное адресное пространство процесса на отведенную процессу физическую память.

При необходимости взаимодействия процессы обращаются к операционной системе, которая, выполняя функции посредника, предоставляет им средства межпроцессной связи – конвейеры, почтовые ящики, разделяемые секции памяти и некоторые другие.

Однако в системах, в которых отсутствует понятие потока, возникают проблемы при организации параллельных вычислений в рамках процесса. А такая необходимость может возникать. Действительно, при мультипрограммировании повышается пропускная способность системы, но отдельный процесс никогда не может быть выполнен быстрее, чем в однопрограммном режиме (всякое разделение ресурсов только замедляет работу одного из участников за счет дополнительных затрат времени на ожидание освобождения ресурса). Однако приложение, выполняемое в рамках одного процесса, может обладать внутренним параллелизмом, который в

принципе мог бы позволить ускорить его решение. Если, например, в программе предусмотрено обращение к внешнему устройству, то на время этой операции можно не блокировать выполнение всего процесса, а продолжить вычисления по другой ветви программы. Параллельное выполнение нескольких работ в рамках одного интерактивного приложения повышает эффективность работы пользователя. Так, при работе с текстовым редактором желательно иметь возможность совмещать набор нового текста с такими продолжительными по времени операциями, как переформатирование значительной части текста, печать документа или его сохранение на локальном или удаленном диске. Еще одним примером необходимости распараллеливания является сетевой сервер баз данных. В этом случае параллелизм желателен как для обслуживания различных запросов к базе данных, так и для более быстрого выполнения отдельного запроса за счет одновременного просмотра различных записей базы.

Потоки возникли в операционных системах как средство распараллеливания вычислений. Конечно, задача распараллеливания вычислений в рамках одного приложения может быть решена и традиционными способами.

Во-первых, прикладной программист может взять на себя сложную задачу организации параллелизма, выделив в приложении некоторую подпрограмму-диспетчер, которая периодически передает управление той или иной ветви вычислений. При этом программа получается логически весьма запутанной, с многочисленными передачами управления, что существенно затрудняет ее отладку и модификацию.

Во-вторых, решением является создание для одного приложения нескольких процессов для каждой из параллельных работ. Однако использование для создания процессов стандартных средств ОС не позволяет учесть тот факт, что эти процессы решают единую задачу, а значит, имеют много общего между собой – они могут работать с одними и теми же данными, использовать один и тот же кодовый сегмент, наделяться одними и теми же правами доступа к ресурсам вычислительной системы. Так, если в примере с сервером баз данных создавать отдельные процессы для каждого запроса, поступающего из сети, то все процессы будут выполнять один и тот же программный код и выполнять поиск в записях, общих для всех процессов файлов данных. А операционная система при таком подходе будет рассматривать эти процессы наравне со всеми остальными процессами и с помощью универсальных механизмов обеспечивать их изоляцию друг от друга. В этом случае эти достаточно громоздкие механизмы используются явно не по назначению, выполняя не только бесполезную, но и вредную работу, затрудняющую обмен данными между различными частями приложения. Кроме того, на создание каждого процесса ОС тратит определенные системные ресурсы, которые в данном случае неоправданно дублируются – каждому процессу выделяются собственное вирту-

альное адресное пространство, физическая память, закрепляются устройства ввода-вывода и т. п.

Из всего вышеизложенного следует, что в операционной системе наряду с процессами нужен другой механизм распараллеливания вычислений, который учитывал бы тесные связи между отдельными ветвями вычислений одного и того же приложения. Для этих целей современные ОС предлагают механизм *многопоточной обработки (multithreading).* При этом вводится новая единица работы – *поток выполнения*, а понятие "процесс" в значительной степени меняет смысл. Понятию "поток" соответствует последовательный переход процессора от одной команды программы к другой. ОС распределяет процессорное время между потоками. Процессу ОС назначает адресное пространство и набор ресурсов, которые совместно используются всеми его потоками.

Создание потоков требует от ОС меньших накладных расходов, чем процессов. В отличие от процессов, которые принадлежат разным, вообще говоря, конкурирующим приложениям, все потоки одного процесса всегда принадлежат одному приложению, поэтому ОС изолирует потоки в гораздо меньшей степени, нежели процессы в традиционной мультипрограммной системе. Все потоки одного процесса используют общие файлы, таймеры, устройства, одну и ту же область оперативной памяти, одно и то же адресное пространство. Это означает, что они разделяют одни и те же глобальные переменные. Поскольку каждый поток может иметь доступ к любому виртуальному адресу процесса, один поток может использовать стек другого потока. Между потоками одного процесса нет полной защиты, потому что, во-первых, это невозможно, а во-вторых, не нужно. Чтобы организовать взаимодействие и обмен данными, потокам вовсе не требуется обращаться к ОС, им достаточно использовать общую память – один поток записывает данные, а другой читает их. С другой стороны, потоки разных процессов по-прежнему хорошо защищены друг от друга.

Итак, мультипрограммирование более эффективно на уровне потоков, а не процессов. Каждый поток имеет собственный счетчик команд и стек. Задача, оформленная в виде нескольких потоков в рамках одного процесса, может быть выполнена быстрее за счет псевдопараллельного (или параллельного в мультипроцессорной системе) выполнения ее отдельных частей. Например, если электронная таблица была разработана с учетом возможностей многопоточной обработки, то пользователь может запросить пересчет своего рабочего листа и одновременно продолжать заполнять таблицу. Особенно эффективно можно использовать многопоточность для выполнения распределенных приложений, например многопоточный сервер может параллельно выполнять запросы сразу нескольких клиентов.

Использование потоков связано не только со стремлением повысить производительность системы за счет параллельных вычислений, но и с целью создания более читабельных, логичных программ. Введение несколь-

ких потоков выполнения упрощает программирование. Например, в задачах типа “писатель-читатель” один поток выполняет запись в буфер, а другой считывает записи из него. Поскольку они разделяют общий буфер, не стоит их делать отдельными процессами. Другой пример использования потоков – управление сигналами, такими как прерывание с клавиатуры (del или break). Вместо обработки сигнала прерывания один поток назначается для постоянного ожидания поступления сигналов. Таким образом, использование потоков может сократить необходимость в прерываниях пользовательского уровня. В этих примерах не столь важно параллельное выполнение, сколь важна ясность программы.

Наибольший эффект от введения многопоточной обработки достигается в мультипроцессорных системах, в которых потоки, в том числе и принадлежащие одному процессу, могут выполняться на разных процессорах действительно параллельно (а не псевдопараллельно).

В завершение можно привести несколько советов по использованию потоков при создании приложений.

1. В случае использования однопроцессорной системы множество параллельных потоков часто не ускоряет работу приложения, поскольку в каждый отдельно взятый промежуток времени возможно выполнение только одного потока. Кроме того, чем больше у вас потоков, тем больше нагрузка на систему, потраченная на переключение между ними. Если ваш проект имеет более двух постоянно работающих потоков, то такая мультизадачность не сделает программу быстрее, если каждый из потоков не будет требовать частого ввода/вывода.

2. Вначале нужно понять, для чего необходим поток. Поток, осуществляющий обработку, может помешать системе быстро реагировать на запросы ввода/вывода. Потоки позволяют программе отзываться на просьбы пользователя и устройств, но при этом сильно загружать процессор. Потоки позволяют компьютеру одновременно обслуживать множество устройств, и созданный вами поток, отвечающий за обработку специфического устройства, в качестве минимума может потребовать столько времени, сколько системе необходимо для обработки запросов всех устройств.

3. Потокам можно назначить определенный приоритет для того, чтобы наименее значимые процессы выполнялись в фоновом режиме. Это путь честного разделения ресурсов CPU. Однако необходимо осознать тот факт, что процессор один на всех, а потоков много. Если в вашей программе главная процедура передает нечто для обработки в низкоприоритетный поток, то сама программа становится просто неуправляемой.

4. Потоки хорошо работают, когда они независимы. Но они начинают работать непродуктивно, если вынуждены часто синхронизироваться для доступа к общим ресурсам. Блокировка и критические секции отнюдь не увеличивают скорость работы системы, хотя без использования этих механизмов взаимодействующие вычисления организовывать нельзя.

5. Помните, что память виртуальна. Механизм виртуальной памяти следит за тем, какая часть виртуального адресного пространства должна находиться в оперативной памяти, а какая должна быть сброшена в файл подкачки. Потоки усложняют ситуацию, если они обращаются в одно и то же время к разным адресам виртуального адресного пространства приложения. Это значительно увеличивает нагрузку на систему, особенно при небольшом объеме кэш-памяти. Помните, что реально память не всегда "свободна", как это пишут в информационных "окошках" "О системе". Всегда отождествляйте доступ к памяти с доступом к файлу на диске и создавайте приложение с учетом вышесказанного.

### 1.3. Идентификатор и дескриптор процесса

Создать процесс – это прежде всего означает создать *описатель процесса,* в качестве которого выступает одна или несколько информационных структур, содержащих все сведения о процессе, необходимые операционной системе для управления им. В число таких сведений могут входить, например, идентификатор процесса, данные о расположении в памяти исполняемого модуля, степень привилегированности процесса (приоритет и права доступа) и т. п. Примерами описателей процесса являются *блок управления задачей* (TCB – Task Control Block) в OS/360, *управляющий блок процесса* (PCB – Process Control Block) в OS/2, *дескриптор процесса* в UNIX, *объект-процесс* (object-process) в Windows NT.

Создание описателя процесса означает появление в системе еще одного претендента на вычислительные ресурсы. Начиная с этого момента при распределении ресурсов ОС должна принимать во внимание потребности нового процесса.

Создание процесса включает загрузку кодов и данных исполняемой программы данного процесса с диска в оперативную память. Для этого ОС должна обнаружить местоположение такой программы на диске, перераспределить оперативную память и выделить память исполняемой программе нового процесса. Затем необходимо считать программу в выделенные для нее участки памяти и, возможно, изменить параметры программы в зависимости от размещения в памяти. В системах с виртуальной памятью в начальный момент может загружаться только часть кодов и данных процесса, с тем чтобы "подкачивать" остальные по мере необходимости. Существуют системы, в которых на этапе создания процесса не требуется непременно загружать коды и данные в оперативную память, вместо этого исполняемый модуль копируется из того каталога файловой системы, в котором он изначально находился, в область подкачки – специальную область диска, отведенную для хранения кодов и данных процессов. При выполнении всех этих действий подсистема управления процессами тесно

взаимодействует с подсистемой управления памятью и файловой системой.

В многопоточной системе при создании процесса ОС создает для каждого процесса как минимум один поток выполнения. При создании потока так же, как и при создании процесса, операционная система генерирует специальную информационную структуру – описатель потока, который содержит идентификатор потока, данные о правах доступа и приоритете, о состоянии потока и другую информацию. В исходном состоянии поток (или процесс, если речь идет о системе, в которой понятие “поток” не определяется) находится в приостановленном состоянии. Момент выборки потока на выполнение осуществляется в соответствии с принятым в данной системе правилом предоставления процессорного времени и с учетом всех существующих в данный момент потоков и процессов. В случае, если коды и данные процесса находятся в области подкачки, необходимым условием активизации потока процесса является также наличие места в оперативной памяти для загрузки его исполняемого модуля.

Во многих системах поток может обратиться к ОС с запросом на создание так называемых потоков-потомков. В разных ОС по-разному строятся отношения между потоками-потомками и их родителями. Например, в одних ОС выполнение родительского потока синхронизируется с его потомками, в частности после завершения родительского потока ОС может снимать с выполнения всех его потомков. В других системах потоки-потомки могут выполняться асинхронно по отношению к родительскому потоку. Потомки, как правило, наследуют многие свойства родительских потоков. Во многих системах порождение потомков является основным механизмом создания процессов и потоков.

При управлении процессами операционная система использует два основных типа информационных структур: дескриптор процесса и контекст процесса. *Дескриптор процесса* содержит такую информацию о процессе, которая необходима ядру в течение всего жизненного цикла процесса независимо от того, находится он в активном или пассивном состоянии, находится образ процесса в оперативной памяти или выгружен на диск (образом процесса называется совокупность его кодов и данных). В общем случае дескриптор процесса содержит следующую информацию:

1) идентификатор процесса (так называемый PID – process identifier);

2) тип (или класс) процесса, который определяет для супервизора некоторые правила предоставления ресурсов;

3) приоритет процесса, в соответствии с которым супервизор предоставляет ресурсы. В рамках одного класса процессов в первую очередь обслуживаются более приоритетные процессы;

4) переменную состояния, которая определяет, в каком состоянии находится процесс (готов к работе, в состоянии выполнения, ожидание устройства ввода/вывода и т. д.);

5) защищенную область памяти (или адрес такой зоны), в которой хранятся текущие значения регистров процессора, если процесс прерывается, не закончив работы;

6) информацию о ресурсах, которыми процесс владеет и/или имеет право пользоваться (указатели на открытые файлы, информация о незавершенных операциях ввода/вывода и т. п.);

7) место (или его адрес) для организации общения с другими процессами;

8) параметры времени запуска (момент времени, когда процесс должен активизироваться, и периодичность этой процедуры);

9) в случае отсутствия системы управления файлами – адрес задачи на диске в ее исходном состоянии и адрес на диске, куда она выгружается из оперативной памяти, если ее вытесняет другая (для диск-резидентных задач, которые постоянно находятся во внешней памяти на системном магнитном диске и загружаются в оперативную память только на время выполнения).

Дескрипторы отдельных процессов объединены в список, образующий таблицу процессов. Память для таблицы процессов отводится динамически в области ядра. На основании информации, содержащейся в таблице процессов, операционная система осуществляет планирование и синхронизацию процессов. В дескрипторе прямо или косвенно (через указатели, на связанные с процессом структуры) содержится информация о состоянии процесса, о расположении образа процесса в оперативной памяти и на диске, о значении отдельных составляющих приоритета, а также о его итоговом значении – глобальном приоритете, об идентификаторе пользователя, создавшего процесс, о родственных процессах, о событиях, осуществления которых ожидает данный процесс, и некоторая другая информация.

*Контекст процесса* содержит менее оперативную, но более объемную часть информации о процессе, необходимую для возобновления выполнения процесса с прерванного места: содержимое регистров процессора, коды ошибок выполняемых процессором системных вызовов, информация обо всех открытых данным процессом файлах и незавершенных операциях ввода-вывода и другие данные, характеризующие состояние вычислительной среды в момент прерывания. Контекст, так же как и дескриптор процесса, доступен только программам ядра, то есть находится в виртуальном адресном пространстве операционной системы, однако он хранится не в области ядра, а непосредственно примыкает к образу процесса и перемещается вместе с ним, если это необходимо, из оперативной памяти на диск.

## 2. Понятие прерываний и их организация в ОС

При рассмотрении состояний процесса и причин перехода его из одного состояния в другое мы установили, что движущей силой, меняющей состояния процессов, являются события. Практически все функции современных вычислительных систем так или иначе сводятся к обработке внешних событий: серверы обрабатывают внешние по отношению к ним запросы клиентов, а персональный компьютер – реагирует на действия пользователя. Различие между управляющими системами (приложениями реального времени) и системами общего назначения состоит лишь в том, что первые должны обеспечивать гарантированное время реакции на событие, в то время как вторые "всего лишь" – предоставить хорошее среднее время такой реакции и/или обработку большого количества событий в секунду. Одним из основных видов событий являются прерывания.

### 2.1. Назначение и типы прерываний

Идея прерываний была предложена в середине 50-х годов и можно без преувеличения сказать, что она внесла наиболее весомый вклад в развитие вычислительной техники. Основная цель введения прерываний – реализация асинхронного режима работы и распараллеливание работы отдельных устройств вычислительного комплекса.

Прерывания представляют собой механизм, позволяющий координировать параллельное функционирование отдельных устройств вычислительной системы и реагировать на особые состояния, возникающие при работе процессора. Таким образом, *прерывание* – это принудительная передача управления от выполняемой программы к системе (а через нее – к соответствующей программе обработки прерывания), происходящая при возникновении определенного события, с последующим возвратом к исходному коду. Из сказанного можно сделать вывод о том, что механизм прерываний очень похож на механизм выполнения процедур. Это на самом деле так, хотя между этими механизмами имеется важное отличие. Переключение по прерыванию отличается от переключения, которое происходит по команде безусловного или условного перехода, предусмотренной программистом в потоке команд приложения. Переход по команде происходит в заранее определенных программистом точках программы в зависимости от исходных данных, обрабатываемых программой. Прерывание же происходит в произвольной точке потока команд программы, которую программист не может прогнозировать. Прерывание возникает либо в зависимости от внешних по отношению к процессу выполнения программы событий, либо при появлении непредвиденных аварийных ситуаций в процессе выполнения данной программы. Сходство же прерываний с процедурами состоит в том, что в обоих случаях выполняется некоторая под-

программа, обрабатывающая специальную ситуацию, а затем продолжается выполнение основной ветви программы.

В зависимости от источника прерывания делятся на три класса:

1) внешние (асинхронные);

2) внутренние (синхронные);

3) программные.

*Внешние прерывания* вызываются асинхронными событиями, которые происходят вне прерываемого процесса, например:

прерывания от таймера;

прерывания от внешних устройств (прерывания по вводу/выводу);

прерывания по нарушению питания;

прерывания с пульта оператора вычислительной системы;

прерывания от другого процессора или другой вычислительной системы.

*Внутренние прерывания*, называемые также исключениями, вызываются событиями, которые связаны с работой процессора и являются синхронными с его операциями. Примерами являются следующие запросы на прерывания:

деление на нуль;

ошибки защиты памяти;

обращения по несуществующему адресу;

попытка выполнить привилегированную инструкцию в пользовательском режиме и т. п.

Исключения возникают непосредственно в ходе выполнения тактов команды (“внутри” выполнения).

Программные прерывания отличаются от предыдущих двух классов тем, что они по своей сути не являются “истинными” прерываниями. Программное прерывание возникает при выполнении особой команды процессора, выполнение которой имитирует прерывание, то есть переход на новую последовательность инструкций. Данный механизм был специально введен для того, чтобы переключение на системные программные модули происходило не просто как переход в подпрограмму, а точно таким же образом, как и обычное прерывание. Этим обеспечивается автоматическое переключение процессора в привилегированный режим с возможностью исполнения любых команд.

## 2.2. Механизм прерываний

Механизм прерываний реализуется аппаратно-программными средствами. Структуры систем прерывания (в зависимости от аппаратной архитектуры) могут быть самыми разными, но все они имеют одну общую особенность – прерывание непременно влечет за собой изменение порядка выполнения команд процессором.

Механизм обработки прерываний независимо от архитектуры вычислительной системы включает следующие элементы:

1. Установление факта прерывания (прием сигнала на прерывание) и идентификация прерывания (в операционных системах иногда осуществляется повторно, на шаге 4).

2. Запоминание состояния прерванного процесса. Состояние процесса определяется прежде всего значением счетчика команд (адресом следующей команды, который, например, в i80x86 определяется регистрами CS и IP – указателем команды), содержимым регистров процессора и может включать также спецификацию режима (например, режим пользовательский или привилегированный) и другую информацию.

3. Управление аппаратно передается подпрограмме обработки прерывания. В простейшем случае в счетчик команд заносится начальный адрес подпрограммы обработки прерываний, а в соответствующие регистры – информация из слова состояния. В более развитых процессорах, например в том же i80286 и последующих 32-битовых микропроцессорах, начиная с i80386, осуществляется достаточно сложная процедура определения начального адреса соответствующей подпрограммы обработки прерывания и не менее сложная процедура инициализации рабочих регистров процессора.

4. Сохранение информации о прерванной программе, которую не удалось спасти на шаге 2 с помощью действий аппаратуры. В некоторых вычислительных системах предусматривается запоминание довольно большого объема информации о состоянии прерванного процесса.

5. Обработка прерывания. Эта работа может быть выполнена той же подпрограммой, которой было передано управление на шаге 3, но в ОС чаще всего она реализуется путем последующего вызова соответствующей подпрограммы.

6. Восстановление информации, относящейся к прерванному процессу (этап, обратный шагу 4).

7. Возврат в прерванную программу.

Шаги 1-3 реализуются аппаратно, а шаги 4-7 – программно.

На рис. 3.2 показано, что при возникновении запроса на прерывание естественный ход вычислений нарушается и управление передается программе обработки возникшего прерывания. При этом средствами аппаратуры сохраняется (как правило, с помощью механизмов стековой памяти) адрес той команды, с которой следует продолжить выполнение прерванной программы. После выполнения программы обработки прерывания управление возвращается прерванной ранее программе посредством занесения в указатель команд сохраненного адреса команды. Однако такая схема используется только в самых простых программных средах. В мультипрограммных операционных системах обработка прерываний происходит по более сложным схемам, что будет более подробно рассмотрено ниже.

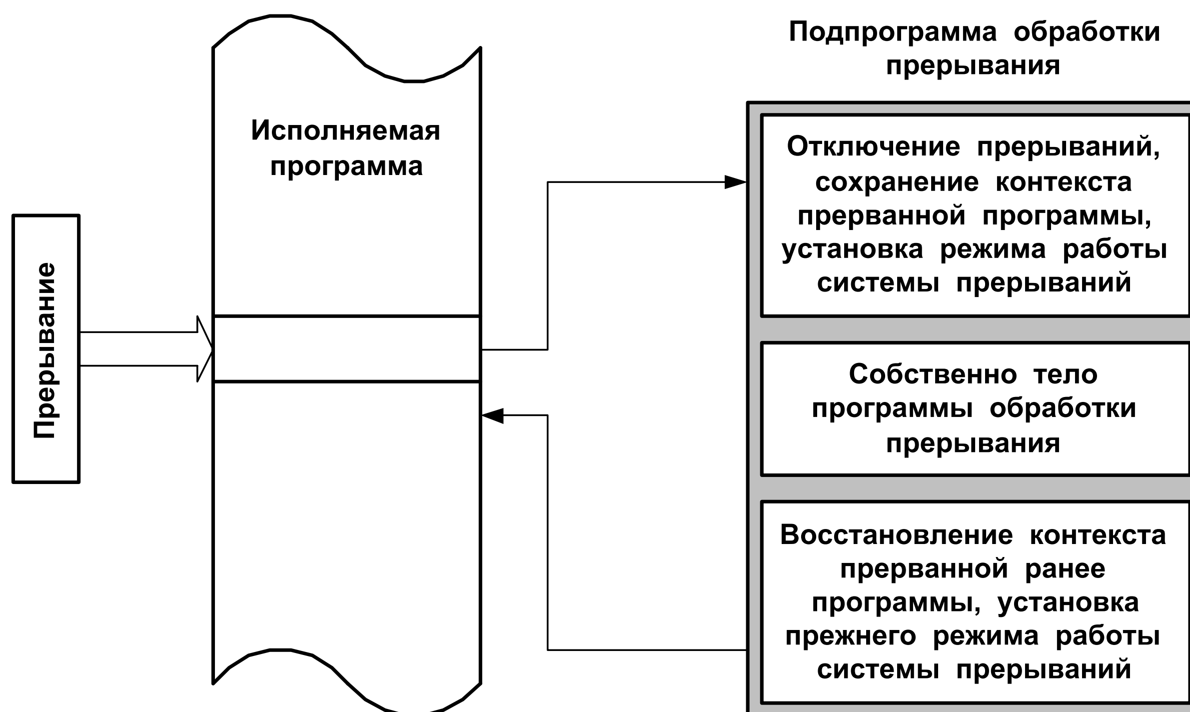



Рис. 3.2. Обработка прерывания

Итак, главные *функции механизма прерываний*:

1) распознавание или классификация прерываний;

2) передача управления соответственно обработчику прерываний;

3) корректное возвращение к прерванной программе.

Переход от прерываемой программы к обработчику и обратно должен выполняться как можно быстрей. Одним из быстрых методов является использование таблицы, содержащей перечень всех допустимых для компьютера прерываний и адреса соответствующих обработчиков. Для корректного возвращения к прерванной программе перед передачей управления обработчику прерываний содержимое регистров процессора запоминается либо в памяти с прямым доступом, либо в системном стеке.

Сигналы, вызывающие прерывания, формируются вне процессора или в самом процессоре; они могут возникать одновременно. Выбор одного из них для обработки осуществляется на основе приоритетов, приписанных каждому типу прерывания. Очевидно, что прерывания от схем контроля процессора должны обладать наивысшим приоритетом (если аппаратура работает неправильно, то не имеет смысла продолжать обработку информации). На рис. 3.3 изображен обычный порядок (приоритеты) обработки прерываний в зависимости от типа прерываний. Учет приоритета может быть встроен в технические средства, а также определяться операционной системой, то есть кроме аппаратно реализованных приоритетов прерывания большинство вычислительных машин и комплексов допускают программно-аппаратное управление порядком обработки сигналов прерывания. Второй способ, дополняя первый, позволяет применять различные дисциплины обслуживания прерываний.

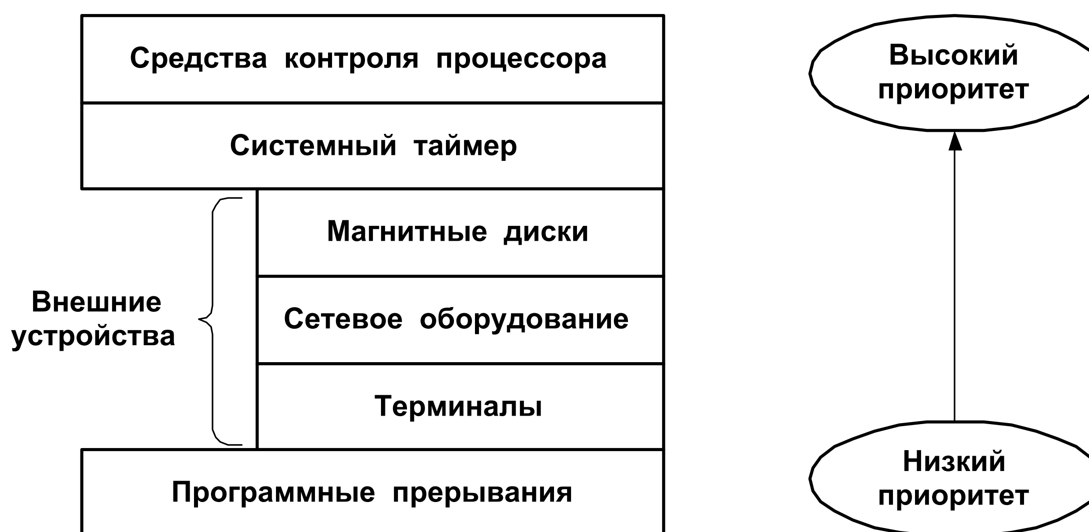



Рис. 3.3. Распределение прерываний по уровням приоритета

Наличие сигнала прерывания не обязательно должно вызывать прерывание исполняющейся программы. Процессор может обладать средствами защиты от прерываний: отключение системы прерываний, маскирование (запрет) отдельных сигналов прерывания. Программное управление этими средствами (существуют специальные команды для управления работой системы прерываний) позволяет операционной системе регулировать обработку сигналов прерывания, заставляя процессор обрабатывать их сразу по приходу, откладывать их обработку на некоторое время или полностью игнорировать. Обычно операция прерывания выполняется только после завершения выполнения текущей команды. Поскольку сигналы прерывания возникают в произвольные моменты времени, то на момент прерывания может существовать несколько сигналов прерывания, которые могут быть обработаны только последовательно. Чтобы обработать сигналы прерывания в разумном порядке им присваиваются приоритеты. Сигнал с более высоким приоритетом обрабатывается в первую очередь, обработка остальных сигналов прерывания откладывается.

Программное управление специальными регистрами маски (маскирование сигналов прерывания) позволяет реализовать различные дисциплины обслуживания:

с *относительными приоритетами*, то есть обслуживание не прерывается даже при наличии запросов с более высокими приоритетами. После окончания обслуживания данного запроса обслуживается запрос с наивысшим приоритетом. Для организации такой дисциплины необходимо в программе обслуживания данного запроса наложить маски на все остальные сигналы прерывания или просто отключить систему прерываний;

с *абсолютными приоритетами*, то есть всегда обслуживается прерывание с наивысшим приоритетом. Для реализации этого режима необходимо на время обработки прерывания замаскировать все запросы с более низким приоритетом. При этом возможно многоуровневое прерывание, то

есть прерывание программ обработки прерываний. Число уровней прерывания в этом режиме изменяется и зависит от приоритета запроса;

по принципу *стека* (по правилу LCFS), то есть запросы с более низким приоритетом могут прерывать обработку прерывания с более высоким приоритетом. Для этого необходимо не накладывать маски ни на один сигнал прерывания и не выключать систему прерываний.

Следует особо отметить, что для правильной реализации последних двух дисциплин нужно обеспечить полное маскирование системы прерываний при выполнении шагов 1-4 и 6-7. Это необходимо для того, чтобы не потерять запрос и правильно его обслужить. Многоуровневое прерывание должно происходить на этапе собственно обработки прерывания, а не на этапе перехода с одного процесса на другой.

Управление ходом выполнения задач со стороны ОС заключается в организации реакций на прерывания, в организации обмена информацией (данными и программами), предоставлении необходимых ресурсов, в динамике выполнения задачи и в организации сервиса. Причины прерываний определяет ОС (модуль, который называют супервизором прерываний), она же и выполняет действия, необходимые при данном прерывании и в данной ситуации. Поэтому в состав любой ОС реального времени прежде всего входят программы управления системой прерываний, контроля состояний задач и событий, синхронизации задач, средства распределения памяти и управления ею, а уже потом средства организации данных (с помощью файловых систем и т. д.). Следует, однако, заметить, что современная ОС реального времени должна вносить в аппаратно-программный комплекс нечто большее, нежели просто обеспечение быстрой реакции на прерывания.

Итак, при появлении запроса на прерывание система прерываний идентифицирует сигнал и, если прерывания разрешены, управление передается на соответствующую подпрограмму обработки. В подпрограмме обработки прерывания имеются две служебные секции (рис. 3.2). Это – первая секция, в которой осуществляется сохранение контекста прерванной задачи, который не смог быть сохранен на 2-м шаге, и последняя, заключительная секция, в которой, наоборот, осуществляется восстановление контекста. Для того чтобы система прерываний не среагировала повторно на сигнал запроса на прерывание, она обычно автоматически “закрывает” (отключает) прерывания, поэтому необходимо потом в подпрограмме обработки прерываний вновь включать систему прерываний. Установка рассмотренных режимов обработки прерываний (с относительными и абсолютными приоритетами, и по правилу LCFS) осуществляется в конце первой секции подпрограммы обработки. Таким образом, на время выполнения центральной секции (в случае работы в режимах с абсолютными приоритетами и по дисциплине LCFS) прерывания разрешены. На время работы заключительной секции подпрограммы обработки система прерываний должна быть отключена и после восстановления контекста вновь

включена. Поскольку эти действия необходимо выполнять практически в каждой подпрограмме обработки прерываний, во многих операционных системах первые секции подпрограмм обработки прерываний выделяются в специальный системный программный модуль, называемый супервизором прерываний.

Супервизор прерываний прежде всего сохраняет в дескрипторе текущей задачи рабочие регистры процессора, определяющие контекст прерываемого вычислительного процесса. Далее он определяет ту подпрограмму, которая должна выполнить действия, связанные с обслуживанием настоящего (текущего) запроса на прерывание. Наконец, перед тем как передать управление этой подпрограмме, супервизор прерываний устанавливает необходимый режим обработки прерывания. После выполнения подпрограммы обработки прерывания управление вновь передается супервизору, на этот раз уже на тот модуль, который занимается диспетчеризацией задач. И уже диспетчер задач, в свою очередь, в соответствии с принятым режимом распределения процессорного времени (между выполняющимися процессами) восстановит контекст той задачи, которой будет решено выделить процессор. Рассмотренная схема проиллюстрирована на рис. 3.4.

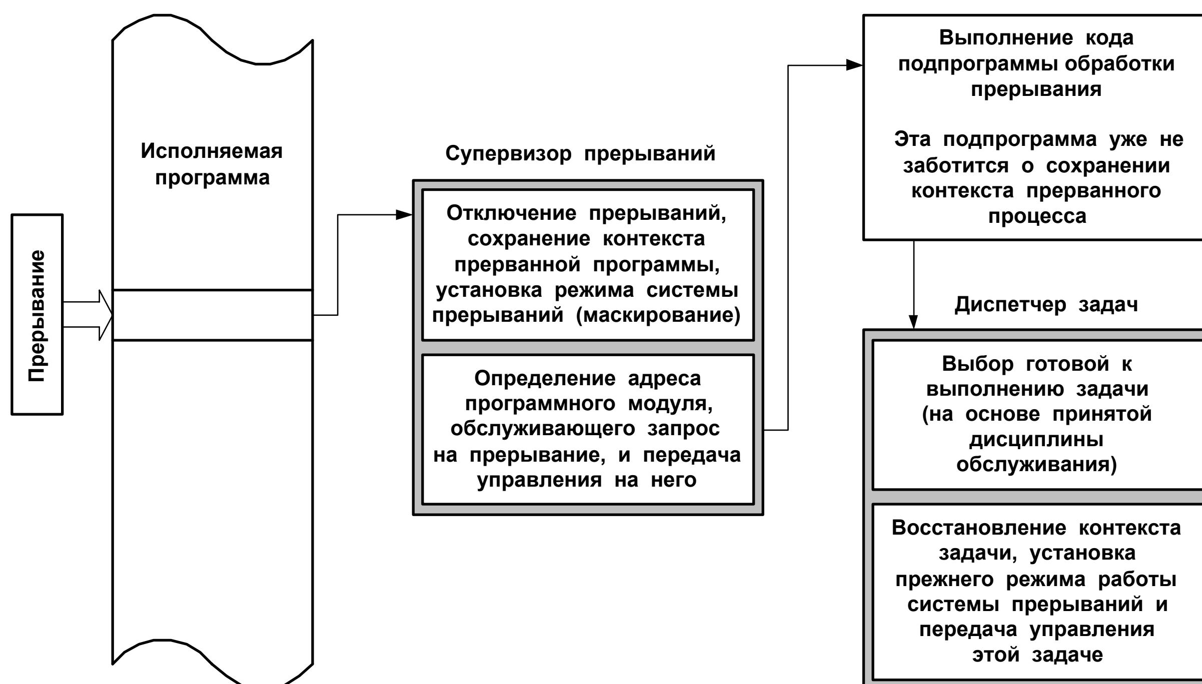



Рис. 3.4. Обработка прерывания при участии супервизоров ОС

Из рассмотренной схемы (рис. 3.4) видно, что здесь нет непосредственного возврата в прерванную ранее программу прямо из самой подпрограммы обработки прерывания. Для прямого непосредственного возврата достаточно адрес возврата сохранить в стеке, что и делает аппаратура

процессора. При этом стек легко обеспечивает возможность возврата в случае вложенных прерываний, поскольку он всегда реализует дисциплину LCFS (last come – first served).

В конкретных процессорах и в конкретных ОС могут существовать некоторые отступления от рассмотренной схемы и/или дополнения к ней. Например, в современных процессорах часто имеются специальные аппаратные возможности для сохранения контекста прерываемого процесса непосредственно в его дескрипторе, то есть дескриптор процесса (по крайней мере, его часть) становится структурой данных, которую поддерживает аппаратура.

## 3. Диспетчеризация и синхронизация процессов

При рассмотрении вопросов диспетчеризации и синхронизации процессов мы не будем разделять понятия процесс (process) и поток (thread), вместо этого используя как бы обобщающий термин *task* (задача). В общем случае под задачей или процессом следует понимать практически одно и то же. Сейчас же мы будем говорить о разделении ресурса центрального процессора, поэтому термин задача может включать в себя и понятие потока.

Операционная система выполняет следующие основные функции, связанные с управлением задачами:

1) создание и удаление задач;
2) планирование процессов и диспетчеризация задач;
3) синхронизация задач, обеспечение их средствами коммуникации.

Система управления задачами обеспечивает прохождение их через компьютер. Создание и удаление задач осуществляется по соответствующим запросам от пользователей или от самих задач. Задача может породить новую задачу. При этом между процессами появляются “родственные” отношения.

При создании современных приложений, позволяющих использовать все возможности операционных систем в плане организации параллельных и распределенных вычислений, одной из важнейших проблем является проблема синхронизации взаимодействия параллельных вычислительных процессов, обмена между ними данными. Существующие методы синхронизации и обмена сообщениями различаются по таким параметрам, как удобство использования при программировании параллельных процессов, стоимость реализации, эффективность функционирования созданных приложений и всей вычислительной системы в целом.

Операционные системы имеют в своем составе различные средства синхронизации. Знание этих средств и их правильное использование позволяет создавать программы, которые при своей работе осуществляют

корректный обмен информацией, а также исключают возможность возникновения тупиковых ситуаций.

## 3.1. Планирование и диспетчеризация процессов и задач

Основным подходом к организации того или иного метода управления процессами, обеспечивающего эффективную загрузку ресурсов или выполнение каких-либо иных целей, является организация очередей процессов и ресурсов.

На распределение ресурсов влияют конкретные потребности тех задач, которые должны выполняться параллельно. Другими словами, можно столкнуться с ситуациями, когда невозможно эффективно распределять ресурсы с тем, чтобы они не простаивали. Таким образом, возникает задача подбора такого множества процессов, что при выполнении они будут как можно реже конфликтовать из-за имеющихся в системе ресурсов. Такая задача называется *планированием вычислительных процессов.*

Задача планирования процессов возникла еще в первых пакетных ОС при планировании пакетов задач, которые должны были выполняться на ЭВМ и оптимально использовать ее ресурсы. В настоящее время актуальность этой задачи не так велика. На первый план уже давно вышли задачи динамического (или краткосрочного) планирования, то есть текущего наиболее эффективного распределения ресурсов, возникающего практически при каждом событии. Задачи динамического планирования стали называть *диспетчеризацией.*

Очевидно, что планирование осуществляется гораздо реже, чем задача текущего распределения ресурсов между уже выполняющимися процессами и потоками. Основное отличие между долгосрочным и краткосрочным планировщиками заключается в частоте запуска: краткосрочный планировщик, например, может запускаться каждые 30 или 100 мс, долгосрочный – один раз за несколько минут.

Долгосрочный планировщик решает, какой из процессов, находящихся во входной очереди, должен быть переведен в очередь готовых процессов в случае освобождения ресурсов памяти. Он выбирает процессы из входной очереди с целью создания неоднородной мультипрограммной смеси. Это означает, что в очереди готовых к выполнению процессов должны находиться – в разной пропорции – как процессы, ориентированные на ввод/вывод, так и процессы, ориентированные на преимущественную работу с центральным процессором.

Краткосрочный планировщик решает, какая из задач, находящихся в очереди готовых к выполнению, должна быть передана на исполнение. В большинстве современных ОС долгосрочный планировщик отсутствует.

### 3.1.1. Стратегии планирования

При рассмотрении стратегий планирования, как правило, идет речь о краткосрочном планировании, то есть о диспетчеризации.

*Стратегия планирования* определяет, какие процессы мы планируем на выполнение для того, чтобы достичь поставленной цели. Известно большое количество различных стратегий выбора процесса, которому необходимо предоставить процессор. Среди них, прежде всего, можно назвать следующие стратегии:

1) по возможности заканчивать вычисления (вычислительные процессы) в том же самом порядке, в котором они были начаты;

2) отдавать предпочтение более коротким процессам;

3) предоставлять всем пользователям (процессам пользователей) одинаковые услуги, в том числе и одинаковое время ожидания.

Когда говорят о стратегии обслуживания, всегда имеют в виду понятие процесса, а не понятие задачи, поскольку процесс, как мы уже знаем, может состоять из нескольких потоков (задач).

### 3.1.2. Дисциплины диспетчеризации

Известно большое количество правил (дисциплин диспетчеризации), в соответствии с которыми формируется список (очередь) готовых к выполнению задач. Различают два больших класса дисциплин обслуживания (рис. 3.5) – *бесприоритетные* и *приоритетные.*

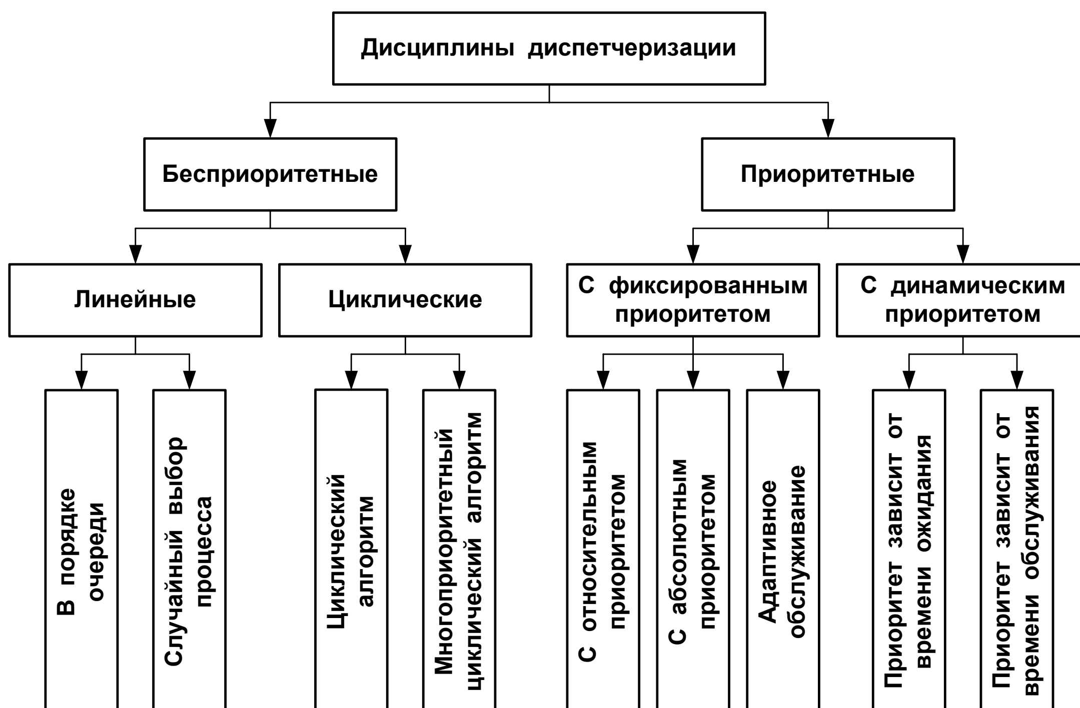



Рис. 3.5. Дисциплины диспетчеризации

При бесприоритетном обслуживании выбор задачи производится в некотором заранее установленном порядке без учета их относительной важности и времени обслуживания. При реализации приоритетных дисциплин обслуживания отдельным задачам предоставляется преимущественное право попасть в состояние исполнения. При этом приоритет, присвоенный задаче, может являться величиной постоянной либо может изменяться в процессе ее решения.

Диспетчеризация с динамическими приоритетами требует дополнительных расходов на вычисление значений приоритетов исполняющихся задач, поэтому во многих ОС реального времени используются методы диспетчеризации на основе статических (постоянных) приоритетов. Однако динамические приоритеты позволяют реализовать гарантии обслуживания задач.

Рассмотрим кратко некоторые основные (наиболее часто используемые) дисциплины диспетчеризации.

Самой простой в реализации является *дисциплина FCFS* (first come – first served), согласно которой задачи обслуживаются “в порядке очереди”, то есть в порядке их появления. Те задачи, которые были заблокированы в процессе работы (попали в какое-либо из состояний ожидания, например, из-за операций ввода/вывода), после перехода в состояние готовности ставятся в эту очередь готовности перед теми задачами, которые еще не выполнялись. Другими словами, образуются две очереди (рис. 3.6): одна очередь образуется из новых задач, а вторая очередь – из ранее выполнявшихся, но попавших в состояние ожидание.

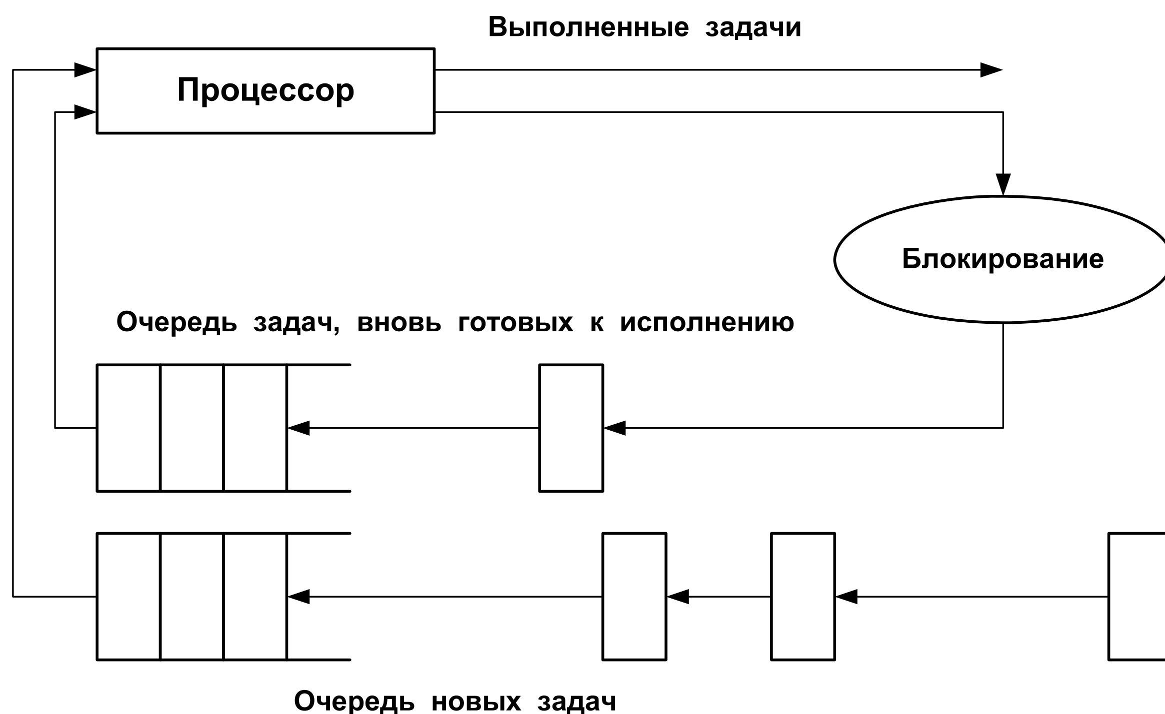



Рис. 3.6. Дисциплина диспетчеризации FCFS

Такой подход позволяет реализовать стратегию обслуживания “по возможности заканчивать вычисления в порядке их появления”. Эта дисциплина обслуживания не требует внешнего вмешательства в ход вычислений, при ней не происходит перераспределение процессорного времени. Существующие дисциплины диспетчеризации процессов могут быть разбиты на два класса – вытесняющие (preemptive) и не вытесняющие (non-preemptive). В первых пакетных ОС часто реализовывали параллельное выполнение заданий без принудительного перераспределения процессора между задачами. В большинстве современных ОС для мощных вычислительных систем, а также и в ОС для ПК, ориентированных на высокопроизводительное выполнение приложений (Windows NT, OS/2, Linux), реализована вытесняющая многозадачность. Можно сказать, что рассмотренная дисциплина относится к не вытесняющим.

*Достоинства* этой дисциплины: простота реализации и малые расходы системных ресурсов на формирование очереди задач.

*Недостатки*: при увеличении загрузки вычислительной системы растет и среднее время ожидания обслуживания, причем короткие задания (требующие небольших затрат машинного времени) вынуждены ожидать столько же, сколько и трудоемкие задания. Избежать этого недостатка позволяют дисциплины SJN и SRT.

<u>*Дисциплина обслуживания SJN*</u> (shortest job next, что означает: следующим будет выполняться кратчайшее задание) требует, чтобы для каждого задания была известна оценка в потребностях машинного времени. Дисциплина обслуживания SJN предполагает, что имеется только одна очередь заданий, готовых к выполнению. И задания, которые в процессе своего исполнения были временно заблокированы (например, ожидали завершения операций ввода/вывода), вновь попадают в конец очереди готовых к выполнению наравне с вновь поступающими. Это приводит к тому, что задания, которым требуется очень немного времени для своего завершения, вынуждены ожидать процессор наравне с длительными работами, что не всегда хорошо.

Для устранения этого недостатка и была предложена <u>*дисциплина SRT*</u> (shortest remaining time, следующее задание требует меньше всего времени для своего завершения).

Все эти три дисциплины обслуживания могут использоваться для пакетных режимов обработки, когда пользователь не вынужден ожидать реакции системы, а просто сдает свое задание и через несколько часов получает свои результаты вычислений. Для интерактивных же вычислений желательно прежде всего обеспечить приемлемое время реакции системы и равенство в обслуживании, если система является мультитерминальной. Если же это однопользовательская система, но с возможностью мультипрограммной обработки, то желательно, чтобы те программы, с которыми пользователь непосредственно работает, имели лучшее время реакции, нежели фоновые задания. При этом пользователь может пожелать, чтобы

некоторые приложения, выполняясь без его непосредственного участия (например, программа получения электронной почты), тем не менее гарантированно получали необходимую им долю процессорного времени. Для решения подобных проблем используется дисциплина обслуживания, называемая RR (round robin, круговая, карусельная), и приоритетные методы обслуживания.

Дисциплина обслуживания RR (рис. 3.7) предполагает, что каждая задача получает процессорное время *квантами*. После окончания кванта времени – задача снимается с процессора и он передается следующей задаче. Снятая задача ставится в конец очереди задач, готовых к выполнению. Для оптимальной работы системы необходимо правильно выбрать закон, по которому кванты времени выделяются задачам.

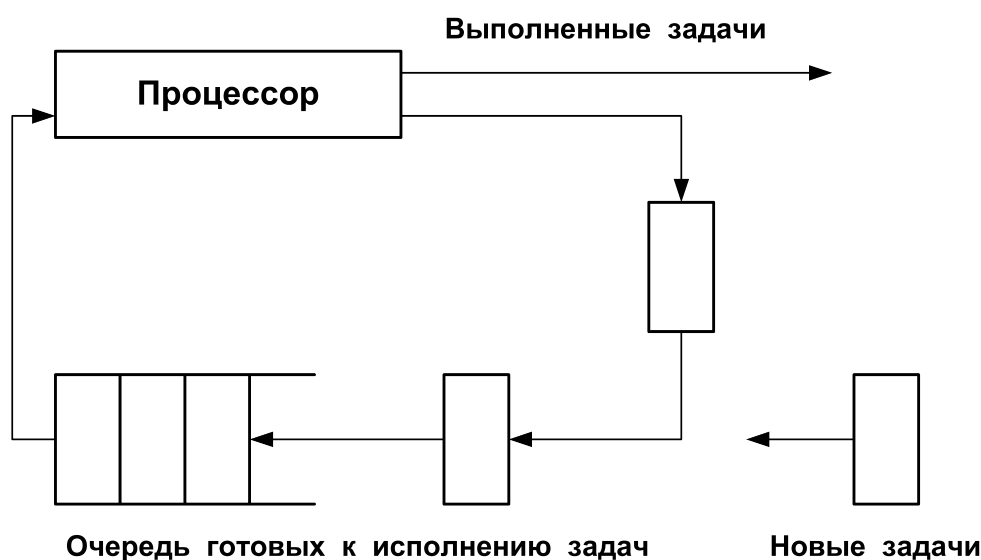



Рис. 3.7. Карусельная дисциплина диспетчеризации (round robin)

Величина кванта времени выбирается как компромисс между приемлемым временем реакции системы на запросы пользователей (с тем, чтобы их простейшие запросы не вызывали длительного ожидания) и накладными расходами на частую смену контекста задач. Очевидно, что при прерываниях ОС вынуждена сохранить достаточно большой объем информации о текущем (прерываемом) процессе, поставить дескриптор снятой задачи в очередь, загрузить контекст задачи, которая теперь будет выполняться (ее дескриптор был первым в очереди готовых к исполнению). Если величина кванта времени велика, то при увеличении очереди готовых к выполнению задач реакция системы станет плохой. Если же она мала, то относительная доля накладных расходов на переключения между исполняющимися задачами станет большой и это ухудшит производительность системы.

Дисциплина диспетчеризации RR – это одна из самых распространенных дисциплин. Однако бывают ситуации, когда ОС не поддерживает в явном виде дисциплину карусельной диспетчеризации. Например, в неко-

торых ОС реального времени используется диспетчер задач, работающий по принципам абсолютных приоритетов (процессор предоставляется задаче с максимальным приоритетом, а при равенстве приоритетов он действует по принципу очередности). Другими словами, снять задачу с выполнения может только появление задачи с более высоким приоритетом. Поэтому если нужно организовать обслуживание задач таким образом, чтобы все они получали процессорное время равномерно и равноправно, то системный оператор может сам организовать эту дисциплину. Для этого достаточно всем пользовательским задачам присвоить одинаковые приоритеты и создать одну высокоприоритетную задачу, которая не должна ничего делать, но которая, тем не менее, будет по таймеру (через указанные интервалы времени) планироваться на выполнение. Эта задача снимет с выполнения текущее приложение, оно будет поставлено в конец очереди, и поскольку этой высокоприоритетной задаче на самом деле ничего делать не надо, то она тут же освободит процессор и из очереди готовности будет взята следующая задача.

В своей простейшей реализации дисциплина карусельной диспетчеризации предполагает, что все задачи имеют одинаковый приоритет. Если же необходимо ввести механизм приоритетного обслуживания, то это, как правило, делается за счет организации нескольких очередей. Процессорное время будет предоставляться в первую очередь тем задачам, которые стоят в самой привилегированной очереди. Если она пустая, то диспетчер задач начнет просматривать остальные очереди. Именно по такому алгоритму действует диспетчер задач в операционных системах OS/2 и Windows NT.

*Вытесняющие и не вытесняющие алгоритмы диспетчеризации*

Диспетчеризация без перераспределения процессорного времени, то есть *не вытесняющая многозадачность* (non-preemptive multitasking) – это такой способ диспетчеризации процессов, при котором активный процесс выполняется до тех пор, пока он сам, что называется “по собственной инициативе”, не отдаст управление диспетчеру задач для выбора из очереди другого, готового к выполнению процесса или потока. К не вытесняющим относятся дисциплины обслуживания FCFS, SJN, SRT.

Диспетчеризация с перераспределением процессорного времени между задачами, то есть *вытесняющая многозадачность* (preemptive multitasking) – это такой способ, при котором решение о переключении процессора с выполнения одного процесса на выполнение другого процесса принимается диспетчером задач, а не самой активной задачей. При вытесняющей многозадачности механизм диспетчеризации задач целиком сосредоточен в операционной системе, и программист может писать свое приложение, не заботясь о том, как оно будет выполняться параллельно с другими задачами. При этом операционная система выполняет следующие функции: определяет момент снятия с выполнения текущей задачи, сохраняет ее контекст в дескрипторе задачи, выбирает из очереди готовых задач сле-

дующую и запускает ее на выполнение, предварительно загрузив ее контекст. Дисциплина RR и многие другие, построенные на ее основе, относятся к вытесняющим.

*Диспетчеризация задач с использованием динамических приоритетов*

При выполнении программ, реализующих какие-либо задачи контроля и управления (что характерно, прежде всего, для систем реального времени), может случиться такая ситуация, когда одна или несколько задач не могут быть реализованы (решены) в течение длительного промежутка времени из-за возросшей нагрузки в вычислительной системе. Потери, связанные с невыполнением таких задач, могут оказаться больше, чем потери от невыполнения программ с более высоким приоритетом. При этом оказывается целесообразным временно изменить приоритет “аварийных” задач (для которых истекает отпущенное для них время обработки). После выполнения этих задач их приоритет восстанавливается. Поэтому почти в любой ОС реального времени имеются средства для изменения приоритета программ.

## 3.2. Приемы и средства синхронизации процессов

### 3.2.1. Независимые и взаимодействующие вычислительные процессы

Основной особенностью мультипрограммных операционных систем является то, что в их среде параллельно развивается несколько (последовательных) вычислительных процессов. С точки зрения внешнего наблюдателя эти последовательные вычислительные процессы выполняются одновременно, поэтому будем использовать термин “параллельно”. При этом под *параллельными* понимаются не только процессы, одновременно развивающиеся на различных процессорах, каналах и устройствах ввода/вывода, но и те последовательные процессы, которые разделяют центральный процессор и хотя бы частично перекрываются во времени. Любая мультипрограммная операционная система вместе с параллельно выполняющимися в ней задачами пользователей может быть логически описана как совокупность последовательных процессов, которые, с одной стороны, состязаются за использование ресурсов, переходя из одного состояния в другое, а с другой – действуют почти независимо друг от друга, но образуют систему вследствие установления всевозможного рода связей между ними (путем пересылки сообщений и синхронизирующих сигналов).

Итак, *параллельными* называются такие последовательные вычислительные процессы, которые одновременно находятся в каком-либо активном состоянии. Два параллельных процесса могут быть *независимыми* (independent processes) либо *взаимодействующими* (cooperating processes).

*Независимыми* являются процессы, множества переменных которых не пересекаются. Под переменными в этом случае понимают файлы данных, а также области оперативной памяти, сопоставленные определенным в программе и промежуточным переменным. Независимые процессы не влияют на результаты работы друг друга, так как не могут изменить значения переменных другого независимого процесса. Они могут только явиться причиной задержек исполнения других процессов, так как вынуждены разделять ресурсы системы.

*Взаимодействующие* процессы совместно используют некоторые (общие) переменные, и выполнение одного процесса может повлиять на выполнение другого.

При выполнении вычислительные процессы разделяют ресурсы системы. Многие ресурсы вычислительной системы могут совместно использоваться несколькими процессами, но в каждый момент времени к разделяемому ресурсу может иметь доступ только один процесс. Ресурсы, которые не допускают одновременного использования несколькими процессами, называются *критическими.*

Если нескольким вычислительным процессам необходимо пользоваться критическим ресурсом в режиме разделения, им следует синхронизировать свои действия таким образом, чтобы ресурс всегда находился в распоряжении не более чем одного из процессов. Если один процесс пользуется в данный момент критическим ресурсом, то все остальные процессы, которым нужен этот ресурс, должны получить отказ и ждать, пока он не освободится. Если в операционной системе не предусмотрена защита от одновременного доступа процессов к критическим ресурсам, в ней могут возникать ошибки, которые трудно обнаружить и исправить. Основной причиной возникновения этих ошибок является то, что процессы в мультипрограммных операционных системах развиваются с различными скоростями, а относительные скорости развития каждого из взаимодействующих процессов не известны и не подвластны ни одному из них. Более того, на их скорости могут влиять решения планировщиков, касающиеся других процессов, с которыми ни одна из этих программ не взаимодействует. Поэтому влияние, которое оказывают друг на друга взаимодействующие процессы, не всегда предсказуемо и воспроизводимо.

Взаимодействовать могут либо *конкурирующие процессы*, либо процессы, *совместно выполняющие* общую работу. Конкурирующие процессы, на первый взгляд, действуют относительно независимо, но они имеют доступ к общим переменным.

Процессы, выполняющие общую совместную работу таким образом, что результаты вычислений одного процесса в явном виде передаются другому, то есть их работа построена именно на обмене данными, называются *сотрудничающими.* Взаимодействие сотрудничающих процессов удобно всего рассматривать в схеме “поставщик – потребитель”.

В качестве первого примера рассмотрим работу двух процессов *P1* и *P2* с общей переменной *X*. Пусть оба процесса асинхронно, независимо один от другого, изменяют (например, увеличивают) значение переменной *X*, считывая ее значение в локальную область памяти $R_i$, при этом каждый процесс выполняет некоторые последовательности операций во времени (рис. 3.8). Здесь мы рассмотрим не все операторы каждого из процессов, а только те, в которых осуществляется работа с общей переменной *X*. Каждому из операторов мы присвоили некоторый условный номер.

| **Процесс P1** | | **Процесс P2** | |
|---|---|---|---|
| **№ оператора** | | **№ оператора** | |
| **1** | **R1 := X** | **4** | **R2 := X** |
| **2** | **R1 := R1 + X** | **5** | **R2 := R2 + X** |
| **3** | **X := R1** | **6** | **X := R2** |

Рис. 3.8. Пример конкурирующих процессов

Поскольку при мультипрограммировании процессы могут иметь различные скорости исполнения, то может иметь место любая последовательность выполнения операций во времени. Если сначала будут выполнены все операции процесса *P1*, а уже потом – все операции процесса *P2* (или, наоборот, сначала операции 4-6, а затем – операции 1-3), то в итоге переменная *X* получит значение, равное *X*+2 (рис. 3.9).

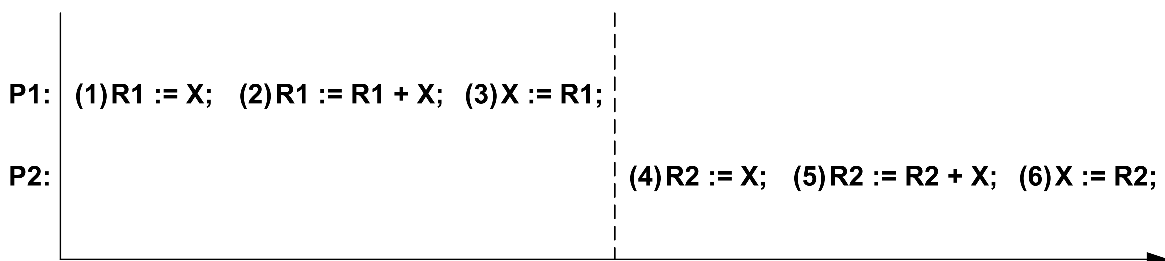



Рис. 3.9. Первый вариант развития событий при выполнении процессов

Однако, если в промежуток времени между выполнением операций 1 и 3 будет выполнена хотя бы одна из операций 4-6 (рис. 3.10), то значение переменной *X* после выполнения всех операций будет не (*X*+2), а (*X*+1).

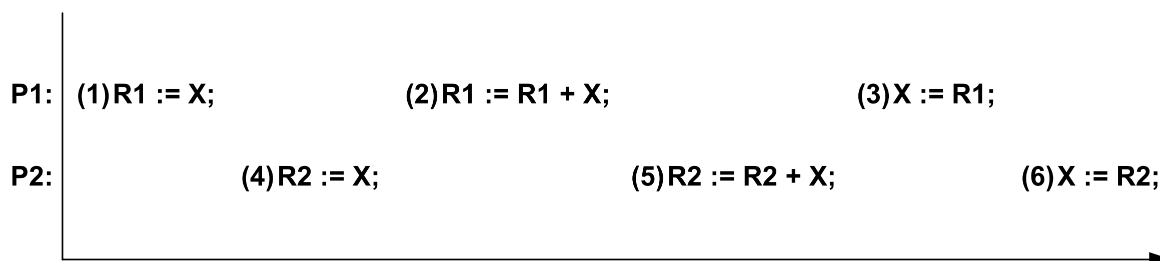



Рис. 3.10. Второй вариант развития событий при выполнении процессов

В качестве второго примера приведем пару процессов, которые изменяют различные поля записей военнослужащих какой-либо воинской части. Пусть процесс АДРЕС изменяет домашний адрес военнослужащего, а процесс СТАТУС – его должность и зарплату. Пусть каждый процесс копирует всю запись ВОЕННОСЛУЖАЩИЙ в свою рабочую область. Предположим, что каждый процесс должен обработать некоторую запись ИВАНОВ. Предположим также, что после того, как процесс АДРЕС скопировал запись ИВАНОВ в свою рабочую область, но до того, как он записал скорректированную запись обратно, процесс СТАТУС скопировал первоначальную запись ИВАНОВ в свою рабочую область. Изменения, выполненные тем из процессов, который первым запишет скорректированную запись назад в файл ВОЕННОСЛУЖАЩИЕ, будут утеряны и, возможно, никто не будет знать об этом.

Чтобы предотвратить некорректное исполнение конкурирующих процессов вследствие нерегламентированного доступа к разделяемым переменным, необходимо ввести механизм *взаимного исключения,* который не позволит двум процессам одновременно обращаться к разделяемым переменным.

Кроме реализации в операционной системе средств, организующих взаимное исключение и тем самым регулирующих доступ процессов к критическим ресурсам, в ней должны быть предусмотрены средства, синхронизирующие работу взаимодействующих процессов. Другими словами, процессы должны обращаться к неким средствам не только ради синхронизации с целью взаимного исключения, но и чтобы обмениваться данными.

Допустим, что “поставщик” – это процесс, который отправляет порции информации (сообщения) другому процессу, имя которого “потребитель”. Например, процесс пользователя, порождающий строки для вывода, может выступать как “поставщик”, а процесс, который выводит эти строки на печать, – как “потребитель”. Один из методов, применяемых при реализации передачи сообщений, состоит в том, что заводится *пул* (pool – совокупность однородных, динамически распределяемых объектов, например, блоков памяти одинаковой длины) свободных буферов, каждый из которых может содержать одно сообщение (длина сообщения может быть произвольной, но ограниченной).

В этом случае между процессами “поставщик” и “потребитель” будем иметь очередь заполненных буферов, содержащих сообщения. Когда “поставщик” хочет послать очередное сообщение, он добавляет в конец этой очереди еще один буфер. “Потребитель”, чтобы получить сообщение, забирает из очереди буфер, который стоит в ее начале. Такое решение, хотя и кажется тривиальным, требует, чтобы “поставщик” и “потребитель”, синхронизировали свои действия. Например, они должны следить за количеством свободных и заполненных буферов. “Поставщик” может передавать сообщения только до тех пор, пока имеются свободные буферы. Аналогично, “потребитель” может получать сообщения только если очередь не пуста. Ясно, что для учета заполненных и свободных буферов нужны разделяемые переменные, поэтому для сотрудничающих процессов, как и для конкурирующих, тоже возникает необходимость во взаимном исключении.

Таким образом, до окончания обращения одной задачи к общим переменным следует исключить возможность обращения к ним другой задачи. Эта ситуация и называется взаимным исключением. Другими словами, при организации различного рода взаимодействующих процессов приходится организовывать взаимное исключение и решать проблему корректного доступа к общим переменным (критическим ресурсам). Те места в программах, в которых происходит обращение к критическим ресурсам, называются *критическими секциями* или критическими интервалами (Critical Section – CS). Решение этой проблемы заключается в организации такого доступа к критическому ресурсу, когда только одному процессу разрешается входить в критическую секцию. Данная задача только на первый взгляд кажется простой, ибо критическая секция, вообще говоря, не является последовательностью операторов программы, а является процессом, то есть последовательностью действий, которые выполняются этими операторами. Другими словами, несколько процессов, которые выполняются по одной и той же программе, могут выполнять критические интервалы, базирующиеся на одной и той же последовательности операторов программы.

Когда какой-либо процесс находится в своем критическом интервале, другие процессы могут, конечно, продолжать свое исполнение, но без входа в их критические секции. Взаимное исключение необходимо только в том случае, когда процессы обращаются к разделяемым, общим данным. Если же они выполняют операции, которые не приводят к конфликтным ситуациям, они должны иметь возможность работать параллельно. Когда процесс выходит из своего критического интервала, то одному из остальных процессов, ожидающих входа в свои критические секции, должно быть разрешено продолжить работу (если в этот момент действительно есть процесс в состоянии ожидания входа в свой критический интервал).

Обеспечение взаимоисключения является одной из ключевых проблем параллельного программирования. При этом можно перечислить следующие требования к критическим секциям:

1) в любой момент времени только один процесс должен находиться в своей критической секции;

2) ни один процесс не должен находиться в своей критической секции бесконечно долго;

3) ни один процесс не должен ждать бесконечно долго входа в свой критический интервал. В частности:

никакой процесс, бесконечно долго находящийся вне своей критической секции (что допустимо), не должен задерживать выполнение других процессов, ожидающих входа в свои критические секции. Другими словами, процесс, работающий вне своей критической секции, не должен блокировать критическую секцию другого процесса;

если два процесса хотят войти в свои критические интервалы, то принятие решения о том, кто первым войдет в критическую секцию, не должно откладываться бесконечно долго;

4) если процесс, находящийся в своем критическом интервале, завершается либо естественным, либо аварийным путем, то режим взаимоисключения должен быть отменен, с тем чтобы другие процессы получили возможность входить в свои критические секции.

Было предложено несколько способов решения этой проблемы – программные и аппаратные, частные низкоуровневые и глобальные высокоуровневые, предусматривающие свободное взаимодействие между процессами и требующие строгого соблюдения жестких протоколов.

### 3.2.2. Средства синхронизации и связи при проектировании взаимодействующих вычислительных процессов

Все известные средства для решения проблемы взаимного исключения основаны на использовании специально введенных аппаратных возможностей, к которым относятся блокировка памяти, специальные команды типа “проверка и установка” и управление системой прерываний, позволяющее организовать такие механизмы, как семафорные операции, мониторы, почтовые ящики и др. С помощью перечисленных средств можно разрабатывать взаимодействующие процессы, при исполнении которых будут корректно решаться все задачи, связанные с проблемой критических интервалов. Рассмотрим эти средства в порядке их появления, а значит, по мере их усложнения, перехода к функциям операционной системы и увеличения предоставляемых ими удобств для пользователя.

*Использование блокировки памяти при синхронизации параллельных процессов*

Все вычислительные машины и системы имеют такое средство для организации взаимного исключения, как *блокировка памяти.* Это средство запрещает одновременное исполнение двух (и более) команд, которые обращаются к одной и той же ячейке памяти. Поскольку в некоторой ячейке памяти хранится значение разделяемой переменной, то получить доступ к ней может только один процесс, несмотря на возможное совмещение выполнения команд во времени на различных процессорах (или на одном процессоре, но с конвейерной организацией параллельного выполнения команд).

Механизм блокировки памяти предотвращает одновременный доступ к разделяемой переменной, но не предотвращает чередование доступа. Таким образом, если критические интервалы исчерпываются одной командой обращения к памяти, данного средства может быть достаточно для непосредственной реализации взаимного исключения. Если же критические секции требуют более одного обращения к памяти, то задача становится сложной, но алгоритмически разрешимой.

Вместо того чтобы связывать с каждым процессом свою собственную переменную, можно со всем множеством конкурирующих критических секций связать одну переменную, которую и рассматривать как некоторый ключ. Вначале доступ к критической секции открыт. Однако перед входом в свой критический интервал процесс забирает "ключ" и тем самым блокирует другие процессы. Покидая критическую секцию, процесс открывает доступ, возвращая "ключ" на место. Если процесс, который хочет войти в свою критическую секцию, обнаруживает, что ключ "отсутствует", то он должен быть переведен в состояние блокирования до тех пор, пока процесс, имеющий ключ, не вернет его. Таким образом, каждый процесс, входящий в критический интервал, должен вначале проверить, доступен ли ключ, и если это так, то сделать его недоступным для других процессов. Причем самым главным является то, что эти два действия должны быть неделимыми, чтобы два или более процессов не могли одновременно получить доступ к ключу. Более того, проверку того, можно ли войти в критический интервал, лучше всего выполнять не самим конкурирующим процессам, так как это приводит к активному ожиданию, а возложить эту функцию на операционную систему. Таким образом, мы подошли к одному из самых главных механизмов решения проблемы взаимного исключения – семафорам Дейкстры.

*Семафорные примитивы Дейкстры*

Понятие семафорных механизмов было введено Дейкстрой. *Семафор* – переменная специального типа, которая доступна параллельным процессам для проведения над ней только двух операций: "закрытия" и "открытия", названных соответственно P- и V-операциями. Эти операции являются примитивами относительно семафора, который указывается в каче-

стве параметра операций. Здесь семафор выполняет роль вспомогательного критического ресурса, так как операции P и V неделимы при своем выполнении и взаимно исключают друг друга.

Семафорный механизм работает по схеме, в которой сначала исследуется состояние критического ресурса, идентифицируемое значением семафора, а затем уже осуществляется допуск к критическому ресурсу или отказ от него на некоторое время. При отказе доступа к критическому ресурсу используется режим “пассивного ожидания”. Поэтому в состав механизма включаются средства формирования и обслуживания очереди ожидающих процессов. Эти средства реализуются супервизором операционной системы. Необходимо отметить, что в силу взаимного исключения примитивов попытка в различных параллельных процессах одновременно выполнить примитив над одним и тем же семафором приведет к тому, что она будет успешной только для одного процесса. Все остальные процессы будут взаимно исключены на время выполнения примитива.

Основным достоинством использования семафорных операций является отсутствие состояния “активного ожидания”, что может существенно повысить эффективность работы мультипрограммной вычислительной системы.

В настоящее время на практике используется много различных видов семафорных механизмов. Варьируемыми параметрами, которые отличают различные виды примитивов, являются начальное значение и диапазон изменения значений семафора, логика действий семафорных операций, количество семафоров, доступных для обработки при исполнении отдельного примитива.

Обобщенный смысл примитива $P(S)$ ($P$ – от голландского *Proberen* – проверить) состоит в проверке текущего значения семафора $S$, и если оно не меньше нуля, то осуществляется переход к следующей за примитивом операции. В противном случае процесс снимается на некоторое время с выполнения и переводится в состояние “пассивного ожидания”. Находясь в списке заблокированных, ожидающий процесс не проверяет семафор непрерывно, как в случае активного ожидания. Вместо него на процессоре может исполняться другой процесс, который реально совершает полезную работу.

Операция $V(S)$ (от голландского *Verhogen* – увеличить) связана с увеличением значения семафора на единицу и переводом одного или нескольких процессов в состояние готовности к центральному процессору.

Отметим еще раз, что операции $P$ и $V$ выполняются операционной системой в ответ на запрос, выданный некоторым процессом и содержащий имя семафора в качестве параметра.

Допустимыми значениями семафоров являются только целые числа. Двоичным семафором называется семафор, максимально возможное значение которого будет равно единице. В противном случае семафоры называют $N$-ичными. Есть реализации, в которых семафорные переменные не

могут быть отрицательными, а есть и такие, где отрицательное значение указывает на длину очереди процессов, стоящих в состоянии ожидания открытия семафора.

*Мьютексы*

Одним из вариантов семафорных механизмов для организации взаимного исключения являются так называемые *мьютексы* (*mutex*). Термин *mutex* произошел от английского словосочетания *mutual exclusion semaphore*, что дословно и переводится как семафор взаимного исключения. Мьютексы реализованы во многих ОС, их основное назначение – организация взаимного исключения для задач (потоков) из одного и того же или из разных процессов. Мьютексы – это простейшие двоичные семафоры, которые могут находиться в одном из двух состояний – отмеченном или неотмеченном (открыт и закрыт соответственно). Когда какая-либо задача, принадлежащая любому процессу, становится владельцем объекта мьютекс, последний переводится в неотмеченное состояние. Если задача освобождает мьютекс, его состояние становится отмеченным.

Организация последовательного (а не параллельного) доступа к ресурсам с использованием мьютексов становится несложной, поскольку в каждый конкретный момент только одна задача может владеть этим объектом. Для того чтобы объект мьютекс стал доступен задачам (потокам), принадлежащим разным процессам, при создании ему необходимо присвоить имя. Потом это имя нужно передать "по наследству" задачам, которые должны его использовать для взаимодействия. Для этого вводятся специальные системные вызовы (*CreateMutex*), в которых указываются начальное значение мьютекса, его имя и, возможно, атрибуты защиты. Если начальное значение мьютекса равно *true*, то считается, что задача, создающая этот объект, будет им сразу владеть. Можно указать в качестве начального значение *false* – в этом случае мьютекс не принадлежит ни одной из задач и только специальным обращением к нему можно изменить его состояние.

*Мониторы Хоара*

Анализ рассмотренных задач показывает, что, несмотря на очевидные достоинства (простота, независимость от количества процессов, отсутствие "активного ожидания"), семафорные механизмы имеют и ряд недостатков. Семафорные механизмы являются слишком примитивными, так как семафор не указывает непосредственно на синхронизирующее условие, с которым он связан, или на критический ресурс. Поэтому при построении сложных схем синхронизации алгоритмы решения задач порой получаются весьма непростыми, ненаглядными и затруднительными для доказательства их правильности.

Необходимо иметь понятные, очевидные решения, которые позволят прикладным программистам без лишних усилий, связанных с доказательством правильности алгоритмов и отслеживанием большого числа взаимосвязанных объектов, создавать параллельные взаимодействующие про-

граммы. К таким решениям можно отнести так называемые *мониторы,* предложенные Хоаром.

В параллельном программировании *монитор* – это пассивный набор разделяемых переменных и повторно входимых процедур доступа к ним, которым процессы пользуются в режиме разделения, причем в каждый момент им может пользоваться только один процесс.

Рассмотрим, например, некоторый ресурс, который разделяется между процессами каким-либо планировщиком. Каждый раз, когда процесс желает получить в свое распоряжение какие-то ресурсы, он должен обратиться к программе-планировщику. Этот планировщик должен иметь переменные, с помощью которых он отслеживает, занят ресурс или свободен. Процедуру планировщика разделяют все процессы, и каждый процесс может в любой момент захотеть обратиться к планировщику. Но планировщик не в состоянии обслуживать более одного процесса одновременно. Такая процедура-планировщик и представляет собой пример монитора.

Таким образом, монитор – это механизм организации параллелизма, который содержит как данные, так, и процедуры, необходимые для реализации динамического распределения конкретного общего ресурса или группы общих ресурсов. Процесс, желающий получить доступ к разделяемым переменным, должен обратиться к монитору, который либо предоставит доступ, либо откажет в нем. Необходимость входа в монитор с обращением к какой-либо его процедуре (например, с запросом на выделение требуемого ресурса) может возникать у многих процессов. Однако вход в монитор находится под жестким контролем – здесь осуществляется взаимоисключение процессов, так что в каждый момент времени только одному процессу разрешается войти в монитор. Процессам, которые хотят войти в монитор, когда он уже занят, приходится ждать, причем режимом ожидания автоматически управляет сам монитор. При отказе в доступе монитор блокирует обратившийся к нему процесс и определяет условие, по которому процесс ждет. Проверка условия выполняется самим монитором, который и деблокирует ожидающий процесс. Поскольку механизм монитора гарантирует взаимоисключение процессов, отсутствуют серьезные проблемы, связанные с организацией параллельных взаимодействующих процессов.

Если процесс обращается к некоторой процедуре монитора и обнаруживается, что соответствующий ресурс уже занят, эта процедура монитора выдает команду ожидания WAIT с указанием условия ожидания. Процесс мог бы оставаться внутри монитора, однако это противоречит принципу взаимоисключения, если в монитор затем вошел бы другой процесс. Поэтому процесс, переводящийся в режим ожидания, должен вне монитора ждать того момента, когда необходимый ему ресурс освободится.

Со временем процесс, который занимал данный ресурс, обратится к монитору, чтобы возвратить ресурс системе. Соответствующая процедура монитора при этом может просто принять уведомление о возвращении ре-

сурса, а затем ждать, пока не поступит запрос от другого процесса, которому потребуется этот ресурс. Однако может оказаться, что уже имеются процессы, ожидающие освобождения данного ресурса. В этом случае монитор выполняет команду извещения (сигнализации) SIGNAL, чтобы один из ожидающих процессов мог получить данный ресурс и покинуть монитор. Если процесс сигнализирует о возвращении (иногда называемом освобождением) ресурса и в это время нет процессов, ожидающих данного ресурса, то подобное оповещение не вызывает никаких других последствий кроме того, что монитор, естественно, вновь вносит ресурс в список свободных. Очевидно, что процесс, ожидающий освобождения некоторого ресурса, должен находиться вне монитора, чтобы другой процесс имел возможность войти в монитор и возвратить ему этот ресурс.

Чтобы гарантировать, что процесс, находящийся в ожидании некоторого ресурса, со временем получит этот ресурс, делается так, что ожидающий процесс имеет более высокий приоритет, чем новый процесс, пытающийся войти в монитор. В противном случае новый процесс мог бы перехватить ожидаемый ресурс до того, как ожидающий процесс вновь войдет в монитор. Если допустить многократное повторение подобной нежелательной ситуации, то ожидающий процесс мог бы откладываться бесконечно. Однако для систем реального времени можно допустить использование дисциплины обслуживания на основе абсолютных или динамически изменяемых приоритетов.

Использование монитора в качестве основного средства синхронизации и связи освобождает процессы от необходимости явно разделять между собой информацию. Напротив, доступ к разделяемым переменным всегда ограничен телом монитора, и поскольку мониторы входят в состав ядра операционной системы, то разделяемые переменные становятся системными переменными. Это автоматически исключает критические интервалы (так как в каждый момент монитором может пользоваться только один процесс, то два процесса никогда не могут получить доступ к разделяемым переменным одновременно).

Монитор является пассивным объектом в том смысле, что это не процесс; его процедуры выполняются только по требованию процесса.

Хотя по сравнению с семафорами мониторы не представляют собой существенно более мощного инструмента для организации параллельных взаимодействующих вычислительных процессов, у них есть некоторые преимущества перед более примитивными синхронизирующими средствами:

1) мониторы очень гибки – в форме мониторов можно реализовать не только семафоры, но и многие другие синхронизирующие операции;

2) локализация всех разделяемых переменных внутри тела монитора позволяет избавиться от малопонятных конструкций в синхронизируемых процессах, сложные взаимодействия процессов можно синхронизировать наглядным образом;

3) мониторы дают процессам возможность совместно использовать программные модули, представляющие собой критические секции – если несколько процессов совместно используют ресурс и работают с ним совершенно одинаково, то в мониторе нужна только одна процедура, тогда как решение с семафорами требует, чтобы в каждом процессе имелся собственный экземпляр критической секции.

Таким образом, мониторы обеспечивают по сравнению с семафорами значительное упрощение организации взаимодействующих вычислительных процессов и большую наглядность при лишь незначительной потере в эффективности.

*Почтовые ящики*

Тесное взаимодействие между процессами предполагает не только синхронизацию – обмен временными сигналами, но и передачу, и получение произвольных данных – обмен сообщениями. В системе с одним процессором посылающий и получающий процессы не могут работать одновременно. В мультипроцессорных системах также нет никакой гарантии их одновременного исполнения. Следовательно, для хранения посланного, но еще не полученного сообщения необходимо место. Оно называется *буфером сообщений* или *почтовым ящиком.*

Если процесс *P1* хочет общаться с процессом *P2*, то *P1* просит систему образовать или предоставить ему почтовый ящик, который свяжет эти два процесса так, чтобы они могли передавать друг другу сообщения. Для того чтобы послать процессу *P2* какое-то сообщение, процесс *P1* просто помещает это сообщение в почтовый ящик, откуда процесс *P2* может его в любое время взять. При применении почтового ящика процесс *P2* в конце концов обязательно получит сообщение, когда обратится за ним, если вообще обратится. Естественно, что процесс *P2* должен знать о существовании почтового ящика. Поскольку в системе может быть много почтовых ящиков, необходимо обеспечить доступ процессу к конкретному почтовому ящику. Почтовые ящики являются системными объектами, и для пользования таким объектом необходимо получить его у операционной системы, что осуществляется с помощью соответствующих запросов.

Если объем передаваемых данных велик, то эффективнее не передавать их непосредственно, а отправлять в почтовый ящик сообщение, информирующее процесс-получатель о том, где можно их найти.

Почтовый ящик может быть связан с парой процессов, только с отправителем, только с получателем, или его можно получить из множества почтовых ящиков, которые используют все или несколько процессов. Почтовый ящик, связанный с процессом-получателем, облегчает посылку сообщений от нескольких процессов в фиксированный пункт назначения. Если почтовый ящик не связан жестко с процессами, то сообщение должно содержать идентификаторы и процесса-отправителя, и процесса-получателя.

Итак, почтовый ящик – это информационная структура, поддерживаемая операционной системой. Она состоит из головного элемента, в котором находится информация о данном почтовом ящике, и из нескольких буферов (гнезд), в которые помещают сообщения. Размер каждого буфера и их количество обычно задаются при образовании почтового ящика.

Правила работы почтового ящика могут быть различными в зависимости от его сложности. В простейшем случае сообщения передаются только в одном направлении. Процесс *P1* может посылать сообщения до тех пор, пока имеются свободные гнезда. Если все гнезда заполнены, то *P1* может либо ждать, либо заняться другими делами и попытаться послать сообщение позже. Аналогично процесс *P2* может получать сообщения до тех пор, пока имеются заполненные гнезда. Если сообщений нет, то он может либо ждать сообщений, либо продолжать свою работу. Эту простую схему работы почтового ящика можно усложнять в нескольких направлениях и получать более хитроумные системы общения – двунаправленные и многовходовые почтовые ящики.

Двунаправленный почтовый ящик, связанный с парой процессов, позволяет подтверждать прием сообщений. Если используется множество гнезд, то каждое из них хранит либо сообщение, либо подтверждение. Чтобы гарантировать передачу подтверждений, когда все гнезда заняты, подтверждение на сообщение помещается в то же гнездо, которое было использовано для сообщения, и оно уже не используется для другого сообщения до тех пор, пока подтверждение не будет получено. Из-за того, что некоторые процессы не забрали свои сообщения, связь может быть приостановлена. Если каждое сообщение снабдить пометкой времени, то управляющая программа может периодически уничтожать старые сообщения.

Процессы могут быть также остановлены в связи с тем, что другие процессы не смогли послать им сообщения. Если время поступления каждого остановленного процесса в очередь заблокированных процессов регистрируется, то управляющая программа может периодически посылать им пустые сообщения.

Основные <u>достоинства</u> почтовых ящиков:

1) процессу не нужно знать о существовании других процессов до тех пор, пока он не получит сообщения от них;

2) два процесса могут обмениваться более чем одним сообщением за один раз;

3) операционная система может гарантировать, что никакой процесс не вмешается в “беседу” других процессов;

4) очереди буферов позволяют процессу-отправителю продолжать работу, не обращая внимания на получателя.

Основным <u>недостатком</u> буферизации сообщений является появление еще одного ресурса, которым нужно управлять, самих почтовых ящиков. Другим недостатком можно считать статический характер этого ресурса:

количество буферов для передачи сообщений через почтовый ящик фиксировано. Поэтому естественным стало появление механизмов, подобных почтовым ящикам, но реализованных на принципах динамического выделения памяти под передаваемые сообщения.

*Конвейеры (программные каналы)*

*Конвейер (pipe* – программный канал (связи), или, как его иногда называют, *транспортер)* является средством, с помощью которого можно производить обмен данными между процессами. Принцип работы конвейера основан на механизме ввода/вывода, который используется для работы с файлами в UNIX, то есть задача, передающая информацию, действует так, как будто она записывает данные в файл, в то время как задача, для которой предназначается эта информация, читает ее из этого файла. Операции записи и чтения осуществляются не записями, как это делается в обычных файлах, а потоком байтов, как это было принято в UNIX-системах. Таким образом, функции, с помощью которых выполняется запись в канал и чтение из него, являются теми же самыми, что и при работе с файлами. По сути, канал представляет собой поток данных между двумя (или более) процессами. Это упрощает программирование и избавляет программистов от использования каких-то новых механизмов. На самом деле конвейеры не являются файлами на диске, а представляют собой буферную память, работающую по принципу FIFO, то есть по принципу обычной очереди. Однако не следует путать конвейеры с очередями сообщений; последние реализуются иначе и имеют другие возможности.

Конвейер имеет определенный размер, который не может превышать 64 Кбайт (конвейер был введен в UNIX-системах и имеет максимальный размер в 64 Кбайт, поскольку в 16-разрядных мини-ЭВМ, для которых создавалась эта система, нельзя было создать массив данных большего размера), и работает циклически. Организация конвейера аналогична реализации очереди на массивах, когда имеются указатели начала и конца очереди, которые перемещаются циклически по массиву. Имеется некий массив и два указателя: один показывает на первый элемент (назовем его условно *head*), а второй – на последний (назовем его *tail*).

В начальный момент оба указателя равны нулю. Добавление самого первого элемента в пустую очередь приводит к тому, что указатели *head* и *tail* принимают значение, равное 1 (в массиве появляется первый элемент). В последующем добавление нового элемента вызывает изменение значения второго указателя, поскольку он отмечает расположение именно последнего элемента очереди. Чтение (и удаление) элемента (читается и удаляется всегда первый элемент из созданной очереди) приводит к необходимости модифицировать значение указателя *head*. В результате операций записи (добавления) и чтения (удаления) элементов в массиве, моделирующем очередь элементов, указатели будут перемещаться от начала массива к его концу. При достижении указателем значения индекса последнего элемента массива значение указателя вновь становится единичным (ес-

ли при этом не произошло переполнение массива, то есть количество элементов в очереди не стало больше числа элементов в массиве). Можно сказать, что массив как бы замыкается в кольцо (рис. 3.11), организуя круговое перемещение указателей *head* и *tail*, которые отслеживают первый и последний элементы в очереди. Именно так и функционирует конвейер.

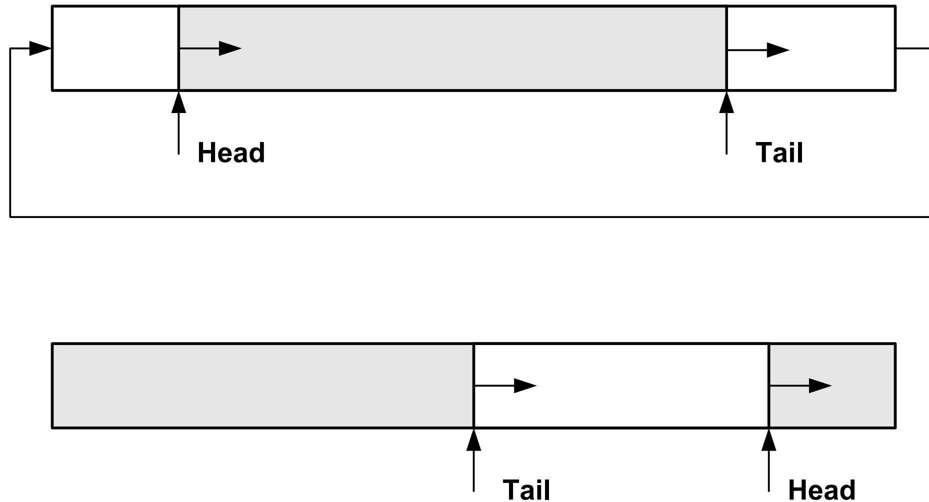



Рис. 3.11. Организация очереди на массиве

Как информационная структура канал описывается идентификатором, размером и двумя указателями. Конвейеры представляют собой системный ресурс. Чтобы начать работу с конвейером, процесс сначала должен заказать его у операционной системы и получить в свое распоряжение. Процессы, знающие идентификатор конвейера, могут через него обмениваться данными.

Читать из конвейера может только тот процесс, который знает идентификатор соответствующего конвейера. При работе с конвейером данные непосредственно помещаются в него. Еще раз отметим, что из-за ограничения на размер конвейера программисты сталкиваются и с ограничениями на размеры передаваемых через него сообщений.

*Очереди сообщений*

*Очереди сообщений* (*Queue*) являются более сложным методом связи между взаимодействующими процессами по сравнению с каналами. С помощью очередей также можно из одной или нескольких задач независимым образом посылать сообщения некоторой задаче-приемнику. При этом только процесс-приемник может читать и удалять сообщения из очереди, а процессы-клиенты имеют право лишь помещать в очередь свои сообщения. Таким образом, очередь работает только в одном направлении. Если же необходима двухсторонняя связь, то можно создать две очереди.

Работа с очередями сообщений имеет много отличий от работы с конвейерами. Во-первых, очереди сообщений предоставляют возможность использовать несколько дисциплин обработки сообщений:

1) FIFO – сообщение, записанное первым, будет первым и прочитано;
2) LIFO – сообщение, записанное последним, будет прочитано первым;
3) приоритетный – сообщения читаются с учетом их приоритетов;
4) произвольный доступ, то есть можно читать любое сообщение, тогда как канал обеспечивает только дисциплину FIFO.

Во-вторых, если при чтении сообщения из канала (конвейера) оно удаляется из него, то при чтении сообщения из очереди этого не происходит, и сообщение при желании может быть прочитано несколько раз.

В третьих, в очередях присутствуют не непосредственно сами сообщения, а только их адреса в памяти и размер. Эта информация размещается системой в сегменте памяти, доступном для всех задач, общающихся с помощью данной очереди.

Каждый процесс, использующий очередь, должен предварительно получить разрешение на использование общего сегмента памяти с помощью системных запросов API, ибо очередь – это системный механизм и для работы с ним требуются системные ресурсы и, соответственно, обращение к самой ОС. Во время чтения из очереди задача-приемник пользуется следующей информацией:

идентификатор процесса (PID – process ID), который передал сообщение;

адрес и длина переданного сообщения;

ждать или нет, если очередь пуста;

приоритет переданного сообщения;

номер освобождаемого семафора, когда сообщение передается в очередь.

## 4. Проблема тупиков и методы борьбы с ними

### 4.1. Понятие тупиковой ситуации и причины их возникновения

При организации параллельного выполнения нескольких процессов одной из главных функций операционной системы является решение сложной задачи корректного распределения ресурсов между выполняющимися процессами и обеспечение последних средствами взаимной синхронизации и обмена данными.

При параллельном исполнении процессов могут возникать ситуации, при которых два или более процесса все время находятся в заблокированном состоянии. Самым простым является случай, когда каждый из двух процессов ожидает ресурс, занятый другим процессом. Из-за такого ожидания ни один из процессов не может продолжить исполнение и освободить в конечном итоге ресурс, необходимый другому процессу. Эта тупиковая ситуация называется *дедлоком*, *тупиком* или *клинчем*. Говорят, что в

мультипрограммной системе процесс находится в состоянии тупика, если он ждет события, которое никогда не произойдет. Тупики чаще всего возникают из-за конкуренции несвязанных параллельных процессов за ресурсы вычислительной системы, но иногда к тупикам приводят и ошибки программирования.

При рассмотрении проблемы тупиков целесообразно понятие ресурсов системы обобщить и разделить их все на два класса – повторно используемые (или системные) ресурсы (типа RR или SR – reusable resource или system resource) и потребляемые (или расходуемые) ресурсы (типа CR – consumable resource).

*Повторно используемый ресурс* (SR) есть конечное множество идентичных единиц со следующими свойствами:

1) число единиц ресурса постоянно;

2) каждая единица ресурса или доступна, или распределена одному и только одному процессу (разделение либо отсутствует, либо не принимается во внимание, так как не оказывает влияния на распределение ресурсов, а значит, и на возникновение тупиковой ситуации);

3) процесс может освободить единицу ресурса (сделать ее доступной), только если он ранее получил эту единицу, то есть никакой процесс не может оказывать какое-либо влияние ни на один ресурс, если он ему не принадлежит.

Данное определение выделяет существенные для изучения проблемы тупика свойства обычных системных ресурсов, к которым относятся такие компоненты аппаратуры, как основная память, вспомогательная (внешняя) память, периферийные устройства и, возможно, процессоры, а также программное и информационное обеспечение, такое как файлы данных, таблицы и “разрешение войти в критическую секцию”.

*Расходуемый ресурс* (CR) отличается от ресурса типа SR в нескольких важных отношениях:

1) число доступных единиц некоторого ресурса типа CR изменяется по мере того, как приобретаются (расходуются) и освобождаются (производятся) отдельные их элементы выполняющимися процессами, и такое число единиц ресурса является потенциально неограниченным; процесс “производитель” увеличивает число единиц ресурса, освобождая одну или более единиц, которые он “создал”;

2) процесс “потребитель”, уменьшает число единиц ресурса, сначала запрашивая и затем приобретая (потребляя) одну или более единиц. Единицы ресурса, которые приобретены, в общем случае не возвращаются ресурсу, а потребляются (расходуются). Эти свойства потребляемых ресурсов присущи многим синхронизирующим сигналам, сообщениям и данным, порождаемым как аппаратурой, так и программным обеспечением, и могут рассматриваться как ресурсы типа CR при изучении тупиков. В их число входят: прерывания от таймера и устройств ввода/вывода; сигналы

синхронизации процессов; сообщения, содержащие запросы на различные виды обслуживания или данные, а также соответствующие ответы.

Для исследования параллельных процессов и, в частности, проблемы тупиков было разработано несколько моделей. Одной из них является модель повторно используемых ресурсов Холта. Согласно этой модели система представляется как набор (множество) процессов и набор ресурсов, причем каждый из ресурсов состоит из фиксированного числа единиц. Любой процесс может изменять состояние системы с помощью запроса, получения или освобождения единицы ресурса.

В графической форме процессы и ресурсы представляются квадратами и кружками соответственно. Каждый кружок содержит некоторое количество маркеров (фишек) в соответствии с существующим количеством единиц этого ресурса. Дуга, указывающая из "процесса" на "ресурс", означает запрос одной единицы этого ресурса. Дуга, указывающая из "ресурса" на "процесс", представляет выделение ресурса процессу. Поскольку каждая единица любого ресурса типа SR может быть выделена одновременно не более чем одному процессу, то число дуг, исходящих из ресурса к различным процессам, не может превышать общего числа единиц этого ресурса. Такая модель называется *графом повторно используемых ресурсов.*

Одно из состояний примера системы из двух процессов с ресурсами типа SR представлено на рис. 3.12.

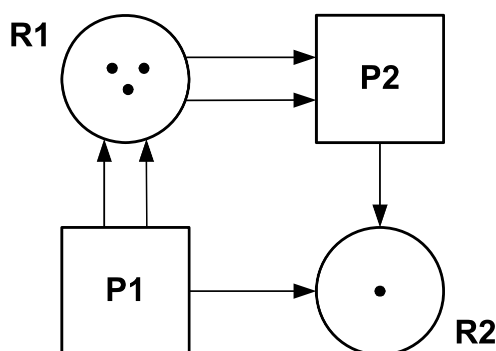



Рис. 3.12. Пример модели Холта для системы из двух процессов

Пусть процесс *P1* запрашивает две единицы ресурса *R1* и одну единицу ресурса *R2*. Процессу *P2* принадлежат две единицы ресурса *R1* и ему нужна одна единица *R2*. Предположим, что процесс *P1* получил бы теперь запрошенную им единицу *R2*. Если принято правило, по которому процесс должен получить все запрошенные им ресурсы, прежде чем освободить хотя бы один из них, то удовлетворение запроса *P1* приведет к тупиковой ситуации: *P1* не сможет продолжиться до тех пор, пока *P2* не освободит единицу ресурса *R1,* а процесс *P2* не сможет продолжиться до тех пор, пока *P1* не освободит единицу *R2*. Причиной этого дедлока являются неупорядоченные попытки процессов войти в критический интервал, связанный с выделением соответствующей единицы ресурса.

## 4.2. Примеры тупиковых ситуаций и причины их возникновения

Для понимания основных причин возникновения тупиков рассмотрим несколько простых характерных примеров.

*Пример тупика на ресурсах типа CR*

Пусть имеются три процесса *ПР1*, *ПР2* и *ПР3*, которые вырабатывают соответственно сообщения *Ml*, *М2* и *М3*. Эти сообщения представляют собой ресурсы типа CR. Пусть процесс *ПР1* является “потребителем” сообщения *М3*, процесс *ПР2* получает сообщение *Ml*, а *ПР3* – сообщение *М2* от процесса *ПР2*, то есть каждый из процессов является и “поставщиком” и “потребителем” одновременно, и вместе они образуют “кольцевую” систему (рис. 3.13) передачи сообщений через почтовые ящики (ПЯ). Если связь с помощью этих сообщений со стороны каждого процесса устанавливается в порядке, изображенном в листинге 3.1, то никаких трудностей не возникает. Однако перестановка этих двух процедур в каждом из процессов (листинг 3.2) вызывает тупик:

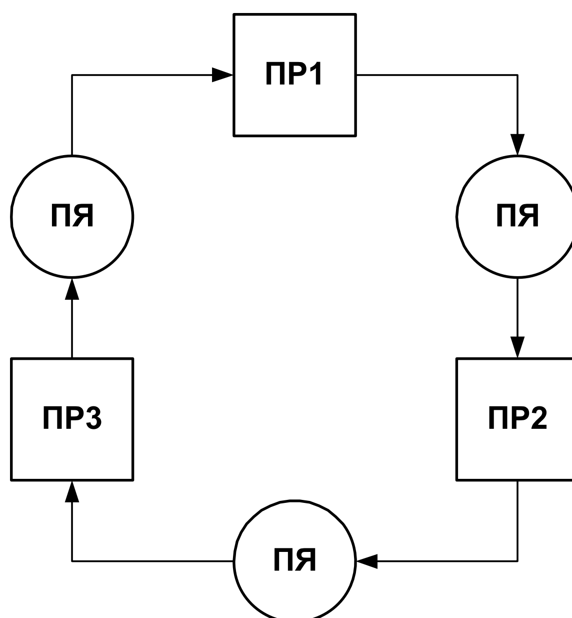



Рис. 3.13. Кольцевая схема взаимодействия процессов

Листинг 3.1. Вариант без тупиковой ситуации

```
ПР1:
        …
        ПОСЛАТЬ СООБЩЕНИЕ (ПР2, М1, ПЯ2);
        ЖДАТЬ СООБЩЕНИЕ (ПР3, М3, ПЯ1);
        …

ПР2:
        …
```

```
        ПОСЛАТЬ СООБЩЕНИЕ (ПР3, М2, ПЯ3);
        ЖДАТЬ СООБЩЕНИЕ (ПР1, М1, ПЯ2);
        …

ПР3:
        …
        ПОСЛАТЬ СООБЩЕНИЕ (ПР1, М3, ПЯ1);
        ЖДАТЬ СООБЩЕНИЕ (ПР2, М2, ПЯ3);
        …
```

Листинг 3.2. Вариант с тупиковой ситуацией

```
ПР1:
        …
        ЖДАТЬ СООБЩЕНИЕ (ПР3, М3, ПЯ1);
        ПОСЛАТЬ СООБЩЕНИЕ (ПР2, М1, ПЯ2);
        …

  ПР2:
        …
        ЖДАТЬ СООБЩЕНИЕ (ПР1, М1, ПЯ2):
        ПОСЛАТЬ СООБЩЕНИЕ (ПР3, М2, ПЯ3):
        …

ПР3:
        …
        ЖДАТЬ СООБЩЕНИЕ (ПР2, М2, ПЯ3):
        ПОСЛАТЬ СООБЩЕНИЕ (ПР1, М3, ПЯ1);
        …
```

В самом деле, во втором варианте ни один из процессов не сможет послать сообщения до тех пор, пока сам его не получит, а этого события никогда не произойдет, поскольку ни один процесс не может этого сделать.

*Пример тупика на ресурсах типа CR и SR*

Пусть некоторый процесс *ПР1* должен обменяться сообщениями с процессом *ПР2* и каждый из них запрашивает некоторый ресурс *R*, причем *ПР1* требует три единицы этого ресурса для своей работы, а *ПР2* – две единицы и только на время обработки сообщения. Всего же имеются только четыре единицы ресурса *R*. Запрос ресурса можно реализовать через соответствующий монитор с процедурами REQUEST (R, N) – запрос *N* единиц ресурса *R* и RELEASE (R, N) – освобождение, возврат *N* единиц ресурса *R*. Обмен сообщениями будем осуществлять через почтовый ящик *MB*. Фрагменты программ *ПР1* и *ПР2* приведены в листинге. 3.3.

Листинг 3.3. Пример тупика на CR- и SR-ресурсах

```
      …
ПР1:  REQUEST (R, 3);
      …
      SEND_MESSAGE (ПР2, сообщение, MB);
      WAIT_ANSWER (ответ, MB);
      …
      RELEASE (R, 3);
      …
------------------------------------------------------------------
      …
ПР2:  WAIT_MESSAGE (ПР1, сообщение, MB);
      REQUEST (R, 2;
      ОБРАБОТКА СООБЩЕНИЯ;
      RELEASE (R, 2);
      SEND_ANSWER (ответ, MB);
      …
```

Эти два процесса всегда будут попадать в тупик. Процесс *ПР2*, если будет выполняться первым, сначала ожидает сообщения от процесса *ПР1*, после чего будет заблокирован при запросе ресурса *R*, часть которого будет уже отдана *ПР1*. Процесс *ПР1*, завладев частью ресурса *R*, будет заблокирован на ожидании ответа от *ПР2*, которого никогда не получит, так как для этого *ПР2* нужно получить ресурс *R,* находящийся в распоряжении *ПР1*. Тупика можно избежать лишь при условии, что на время ожидания ответа от *ПР2* процесс *ПР1* будет отдавать хотя бы одну единицу ресурса *R*, которыми он сейчас владеет. В данном примере, как и в предыдущем, причиной тупика являются ошибки программирования.

*Пример тупика на ресурсах типа SR*

Предположим, что существуют два процесса *ПР1* и *ПР2*, разделяющих два ресурса типа *SR*: *R1* и *R2*. Пусть взаимное исключение доступов к этим ресурсам реализуется с помощью семафоров *S1* и *S2* соответственно. Процессы *ПР1* и *ПР2* обращаются к ресурсам в порядке, проиллюстрированном на рис. 3.14. Здесь несущественные (с точки зрения обращения к ресурсам) детали опущены. Считаем, что оба семафора первоначально установлены в единицу. Пространство возможных вычислений приведено на рис. 3.15.

| **Процесс ПР1** | **Процесс ПР2** |
|---|---|
| ⋮ | ⋮ |
| **1: P(S2);** | **5: P(S1);** |
| ⋮ | ⋮ |
| **2: P(S1);** | **6: P(S2);** |
| ⋮ | ⋮ |
| **3: V(S1);** | **7: V(S1);** |
| ⋮ | ⋮ |
| **4: V(S2);** | **8: V(S2);** |
| ⋮ | ⋮ |

Рис. 3.14. Пример последовательности операторов для двух процессов, которые могут привести к тупиковой ситуации

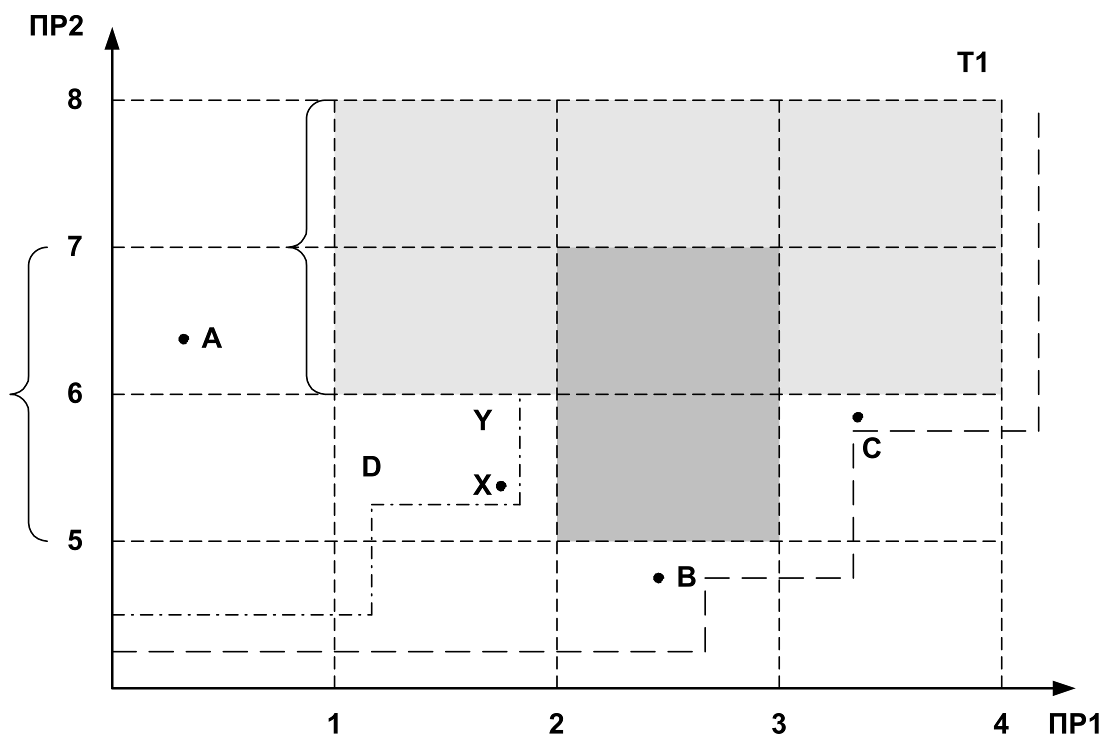



Рис. 3.15. Пространство состояний системы двух параллельных конкурирующих процессов

Горизонтальная ось задает выполнение процесса *ПР1*, вертикальная – *ПР2*. Вертикальные линии, пронумерованные от 1 до 4, соответствуют операторам 1-4 процесса *ПР1*. Аналогично горизонтальные линии, пронумерованные от 5 до 8, соответствуют операторам 5-8 программы *ПР2*. Точка на плоскости определяет состояние вычислений в некоторый момент времени. Так, точка *A* соответствует ситуации, при которой *ПР1* начал исполнение, но не достиг оператора 1, а *ПР2* выполнил оператор 6, но не дошел до оператора 7. По мере выполнения точка будет двигаться го-

ризонтально вправо, если исполняется *ПР1*, и вертикально вверх, если исполняется *ПР2*.

Интервалы исполнения, во время которых ресурсы *R1* и *R2* используются каждым процессом, показаны с помощью фигурных скобок. Линии 1-8 делят пространство вычислений на 25 прямоугольников, каждый из которых задает состояние вычислений. Закрашенные серым цветом состояния являются недостижимыми из-за взаимного исключения *ПР1* и *ПР2* при доступе к ресурсам *R1* и *R2*.

Рассмотрим последовательность исполнения 1-2-5-3-6-4-7-8, представленную траекторией *T1*. Когда процесс *ПР2* запрашивает ресурс *R1* (оператор 5), ресурс недоступен (оператор выполнен, семафор закрыт). Поэтому процесс *ПР2* заблокирован в точке *B*. Как только процесс *ПР1* достигнет оператора 3, процесс *ПР2* деблокируется по ресурсу *R1*. Аналогично в точке *C* процесс *ПР2* будет заблокирован при попытке доступа к ресурсу *R2* (оператор 6). Как только процесс *ПР1* достигнет оператора 4, процесс *ПР2* деблокируется по ресурсу *R2*.

Если же, например, выполняется последовательность 1-5-2-6, то процесс *ПР1* заблокируется в точке *X* при выполнении оператора 2, а процесс *ПР2* заблокируется в точке *Y* при выполнении оператора 6. При этом процесс *ПР1* ждет, когда процесс *ПР2* выполнит оператор 7, а *ПР2* ждет, когда *ПР1* выполнит оператор 4. Оба процесса будут находиться в тупике, ни *ПР1*, ни *ПР2* не могут закончить выполнение. При этом все ресурсы, которые получили *ПР1* и *ПР2*, становятся недоступными для других процессов, что резко снижает возможности вычислительной системы по обслуживанию их. Отметим одно очень важное обстоятельство: тупик будет неизбежным, если вычисления зашли в прямоугольник *D*, являющийся критическим состоянием.

Для того чтобы возник тупик, необходимо, чтобы одновременно выполнялись четыре условия:

1) взаимного исключения, при котором процессы осуществляют монопольный доступ к ресурсам;

2) ожидания, при котором процесс, запросивший ресурс, ждет до тех пор, пока запрос не будет удовлетворен, при этом удерживая ранее полученные ресурсы;

3) отсутствия перераспределения, при котором ресурсы нельзя отобрать у процесса, если они ему уже выделены;

4) кругового ожидания, при котором существует замкнутая цепь процессов, каждый из которых ждет ресурс, удерживаемый его предшественником в этой цепи.

Проанализировав содержательный смысл этих четырех условий, легко убедиться, что все они выполняются в точке *Y* (см. рис. 3.15).

## 4.3. Методы борьбы с тупиками

Проблема тупиков является чрезвычайно серьезной и сложной. В настоящее время разработано несколько подходов и методов разрешения этой проблемы, однако ни один из них нельзя считать панацеей. В некоторых случаях цена, которую приходится платить за то, чтобы сделать систему свободной от тупиков, слишком высока. В других случаях, например в системах управления процессами реального времени, просто нет иного выбора, поскольку возникновение тупика может привести к катастрофическим последствиям.

Проблема борьбы с тупиками становится все более актуальной и сложной по мере развития и внедрения параллельных вычислительных систем. При проектировании таких систем разработчики стараются проанализировать возможные неприятные ситуации, используя специальные модели и методы. Борьба с тупиковыми ситуациями основывается на одной из трех стратегий:

предотвращение тупика;

обход тупика;

распознавание тупика с последующим восстановлением.

### 4.3.1. Предотвращение тупиков

Предотвращение тупика основывается на предположении о чрезвычайно высокой его стоимости, поэтому лучше потратить дополнительные ресурсы системы, чтобы исключить вероятность возникновения тупика при любых обстоятельствах. Этот подход используется в наиболее ответственных системах, часто это системы реального времени.

Предотвращение можно рассматривать как запрет существования опасных состояний. Поэтому дисциплина, предотвращающая тупик, должна гарантировать, что какое-либо из четырех условий, необходимых для его наступления, не может возникнуть.

*Условие взаимного исключения* можно подавить путем разрешения неограниченного разделения ресурсов. Это удобно для повторно входимых программ и ряда драйверов, но совершенно неприемлемо к совместно используемым переменным в критических интервалах.

*Условие ожидания* можно подавить, предварительно выделяя ресурсы. При этом процесс может начать исполнение, только получив все необходимые ресурсы заранее. Следовательно, общее число затребованных параллельными процессами ресурсов должно быть не больше возможностей системы. Поэтому предварительное выделение может привести к снижению эффективности работы вычислительной системы в целом. Необходимо также отметить, что предварительное выделение зачастую невозможно,

так как необходимые ресурсы становятся известны процессу только после начала исполнения.

*Условие отсутствия перераспределения* можно исключить, позволяя операционной системе отнимать у процесса ресурсы. Для этого в операционной системе должен быть предусмотрен механизм запоминания состояния процесса с целью последующего восстановления. Перераспределение процессора реализуется достаточно легко, в то время как перераспределение устройств ввода/вывода крайне нежелательно.

*Условие кругового ожидания* можно исключить, предотвращая образование цепи запросов. Это можно обеспечить с помощью принципа *иерархического выделения ресурсов.* Все ресурсы образуют некоторую иерархию. Процесс, затребовавший ресурс на одном уровне, может затем потребовать ресурсы только на более высоком уровне. Он может освободить ресурсы на данном уровне только после освобождения всех ресурсов на всех более высоких уровнях. После того как процесс получил, а потом освободил ресурсы данного уровня, он может запросить ресурсы на том же самом уровне. Пусть имеются процессы *ПР1* и *ПР2*, которые могут иметь доступ к ресурсам *R1* и *R2*, причем *R2* находится на более высоком уровне иерархии. Если *ПР1* захватил *R1*, то *ПР2* не может захватить *R2*, так как доступ к нему проходит через доступ к *R1*, который уже захвачен *ПР1*. Таким образом, создание замкнутой цепи исключается. Иерархическое выделение ресурсов часто не дает никакого выигрыша, если порядок использования ресурсов, определенный в описании процессов, отличается от порядка уровней иерархии. В этом случае ресурсы будут использоваться крайне неэффективно.

В целом стратегия предотвращения тупиков – это очень дорогое решение проблемы, и она используется нечасто.

### 4.3.2. Обход тупиков

Обход тупика можно интерпретировать как запрет входа в опасное состояние. Если ни одно из упомянутых четырех условий не исключено, то вход в опасное состояние можно предотвратить при наличии у системы информации о последовательности запросов, связанных с каждым параллельным процессом. Доказано, что если вычисления находятся в любом неопасном состоянии, то существует по крайней мере одна последовательность состояний, которая обходит опасное. Следовательно; достаточно проверить, не приведет ли выделение затребованного ресурса сразу же к опасному состоянию. Если да, то запрос отклоняется. Если нет, его можно выполнить. Определение, того, является ли состояние опасным или нет, требует анализа последующих запросов процессов.

Рассмотрим следующий пример. Пусть имеется система из трех вычислительных процессов, которые потребляют некоторый ресурс *R* типа *SR*,

который выделяется дискретными взаимозаменяемыми единицами, причем существует всего десять единиц этого ресурса. В табл. 3.1 приведены сведения о текущем распределении процессами этого ресурса *R*, об их текущих запросах на этот ресурс и о максимальных потребностях процессов в ресурсе *R*.

Последний столбец в табл. 3.1 показывает, сколько еще единиц ресурса может затребовать каждый из процессов, если получит ресурс на свой текущий запрос.

Таблица 3.1

Пример распределения ресурсов

| Имя процесса | Выделено | Запрос | Максимальная потребность | “Остаток” потребностей |
|---|---|---|---|---|
| A | 2 | 3 | 6 | 1 |
| B | 3 | 2 | 7 | 2 |
| C | 2 | 3 | 5 | 0 |

Если запрос процесса *A* будет удовлетворен первым, то он в принципе может запросить еще одну единицу ресурса *R*, и уже в этом случае мы тогда получим тупиковую ситуацию, поскольку ни один из процессов не сможет продолжить свои вычисления. Следовательно, при выполнении запроса процесса *A* мы попадаем в *ненадежное* состояние. (Термин “ненадежное состояние” не предполагает, что в данный момент существует или в какое-то время обязательно возникнет тупиковая ситуация. Он просто говорит о том, что в случае некоторой неблагоприятной последовательности событий система может зайти в тупик.)

Если первым будет выполнен запрос процесса *B*, то у нас останется свободной еще одна единица ресурса *R*. Однако если процесс *B* запросит еще две, а не одну единицу ресурса *R*, а он может это сделать, то мы опять получим тупиковую ситуацию.

Если же мы сначала выполним запрос процесса *C* и выделим ему не две (как у процесса *B*), а все три единицы ресурса *R* и у нас при этом даже не останется никакого резерва, то, поскольку на этом его потребности в ресурсах заканчиваются, процесс *C* сможет благополучно завершиться и вернуть системе все свои ресурсы. Это приведет к тому, что свободное количество ресурса *R* станет равно пяти. Теперь уже можно будет выполнить запрос либо процесса *B*, либо процесса *A*, но не обоих сразу.

Часто бывает так, что последовательность запросов, связанных с каждым процессом, неизвестна заранее. Но если заранее известен общий запрос на ресурсы каждого типа, то выделение ресурсов можно контролировать. В этом случае необходимо для каждого требования, предполагая, что оно удовлетворено, определить, существует ли среди общих запросов от всех процессов некоторая последовательность требований, которая может

привести к опасному состоянию. Данный подход является примером контролируемого выделения ресурса.

Классическое решение этой задачи известно как *алгоритм банкира* Дейкстры. Алгоритм банкира напоминает процедуру принятия решения, может ли банк безопасно для себя дать взаймы денег. Принятие решения основывается на информации о потребностях клиента (текущих и максимально возможных) и учете текущего баланса банка. Несмотря на то, что этот алгоритм нигде практически не используется, рассмотрим его, так как он интересен с методической и академической точек зрения. Текст программы алгоритма банкира приведен в листинге 3.4.

Листинг 3.4. Алгоритм банкира Дейкстры

```
begin
  Своб_рес := Всего_рес;
  for i := 1 to N do
  begin
    Своб_рес := Своб_рес - Получ(i);
    Остаток(i) := Max(i) - Получ(i);
    Заверш(i) := false;   {процесс может не завер-
    шиться }
  end
  flag := true;           { признак продолжения ана-
  лиза }
  white flag do
  begin
    flag := false;
    for i := 1 to N do
    begin
      if ( not ( Заверш(i)) and (Остаток(i) <=
      Своб_рес))
        then begin
          Заверш(i) := true;
          Своб_рес := Своб_рес + Получ(i);
          Flag := true;
        end
    end
  end;
  if Своб_рес = Всего_рес
    then  Состояние системы безопасное и можно вы-
    дать
          ресурс
    else  Состояние не безопасное и выдавать ресурс
          нельзя
end.
```

Пусть существует N процессов, для каждого из которых известно максимальное количество потребностей в некотором ресурсе R (обозначим эти потребности через Max(i)). Ресурсы выделяются не сразу все, а в соответствии с текущим запросом. Считается, что все ресурсы *i*-го процесса будут освобождены по его завершении. Количество полученных ресурсов для *i*-го процесса обозначим Получ(i). Остаток в потребностях *i*-го процесса на ресурс *R* обозначим через Остаток(i). Признак того, что процесс может не завершиться – это значение false для переменной Заверш(i). Наконец, переменная Своб_рес будет означать количество свободных единиц ресурса *R*, а максимальное количество ресурсов в системе определено значением Всего_рес.

Каждый раз, когда какой-то остаток может быть выделен из числа остающихся незанятыми ресурсов, предполагается, что соответствующий процесс работает, пока не окончится, а затем его ресурсы освобождаются. Если в конце концов все ресурсы освобождаются, значит, все процессы могут завершиться и система находится в безопасном состоянии. Другими словами, согласно алгоритму банкира система удовлетворяет только те запросы, при которых ее состояние остается надежным. Новое состояние безопасно тогда и только тогда, когда каждый процесс все же может окончиться. Именно это условие и проверяется в алгоритме банкира. Запросы процессов, приводящие к переходу системы в ненадежное состояние, не выполняются и откладываются до момента, когда его все же можно будет выполнить.

Алгоритм банкира позволяет продолжать выполнение таких процессов, которым в случае системы с предотвращением тупиков пришлось бы ждать. Хотя алгоритм банкира относительно прост, его реализация может обойтись довольно дорого. Основным накладным расходом стратегии обхода тупика с помощью контролируемого выделения ресурса является время выполнения алгоритма, так как он выполняется при каждом запросе. Причем алгоритм работает медленнее всего, когда система близка к тупику. Необходимо отметить, что обход тупика неприменим при отсутствии информации о требованиях процессов на ресурсы.

Рассмотренный алгоритм примитивен, в нем учитывается только один вид ресурса, тогда как в реальных системах количество различных типов ресурсов бывает очень большим. Были опубликованы варианты этого алгоритма для большого числа различных типов системных ресурсов. Однако все равно алгоритм не получил распространения. Причин тому несколько:

Алгоритм исходит из того, что количество распределяемых ресурсов в системе фиксировано, постоянно. Иногда это не так, например вследствие неисправности отдельных устройств,

Алгоритм требует, чтобы пользователи заранее указывали свой максимальные потребности в ресурсах. Это чрезвычайно трудно реализовать. Часть таких сведений, конечно, могла бы подготавливать система про-

граммирования, но все равно часть информации о потребностях в ресурсах должны давать пользователи. Однако, поскольку компьютеры становятся все более дружественными по отношению к пользователям, все чаще встречаются пользователи, которые не имеют ни малейшего представления о том, какие ресурсы им потребуются.

Алгоритм требует, чтобы число работающих процессов оставалось постоянным. Это возможно только для очень редких случаев. Очевидно, что выполнение этого требования в общем случае не реально, особенно в мультитер-минальных системах либо если пользователь может запускать по несколько процессов параллельно.

### 4.3.3. Обнаружение тупика

Чтобы распознать тупиковое состояние, необходимо для каждого процесса определить, сможет ли он когда-либо снова развиваться, то есть изменять свои состояния. Так как нас интересует возможность развития процесса, а не сам процесс смены состояния, то достаточно рассмотреть только самые благоприятные изменения состояния.

Очевидно, что незаблокированный процесс (он только что получил ресурс и поэтому не заблокирован) через некоторое время освобождает все свои ресурсы и затем благополучно завершается. Освобождение ранее занятых ресурсов может “разбудить” некоторые ранее заблокированные процессы, которые, в свою очередь, развиваясь, смогут освободить другие ранее занятые ресурсы. Это может продолжаться до тех пор, пока либо не останется незаблокированных процессов, либо какое-то количество процессов все же останется заблокированными. В последнем случае (если существуют заблокированные процессы при завершении указанной последовательности действий) начальное состояние $S$ является состоянием тупика, а оставшиеся процессы находятся в тупике в состоянии $S$. В противном случае $S$ не есть состояние тупика.

*Обнаружение тупика посредством редукции графа повторно используемых ресурсов*

Наиболее благоприятные условия для незаблокированного процесса $P$, могут быть представлены редукцией (сокращением) графа повторно используемых ресурсов. Редукция графа может быть описана следующим образом:

Граф повторно используемых ресурсов сокращается процессом $P_i$, который не является ни заблокированной, ни изолированной вершиной, с помощью удаления всех ребер, входящих в вершину $P_i$ и выходящих из $P_i$. Эта процедура является эквивалентной приобретению процессом $P$, неких ресурсов, на которые он ранее выдавал запросы, а затем освобождению всех его ресурсов. Тогда $P_i$, становится изолированной вершиной.

Граф повторно используемых ресурсов несокращаем (не редуцируется), если он не может быть сокращен ни одним процессом.

Граф ресурсов типа RS является полностью сокращаемым, если существует последовательность сокращений, которая удаляет все дуги графа.

*Алгоритм обнаружения тупика по наличию замкнутой цепочки запросов*

Распознавание тупика может быть основано на анализе модели распределения ресурсов. Один из алгоритмов, основанный на методе обнаружения замкнутой цепи запросов, был разработан сотрудниками фирмы IBM; этот алгоритм использовался в одной из ОС этой компании. Он использует информацию о состоянии системы, содержащуюся в двух таблицах: таблице текущего распределения (назначения) ресурсов RATBL и "таблице" заблокированных процессов PWTBL (для каждого вида ресурса может быть свой список заблокированных процессов). При каждом запросе на получение или освобождении ресурсов содержимое этих таблиц модифицируется, а при запросе – анализируется в соответствии со следующим алгоритмом, который описан по пунктам.

1. Запрос от процесса *J* на занятый ресурс *I*.

2. Поместить номер ресурса *I* в PWTBL в строке с номером процесса *J*.

3. Использовать *I* в качестве смещения в RATBL, чтобы найти номер процесса *К*, который владеет ресурсом.

4. Использовать К в качестве смещения в PWTBL.

5. Проверить, ждет ли процесс К освобождения какого-либо ресурса I'. Если нет, то перейти к шагу 6, в противном случае – к шагу 7.

6. Перевести J в состояние ожидания и выйти из алгоритма.

7. Использовать I' в качестве смещения в RATBL, чтобы найти номер блокирующего его процесса К'.

8. Проверить К' = J. Если нет, то перейти к шагу 9, в противном случае – к шагу 11.

9. Проверить, вся ли таблица PWTBL просмотрена. Если да, то перейти к шагу 6, в противном случае – к шагу 10.

10. Присвоить К := К' и перейти к шагу 4.

11. Вывод о наличии тупика с последующим восстановлением.

12. Конец алгоритма.

Распознавание тупика требует дальнейшего восстановления.

Восстановление можно интерпретировать как запрет постоянного пребывания в опасном состоянии. Существуют следующие методы восстановления:

принудительный перезапуск системы, характеризующийся потерей информации обо всех процессах, существовавших до перезапуска;

принудительное завершение процессов, находящихся в тупике;

принудительное последовательное завершение процессов, находящихся в тупике, с последующим вызовом алгоритма распознавания после каждого завершения до исчезновения тупика;

перезапуск процессов, находящихся в тупике, с некоторой контрольной точки, то есть из состояния, предшествовавшего запросу на ресурс;

перераспределение ресурсов с последующим последовательным перезапуском процессов, находящихся в тупике.

Основные издержки восстановления составляют потери времени на повторные вычисления, которые могут оказаться весьма существенными. К сожалению, в ряде случаев восстановление может стать невозможным: например, исходные данные, поступившие с каких-либо датчиков, могут уже измениться, а предыдущие значения будут безвозвратно потеряны.

# ГЛАВА 4

# УПРАВЛЕНИЕ ПАМЯТЬЮ И ДАННЫМИ

## 1. Организация и распределение памяти

Модули управления памятью в операционной системе обеспечивают управление основной памятью, под которой понимается оперативная память. В иерархической структуре ОС модули управления памятью обычно помещаются на некотором промежуточном уровне, т.е. не на самом высоком уровне (поскольку к нм обращаются модули других уровней) и не на самом низком уровне. Однако целесообразно рассмотреть в первую очередь именно управление памятью, поскольку в настоящее время главной отличительной чертой многих современных ОС являются реализованные в них алгоритмы распределения памяти.

### 1.1. Задачи управления памятью

Память вычислительной системы часто оказывается тем узким местом, которое сдерживает повышение производительности ЭВМ. Как ресурс память характеризуется следующими особенностями.

1. Допускает параллельное пользование различными заданиями, что приводит к необходимости защиты определенных областей от взаимного влияния программ.
2. Память вычислительной системы включает два уровня, оперативную и внешнюю память, различных по назначению, между которыми осуществляется взаимодействие путем обмена данными.
3. Любое задание требует для своего выполнения определенного объема оперативной памяти, без которого оно не может выполняться. Во внешней памяти нуждается не каждое задание.
4. Оперативная память является одним из дорогостоящих и дефицитных ресурсов, и от организации ее использования во многом зависит производительность ЭВМ.

Для эффективного использования оперативной памяти необходимо ею управлять. Основными функциями управления оперативной памятью являются:

1. Учет наличия свободного пространства памяти.
2. Распределение памяти, т.е. принятие решения о том, кому распределяется память и в каком количестве. Если оперативная память одновременно используется несколькими заданиями, то модули управления памятью должны решать, запрос какого задания должен быть удовлетворен первым.
3. Выделение памяти – если решение о распределении памяти принято, конкретная область памяти выделяется заданию, а информация о состоянии памяти должна быть откорректирована соответствующим образом.
4. Освобождение – задание может освобождать выделенную ему память явно, или модули управления памятью могут односторонне забирать память, руководствуясь стратегией освобождения. После освобождения памяти информация о состоянии памяти должна быть уточнена.

При разработке конкретной ОС на выбор конкретной стратегии управления памятью оказывают влияние различные соображения. Основными из них являются: обеспечение максимальной загрузки оперативной памяти и минимизация времени передачи данных между уровнями памяти в процессе решения задач.

Выполнение этих требований существенно зависит от организации памяти в системе.

## 1.2. Способы распределения памяти

### 1.2.1. Память и отображения, виртуальное адресное пространство

Если не принимать во внимание программирование на машинном языке (эта технология практически не используется уже очень давно), то можно сказать, что программист обращается к памяти с помощью некоторого набора логических имен, которые чаще всего являются символьными, а не числовыми и для которого отсутствует отношение порядка. Другими словами, в общем случае множество переменных неупорядочено, хотя отдельные переменные и могут иметь частичную упорядоченность (например, элементы массива). Имена переменных и входных точек программных модулей составляют пространство имен.

С другой стороны, существует понятие физической оперативной памяти, собственно с которой и работает процессор, извлекая из нее команды и данные и помещая в нее результаты вычислений. *Физическая память* представляет собой упорядоченное множество ячеек, и все они пронумерованы, то есть к каждой из них можно обратиться, указав ее порядковый

номер (адрес). Количество ячеек физической памяти ограничено и фиксировано.

Системное программное обеспечение должно связать каждое указанное пользователем имя с физической ячейкой памяти, то есть осуществить отображение пространства имен на физическую память компьютера. В общем случае это отображение осуществляется в два этапа (рис. 4.1): сначала системой программирования, а затем операционной системой (с помощью специальных программных модулей управления памятью и использования соответствующих аппаратных средств вычислительной системы). Между этими этапами обращения к памяти имеют форму *виртуального* или логического адреса. При этом можно сказать, что множество всех допустимых значений виртуального адреса для некоторой программы определяет ее *виртуальное адресное пространство* или виртуальную память. Виртуальное адресное пространство программы прежде всего зависит от архитектуры процессора и от системы программирования и практически не зависит от объема реальной физической памяти, установленной в компьютер. Можно еще сказать, что адреса команд и переменных в готовой машинной программе, подготовленной к выполнению системой программирования, как раз и являются виртуальными адресами.

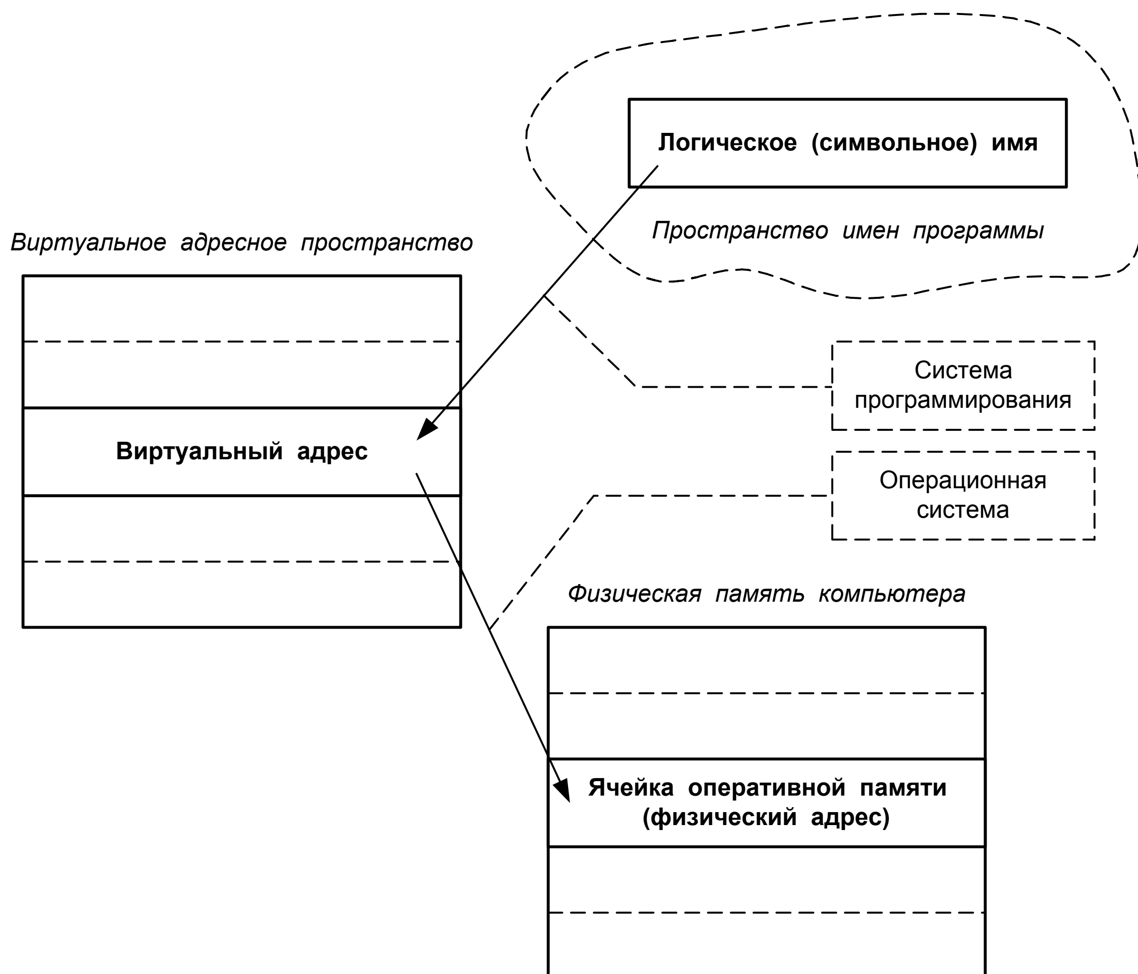



Рис. 4.1. Память и отображения

Как мы знаем, система программирования осуществляет трансляцию и компоновку программы, используя библиотечные программные модули. В результате работы системы программирования полученные виртуальные адреса могут иметь как двоичную форму, так и символьно-двоичную, то есть некоторые программные модули (их, как правило, большинство) и их переменные получают какие-то числовые значения, а те модули, адреса для которых не могут быть сейчас определены, имеют по-прежнему символьную форму и окончательная привязка их к физическим ячейкам будет осуществлена на этапе загрузки программы в память перед ее непосредственным выполнением.

Одним из частных случаев отображения пространства имен на физическую память является тождественность виртуального адресного пространства физической памяти. При этом нет необходимости осуществлять второе отображение. В данном случае говорят, что система программирования генерирует *абсолютную двоичную программу;* в этой программе все двоичные адреса таковы, что программа может исполняться только в том случае, если ее виртуальные адреса будут точно соответствовать физическим. Часть программных модулей любой операционной системы обязательно должна быть абсолютными двоичными программами. Эти программы размещаются по фиксированным адресам и с их помощью уже можно впоследствии реализовывать размещение остальных программ, подготовленных системой программирования таким образом, что они могут работать на различных физических адресах (то есть на тех адресах, на которые их разместит операционная система).

Другим частным случаем этой общей схемы трансляции адресного пространства является тождественность виртуального адресного пространства исходному пространству имен. Здесь уже отображение выполняется самой ОС, которая во время исполнения использует таблицу символьных имен. Такая схема отображения используется чрезвычайно редко, так как отображение имен на адреса необходимо выполнять для каждого вхождения имени (каждого нового имени) и особенно много времени тратится на квалификацию имен. Данную схему можно было встретить в интерпретаторах, в которых стадии трансляции и исполнения практически неразличимы. Это характерно для простейших компьютерных систем, в которых вместо операционной системы использовался встроенный интерпретатор (например, Basic).

Возможны и промежуточные варианты. В простейшем случае транслятор-компилятор генерирует относительные адреса, которые, по сути, являются виртуальными адресами с последующей настройкой программы на один из непрерывных разделов. Второе отображение осуществляется перемещающим загрузчиком. После загрузки программы виртуальный адрес теряется, и доступ выполняется непосредственно к физическим ячейкам. Более эффективное решение достигается в том случае, когда транслятор вырабатывает в качестве виртуального адреса относительный адрес и ин-

формацию о начальном адресе, а процессор, используя подготавливаемую операционной системой адресную информацию, выполняет второе отображение не один раз (при загрузке программы), а при каждом обращении к памяти.

Термин *виртуальная память* фактически относится к системам, которые сохраняют виртуальные адреса во время исполнения. Так как второе отображение осуществляется в процессе исполнения задачи, то адреса физических ячеек могут изменяться. При правильном применении такие изменения могут улучшить использование памяти, избавив программиста от деталей управления ею, и даже увеличить надежность вычислений.

Если рассматривать общую схему двухэтапного отображения адресов, то с позиции соотношения объемов упомянутых адресных пространств можно отметить наличие следующих трех ситуаций:

1) объем виртуального адресного пространства программы $V_V$ меньше объема физической памяти $V_P$;

2) $V_V = V_P$;

3) $V_V > V_P$.

Первая ситуация, при которой $V_V < V_P$, ныне практически не встречается, но тем не менее это реальное соотношение. Скажем, не так давно 16-разрядные мини-ЭВМ имели систему команд, в которых пользователи-программисты могли адресовать до $2^{16}$ = 64K адресов (обычно в качестве адресуемой единицы выступала ячейка памяти размером с байт). А физически старшие модели этих мини-ЭВМ могли иметь объем оперативной памяти в несколько мегабайт. Обращение к памяти столь большого объема осуществлялось с помощью специальных регистров, содержимое которых складывалось с адресом операнда (или команды), извлекаемым и/или определяемым из поля операнда (или из указателя команды). Соответствующие значения в эти специальные регистры, выступающие как базовое смещение в памяти, заносила операционная система. Для одной задачи в регистр заносилось одно значение, а для второй (третьей, четвертой и т. д.) задачи, размещаемой одновременно с первой, но в другой области памяти, заносилось, соответственно, другое значение. Вся физическая память, таким образом, разбивалась на разделы объемом по 64 Кбайт, и на каждый такой раздел осуществлялось отображение своего виртуального адресного пространства.

Ситуация, когда $V_V = V_P$ встречалась достаточно часто, особенно характерна она была для недорогих вычислительных комплексов. Для этого случая имеется большое количество методов распределения оперативной памяти.

Наконец, в наше время мы уже достигли того, что ситуация $V_V > V_P$ встречается даже в ПК, то есть в самых распространенных и недорогих компьютерах. Теперь это самая распространенная ситуация, и для нее имеется несколько методов распределения памяти, отличающихся как сложностью, так и эффективностью.

### 1.2.2. Простое непрерывное распределение и распределение с перекрытием (оверлейные структуры)

Простое непрерывное распределение – это самая простая схема, согласно которой вся память условно может быть разделена на три части:

1) область, занимаемая операционной системой;

2) область, в которой размещается исполняемая задача;

3) незанятая ничем (свободная) область памяти.

Изначально являясь самой первой схемой, она продолжает и сегодня быть достаточно распространенной. Эта схема предполагает, что ОС не поддерживает мультипрограммирование, поэтому не возникает проблемы распределения памяти между несколькими задачами. Программные модули, необходимые для всех программ, располагаются в области самой ОС, а вся оставшаяся память может быть предоставлена задаче. Эта область памяти при этом получается непрерывной, что облегчает работу системы программирования. Поскольку в различных однотипных вычислительных комплексах может быть разный состав внешних устройств (и, соответственно, они содержат различное количество драйверов), для системных нужд могут быть отведены отличающиеся объемы оперативной памяти, и получается, что можно не привязывать жестко виртуальные адреса программы к физическому адресному пространству. Эта привязка осуществляется на этапе загрузки задачи в память.

Чтобы для задач отвести как можно больший объем памяти, операционная система строится таким образом, что постоянно в оперативной памяти располагается только самая нужная ее часть. Эту часть ОС стали называть ядром. Остальные модули ОС могут быть обычными диск-резидентными (или транзитными), то есть загружаться в оперативную память только по необходимости, и после своего выполнения вновь освобождать память.

Такая схема распределения влечет за собой два вида потерь вычислительных ресурсов – потеря процессорного времени, потому что процессор простаивает, пока задача ожидает завершения операций ввода/вывода, и потеря самой оперативной памяти, потому что далеко не каждая программа использует всю память, а режим работы в этом случае однопрограммный. Однако это очень недорогая реализация и можно отказаться от многих функций операционной системы. В частности, такая сложная проблема, как защита памяти, здесь почти не стоит.

Если есть необходимость создать программу, логическое (и виртуальное) адресное пространство которой должно быть больше, чем свободная область памяти, или даже больше, чем весь возможный объем оперативной памяти, то используется распределение с перекрытием (так называемые *оверлейные структуры* – от *overlay* – перекрытие, расположение поверх чего-то). Этот метод распределения предполагает, что вся программа может быть разбита на части – сегменты. Каждая оверлейная программа

имеет одну главную часть (main) и несколько сегментов (segment), причем в памяти машины одновременно могут находиться только ее главная часть и один или несколько не перекрывающихся сегментов.

Пока в оперативной памяти располагаются выполняющиеся сегменты, остальные находятся во внешней памяти. После того как текущий (выполняющийся) сегмент завершит свое выполнение, возможны два варианта. Либо он сам (если данный сегмент не нужно сохранить во внешней памяти в его текущем состоянии) обращается к ОС с указанием, какой сегмент должен быть загружен в память следующим. Либо он возвращает управление главному сегменту задачи (в модуль main), и уже тот обращается к ОС с указанием, какой сегмент сохранить (если это нужно), а какой сегмент загрузить в оперативную память, и вновь отдает управление одному из сегментов, располагающихся в памяти. Простейшие схемы сегментирования предполагают, что в памяти в каждый конкретный момент времени может располагаться только один сегмент (вместе с модулем main). Более сложные схемы, используемые в больших вычислительных системах, позволяют располагать по несколько сегментов. В некоторых вычислительных комплексах могли существовать отдельно сегменты кода и сегменты данных. Сегменты кода, как правило, не претерпевают изменений в процессе своего исполнения, поэтому при загрузке нового сегмента кода на место отработавшего последний можно не сохранять во внешней памяти, в отличие от сегментов данных, которые сохранять необходимо.

Первоначально программисты сами должны были включать в тексты своих программ соответствующие обращения к ОС (их называют вызовами) и тщательно планировать, какие сегменты могут находиться в оперативной памяти одновременно, чтобы их адресные пространства не пересекались. Однако с некоторых пор эти вызовы система программирования стала подставлять в код программы сама, автоматически, если в том возникает необходимость. Так, в известной и популярной системе программирования Turbo Pascal, начиная с третьей версии, программист просто указывал, что данный модуль является оверлейным. И при обращении к нему из основной программы модуль загружался в память и ему передавалось управление. Все адреса определялись системой программирования автоматически, обращения к DOS для загрузки оверлеев тоже генерировались системой Turbo Pascal.

### 1.2.3. Распределение статическими и динамическими разделами

Для организации мультипрограммного режима необходимо обеспечить одновременное расположение в оперативной памяти нескольких задач (целиком или их частями). Самая простая схема распределения памяти между несколькими задачами предполагает, что память, незанятая ядром ОС, может быть разбита на несколько непрерывных частей (зон, разделов).

Разделы характеризуются именем, типом, границами (как правило, указываются начало раздела и его длина).

Разбиение памяти на несколько непрерывных разделов может быть фиксированным (статическим), либо динамическим (то есть процесс выделения нового раздела памяти происходит непосредственно при появлении новой задачи).

Вначале мы кратко рассмотрим статическое распределение памяти на несколько разделов.

*Разделы с фиксированными границами*

Разбиение всего объема оперативной памяти на несколько разделов может осуществляться единовременно (то есть в процессе генерации варианта ОС, который потом и эксплуатируется) или по мере необходимости оператором системы. Однако и во втором случае при выполнении разбиения памяти на разделы вычислительная система более ни для каких целей в этот момент не используется. Пример разбиения памяти на несколько разделов приведен на рис. 4.2.

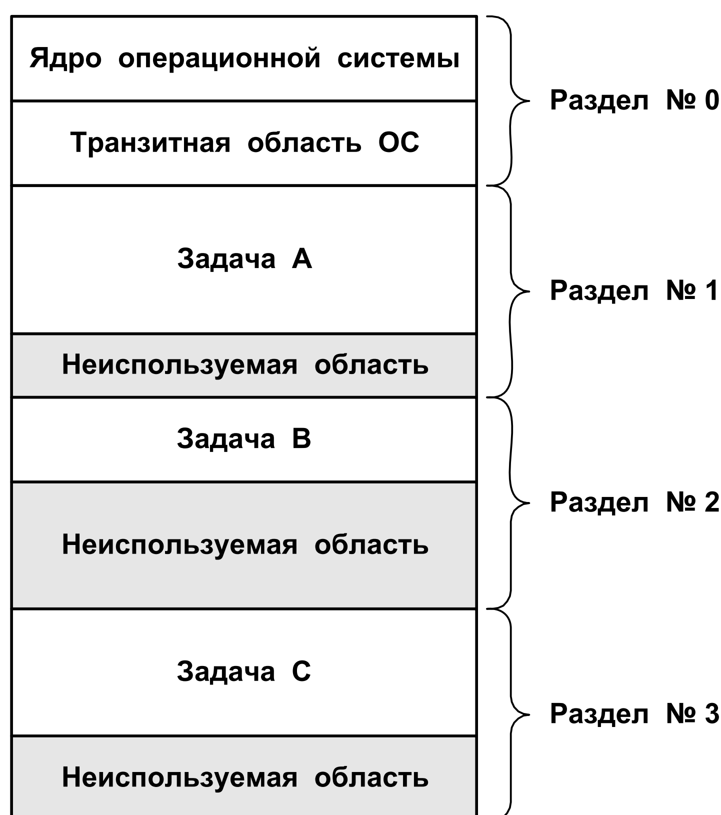



Рис. 4.2. Распределение памяти разделами с фиксированными границами

В каждом разделе в каждый момент времени может располагаться по одной программе (задаче). В этом случае по отношению к каждому разделу можно применить все те методы создания программ, которые используются для однопрограммных систем. Возможно использование оверлейных структур, что позволяет создавать большие сложные программы и в

то же время поддерживать коэффициент мультипрограммирования (под коэффициентом мультипрограммирования $\mu$ понимают количество параллельно выполняемых программ) на должном уровне. Первые мультипрограммные ОС строились по этой схеме. Использовалась эта схема и много лет спустя при создании недорогих вычислительных систем, ибо она является несложной и обеспечивает возможность параллельного выполнения программ. Иногда в некотором разделе размещалось по несколько небольших программ, которые постоянно в нем и находились. Такие программы назывались ОЗУ-резидентными (или просто – *резидентными).* Они же используются и в современных встроенных системах; правда, для них характерно, что все программы являются резидентными и внешняя память во время работы вычислительного оборудования не используется.

При небольшом объеме памяти и, следовательно, небольшом количестве разделов увеличить количество параллельно выполняемых приложений (особенно когда эти приложения интерактивны и во время своей работы они фактически не используют процессорное время, а в основном ожидают операций ввода/вывода) можно за счет *своппинга* (swapping). При своппинге задача может быть целиком выгружена на магнитный диск (перемещена во внешнюю память), а на ее место загружается либо более привилегированная, либо просто готовая к выполнению другая задача, находившаяся на диске в приостановленном состоянии. При своппинге из основной памяти во внешнюю (и обратно) перемещается вся программа, а не ее отдельная часть.

Серьезная проблема, которая возникает при организации мультипрограммного режима работы вычислительной системы, – это защита как самой ОС от ошибок и преднамеренного вмешательства задач в ее работу, так и самих задач друг от друга.

В самом деле, программа может обращаться к любым ячейкам в пределах своего виртуального адресного пространства. Если система отображения памяти не содержит ошибок и в самой программе их тоже нет, то возникать ошибок при выполнении программы не должно. Однако в случае ошибок адресации, что не так уж и редко случается, исполняющаяся программа может начать “обработку” чужих данных или кодов с непредсказуемыми последствиями. Одной из простейших, но достаточно сильных мер является введение регистров защиты памяти. В эти регистры ОС заносит граничные значения области памяти раздела текущей исполняющейся задачи. При нарушении адресации возникает прерывание и управление передается супервизору ОС. Обращения к ОС за необходимыми сервисами осуществляются не напрямую, а через команды программных прерываний, что обеспечивает передачу управления только в предопределенные входные точки кода ОС, и в системном режиме работы процессора, при котором регистры защиты памяти игнорируются. Таким образом, выполнение функции защиты требует введения специальных аппаратных механизмов, используемых операционной системой.

Основным недостатком такого способа распределения памяти является наличие порой достаточно большого объема неиспользуемой памяти (рис. 4.2). Неиспользуемая память может быть в каждом из разделов. Поскольку разделов несколько, то и неиспользуемых областей получается несколько, поэтому такие потери стали называть *фрагментацией памяти.* В отдельных разделах потери памяти могут быть очень значительными, однако использовать фрагменты свободной памяти при таком способе распределения не представляется возможным. Желание разработчиков сократить столь значительные потери привело их к следующим двум решениям:

1) выделять раздел ровно такого объема, который нужен под текущую задачу;

2) размещать задачу не в одной непрерывной области памяти, а в нескольких областях.

Второе решение реализовалось в нескольких способах организации виртуальной памяти, которые будут рассмотрены в п. 2 настоящей главы, а сейчас кратко рассмотрим первое решение.

*Разделы с подвижными границами*

Чтобы избавиться от фрагментации, можно попробовать размещать в оперативной памяти задачи плотно, одну за другой, выделяя ровно столько памяти, сколько задача требует. Одной из первых ОС, реализовавшей такой способ распределения памяти, была OS MVT. Специальный планировщик (диспетчер памяти) ведет список адресов свободной оперативной памяти. При появлении новой задачи диспетчер памяти просматривает этот список и выделяет для задачи раздел, объем которого либо равен необходимому, либо чуть больше, если память выделяется не ячейками, а некими дискретными единицами. При этом модифицируется список свободной памяти. При освобождении раздела диспетчер памяти пытается объединить освобождающийся раздел с одним из свободных участков, если таковой является смежным.

При этом список свободных участков может быть упорядочен либо по адресам, либо по объему. Выделение памяти под новый раздел может осуществляться одним из трех способов:

1) первый подходящий участок;

2) самый подходящий участок;

3) самый неподходящий участок.

В первом случае список свободных областей упорядочивается по адресам (например, по возрастанию адресов). Диспетчер памяти просматривает этот список и выделяет задаче раздел в той области, которая первой подойдет по объему. В этом случае, если такой фрагмент имеется, то в среднем необходимо просмотреть половину списка. При освобождении раздела также необходимо просмотреть половину списка. Правило “первый подходящий” приводит к тому, что память для небольших задач преимущественно будет выделяться в области младших адресов и, следовательно,

это будет увеличивать вероятность того, что в области старших адресов будут образовываться фрагменты достаточно большого объема.

Способ “самый подходящий” предполагает, что список свободных областей упорядочен по возрастанию объема этих фрагментов. В этом случае при просмотре списка для нового раздела будет использован фрагмент свободной памяти, объем которой наиболее точно соответствует требуемому. Требуемый раздел будет определяться по-прежнему в результате просмотра в среднем половины списка.

Однако оставшийся фрагмент оказывается настолько малым, что в нем уже вряд ли удастся разместить какой-либо еще раздел и при этом этот фрагмент попадет в самое начало списка. Поэтому в целом такую дисциплину нельзя назвать эффективной.

Как ни странно, самым эффективным правилом является последнее, по которому для нового раздела выделяется “самый неподходящий” фрагмент свободной памяти. Для этой дисциплины список свободных областей упорядочивается по убыванию объема свободного фрагмента. Очевидно, что если есть подходящий фрагмент памяти, то он сразу же и будет найден, и поскольку этот фрагмент является самым большим, то, скорее всего, после выделения из него раздела памяти для задачи оставшаяся область памяти еще сможет быть использована в дальнейшем.

Однако очевидно, что при любой дисциплине обслуживания, по которой работает диспетчер памяти, из-за того, что задачи появляются и завершаются в произвольные моменты времени и при этом они имеют разные объемы, то в памяти всегда будет наблюдаться сильная фрагментация. При этом возможны ситуации, когда из-за сильной фрагментации памяти диспетчер памяти не сможет образовать новый раздел, хотя суммарный объем свободных областей будет больше, чем необходимо для задачи. В этой ситуации возможно организовать так называемое *“уплотнение памяти”*. Для уплотнения памяти все вычисления приостанавливаются, и диспетчер памяти корректирует свои списки, перемещая разделы в начало памяти (или, наоборот, в область старших адресов). При определении физических адресов задачи будут участвовать новые значения базовых регистров, с помощью которых и осуществляется преобразование виртуальных адресов в физические. Недостатком этого решения является потеря времени на уплотнение и, что самое главное, невозможность при этом выполнять сами вычислительные процессы.

Данный способ распределения памяти тем не менее применялся достаточно длительное время в нескольких операционных системах, поскольку в нем для задач выделяется непрерывное адресное пространство, а это упрощает создание систем программирования и их работу.

## 1.3. Распределение оперативной памяти в современных операционных системах для ПК

Сначала определим – какие ОС следует относить к современным, а какие – нет? Стоит ли в изучать такую “несовременную” ОС, как MS-DOS? К современным ОС, прежде всего, относят те, что используют аппаратные возможности микропроцессоров, специально заложенные для организации высокопроизводительных и надежных вычислений. Однако эти ОС, как правило, очень сложны и громоздки. Они занимают большое дисковое пространство, требуют и большого объема оперативной памяти. Поэтому для решения некоторого класса задач вполне подходят и системы, использующие микропроцессоры в так называемом реальном режиме работы. Кроме того, достаточно часто для обслуживания компьютера необходимо выполнить простейшие программы – утилиты. Эти программы были созданы для DOS, они не требуют больших ресурсов, для их функционирования достаточно запустить MS-DOS или аналогичную простую ОС. Однако без выполнения этих программ невозможно порой установить или загрузить иные ОС (хоть и современные, но очень сложные и громоздкие). Поэтому целесообразно сделать хотя бы первичное, пусть не очень глубокое, ознакомление с MS-DOS.

### 1.3.1. Распределение оперативной памяти в MS-DOS

Как известно, MS-DOS – это однопрограммная ОС. В ней, конечно, можно организовать запуск резидентных или TSR-задач (*TSR* – terminate and stay resident – резидентная в памяти программа, которая благодаря изменениям в таблице векторов прерываний позволяет перехватывать прерывания и в случае обращения к ней выполнять необходимые нам действия), но в целом она предназначена для выполнения только одного вычислительного процесса. Поэтому распределение памяти в ней построено по самой простой схеме, которая была рассмотрена в п.п. “1.2.2. Простое непрерывное распределение и распределение с перекрытием (оверлейные структуры)”. Здесь лишь уточним некоторые характерные детали.

В IBM PC использовался 16-разрядный микропроцессор i8088, который за счет введения сегментного способа адресации позволял адресоваться к памяти объемом до 1 Мбайт. В последующих ПК (IBM PC AT, AT386 и др.) было принято решение поддерживать совместимость с первыми, поэтому при работе с DOS прежде всего рассматривают первый мегабайт. Вся эта память разделялась на несколько областей (рис. 4.3).

| Адрес | Размер | Содержимое | Примечание |
|---|---|---|---|
| 00000-003FF | 1 Кб | Таблица векторов прерываний | |
| 00400-005FF | 512 байт | Глобальные переменные BIOS<br>Глобальные переменные DOS | В ранних версиях здесь располагались глобальные переменные интерпретатора BASIC |
| 00600-0A000 | 35-60 Кб | Модуль IO.SYS<br>Модуль MSDOS.SYS:<br>- обслуживающие функции;<br>- буферы, рабочие и управляющие области;<br>- инсталлируемые драйверы<br>Резидентная часть COMMAND.COM:<br>- обработка программных прерываний;<br>- системная программа загрузки;<br>- программа загрузки транзитной части COMMAND.COM | Размер этой области зависит от версии MS-DOS и, главное, от конфигурационного файла CONFIG.SYS |
| | ≈ 580 Кб | Область памяти для выполнения программ пользователя и утилит MS-DOS. В эту область попадают програмы типа *.COM и *.EXE<br>Область расположения стека исполняющей-ся программы | Размер этой области сильно зависит от объема, занима-емого ядром ОС. Программа может перекрывать транзит-ную область COMMAND.COM<br>Стек "растет" снизу вверх |
| | 18 Кб | Транзитная часть командного процессора COMMAND.COM | Собственно командный интерпретатор |
| A0000-C7FFF | 160 Кб | Видеопамять. Область и размер используе-мого видеобуфера зависит от используе-мого режима | При работе в текстовом ре-жиме область памяти A0000-B0000 свободна и мо-жет быть использована в программе |
| C8000-E0000 | 96 Кб | Зарезервировано для расширения BIOS | |
| F0000-FFFFF | 64 Кб | Область ROM BIOS (System BIOS) | Обычно объем этой области равен 32 Кб, но может дости-гать и 128 Кб, занимая и млад-шие адреса |
| 100000-10FFFF | | High Memory Area<br>При наличии драйвера HIMEM.SYS здесь мож-но расположить основные системные файлы MS-DOS, освобождая тем самым область ос-новной памяти в первом мегабайте | Может использоваться при наличии специальных драй-веров. Используются специ-фикации XMS и EMS |

Рис. 4.3. Распределение оперативной памяти в MS-DOS

В состав MS-DOS входят следующие основные компоненты:

1. Базовая подсистема ввода/вывода – BIOS (base input-output system), включающая в себя помимо программы тестирования ПК (*POST* – power on self test – программа самотестирования при включении компьютера) обработчики прерываний (драйверы), расположенные в постоянном запоминающем устройстве. В конечном итоге, почти все остальные модули MS-DOS обращаются к BIOS. Если и не напрямую, то через модули более высокого уровня иерархии.
2. Модуль расширения BIOS – файл IO.SYS.
3. Основной базовый модуль обработки прерываний DOS – файл MSDOS.SYS. Именно этот модуль в основном реализует работу с файловой системой.

4. Командный процессор (интерпретатор команд) – файл COMMAND.COM.

5. Утилиты и драйверы, расширяющие возможности системы.

6. Программа загрузки MS-DOS – загрузочная запись (boot record), расположенная на дискете.

Вся память в соответствии с архитектурой IBM PC условно может быть разбита на три части.

В самых младших адресах памяти (первые 1024 ячейки) размещается таблица векторов прерываний. Это связано с аппаратной реализацией процессора i8088, на котором была реализована ПК. В последующих процессорах (начиная с i80286) адрес таблицы прерываний определяется через содержимое соответствующего регистра, но для обеспечения полной совместимости с первым процессором при включении или аппаратном сбросе в этот регистр заносятся нули. При желании, однако, в случае использования современных микропроцессоров i80x86 можно разместить векторы прерываний и в другой области.

Вторая часть памяти отводится для размещения программных модулей самой MS-DOS и для программ пользователя. Эта область памяти называется *Conventional Memory* (основная, стандартная память).

Наконец, третья часть адресного пространства отведена для постоянных запоминающих устройств и функционирования некоторых устройств ввода/вывода. Эта область памяти получила название *UMA* (upper memory areas – область верхней памяти).

В младших адресах основной памяти размещается то, что можно назвать ядром этой ОС – системные переменные, основные программные модули, блоки данных для буферирования операций ввода/вывода. Для управления устройствами, драйверы которых не входят в базовую подсистему ввода/вывода, загружаются так называемые *загружаемые* (или *инсталлируемые)* драйверы. Перечень инсталлируемых драйверов определяется специальным конфигурационным файлом CONFIG.SYS. После загрузки расширения BIOS – файла IO.SYS – последний (загрузив модуль MSDOS.SYS) считывает файл CONFIG.SYS и уже в соответствии с ним подгружает в память необходимые драйверы. В случае использования микропроцессоров i80x86 и наличия в памяти драйвера HIMEM.SYS модули IO.SYS и MSDOS.SYS могут быть размещены за пределами первого мегабайта в области, которая получила название *HMA* (high memory area).

Память с адресами, большими чем 10FFFFh, может быть использована в DOS-программах при выполнении их на микропроцессорах, имеющих такую возможность. Так, например, микропроцессор i80286 имел 24-разрядную шину адреса, а i80386 – уже 32-разрядную шину адреса. Но для этого с помощью специальных драйверов необходимо переключать процессор в другой режим работы, при котором он сможет использовать адреса выше 10FFFFh. Широкое распространение получили две основные

спецификации: *XMS* (extended memory specification) и *EMS* (expanded memory specification).

Остальные программные модули MS-DOS (в принципе, большинство из них является утилитами) оформлены как обычные исполняемые файлы. В основном они являются транзитными модулями, то есть загружаются в память только на время своей работы, хотя среди них имеются и TSR-программы.

Для того чтобы предоставить больше памяти программам пользователя, в MS-DOS применено то же решение, что и во многих других простейших ОС – командный процессор COMMAND.COM сделан состоящим из двух частей. Первая часть является резидентной, она размещается в области ядра. Вторая часть – транзитная; она размещается в области старших адресов раздела памяти, выделяемой для программ пользователя. И если программа пользователя перекрывает собой область, в которой была расположена транзитная часть командного процессора, то последний при необходимости восстанавливает в памяти свою транзитную часть, поскольку после выполнения программы она возвращает управление резидентной части COMMAND.COM.

Поскольку размер основной памяти (conventional memory) относительно небольшой, то очень часто системы программирования реализуют оверлейные структуры. Для этого в MS-DOS есть специальные вызовы.

### 1.3.2. Распределение оперативной памяти в Microsoft Windows 95/98

С точки зрения базовой архитектуры ОС Windows 95/98 они обе являются 32-разрядными, многопотоковыми ОС с вытесняющей многозадачностью. Основной пользовательский интерфейс этих ОС – графический.

Для своей загрузки они используют операционную систему MS-DOS 7.X (MS-DOS 98), и в случае если в файле MSDOS.SYS в секции [Options] прописано BootGUI = 0, то процессор работает в обычном реальном режиме. Распределение памяти в MS-DOS 7.X. такое же, как и в предыдущих версиях DOS. Однако при загрузке GUI-интерфейса перед загрузкой ядра Windows 95/98 процессор переключается в защищенный режим работы и начинает распределять память уже с помощью страничного механизма.

Таким образом, в системе фактически действует только страничный механизм преобразования виртуальных адресов в физические. Программы используют классическую “small” (малую) модель памяти. Каждая прикладная программа определяется 32-битными адресами, в которых сегмент кода имеет то же значение, что и сегменты данных. Единственный сегмент программы отображается непосредственно в область виртуального линейного адресного пространства, который, в свою очередь, состоит из 4 килобайтных страниц. Каждая страница может располагаться где

угодно в оперативной памяти (естественно, в том месте, куда ее разместит диспетчер памяти, который сам находится в невыгружаемой области) или может быть перемещена на диск, если не запрещено использовать страничный файл.

Младшие адреса виртуального адресного пространства совместно используются всеми процессами. Это сделано для обеспечения совместимости с драйверами устройств реального режима, резидентными программами и некоторыми 16-разрядными программами Windows. Безусловно, это плохое решение с точки зрения надежности, поскольку оно приводит к тому, что любой процесс может непреднамеренно (или же, наоборот, специально) испортить компоненты, находящиеся в этих адресах.

В Windows 95/98 каждая 32-разрядная прикладная программа выполняется в своем собственном адресном пространстве, но все они используют совместно один и тот же 32-разрядный системный код. Доступ к чужим адресным пространствам в принципе возможен. Другими словами, виртуальные адресные пространства не используют всех аппаратных средств защиты, заложенных в микропроцессор. В результате неправильно написанная 32-разрядная прикладная программа может привести к аварийному сбою всей системы. Все 16-битовые прикладные программы Windows разделяют общее адресное пространство, поэтому они так же уязвимы друг перед другом, как и в среде Windows 3.X.

Системный код Windows 95 размещается выше границы 2 Гбайт. В пространстве с отметками 2 и 3 Гбайт находятся системные библиотеки DLL, используемые несколькими программами. Заметим, что в 32-битовых микропроцессорах семейства i80x86 имеются четыре уровня защиты, именуемые кольцами с номерами от 0 до 3. Кольцо с номером 0 является наиболее привилегированным, то есть максимально защищенным. Компоненты системы Windows 95, относящиеся к кольцу 0, отображаются на виртуальное адресное пространство между 3 и 4 Гбайт. К этим компонентам относятся собственно ядро Windows, подсистема управления виртуальными машинами, модули файловой системы и виртуальные драйверы (VxD).

Область памяти между 2 и 4 Гбайт адресного пространства каждой 32-разрядной прикладной программы совместно используется всеми 32-разрядными прикладными программами. Такая организация позволяет обслуживать вызовы API непосредственно в адресном пространстве прикладной программы и ограничивает размер рабочего множества. Однако за это приходится расплачиваться снижением надежности. Ничто не может помешать программе, содержащей ошибку, произвести запись в адреса, принадлежащие системным DLL, и вызвать крах всей системы.

В области между 2 и 3 Гбайт также находятся все запускаемые 16-разрядные прикладные программы Windows. С целью обеспечения совместимости эти программы выполняются в совместно используемом ад-

ресном пространстве, где они могут испортить друг друга так же, как и в Windows 3.x.

Адреса памяти ниже 4 Мбайт также отображаются в адресное пространство каждой прикладной программы и совместно используются всеми процессами. Благодаря этому становится возможной совместимость с существующими драйверами реального режима, которым необходим доступ к этим адресам. Это делает еще одну область памяти незащищенной от случайной записи. К самым нижним 64 Кбайт этого адресного пространства 32-разрядные прикладные программы обращаться не могут, что дает возможность перехватывать неверные указатели, но 16-разрядные программы, которые, возможно, содержат ошибки, могут записывать туда данные.

Вышеизложенную модель распределения памяти можно проиллюстрировать с помощью рис. 4.4.

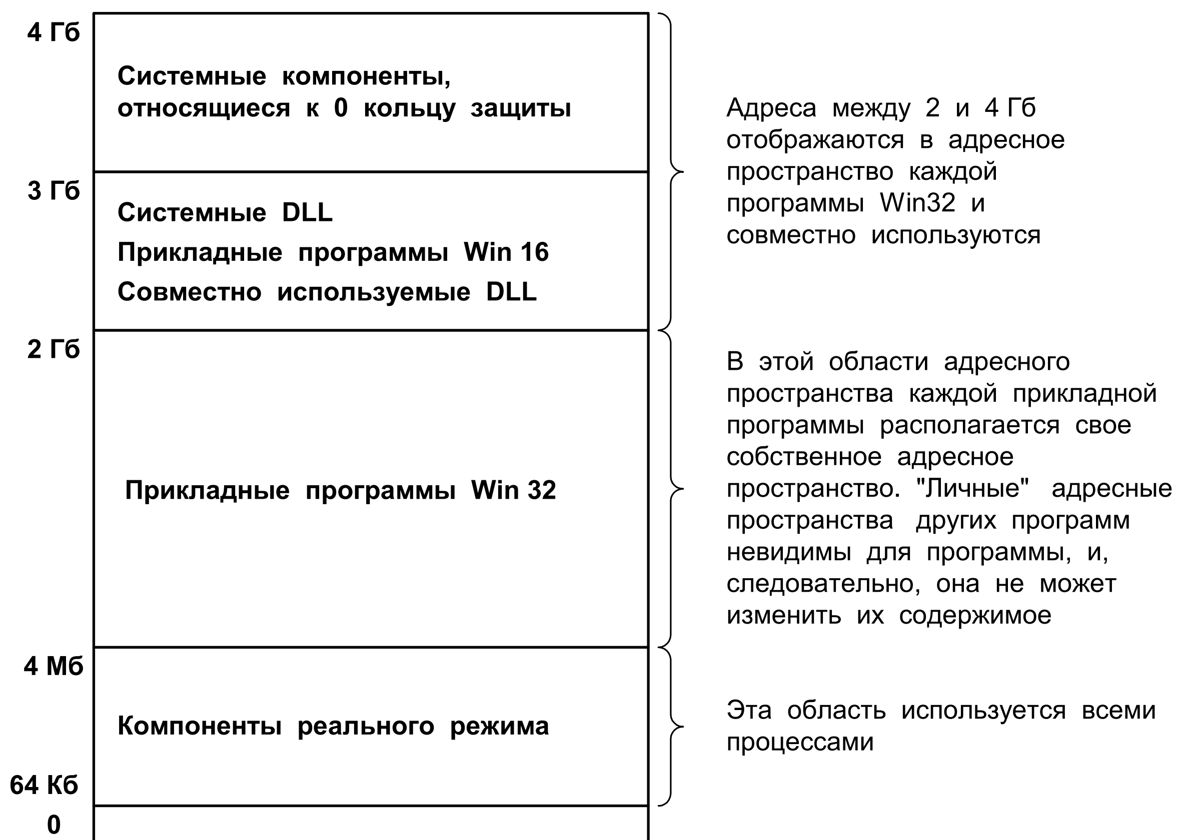



Рис. 4.4. Модель памяти ОС Windows 95/98

Минимально допустимый объем оперативной памяти, начиная с которого ОС Windows 95 может функционировать, равен 4 Мбайт, однако при таком объеме пробуксовка столь велика, что практически работать нельзя. Страничный файл, с помощью которого реализуется механизм виртуальной памяти, по умолчанию располагается в каталоге самой Windows и имеет переменный размер. Система отслеживает его длину, увеличивая или сокращая этот файл при необходимости. Вместе с фрагментацией

файла подкачки это приводит к тому, что быстродействие системы становится меньше, чем если бы файл был фиксированного размера и располагался в смежных кластерах.

### 1.3.3. Распределение оперативной памяти в Microsoft Windows NT

Схема распределения возможного виртуального адресного пространства в системах Windows NT серьезно отличается от модели памяти Windows 95/98. Прежде всего, в отличие от Windows 95/98 в гораздо большей степени используется ряд серьезных аппаратных средств защиты, имеющихся в микропроцессорах, а также применено принципиально другое логическое распределение адресного пространства.

Во-первых, все системные программные модули находятся в своих собственных виртуальных адресных пространствах, и доступ к ним со стороны прикладных программ невозможен. Ядро системы и несколько драйверов работают в нулевом кольце защиты в отдельном адресном пространстве.

Во-вторых, остальные программные модули самой операционной системы, которые выступают как серверные процессы по отношению к прикладным программам (клиентам), функционируют также в своем собственном системном виртуальном адресном пространстве, невидимом для прикладных процессов. Логическое распределение адресных пространств приведено на рис. 4.5.

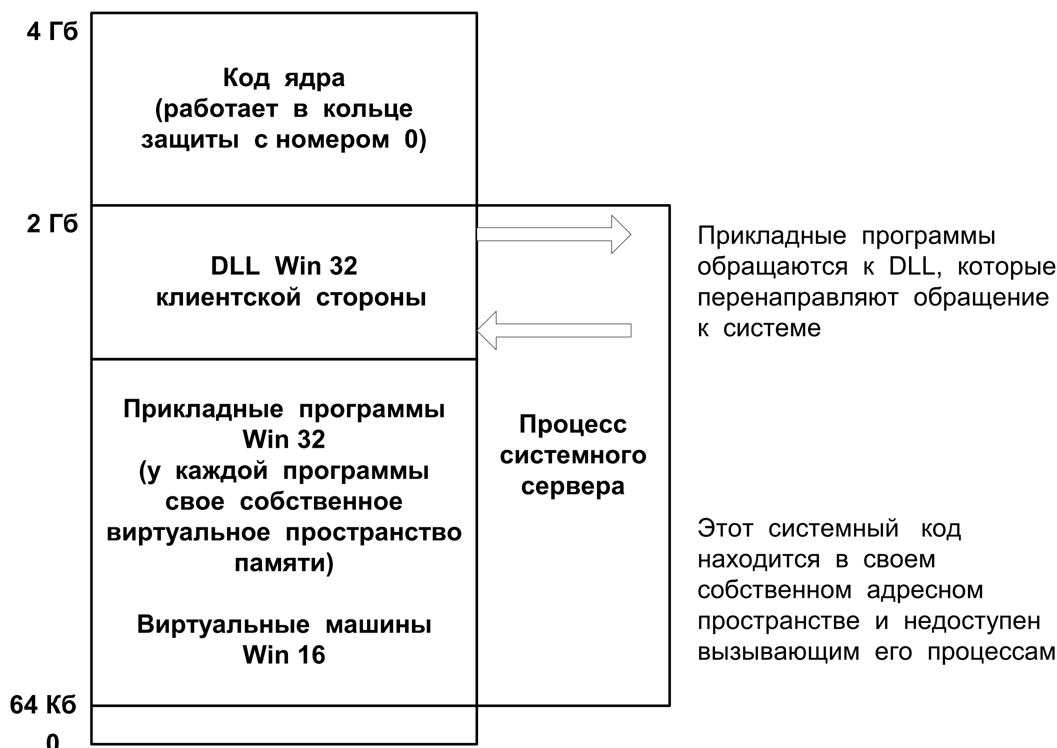



Рис. 4.5. Модель распределения виртуальной памяти в Windows NT

Прикладным программам выделяется 2 Гбайт локального (собственного) линейного (неструктурированного) адресного пространства от границы 64 Кбайт до 2 Гбайт (первые 64 Кбайт полностью недоступны). Прикладные программы изолированы друг от друга, хотя могут общаться через буфер обмена (clipboard), механизмы DDE и OLE.

В верхней части каждой 2-гигабайтной области прикладной программы размещен код системных DLL кольца 3, который выполняет перенаправление вызовов в совершенно изолированное адресное пространство, где содержится уже собственно системный код. Этот системный код, выступающий как сервер-процесс (server process), проверяет значения параметров, исполняет запрошенную функцию и пересылает результаты назад в адресное пространство прикладной программы. Хотя сервер-процесс сам по себе остается процессом прикладного уровня, он полностью защищен от вызывающей его прикладной программы и изолирован от нее.

Между отметками 2 и 4 Гбайт расположены низкоуровневые системные компоненты Windows NT кольца 0, в том числе ядро, планировщик потоков и диспетчер виртуальной памяти. Системные страницы в этой области наделены привилегиями супервизора, которые задаются физическими схемами кольцевой защиты процессора. Это делает низкоуровневый системный код невидимым и недоступным для записи для программ прикладного уровня, но приводит к падению производительности во время переходов между кольцами.

Процессами выделения памяти, ее резервирования, освобождения и подкачки управляет диспетчер виртуальной памяти Windows NT (Windows NT virtual memory manager, VMM). В своей работе этот компонент реализует сложную стратегию учета требований к коду и данным процесса для минимизации доступа к диску, поскольку реализация виртуальной памяти часто приводит к большому количеству дисковых операций.

Вся виртуальная память в Windows NT подразделяется на классы: зарезервированную (reserved), выделенную (committed) и доступную (available).

*Зарезервированная память* представляет собой набор непрерывных адресов, которые диспетчер виртуальной памяти выделяет для процесса, но не учитывает в общей квоте памяти процесса до тех пор, пока она не будет фактически использована. Когда процессу требуется выполнить запись в память, ему выделяется нужный объем из зарезервированной памяти. Если процессу потребуется больший объем памяти, то дополнительная память может быть одновременно зарезервирована и использована, если в системе имеется доступная память

*Память выделена,* если диспетчер VMM резервирует для нее место в файле Pagefile.sys на тот случай, когда потребуется выгрузить содержимое памяти на диск. Объем выделенной памяти процесса характеризует фактически потребляемый им объем памяти. Выделенная память ограничивается размером файла подкачки. Предельный объем выделенной памяти в

системе (commit limit) определяется тем, какой объем памяти можно выделить процессам без увеличения размеров файла подкачки.

Вся память, которая не является ни выделенной, ни зарезервированной, является *доступной.* К доступной относится свободная память, обнуленная память, а также память, находящаяся в *списке ожидания* (standby list), которая была удалена из рабочего набора процесса, но может быть затребована вновь.

## 2. Механизмы реализации виртуальной памяти

Методы распределения памяти, при которых задаче уже может не предоставляться сплошная (непрерывная) область памяти, называют разрывными. Идея выделять память задаче не одной сплошной областью, а фрагментами требует для своей реализации соответствующей аппаратной поддержки – нужно иметь относительную адресацию. Если указывать адрес начала текущего фрагмента программы и величину смещения относительно этого начального адреса, то можно указать необходимую нам переменную или команду. Таким образом, виртуальный адрес можно представить состоящим из двух полей. Первое поле будет указывать часть программы (с которой сейчас осуществляется работа) для определения местоположения этой части в памяти, а второе поле виртуального адреса позволит найти нужную нам ячейку относительно найденного адреса. Программист может либо самостоятельно разбивать программу на фрагменты, либо автоматизировать эту задачу и возложить ее на систему программирования.

### 2.1. Сегментный способ организации виртуальной памяти

Первым среди разрывных методов распределения памяти был сегментный. Для этого метода программу необходимо разбивать на части и уже каждой такой части выделять физическую память. Естественным способом разбиения программы на части является разбиение ее на логические элементы – так называемые сегменты. В принципе каждый программный модуль (или их совокупность, если мы того пожелаем) может быть воспринят как отдельный сегмент, и вся программа тогда будет представлять собой множество сегментов. Каждый сегмент размещается в памяти как до определенной степени самостоятельная единица. Логически обращение к элементам программы в этом случае будет представляться как указание имени сегмента и смещения относительно начала этого сегмента. Физически имя (или порядковый номер) сегмента будет соответствовать некоторому адресу, с которого этот сегмент начинается при его размещении в памяти, и смещение должно прибавляться к этому базовому адресу.

Преобразование имени сегмента в его порядковый номер осуществит система программирования, а операционная система будет размещать сег-

менты в память и для каждого сегмента получит информацию о его начале. Таким образом, виртуальный адрес для этого способа будет состоять из двух полей – номер сегмента и смещение относительно начала сегмента. Соответствующая иллюстрация приведена на рис. 4.6. На этом рисунке изображен случай обращения к ячейке, виртуальный адрес которой равен сегменту с номером 11 и смещением от начала этого сегмента, равным 612. Как видно, операционная система разместила данный сегмент в памяти, начиная с ячейки с номером 19700.

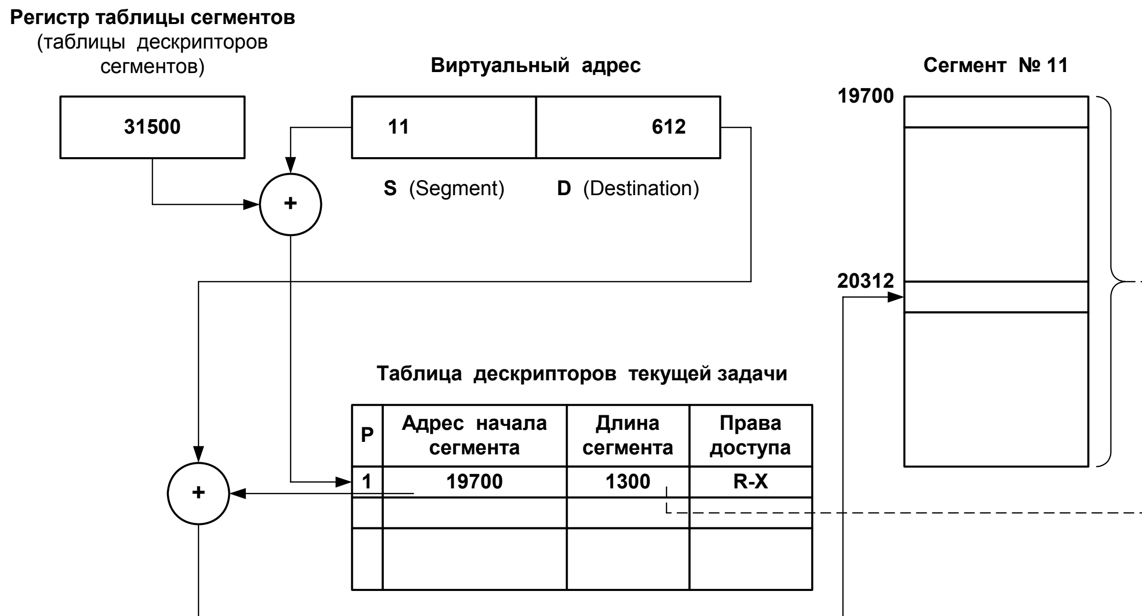



Рис. 4.6. Сегментный способ организации виртуальной памяти

Каждый сегмент, размещаемый в памяти, имеет соответствующую информационную структуру, часто называемую *дескриптором сегмента.* Именно операционная система строит для каждого исполняемого процесса соответствующую таблицу дескрипторов сегментов и при размещении каждого из сегментов в оперативной или внешней памяти в дескрипторе отмечает его текущее местоположение. Если сегмент задачи в данный момент находится в оперативной памяти, то об этом делается пометка в дескрипторе. Как правило, для этого используется *"бит присутствия" P* (present). В этом случае в поле "адрес" диспетчер памяти записывает адрес физической памяти, с которого сегмент начинается, а в поле "длина сегмента" (limit) указывается количество адресуемых ячеек памяти. Это поле используется не только для того, чтобы размещать сегменты без наложения один на другой, но и для того, чтобы проконтролировать, не обращается ли код исполняющейся задачи за пределы текущего сегмента. В случае превышения длины сегмента вследствие ошибок программирования можно говорить о нарушении адресации и с помощью введения специальных аппаратных средств генерировать сигналы прерывания, которые позволят фиксировать (обнаруживать) такого рода ошибки.

Если бит *present* в дескрипторе указывает, что сейчас этот сегмент находится не в оперативной, а во внешней памяти (например, на винчестере), то названные поля адреса и длины используются для указания адреса сегмента в координатах внешней памяти. Помимо информации о местоположении сегмента, в дескрипторе сегмента, как правило, содержатся данные о его типе (сегмент кода или сегмент данных), правах доступа к этому сегменту (можно или нельзя его модифицировать, предоставлять другой задаче), отметка об обращениях к данному сегменту (информация о том, как часто или как давно/недавно этот сегмент используется или не используется, на основании которой можно принять решение о том, чтобы предоставить место, занимаемое текущим сегментом, другому сегменту).

При передаче управления следующей задаче ОС должна занести в соответствующий регистр адрес таблицы дескрипторов сегментов этой задачи. Сама *таблица дескрипторов сегментов,* в свою очередь, также представляет собой сегмент данных, который обрабатывается диспетчером памяти операционной системы.

При таком подходе появляется возможность размещать в оперативной памяти не все сегменты задачи, а только те, с которыми в настоящий момент происходит работа. С одной стороны, становится возможным, чтобы общий объем виртуального адресного пространства задачи превосходил объем физической памяти компьютера, на котором эта задача будет выполняться. С другой стороны, даже если потребности в памяти не превосходят имеющуюся физическую память, появляется возможность размещать в памяти как можно больше задач. А увеличение *коэффициента мультипрограммирования* μ позволяет увеличить загрузку системы и более эффективно использовать ресурсы вычислительной системы. Очевидно, однако, что увеличивать количество задач можно только до определенного предела, ибо если в памяти не будет хватать места для часто используемых сегментов, то производительность системы резко упадет. Ведь сегмент, который сейчас находится вне оперативной памяти, для участия в вычислениях должен быть перемещен в оперативную память. При этом если в памяти есть свободное пространство, то необходимо всего лишь найти его во внешней памяти и загрузить в оперативную память. А если свободного места сейчас нет, то необходимо будет принять решение – на место какого из ныне присутствующих сегментов будет загружаться требуемый.

Итак, если требуемого сегмента в оперативной памяти нет, то возникает прерывание и управление передается через диспетчер памяти программе загрузки сегмента. Пока происходит поиск сегмента во внешней памяти и загрузка его в оперативную, диспетчер памяти определяет подходящее для сегмента место. Возможно, что свободного места нет, и тогда принимается решение о выгрузке какого-нибудь сегмента и его перемещение во внешнюю память. Если при этом еще остается время, то процессор передается другой готовой к выполнению задаче. После загрузки необхо-

димого сегмента процессор вновь передается задаче, вызвавшей прерывание из-за отсутствия сегмента. Всякий раз при считывании сегмента в оперативную память в таблице дескрипторов сегментов необходимо установить адрес начала сегмента и признак присутствия сегмента.

При поиске свободного места используется одна из изученных ранее дисциплин работы диспетчера памяти (применяются правила "первого подходящего" и "самого неподходящего" фрагментов).

В идеальном случае размер сегмента должен быть достаточно малым, чтобы его можно было разместить в случайно освобождающихся фрагментах оперативной памяти, но достаточно большим, чтобы содержать логически законченную часть программы с тем, чтобы минимизировать межсегментные обращения.

Для решения проблемы замещения (определения того сегмента, который должен быть либо перемещен во внешнюю память, либо просто замещен новым) используются следующие дисциплины, называемые "дисциплинами замещения":

правило *FIFO* (first in – first out, что означает: "первый пришедший первым и выбывает");

правило *LRU* (least recently used, что означает "последний из недавно использованных" или, иначе говоря, "дольше всего неиспользуемый");

правило *LFU* (least frequently used, что означает: "используемый реже всех остальных");

случайный (random) выбор сегмента.

Первая и последняя дисциплины являются самыми простыми в реализации, но они не учитывают, насколько часто используется тот или иной сегмент и, следовательно, диспетчер памяти может выгрузить или расформировать тот сегмент, к которому в самом ближайшем будущем будет обращение. Безусловно, достоверной информации о том, какой из сегментов потребуется в ближайшем будущем, в общем случае иметь нельзя, но вероятность ошибки для этих дисциплин многократно выше, чем у второй и третьей дисциплины, которые учитывают информацию об использовании сегментов.

Алгоритм FIFO ассоциирует с каждым сегментом время, когда он был помещен в память. Для замещения выбирается наиболее старый сегмент. Учет времени необязателен, когда все сегменты в памяти связаны в FIFO-очередь и каждый помещаемый в память сегмент добавляется в хвост этой очереди. Алгоритм учитывает только время нахождения сегмента в памяти, но не учитывает фактическое использование сегментов. Например, первые загруженные сегменты программы могут содержать переменные, используемые на протяжении работы всей программы. Это приводит к немедленному возвращению к только что замещенному сегменту.

Для реализации дисциплин LRU и LFU необходимо, чтобы процессор имел дополнительные аппаратные средства. Минимальные требования – достаточно, чтобы при обращении к дескриптору сегмента для получения

физического адреса, с которого сегмент начинает располагаться в памяти, соответствующий бит обращения менял свое значение (скажем, с нулевого, которое установила ОС, в единичное). Тогда диспетчер памяти может время от времени просматривать таблицы дескрипторов исполняющихся задач и собирать для соответствующей обработки статистическую информацию об обращениях к сегментам. В результате можно составить список, упорядоченный либо по длительности не использования (для дисциплины LRU), либо по частоте использования (для дисциплины LFU).

Важнейшей проблемой, которая возникает при организации мультипрограммного режима, является защита памяти. Для того чтобы выполняющиеся приложения не смогли испортить саму ОС и другие вычислительные процессы, необходимо, чтобы доступ к таблицам сегментов с целью их модификации был обеспечен только для кода самой ОС. Для этого код ОС должен выполняться в некотором привилегированном режиме, из которого можно осуществлять манипуляции с дескрипторами сегментов, тогда как выход за пределы сегмента в обычной прикладной программе должен вызывать прерывание по защите памяти. Каждая прикладная задача должна иметь возможность обращаться только к своим собственным сегментам.

При использовании сегментного способа организации виртуальной памяти появляется несколько интересных возможностей. Во-первых, появляется возможность при загрузке программы на исполнение размещать ее в памяти не целиком, а “по мере необходимости”. Действительно, поскольку в подавляющем большинстве случаев алгоритм, по которому работает код программы, является разветвленным, а не линейным, то в зависимости от исходных данных некоторые части программы, расположенные в самостоятельных сегментах, могут быть и не задействованы; значит, их можно и не загружать в оперативную память.

Во-вторых, некоторые программные модули могут быть разделяемыми. Эти программные модули являются сегментами, и в этом случае относительно легко организовать доступ к таким сегментам. Сегмент с разделяемым кодом располагается в памяти в единственном экземпляре, а в нескольких таблицах дескрипторов сегментов исполняющихся задач будут находиться указатели на такие разделяемые сегменты.

Однако у сегментного способа распределения памяти есть и недостатки. Прежде всего, из рис. 4.5 видно, что для получения доступа к искомой ячейке памяти необходимо потратить намного больше времени. Мы должны сначала найти и прочитать дескриптор сегмента, а уже потом, используя данные из него о местонахождении нужного нам сегмента, можем вычислить и конечный физический адрес. Для того чтобы уменьшить эти потери, используется кэширование – то есть те дескрипторы, с которыми мы имеем дело в данный момент, могут быть размещены в сверхоперативной памяти (специальных регистрах, размещаемых в процессоре).

Несмотря на то, что этот способ распределения памяти приводит к существенно меньшей фрагментации памяти, нежели способы с неразрывным распределением, фрагментация остается. Кроме этого, мы имеем большие потери памяти и процессорного времени на размещение и обработку дескрипторных таблиц. Ведь на каждую задачу необходимо иметь свою таблицу дескрипторов сегментов. А при определении физических адресов необходимо выполнять операции сложения.

Поэтому следующим способом разрывного размещения задач в памяти стал способ, при котором все фрагменты задачи одинакового размера и длины, кратной степени двойки, чтобы операции сложения можно было заменить операциями конкатенации (слияния). Это – страничный способ организации виртуальной памяти.

Примером использования сегментного способа организации виртуальной памяти является операционная система для ПК OS/2 первого поколения, которая была создана для процессора i80286. В этой ОС в полной мере использованы аппаратные средства микропроцессора, который специально проектировался для поддержки сегментного способа распределения памяти.

## 2.2. Страничный способ организации виртуальной памяти

При страничном способе организации виртуальной памяти все фрагменты программы, на которые она разбивается (за исключением последней ее части), получаются одинаковыми. Одинаковыми полагаются и единицы памяти, которые мы предоставляем для размещения фрагментов программы. Эти одинаковые части называют страницами и говорят, что память разбивается на физические страницы, а программа – на виртуальные страницы. Часть виртуальных страниц задачи размещается в оперативной памяти, а часть – во внешней. Обычно место во внешней памяти, в качестве которой в абсолютном большинстве случаев выступают накопители на магнитных дисках (поскольку они относятся к быстродействующим устройствам с прямым доступом), называют файлом подкачки или *страничным файлом* (paging file). Иногда этот файл называют *swap-файлом,* тем самым подчеркивая, что записи этого файла – страницы – замещают друг друга в оперативной памяти. В некоторых ОС выгруженные страницы располагаются не в файле, а в специальном разделе дискового пространства. В UNIX-системах для этих целей выделяется специальный раздел, но кроме него могут быть использованы и файлы, выполняющие те же функции, если объема раздела недостаточно.

Разбиение всей оперативной памяти на страницы одинаковой величины (величина каждой страницы выбирается кратной степени двойки) приводит к тому, что вместо одномерного адресного пространства памяти можно говорить о двумерном. Первая координата адресного пространства –

это номер страницы, а вторая координата – номер ячейки внутри выбранной страницы (его называют индексом). Таким образом, физический адрес определяется парой ($P_p$, $i$), а виртуальный адрес – парой ($P_v$, $i$), где $P_v$ – это номер виртуальной страницы, $P_p$ – это номер физической страницы и $i$ – это индекс ячейки внутри страницы. Количество битов, отводимое под индекс, определяет размер страницы, а количество битов, отводимое под номер виртуальной страницы, – объем возможной виртуальной памяти, которой может пользоваться программа. Отображение, осуществляемое системой во время исполнения, сводится к отображению $P_v$ в $P_p$ и приписывании к полученному значению битов адреса, задаваемых величиной $i$. При этом нет необходимости ограничивать число виртуальных страниц числом физических, то есть не поместившиеся страницы можно размещать во внешней памяти, которая в данном случае служит расширением оперативной.

Для отображения виртуального адресного пространства задачи на физическую память, как и в случае с сегментным способом организации, для каждой задачи необходимо иметь *таблицу страниц* для трансляции адресных пространств. Для описания каждой страницы диспетчер памяти ОС заводит соответствующий дескриптор, который отличается от дескриптора сегмента прежде всего тем, что в нем нет необходимости иметь поле длины – ведь все страницы имеют одинаковый размер. По номеру виртуальной страницы в таблице дескрипторов страниц текущей задачи находится соответствующий элемент (дескриптор). Если бит присутствия имеет единичное значение, значит, данная страница сейчас размещена в оперативной, а не во внешней памяти и мы в дескрипторе имеем номер физической страницы, отведенной под данную виртуальную. Если же бит присутствия равен нулю, то в дескрипторе мы будем иметь адрес виртуальной страницы, расположенной сейчас во внешней памяти. Таким образом и осуществляется трансляция виртуального адресного пространства на физическую память. Этот механизм трансляции проиллюстрирован на рис. 4.6.

Защита страничной памяти, как и в случае с сегментным механизмом, основана на контроле уровня доступа к каждой странице. Как правило, возможны следующие уровни доступа: только чтение; чтение и запись; только выполнение. В этом случае каждая страница снабжается соответствующим кодом уровня доступа. При трансформации логического адреса в физический сравнивается значение кода разрешенного уровня доступа с фактически требуемым. При их несовпадении работа программы прерывается.

При обращении к виртуальной странице, не оказавшейся в данный момент в оперативной памяти, возникает прерывание и управление передается диспетчеру памяти, который должен найти свободное место. Обычно предоставляется первая же свободная страница. Если свободной физической страницы нет, то диспетчер памяти по одной из вышеупомянутых

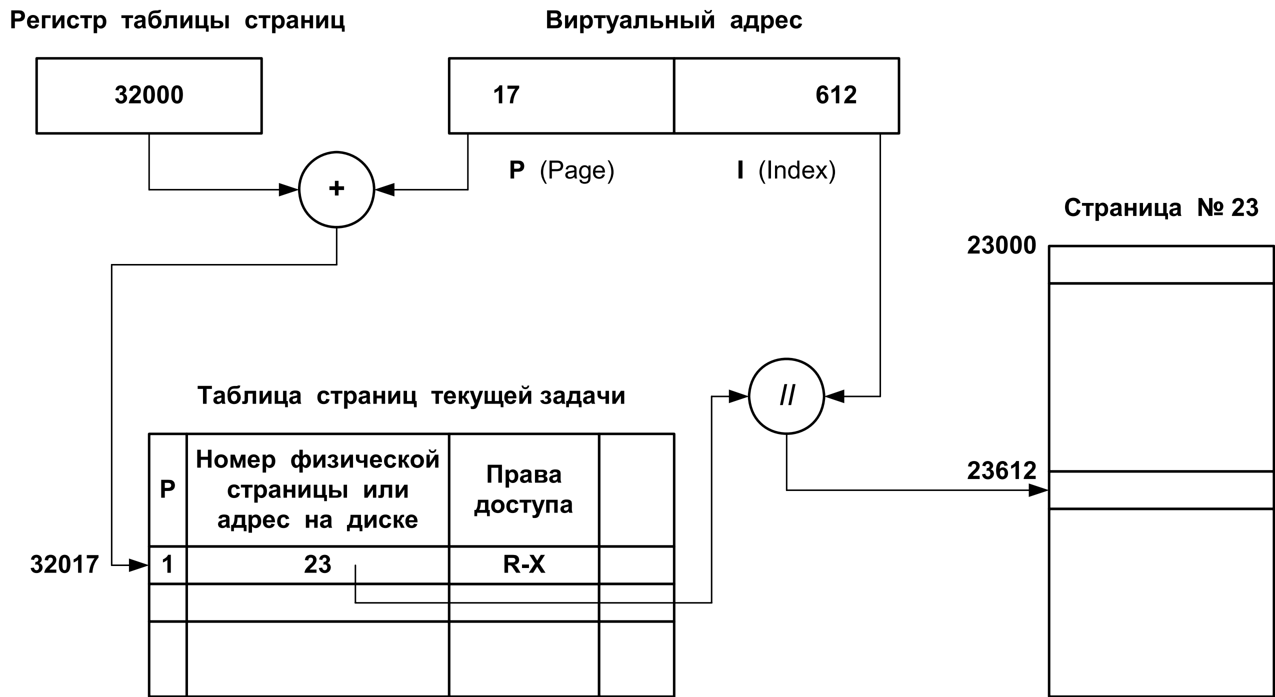



Рис. 4.6. Страничный способ организации виртуальной памяти

дисциплин замещения (LRU, LFU, FIFO, random) определит страницу, подлежащую расформированию или сохранению во внешней памяти. На ее место он разместит ту новую виртуальную страницу, к которой было обращение из задачи, но ее не оказалось в оперативной памяти.

Для использования дисциплин LRU и LFU в процессоре должны быть соответствующие аппаратные средства. В дескрипторе страницы размещается бит обращения (подразумевается, что на рис. 4.6 этот бит расположен в последнем поле), и этот бит становится единичным при обращении к дескриптору.

Если объем физической памяти небольшой и даже часто требуемые страницы не удается разместить в оперативной памяти, то возникает так называемая “пробуксовка”. Другими словами, *пробуксовка* – это ситуация, при которой загрузка нужной нам страницы вызывает перемещение во внешнюю память той страницы, с которой мы тоже активно работаем. Очевидно, что это очень плохое явление. Чтобы его не допускать, желательно увеличить объем оперативной памяти (сейчас это стало самым простым решением), уменьшить количество параллельно выполняемых задач, либо попробовать использовать более эффективные дисциплины замещения.

В абсолютном большинстве современных ОС используется дисциплина замещения страниц LRU как самая эффективная. Так, именно эта дисциплина используется в OS/2 и Linux. Однако в такой ОС, как Windows NT, разработчики, желая сделать систему максимально независимой от аппаратных возможностей процессора, пошли на отказ от этой дисциплины и

применили правило FIFO. А для того, чтобы хоть как-нибудь сгладить ее неэффективность, была введена "буферизация" тех страниц, которые должны быть записаны в файл подкачки на диск или просто расформированы. Принцип буферирования прост. Прежде чем замещаемая страница действительно будет перемещена во внешнюю память или просто расформирована, она помечается как кандидат на выгрузку. Если в следующий раз произойдет обращение к странице, находящейся в таком "буфере", то страница никуда не выгружается и уходит в конец списка FIFO. В противном случае страница действительно выгружается, а на ее место в "буфере" попадает следующий "кандидат". Величина такого "буфера" не может быть большой, поэтому эффективность страничной реализации памяти в Windows NT намного ниже, чем у вышеназванных ОС, и явление пробуксовки начинается даже при существенно большем объеме оперативной памяти.

В ряде ОС с пакетным режимом работы для борьбы с пробуксовкой используется метод "рабочего множества". *Рабочее множество* – это множество "активных" страниц задачи за некоторый интервал, то есть тех страниц, к которым было обращение за этот интервал времени. Реально количество активных страниц задачи (за интервал Т) все время изменяется, и это естественно, но тем не менее для каждой задачи можно определить среднее количество ее активных страниц. Это среднее число активных страниц и есть рабочее множество задачи. Наблюдения за исполнением множества различных программ показали, что даже если Т равно времени выполнения всей работы, то размер рабочего множества часто существенно меньше, чем общее число страниц программы. Таким образом, если ОС может определить рабочие множества исполняющихся задач, то для предотвращения пробуксовки достаточно планировать на выполнение только такое количество задач, чтобы сумма их рабочих множеств не превышала возможности системы.

Как и в случае с сегментным способом организации виртуальной памяти, страничный механизм приводит к тому, что без специальных аппаратных средств он будет существенно замедлять работу вычислительной системы. Поэтому обычно используется кэширование страничных дескрипторов. Наиболее эффективным способом кэширования является использование ассоциативного кэша. Именно такой ассоциативный кэш и создан в 32-разрядных микропроцессорах i80x86. Начиная с i80386, который поддерживает страничный способ распределения памяти, в этих микропроцессорах имеется кэш на 32 страничных дескриптора. Поскольку размер страницы в этих микропроцессорах равен 4 Кбайт, возможно быстрое обращение к 128 Кбайт памяти.

Итак, основным достоинством страничного способа распределения памяти является минимально возможная фрагментация. Поскольку на каждую задачу может приходиться по одной незаполненной странице, то становится очевидно, что память можно использовать достаточно эффектив-

но; этот метод организации виртуальной памяти был бы одним из самых лучших, если бы не два следующих обстоятельства.

Первое – это то, что страничная трансляция виртуальной памяти требует существенных накладных расходов. В самом деле, таблицы страниц нужно тоже размещать в памяти. Кроме этого, эти таблицы нужно обрабатывать; именно с ними работает диспетчер памяти.

Второй существенный недостаток страничной адресации заключается в том, что программы разбиваются на страницы случайно, без учета логических взаимосвязей, имеющихся в коде. Это приводит к тому, что межстраничные переходы, как правило, осуществляются чаще, нежели межсегментные, и к тому, что становится трудно организовать разделение программных модулей между выполняющимися процессами.

Для того чтобы избежать второго недостатка, постаравшись сохранить достоинства страничного способа распределения памяти, был предложен еще один способ – сегментно-страничный. Правда, за счет дальнейшего увеличения накладных расходов на его реализацию.

### 2.3. Сегментно-страничный способ организации виртуальной памяти

Как и в сегментном способе распределения памяти, программа разбивается на логически законченные части – сегменты – и виртуальный адрес содержит указание на номер соответствующего сегмента. Вторая составляющая виртуального адреса – смещение относительно начала сегмента – в свою очередь, может состоять из двух полей: виртуальной страницы и индекса. Другими словами, получается, что виртуальный адрес теперь состоит из трех компонентов: сегмент, страница, индекс. Получение физического адреса и извлечение из памяти необходимого элемента для этого способа представлено на рис. 4.7.

Из рисунка сразу видно, что этот способ организации виртуальной памяти вносит еще большую задержку доступа к памяти. Необходимо сначала вычислить адрес дескриптора сегмента и прочитать его, затем вычислить адрес элемента таблицы страниц этого сегмента и извлечь из памяти необходимый элемент, и уже только после этого можно к номеру физической страницы приписать номер ячейки в странице (индекс). Задержка доступа к искомой ячейке получается по крайней мере в три раза больше, чем при простой прямой адресации. Чтобы избежать этой неприятности, вводится кэширование, причем кэш, как правило, строится по ассоциативному принципу. Другими словами, просмотры двух таблиц в памяти могут быть заменены одним обращением к ассоциативной памяти.

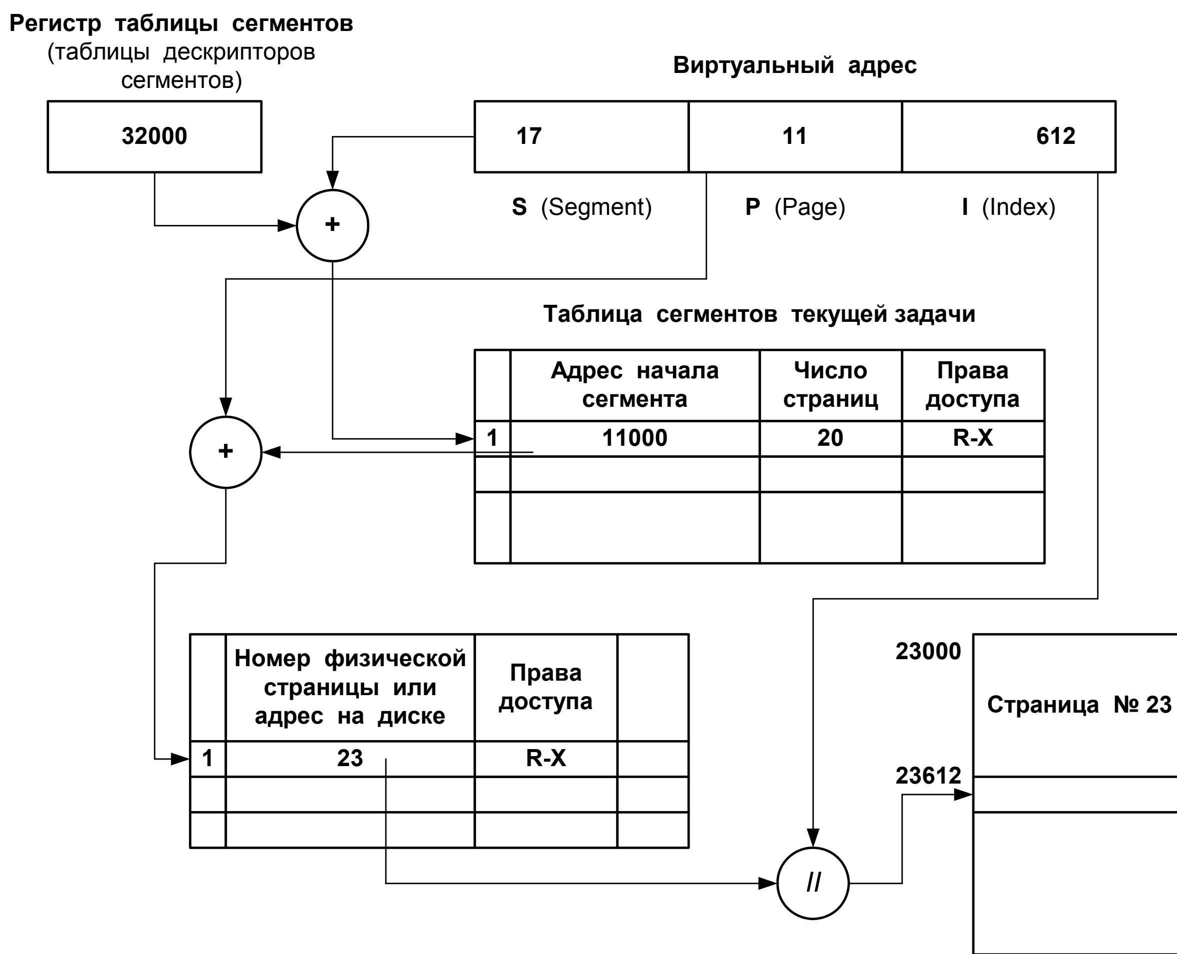



Рис. 4.7. Сегментно-страничный способ организации виртуальной памяти

Принцип действия ассоциативного запоминающего устройства предполагает, что каждой ячейке памяти такого устройства ставится в соответствие ячейка, в которой записывается некий ключ (признак, адрес), позволяющий однозначно идентифицировать содержимое ячейки памяти. Сопутствующую ячейку с информацией, позволяющей идентифицировать основные данные, обычно называют *полем тега.* Просмотр полей тега всех ячеек ассоциативного устройства памяти осуществляется одновременно, то есть в каждой ячейке тега есть необходимая логика, позволяющая посредством побитовой конъюнкции найти данные по их признаку за одно обращение к памяти (если они там, конечно, присутствуют). Часто поле тегов называют аргументом, а поле с данными – функцией. В качестве аргумента при доступе к ассоциативной памяти выступают номер сегмента и номер виртуальной страницы, а в качестве функции от этих аргументов получаем номер физической страницы. Остается приписать номер ячейки в странице к полученному номеру, и мы получаем искомую команду или операнд.

Достоинства сегментно-страничного способа следующие. Разбиение программы на сегменты позволяет размещать сегменты в памяти целиком. Сегменты разбиты на страницы, все страницы сегмента загружаются в па-

мять. Это позволяет уменьшить обращения к отсутствующим страницам, поскольку вероятность выхода за пределы сегмента меньше вероятности выхода за пределы страницы. Страницы исполняемого сегмента находятся в памяти, но при этом они могут находиться не рядом друг с другом, а “россыпью”, поскольку диспетчер памяти манипулирует страницами. Наличие сегментов облегчает реализацию разделения программных модулей между параллельными процессами. Возможна и динамическая компоновка задачи. А выделение памяти страницами позволяет минимизировать фрагментацию.

Однако, поскольку этот способ распределения памяти требует очень значительных затрат вычислительных ресурсов и его не так просто реализовать, используется он редко, причем в дорогих, мощных вычислительных системах. Возможность реализовать сегментно-страничное распределение памяти заложена и в семейство микропроцессоров i80x86, однако вследствие слабой аппаратной поддержки, трудностей при создании систем программирования и операционной системы, практически он не используется в ПК.

## 3. Управление вводом-выводом

Необходимость обеспечить программам возможность осуществлять обмен данными с внешними устройствами и при этом не включать в каждую двоичную программу соответствующий двоичный код, осуществляющий собственно управление устройствами ввода/вывода, привела разработчиков к созданию системного программного обеспечения и, в частности, самих операционных систем. Программирование задач управления вводом/выводом является наиболее сложным и трудоемким, требующим очень высокой квалификации. Поэтому код, позволяющий осуществлять операции ввода/вывода, стали оформлять в виде системных библиотечных процедур; потом его стали включать не в системы программирования, а в операционную систему с тем, чтобы в каждую отдельно взятую программу его не вставлять, а только позволить обращаться к такому коду. Системы программирования стали генерировать обращения к этому системному коду ввода/вывода и осуществлять только подготовку к собственно операциям ввода/вывода, то есть автоматизировать преобразование данных к соответствующему формату, понятному устройствам, избавляя прикладных программистов от этой сложной и трудоемкой работы. Другими словами, системы программирования вставляют в машинный код необходимые библиотечные подпрограммы ввода/вывода и обращения к тем системным программным модулям, которые, собственно, и управляют операциями обмена между оперативной памятью и внешними устройствами. Таким образом, управление вводом/выводом – это одна из основных функций любой ОС.

С одной стороны, в организации ввода/вывода в различных ОС много общего. С другой стороны, реализация ввода/вывода в ОС так сильно отличается от системы к системе, что очень нелегко выделить и описать именно основные принципы реализации этих функций. Проблема усугубляется еще тем, что в большинстве ныне используемых систем эти моменты вообще, как правило, подробно не описаны, и исключение по этому вопросу касается только системы Linux, для которой имеются комментированные исходные тексты. Детально описываются функции API, реализующие ввод/вывод. Поэтому мы рассмотрим только основные идеи и концепции.

## 3.1. Основные понятия и концепция организации ввода-вывода в ОС

Ввод/вывод считается одной из самых сложных областей проектирования операционных систем, в которой сложно применить общий подход из-за изобилия частных методов. Сложность возникает из-за огромного числа устройств ввода/вывода разнообразной природы, которые должна поддерживать ОС. При этом перед создателями ОС встает очень непростая задача – не только обеспечить эффективное управление устройствами ввода/вывода, но и создать удобный и эффективный виртуальный интерфейс устройств ввода/вывода, позволяющий прикладным программистам просто считывать или сохранять данные, не обращая внимание на специфику устройств и проблемы распределения устройств между выполняющимися задачами. Система ввода/вывода, способная объединить в одной модели широкий набор устройств, должна быть универсальной. Она должна учитывать потребности существующих устройств, от простой мыши до клавиатур, принтеров, графических дисплеев, дисковых накопителей, компакт-дисков и даже сетей. С другой стороны, необходимо обеспечить доступ к устройствам ввода/вывода для множества параллельно выполняющихся задач, причем так, чтобы они как можно меньше мешали друг другу.

Поэтому самым главным является следующий принцип: любые операции по управлению вводом/выводом объявляются привилегированными и могут выполняться только кодом самой ОС. Для обеспечения этого принципа в большинстве процессоров даже вводятся *режимы пользователя* и *супервизора.* Как правило, в *режиме супервизора* выполнение команд ввода/вывода разрешено, а в пользовательском режиме – запрещено. Использование команд ввода/вывода в пользовательском режиме вызывает *исключение* и управление через механизм прерываний передается коду ОС. Хотя возможны и более сложные системы, в которых в ряде случаев пользовательским программам разрешено непосредственное выполнение команд ввода/вывода.

Одним из основных видов ресурсов являются устройства ввода/вывода и соответствующее программное обеспечение, с помощью которого осуществляется управление обменом данными между внешними устройствами и оперативной памятью. Помимо разделяемых устройств ввода/вывода (эти устройства допускают разделение посредством механизма доступа) существуют неразделяемые устройства. Примерами разделяемого устройства могут служить накопитель на магнитных дисках, устройство для чтения компакт-дисков. Это устройства с прямым доступом. Примеры неразделяемых устройств – принтер, накопитель на магнитных лентах. Это устройства с последовательным доступом. Операционные системы должны управлять и теми и другими устройствами, предоставляя возможность параллельно выполняющимся задачам использовать различные устройства ввода/вывода.

Можно назвать *три основные причины*, по которым нельзя разрешать каждой отдельной пользовательской программе обращаться к внешним устройствам непосредственно:

1. Необходимость разрешать возможные конфликты доступа к устройствам ввода/вывода. Например, две параллельно выполняющиеся программы пытаются вывести на печать результаты своей работы. Если не предусмотреть внешнее управление устройством печати, то в результате мы можем получить абсолютно нечитаемый текст, так как каждая программа будет время от времени выводить свои данные, которые будут перемежаться данными другой программы. Другой пример: ситуация, когда одной программе необходимо прочитать данные с некоторого сектора магнитного диска, а другой – записать результаты в другой сектор того же накопителя. Если операции ввода/вывода не будут отслеживаться каким-то третьим (внешним) процессом-арбитром, то после позиционирования магнитной головки для первого запроса может тут же появиться команда позиционирования головки для второй задачи, и обе операции ввода/вывода не смогут быть выполнены корректно.

2. Желание увеличить эффективность использования этих ресурсов. Например, у накопителя на магнитных дисках время подвода головки чтения/записи к необходимой дорожке и обращение к определенному сектору может значительно (до тысячи раз) превышать время пересылки данных. В результате, если задачи по очереди обращаются к цилиндрам, далеко отстоящим друг от друга, то полезная работа, выполняемая накопителем, может быть существенно снижена.

3. Ошибки в программах ввода/вывода могут привести к краху всех вычислительных процессов, ибо часть операций ввода/вывода осуществляется для самой операционной системы. В ряде ОС системный ввод/вывод имеет существенно более высокие привилегии, чем ввод/вывод задач пользователя. Поэтому системный код, управляющий операциями ввода/вывода, очень тщательно отлаживается и оптимизиру-

ется для повышения надежности вычислений и эффективности использования оборудования.

Итак, управление вводом/выводом осуществляется операционной системой, компонентом, который чаще всего называют *супервизором ввода/вывода.* В перечень основных задач, возлагаемых на супервизор, входят следующие:

1. Супервизор ввода/вывода получает запросы на ввод/вывод от прикладных задач и от программных модулей самой операционной системы. Эти запросы проверяются на корректность, и если запрос выполнен по спецификациям и не содержит ошибок, он обрабатывается дальше, в противном случае пользователю (задаче) выдается соответствующее диагностическое сообщение о недействительности (некорректности) запроса.

2. Супервизор ввода/вывода вызывает соответствующие распределители каналов и контроллеров, планирует ввод/вывод (определяет очередность предоставления устройств ввода/вывода задачам, затребовавшим их). Запрос на ввод/ вывод либо тут же выполняется, либо ставится в очередь на выполнение.

3. Супервизор ввода/вывода инициирует операции ввода/вывода (передает управление соответствующим драйверам) и в случае управления вводом/выводом с использованием прерываний предоставляет процессор диспетчеру задач с тем, чтобы передать его первой задаче, стоящей в очереди на выполнение.

4. При получении сигналов прерываний от устройств ввода/вывода супервизор идентифицирует их и передает управление соответствующей программе обработки прерывания (как правило, на секцию продолжения драйвера).

5. Супервизор ввода/вывода осуществляет передачу сообщений об ошибках, если таковые происходят в процессе управления операциями ввода/вывода.

6. Супервизор ввода/вывода посылает сообщения о завершении операции ввода/вывода запросившему эту операцию процессу и снимает его с состояния ожидания ввода/вывода, если процесс ожидал завершения операции.

В случае если устройство ввода/вывода является *инициативным*, управление со стороны супервизора ввода/вывода будет заключаться в активизации соответствующего вычислительного процесса (перевод его в состояние готовности к выполнению).

Таким образом, прикладные программы (а в общем случае – все обрабатывающие программы) не могут непосредственно связываться с устройствами ввода/ вывода независимо от использования устройств (монопольно или совместно). Установив соответствующие значения параметров в *запросе на ввод/вывод,* определяющих требуемую операцию и количество потребляемых ресурсов, они могут передать управление супервизору вво-

да/вывода, который и запускает необходимые логические и физические операции.

Упомянутый выше запрос на ввод/вывод должен удовлетворять требованиям API той операционной системы, в среде которой выполняется приложение. Параметры, указываемые в запросах на ввод/вывод, передаются не только в вызывающих последовательностях, создаваемых по спецификациям API, но и как данные, хранящиеся в соответствующих системных таблицах. Все параметры, которые будут стоять в вызывающей последовательности, поставляются компилятором и отражают требования программиста и постоянные сведения об операционной системе и архитектуре компьютера в целом. Переменные сведения о вычислительной системе (ее конфигурация, состав оборудования, состав и особенности системного программного обеспечения) содержатся в специальных системных таблицах. Процессору, каналам прямого доступа в память, контроллерам необходимо передавать конкретную двоичную информацию, с помощью которой и осуществляется управление оборудованием. Эта конкретная двоичная информация в виде кодов и данных часто готовится с помощью препроцессоров, но часть ее хранится в системных таблицах.

## 3.2. Режимы управления вводом-выводом

### 3.2.1. Сущность основных режимов ввода-вывода

Имеются два основных режима ввода/вывода: *режим обмена с опросом готовности* устройства ввода/вывода и *режим обмена с прерываниями.* Рассмотрим рис. 4.8.

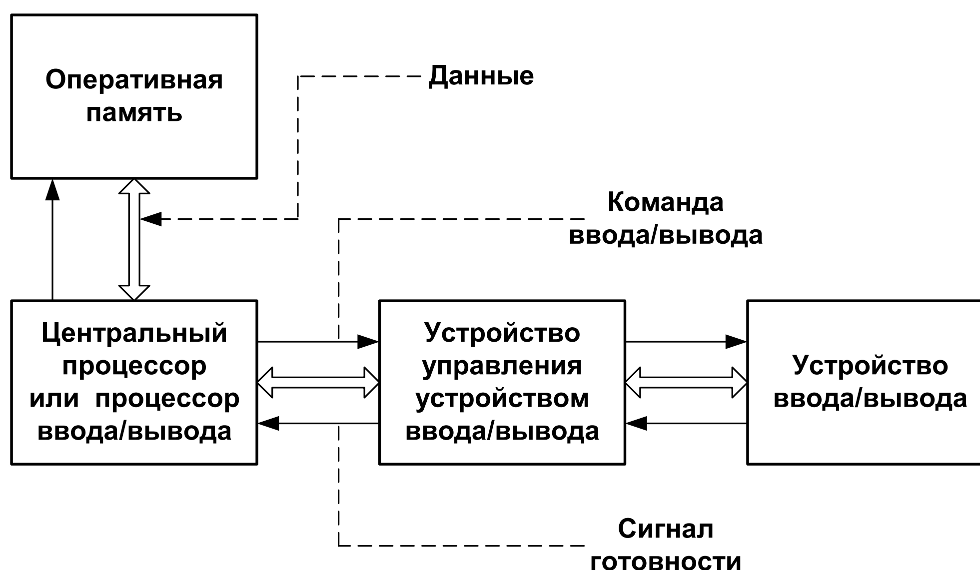



Рис. 4.8. Управление вводом/выводом

Пусть для простоты управление вводом/выводом осуществляет центральный процессор (в этом случае часто говорят о наличии программного канала обмена данными между внешними устройством и оперативной памятью, в отличие от канала прямого доступа к памяти, при котором управление вводом/выводом осуществляет специальное дополнительное оборудование). Центральный процессор посылает устройству управления команду выполнить некоторое действие устройству ввода/вывода. Последнее исполняет команду, транслируя сигналы, понятные центральному устройству и устройству управления в сигналы, понятные устройству ввода/вывода. Но быстродействие устройства ввода/вывода намного меньше быстродействия центрального процессора (порой на несколько порядков). Поэтому сигнал готовности (транслируемый или генерируемый устройством управления и сигнализирующий процессору о том, что команда ввода/вывода выполнена и можно выдать новую команду для продолжения обмена данными) приходится очень долго ожидать, постоянно опрашивая соответствующую линию интерфейса на наличие или отсутствие нужного сигнала. Посылать новую команду, не дождавшись сигнала готовности, сообщающего об исполнении предыдущей команды, бессмысленно. В режиме опроса готовности драйвер, управляющий процессом обмена данными с внешним устройством, как раз и выполняет в цикле команду “проверить наличие сигнала готовности”. До тех пор пока сигнал готовности не появится, драйвер ничего другого не делает. При этом, естественно, нерационально используется время центрального процессора. Гораздо выгоднее, выдав команду ввода/вывода, на время забыть об устройстве ввода/вывода и перейти на выполнение другой программы. А появление сигнала готовности трактовать как запрос на прерывание от устройства ввода/вывода. Именно эти сигналы готовности и являются сигналами запроса на прерывание.

Режим обмена с прерываниями по своей сути является режимом асинхронного управления. Для того чтобы не потерять связь с устройством (после того как процессор выдал очередную команду по управлению обменом данными и переключился на выполнение других программ), может быть запущен отсчет времени, в течение которого устройство обязательно должно выполнить команду и выдать таки сигнал запроса на прерывание. Максимальный интервал времени, в течение которого устройство ввода/вывода или его контроллер должны выдать сигнал запроса на прерывание, часто называют *уставкой тайм-аута.* Если это время истекло после выдачи устройству очередной команды, а устройство так и не ответило, то делается вывод о том, что связь с устройством потеряна и управлять им больше нет возможности. Пользователь и/или задача получают соответствующее диагностическое сообщение.

Драйверы, работающие в режиме прерываний, представляют собой сложный комплекс программных модулей и могут иметь несколько сек-

ций: секцию запуска, одну или несколько секций продолжения и секцию завершения.

*Секция запуска* инициирует операцию ввода/вывода. Эта секция запускается для включения устройства ввода/вывода либо просто для инициации очередной операции ввода/вывода.

*Секция продолжения* (их может быть несколько, если алгоритм управления обменом данными сложный и требуется несколько прерываний для выполнения одной логической операции) осуществляет основную работу по передаче данных. Секция продолжения, собственно говоря, и является основным обработчиком прерывания. Используемый интерфейс может потребовать для управления вводом/выводом несколько последовательностей управляющих команд, а сигнал прерывания у устройства, как правило, только один. Поэтому после выполнения очередной секции прерывания супервизор прерываний при следующем сигнале готовности должен передать управление другой секции. Это делается за счет изменения адреса обработки прерывания после выполнения очередной секции, если же имеется только одна секция прерываний, то она сама передает управление тому или иному модулю обработки.

*Секция завершения* обычно выключает устройство ввода/вывода либо просто завершает операцию.

Управление операциями ввода/вывода в режиме прерываний требует больших усилий со стороны системных программистов – такие программы создавать сложнее, чем те, что работают в режиме опроса готовности.

### 3.2.2. Закрепление устройств, общие устройства ввода/вывода

Как известно, многие устройства не допускают совместного использования. Прежде всего, это устройства с последовательным доступом. Такие устройства могут стать закрепленными, то есть быть предоставленными некоторому вычислительному процессу на все время жизни этого процесса. Однако это приводит к тому, что вычислительные процессы часто не могут выполняться параллельно – они ожидают освобождения устройств ввода/вывода. Для организации использования многими параллельно выполняющимися задачами устройств ввода/вывода которые не могут быть разделяемыми, вводится понятие *виртуальных* устройств Использование принципа виртуализации позволяет повысить эффективность вычислительной системы.

Вообще говоря, понятие виртуального устройства шире, нежели использование этого термина для обозначения *спулинга* (SPOOLing – simultaneous peripheral operation on-line, то есть имитация работы с устройством в режиме “он-лайн”). Главная задача спулинга – создать видимость параллельного разделения устройства ввода/вывода с последовательным доступом, которое фактически должно использоваться только монопольно и

быть закрепленным. Например, в случае, когда несколько приложений должны выводить на печать результаты своей работы, если разрешить каждому такому приложению печатать строку по первому же требованию, то это приведет к потоку строк, не представляющих никакой ценности. Однако можно каждому вычислительному процессу предоставлять не реальный, а виртуальный принтер, и поток выводимых символов (или управляющих кодов для их печати) сначала направлять в специальный файл на магнитном диске. Затем, по окончании виртуальной печати, в соответствии с принятой дисциплиной обслуживания и приоритетами приложений выводить содержимое спул-файла на принтер. Системный процесс, который управляет спул-файлом, называется *спулером* (spool-reader или spool-writer).

### 3.2.3. Основные системные таблицы ввода/вывода

Каждая ОС имеет свои таблицы ввода/вывода, их состав (количество и назначение каждой таблицы) может сильно отличаться. В некоторых ОС вместо таблиц создаются списки, хотя использование статических структур данных для организации ввода/вывода, как правило, приводит к большему быстродействию.

Исходя из принципа управления вводом/выводом через супервизор ОС и учитывая, что драйверы устройств ввода/вывода используют механизм прерываний для установления обратной связи центральной части с внешними устройствами, можно сделать вывод о необходимости создания по крайней мере трех системных таблиц.

Первая таблица (или список) содержит информацию обо всех устройствах ввода/вывода, подключенных к вычислительной системе. Эта таблица обычно называется *таблицей оборудования* (equipment table), а каждый ее элемент называется UCB (unit control block, блок управления устройством ввода/вывода). Каждый элемент UCB таблицы оборудования, как правило, содержит следующую информацию об устройстве:

тип устройства, его конкретная модель, символическое имя и характеристики устройства;

как это устройство подключено (через какой интерфейс, к какому разъему, какие порты и линия запроса прерывания используются и т. д.);

номер и адрес канала (и подканала), если такие используются для управления устройством;

указание на драйвер, который должен управлять этим устройством, адрес секции запуска и секции продолжения драйвера;

информация о том, используется или нет буферирование при обмене данными с этим устройством, “имя” (или просто адрес) буфера, если такой выделяется из системной области памяти;

уставка тайм-аута и ячейки для счетчика тайм-аута;

состояние устройства;

поле указателя для связи задач, ожидающих устройство, и, возможно, много еще каких сведений.

Поскольку во многих ОС драйверы могут обладать свойством реентерабельности (один и тот же экземпляр программного модуля может обеспечить параллельное обслуживание сразу нескольких однотипных устройств), то в элементе UCB должна храниться либо непосредственно сама информация о текущем состоянии устройства и сами переменные для реентерабельной обработки, либо указание на место, где такая информация может быть найдена. Наконец, важнейшим компонентом элемента таблицы оборудования является указатель на дескриптор той задачи, которая сейчас использует данное устройство. Если устройство свободно, то поле указателя будет иметь нулевое значение. Если же устройство уже занято и рассматриваемый указатель не нулевой, то новые запросы к устройству фиксируются посредством образования списка из дескрипторов тех задач, которые сейчас ожидают данное устройство.

Вторая таблица предназначена для реализации еще одного принципа виртуализации устройств ввода/вывода – независимости от устройства. Желательно, чтобы программист не был озабочен учетом конкретных параметров (и/или возможностей) того или иного устройства ввода/вывода, которое установлено (или не установлено) в компьютер. Для него должны быть важны только самые общие возможности, характерные для данного класса устройств ввода/вывода, которыми он желает воспользоваться. Например, принтер должен уметь выводить (печатать) символы или графическое изображение. А накопитель на магнитных дисках – считывать или записывать по указанному адресу (в координатах C-H-S – cylinder-head-sector) порцию данных. Ни программист, ни разработчики файловой системы не должны зависеть от того, накопитель какого конкретного типа и модели, а также какого производителя используется в данном конкретном компьютере. Важным должен быть только сам факт существования накопителя, имеющего некоторое количество цилиндров, головок чтения/записи и секторов на дорожке магнитного диска. Упомянутые значения количества цилиндров, головок и секторов должны быть взяты из элемента таблицы оборудования. При этом для программиста также не должно иметь значения, каким образом то или иное устройство подключено к вычислительной системе, а не только какая конкретная модель устройства используется. Поэтому в запросе на ввод/вывод программист указывает именно *логическое имя устройства.* Действительное устройство, которое сопоставляется виртуальному (логическому), выбирается супервизором с помощью таблицы о которой сейчас идет речь. Способ подключения устройства, его конкретная модель и соответствующий ей драйвер содержатся в уже рассмотренной таблице оборудования. А для того, чтобы связать некоторое виртуальное устройство, использованное программистом при создании приложения с системной таблицей, отображающей

информацию о том, какое конкретно устройство и каким образом подключено к компьютеру, используется вторая системная таблица. Обычно ее называют *таблицей описания виртуальных логических устройств* (DRT, device reference table). Назначение этой второй таблицы – установление связи между виртуальными (логическими) устройствами и реальными устройствами, описанными посредством первой таблицы оборудования. Другими словами, вторая таблица позволяет супервизору перенаправить запрос на ввод/вывод из приложения на те программные модули и структуры данных, которые (или адреса которых) хранятся в соответствующем элементе первой таблицы. Во многих многопользовательских системах такая таблица не одна, а несколько: одна общая и по одной – на каждого пользователя, что позволяет строить необходимые связи между логическими (символьными) именами устройств и реальными физическими устройствами, которые имеются в системе.

Наконец, третья таблица необходима для организации обратной связи между центральной частью и устройствами ввода/вывода. Это *таблица прерываний,* которая указывает для каждого сигнала запроса на прерывание тот элемент UCB, который сопоставлен данному устройству, подключенному так, что оно использует настоящую линию (сигнал) прерывания. Как системная таблица ввода/вывода, таблица прерываний может в явном виде и не присутствовать. В принципе можно сразу из основной таблицы прерываний попадать на программу обработки (драйвер), имеющей связи с элементом UCB. Важно наличие связи между сигналами прерываний и таблицей оборудования.

В современных сложных ОС имеется гораздо больше системных таблиц или списков, используемых для организации процесса управления операциями ввода/вывода. Например, одной из часто реализуемых информационных структур, сопровождающих практически каждый запрос на ввод/вывод, является блок управления данными (data control block, DCB). Назначение DCB – подключение препроцессоров к процессу подготовки данных на ввод/вывод, то есть учет конкретных технических характеристик и используемых преобразований. Это необходимо для того, чтобы имеющееся устройство получало не какие-то непонятные ему коды либо форматы данных, которые не соответствуют режиму его работы, а коды, созданные специально под данное устройство и используемый в настоящий момент формат представления данных.

Взаимосвязи между описанными таблицами изображены на рис. 4.9.

Теперь еще раз, с учетом изложенных принципов и таблиц, рассмотрим процесс управления вводом/вывода с помощью рис. 4.10.

Запрос на операцию ввода/вывода от выполняющейся программы поступает на супервизор (действие *1).* Тот проверяет системный вызов на соответствие принятым спецификациям и в случае ошибки возвращает задаче соответствующее сообщение (действие *1-1).* Если же запрос корректен, то он перенаправляется в супервизор ввода/вывода (действие 2).

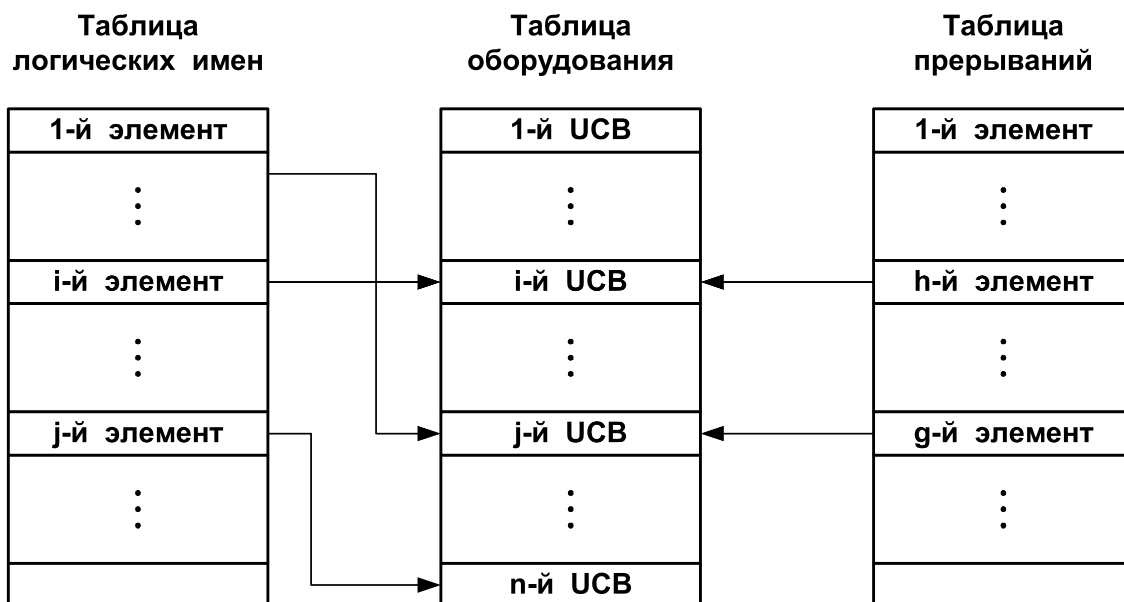



Рис. 4.9. Взаимосвязь системных таблиц ввода/вывода

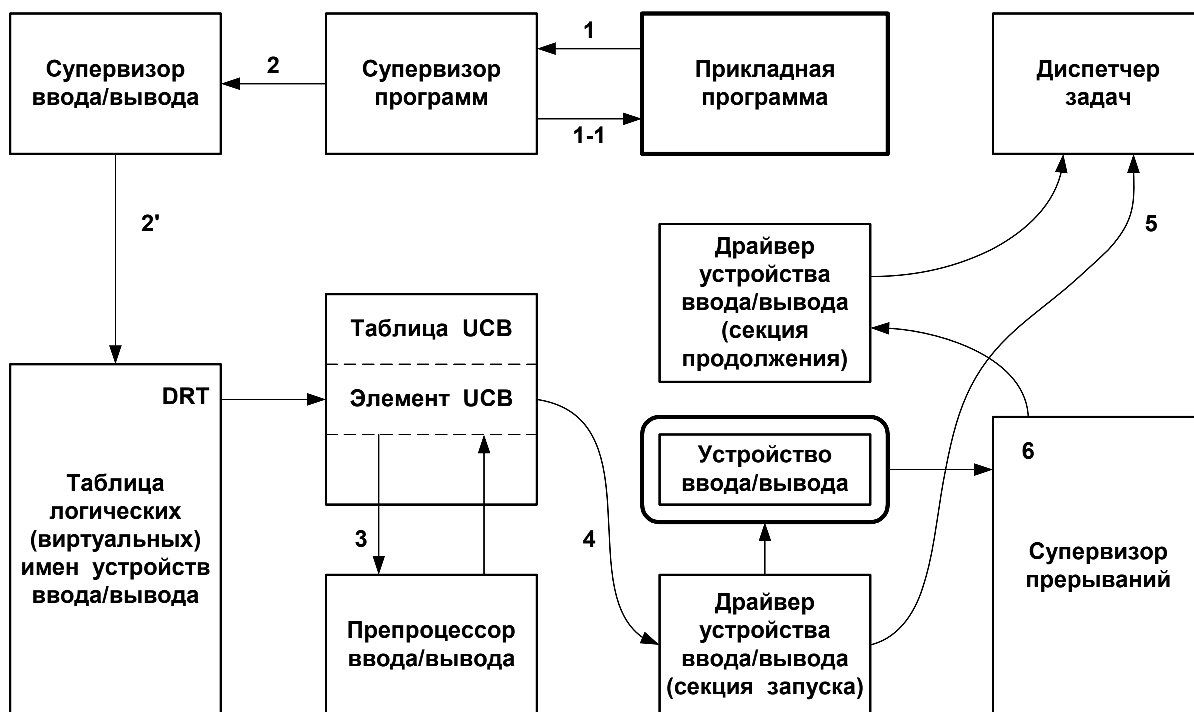



Рис. 4.10. Процесс управления вводом/выводом

Супервизор ввода/вывода по логическому (виртуальному) имени с помощью таблицы DRT находит соответствующий элемент UCB в таблице оборудования. Если устройство уже занято, то описатель задачи, запрос которой сейчас обрабатывается супервизором ввода/вывода, помещается в список задач, ожидающих настоящее устройство. Если же устройство свободно, то супервизор ввода/вывода определяет из UCB тип устройства и

при необходимости запускает препроцессор, позволяющий получить последовательность управляющих кодов и данных, которую сможет правильно понять и отработать устройство (действие *3)*. Когда "программа" управления операцией ввода/вывода будет готова, супервизор ввода/вывода передаст управление соответствующему драйверу на секцию запуска (действие *4)*. Драйвер инициализирует операцию управления, обнуляет счетчик тайм-аута и возвращает управление супервизору (диспетчеру задач) с тем, чтобы он поставил на процессор готовую к исполнению задачу (действие *5)*. Система работает своим чередом, но когда устройство ввода/вывода отработает посланную ему команду, оно выставляет сигнал запроса на прерывания, по которому через таблицу прерываний управление передается на секцию продолжения (действие *6*). Получив новую команду, устройство вновь начинает ее обрабатывать, а управление процессором опять передается диспетчеру задач, и процессор продолжает полезную работу. Таким образом, получается параллельная обработка задач, на фоне которой процессор осуществляет управление операциями ввода/вывода.

Очевидно, что если имеются специальные аппаратные средства для управления вводом/выводом, снимающие эту работу с центрального процессора (речь идет о каналах прямого доступа к памяти), то в функции центрального процессора будут по-прежнему входить все только что рассмотренные шаги, за исключением последнего – непосредственного управления операциями ввода/вывода. В случае использования каналов прямого доступа к памяти последние исполняют соответствующие канальные программы и разгружают центральный процессор, избавляя его от непосредственного управления обменом данными между памятью и внешними устройствами.

### 3.2.4. Синхронный и асинхронный ввод/вывод

Задача, выдавшая запрос на операцию ввода/вывода, переводится супервизором в состояние ожидания завершения заказанной операции. Когда супервизор получает от секции завершения сообщение о том, что операция завершилась, он переводит задачу в состояние готовности к выполнению, и она продолжает свою работу. Эта ситуация соответствует синхронному вводу/выводу. Синхронный ввод/вывод является стандартным для большинства ОС. Чтобы увеличить скорость выполнения приложений, было предложено при необходимости использовать асинхронный ввод/вывод.

Простейшим вариантом асинхронного вывода является так называемый буферированный вывод данных на внешнее устройство, при котором данные из приложения передаются не непосредственно на устройство ввода/вывода, а в специальный системный буфер. В этом случае логически

операция вывода для приложения считается выполненной сразу же, и задача может не ожидать окончания действительного процесса передачи данных на устройство. Процессом реального вывода данных из системного буфера занимается супервизор ввода/ вывода. Естественно, что выделением буфера из системной области памяти занимается специальный системный процесс по указанию супервизора ввода/вывода. Итак, для рассмотренного случая вывод будет асинхронным, если, во-первых, в запросе на ввод/вывод было указано на необходимость буферирования данных, а во-вторых, если устройство ввода/вывода допускает такие асинхронные операции и это отмечено в UCB.

Можно организовать и асинхронный ввод данных. Однако для этого необходимо не только выделить область памяти для временного хранения считываемых с устройства данных и связать выделенный буфер с задачей, заказавшей операцию, но и сам запрос на операцию ввода/вывода разбить на две части (на два запроса). В первом запросе указывается операция на считывание данных, подобно тому как это делается при синхронном вводе/выводе. Однако тип (код) запроса используется другой, и в запросе указывается еще по крайней мере один дополнительный параметр – имя (код) того системного объекта, которое получает задача в ответ на запрос и которое идентифицирует выделенный буфер. Получив имя буфера (будем этот системный объект условно называть таким образом, хотя в различных ОС для его обозначения используются и другие термины, например – класс), задача продолжает свою работу. Здесь очень важно подчеркнуть, что в результате запроса на асинхронный ввод данных задача не переводится супервизором ввода/вывода в состояние ожидания завершения операции ввода/ вывода, а остается в состоянии выполнения или в состоянии готовности к выполнению. Через некоторое время, выполнив необходимый код, который был определен программистом, задача выдает второй запрос на завершение операции ввода/вывода. В этом втором запросе к тому же устройству, который, естественно, имеет другой код (или имя запроса), задача указывает имя системного объекта (буфера для асинхронного ввода данных) и в случае успешного завершения операции считывания данных тут же получает их из системного буфера. Если же данные еще не успели до конца переписаться с внешнего устройства в системный буфер, супервизор ввода/вывода переводит задачу в состояние ожидания завершения операции ввода/вывода, и далее все напоминает обычный синхронный ввод данных.

Обычно асинхронный ввод/вывод предоставляется в большинстве мультипрограммных ОС, особенно если ОС поддерживает мультизадачность с помощью механизма потоков. Однако если асинхронный ввод/вывод в явном виде отсутствует, его идеи можно реализовать самому, организовав для вывода данных самостоятельный поток.

Аппаратуру ввода/вывода можно рассматривать как совокупность аппаратурных процессоров, которые способны работать параллельно отно-

сительно друг друга, а также относительно центрального процессора (процессоров). На таких "процессорах" выполняются так называемые *внешние процессы*. Например, для внешнего устройства (устройства ввода/вывода) внешний процесс может представлять собой совокупность операций, обеспечивающих перевод печатающей головки, продвижение бумаги на одну позицию, смену цвета чернил или печать каких-то символов. Внешние процессы, используя аппаратуру ввода/вывода, взаимодействуют как между собой, так и с обычными "программными" процессами, выполняющимися на центральном процессоре. Важным при этом является то обстоятельство, что скорости выполнения внешних процессов будут существенно (порой, на порядок или больше) отличаться от скорости выполнения обычных *("внутренних")* процессов. Для своей нормальной работы внешние и внутренние процессы обязательно должны синхронизироваться. Для сглаживания эффекта сильного несоответствия скоростей между внутренними и внешними процессами используют упомянутое выше буферирование. Таким образом, можно говорить о системе параллельных взаимодействующих процессов.

Буферы являются критическим ресурсом в отношении внутренних (программных) и внешних процессов, которые при параллельном своем развитии информационно взаимодействуют. Через буфер (буферы) данные либо посылаются от некоторого процесса к адресуемому внешнему (операция вывода данных на внешнее устройство), либо от внешнего процесса передаются некоторому программному процессу (операция считывания данных). Введение буферирования как средства информационного взаимодействия выдвигает проблему управления этими системными буферами, которая решается средствами супервизорной части ОС. При этом на супервизор возлагаются задачи не только по выделению и освобождению буферов в системной области памяти, но и синхронизации процессов в соответствии с состоянием операций по заполнению или освобождению буферов, а также их ожидания, если свободных буферов в наличии нет, а запрос на ввод/вывод требует буферирования. Обычно супервизор ввода/вывода для решения перечисленных задач использует стандартные средства синхронизации, принятые в данной ОС. Поэтому если ОС имеет развитые средства для решения проблем параллельного выполнения взаимодействующих приложений и задач, то, как правило, она реализует и асинхронный ввод/вывод.

### 3.2.5. Кэширование операций ввода/вывода при работе с накопителями на магнитных дисках

Как известно, накопители на магнитных дисках обладают крайне низкой скоростью по сравнению с быстродействием центральной части компьютера. Разница в быстродействии отличается на несколько порядков.

Например, современные процессоры за один такт работы, а они работают уже с частотами в 1 ГГц и более, могут выполнять по две операции. Таким образом, время выполнения операции (с позиции внешнего наблюдателя, не видящего конвейеризации при выполнении машинных команд, благодаря которой производительность возрастает в несколько раз) может составлять 0,5 нс (!). В то же время переход магнитной головки с дорожки на дорожку составляет несколько миллисекунд. Такие же временные интервалы имеют место и при ожидании, пока под головкой чтения/ записи не окажется нужный сектор данных. Как известно, в современных приводах средняя длительность на чтение случайным образом выбранного сектора данных составляет около 20 мс, что существенно медленнее, чем выборка команды или операнда из оперативной памяти и уж тем более из кэша. Правда, после этого данные читаются большим пакетом (сектор, как мы уже говорили, имеет размер в 512 байтов, а при операциях с диском часто читается или записывается сразу несколько секторов). Таким образом, средняя скорость работы процессора с оперативной памятью на 2-3 порядка выше, чем средняя скорость передачи данных из внешней памяти на магнитных дисках в оперативную память.

Для того чтобы сгладить такое сильное несоответствие в производительности основных подсистем, используется буферирование и/или кэширование данных. Простейшим вариантом ускорения дисковых операций чтения данных можно считать использование двойного буферирования. Его суть заключается в том, что пока в один буфер заносятся данные с магнитного диска, из второго буфера ранее считанные данные могут быть прочитаны и переданы запросившей их задаче. Аналогичный процесс происходит и при записи данных. Буферирование используется во всех операционных системах, но помимо буферирования применяется и кэширование. Кэширование исключительно полезно в том случае, когда программа неоднократно читает с диска одни и те же данные. После того как они один раз будут помещены в кэш, обращений к диску больше не потребуется и скорость работы программы значительно возрастет.

Если не вдаваться в подробности, то под кэшем можно понимать некий пул буферов, которыми мы управляем с помощью соответствующего системного процесса. Если мы считываем какое-то множество секторов, содержащих записи того или иного файла, то эти данные, пройдя через кэш, там остаются (до тех пор, пока другие секторы не заменят эти буферы). Если впоследствии потребуется повторное чтение, то данные могут быть извлечены непосредственно из оперативной памяти без фактического обращения к диску. Ускорить можно и операции записи: данные помещаются в кэш, и для запросившей эту операцию задачи можно считать, что они уже фактически и записаны. Задача может продолжить свое выполнение, а системные внешние процессы через некоторое время запишут данные на диск. Это называется *операцией отложенной записи* (lazy write, “ленивая запись”). Если отложенная запись отключена, только одна задача может

записывать на диск свои данные. Остальные приложения должны ждать своей очереди. Это ожидание подвергает информацию риску не меньшему (если не большему), чем отложенная запись, которая к тому же и более эффективна по скорости работы с диском.

Интервал времени, после которого данные будут фактически записываться, с одной стороны, желательно выбрать больше, поскольку если потребуется еще раз прочитать эти данные, то они уже и так фактически находятся в кэше. И после модификации эти данные опять же помещаются в быстродействующий кэш. С другой стороны, для большей надежности данные желательно поскорее отправить во внешнюю память, поскольку она энергонезависима и в случае какой-нибудь аварии (например, нарушения питания) данные в оперативной памяти пропадут, в то время как на магнитном диске они с большой вероятностью останутся в безопасности.

Кэширование дисковых операций может быть существенно улучшено за счет введения техники *упреждающего чтения* (read ahead). Она основана на чтении с диска гораздо большего количества данных, чем на самом деле запросила операционная система или приложение. Когда некоторой программе требуется считать с диска только один сектор, программа кэширования читает еще и несколько дополнительных блоков данных. А операции последовательного чтения нескольких секторов фактически несущественно замедляют операцию чтения затребованного сектора с данными. Поэтому, если программа вновь обратится к диску, вероятность того, что нужные ей данные уже находятся в кэше, достаточно высока. Поскольку передача данных из одной области памяти в другую происходит во много раз быстрее, чем чтение их с диска, кэширование существенно сокращает время выполнения операций с файлами.

Помимо описанных действий ОС может выполнять и работу по оптимизации перемещения головок чтения/записи данных, связанного с выполнением запросов от параллельно выполняющихся задач. Время, необходимое на получение данных с магнитного диска, складывается из времени перемещения магнитной головки на требуемый цилиндр и времени ожидания заданного сектора; временем считывания найденного сектора и затратами на передачу этих данных в оперативную память мы можем пренебречь. Таким образом, основные затраты времени уходят на поиск данных. В мультипрограммных ОС при выполнении многих задач запросы на чтение и запись данных могут идти таким потоком, что при их обслуживании образуется очередь. Если выполнять эти запросы в порядке поступления их в очередь, то вследствие случайного характера обращений к тому или иному сектору магнитного диска мы можем иметь значительные потери времени на поиск данных. Напрашивается очевидное решение: поскольку выполнение переупорядочивания запросов с целью минимизации затрат времени на поиск данных можно выполнить очень быстро (практически этим временем можно пренебречь, учитывая разницу в быстродействии центральной части и устройств ввода/вывода), то необходимо найти

метод, позволяющий перестраивать очередь запросов оптимальным образом. Изучение этой проблемы позволило найти наиболее эффективные дисциплины планирования.

В настоящее время в ОС используются следующие дисциплины, в соответствии с которыми можно перестраивать очередь запросов на операции чтения/записи данных:

1. *SSTF* (shortest seek time – first) – с наименьшим временем поиска – первым. В соответствии с этой дисциплиной при позиционировании магнитных головок следующим выбирается запрос, для которого необходимо минимальное перемещение с цилиндра на цилиндр, даже если этот запрос не был первым в очереди на ввод/вывод. Однако для этой дисциплины характерна резкая дискриминация определенных запросов, а ведь они могут идти от высокоприоритетных задач. Обращения к диску проявляют тенденцию концентрироваться, в результате чего запросы на обращение к самым внешним и самым внутренним дорожкам могут обслуживаться существенно дольше и нет никакой гарантии обслуживания. Достоинством такой дисциплины является максимально возможная пропускная способность дисковой подсистемы.

2. *Scan* (сканирование). По этой дисциплине головки перемещаются то в одном, то в другом "привилегированном" направлении, обслуживая "по пути" подходящие запросы. Если при перемещении головок чтения/записи более нет попутных запросов, то движение начинается в обратном направлении.

3. *Next-Step Scan* – отличается от предыдущей дисциплины тем, что на каждом проходе обслуживаются только запросы, которые уже существовали на момент начала прохода. Новые запросы, появляющиеся в процессе перемещения головок чтения/записи, формируют новую очередь запросов, причем таким образом, чтобы их можно было оптимально обслужить на обратном ходу.

4. *C-Scan* (циклическое сканирование). По этой дисциплине головки перемещаются циклически с самой наружной дорожки к внутренним, по пути обслуживая имеющиеся запросы, после чего вновь переносятся к наружным цилиндрам. Эту дисциплину иногда реализуют таким образом, чтобы запросы, поступающие во время текущего прямого хода головок, обслуживались не попутно, а при следующем ходе, что позволяет исключить дискриминацию запросов к самым крайним цилиндрам; она характеризуется очень малой дисперсией времени ожидания обслуживания. Эту дисциплину обслуживания часто называют *"элеваторной"*.

# 4. Управление данными

Одной из основных задач операционной системы является предоставление удобств пользователю при работе с данными, хранящимися на дисках. В современных ОС имеются специальные модули, предназначенные для работы с данными. Эту совокупность модулей часто называют системой управления файлами или файловой системой. Она представляет собой наиболее полную, объемную и сложную часть современных ОС, что объясняется большим разнообразием тех функций, которые она выполняет в интересах пользователей и самой операционной системы.

Для обеспечения удобства работы пользователя с данными ОС подменяет физическую структуру хранящихся данных некоторой удобной для пользователя логической моделью. Логическая модель файловой системы материализуется в виде дерева каталогов, выводимого на экран такими утилитами, как Norton Commander или Windows Explorer, в символьных составных именах файлов, в командах работы с файлами. Базовым элементом этой модели является файл, который так же, как и файловая система в целом, может характеризоваться как логической, так и физической структурой.

## 4.1. Задачи управления данными

Основной целью управления данными является организация их хранения, обеспечение правильного доступа к ним и выдача информации о данных.

Задачи управления данными:

1. Организация данных в наборы в соответствии с принятыми методами и указаниями пользователя.
2. Идентификация наборов данных, размещение их на внешних носителях и организация хранения.
3. Ведение каталога данных и организация поиска данных по их именам.
4. Защита данных от их несанкционированного использования.
5. Взаимодействие с программами пользователей на уровне, не зависящем от характера конкретных носителей данных и устройств ввода-вывода.

Модули управления данными ОС решают эти задачи, если структура данных отвечает требованиям, предъявленным к ней операционной системой. В противном случае ОС не может обеспечить решение всех перечисленных задач и пользователь должен это сделать самостоятельно. С целью распределения функций между системными программами и программами пользователей в ОС выделяют два уровня управления данными: физический и логический.

На физическом уровне управления пользователь должен сам обеспечивать выполнение всех вспомогательных операций по чтению и записи наборов данных. Этот уровень предполагает наличие у пользователя необходимой информации о размещении его данных в вычислительной машине и о периферийных устройствах, выделенных в его распоряжение. Пользователь сам программирует перепись нужного набора данных в оперативную память, извлечение необходимых для работы элементов из этого набора и обратную перепись преобразованного набора во внешнюю память.

Логический уровень управления избавляет пользователя от необходимости знания характеристик конкретных периферийных устройств и адресов хранения данных. Системные модули выполняют все вспомогательные функции так, что при указании наименования набора данных требуемый набор к началу обработки будет передан в нужную область оперативной памяти.

## 4.2. Понятие и функции файловой системы

Базовым элементом файловой системы является файл.

*Файл* – это именованная область внешней памяти, в которую можно записывать и из которой можно считывать данные.

Файлы хранятся в памяти, не зависящей от энергопитания, обычно – на магнитных дисках. Однако нет правил без исключения. Одним из таких исключений является так называемый электронный диск, когда в оперативной памяти создается структура, имитирующая файловую систему.

Основные *цели использования файла* перечислены ниже.

*Долговременное и надежное хранение информации*. Долговременность достигается за счет использования запоминающих устройств, не зависящих от питания, а высокая надежность определяется средствами защиты доступа к файлам и общей организацией программного кода ОС, при которой сбои аппаратуры чаще всего не разрушают информацию, хранящуюся в файлах.

*Совместное использование информации*. Файлы обеспечивают естественный и легкий способ разделения информации между приложениями и пользователями за счет наличия понятного человеку символьного имени и постоянства хранимой информации и расположения файла. Пользователь должен иметь удобные средства работы с файлами, включая каталоги-справочники, объединяющие файлы в группы, средства поиска файлов по признакам, набор команд для создания, модификации и удаления файлов. Файл может быть создан одним пользователем, а затем использоваться совсем другим пользователем, при этом создатель файла или администратор могут определить права доступа к нему других пользователей. Эти цели реализуются в ОС файловой системой.

Для организации и управления файлами создаются соответствующие файловые системы, предоставляющие возможность иметь дело с логическим уровнем структуры данных и операций, выполняемых над ними в процессе их обработки.

Таким образом, *файловая система* – это часть операционной системы, включающая:

совокупность всех файлов на диске;

наборы структур данных, используемых для управления файлами, такие, например, как каталоги файлов, дескрипторы файлов, таблицы распределения свободного и занятого пространства на диске;

комплекс системных программных средств, реализующих различные операции над файлами, такие как создание, уничтожение, чтение, запись, именование и поиск файлов.

Именно файловая система определяет способ организации данных на диске или на каком-нибудь ином носителе данных. В качестве примера можно привести файловую систему FAT, реализация для которой имеется в абсолютном большинстве ОС, работающих в современных ПК. Так, например, в MS-DOS, OS/2, Windows 95, Windows NT, Linux имеется поддержка работы с файлами, организованными по принципам FAT. Однако программные модули соответствующих файловых систем не взаимозаменяемы. Кроме этого, все эти файловые системы имеют свои индивидуальные особенности и ограничения.

Задачи, решаемые файловой системой, зависят от способа организации вычислительного процесса в целом. Основными функциями файловых систем являются:

1) создание, удаление, переименование (и другие операции) файлов;

2) обеспечение программного интерфейса для приложений;

3) отображение логической модели файловой системы на физическую организацию хранилища данных;

4) работа с не дисковыми периферийными устройствами как с файлами;

5) обмен данными между файлами, между устройствами, между файлом и устройством (и наоборот);

6) защита файлов от несанкционированного доступа;

7) обеспечение устойчивости файловой системы к сбоям питания, ошибкам аппаратных и программных средств.

Таким образом, файловая система позволяет программам обходиться набором достаточно простых операций для выполнения действий над некоторым абстрактным объектом, представляющим файл. При этом программистам не нужно иметь дело с деталями действительного расположения данных на диске, буферизацией данных и другими низкоуровневыми проблемами передачи данных с долговременного запоминающего устройства. Все эти функции файловая система берет на себя. Файловая система распределяет дисковую память, поддерживает именование файлов, ото-

бражает имена файлов в соответствующие адреса во внешней памяти, обеспечивает доступ к данным, поддерживает разделение, защиту и восстановление файлов.

## 4.3. Логическая организация файловой системы

### 4.3.1. Типы файлов

Файловые системы поддерживают несколько функционально различных типов файлов, в число которых, как правило, входят обычные файлы, файлы-каталоги, специальные файлы, именованные конвейеры, отображаемые в память файлы и другие.

*Обычные файлы*, или просто файлы, содержат информацию произвольного характера, которую заносит в них пользователь или которая образуется в результате работы системных и пользовательских программ. Большинство современных операционных систем (например, UNIX, Windows, OS/2) никак не ограничивает и не контролирует содержимое и структуру обычного файла. Содержание обычного файла определяется приложением, которое с ним работает. Например, текстовый редактор создает текстовые файлы, состоящие из строк символов, представленных в каком-либо коде. Это могут быть документы, исходные тексты программ и т. п. Текстовые файлы можно прочитать на экране и распечатать на принтере. Двоичные файлы не используют коды символов, они часто имеют сложную внутреннюю структуру, например исполняемый код программы или архивный файл. Все операционные системы должны уметь распознавать хотя бы один тип файлов – их собственные исполняемые файлы.

*Каталоги* – это особый тип файлов, которые содержат системную справочную информацию о наборе файлов, сгруппированных пользователями по какому-либо неформальному признаку (например, в одну группу объединяются файлы, содержащие документы одного договора, или файлы, составляющие один программный пакет). Во многих операционных системах в каталог могут входить файлы любых типов, в том числе другие каталоги, за счет чего образуется древовидная структура, удобная для поиска. Каталоги устанавливают соответствие между именами файлов и их характеристиками, используемыми файловой системой для управления файлами. В число таких характеристик входит, в частности, информация (или указатель на другую структуру, содержащую эти данные) о типе файла и расположении его на диске, правах доступа к файлу и датах его создания и модификации. Во всех остальных отношениях каталоги рассматриваются файловой системой как обычные файлы.

*Специальные файлы* – это фиктивные файлы, ассоциированные с устройствами ввода-вывода, которые используются для унификации меха-

низма доступа к файлам и внешним устройствам. Специальные файлы позволяют пользователю выполнять операции ввода-вывода посредством обычных команд записи в файл или чтения из файла. Эти команды обрабатываются сначала программами файловой системы, а затем на некотором этапе выполнения запроса преобразуются операционной системой в команды управления соответствующим устройством.

Современные файловые системы поддерживают и другие типы файлов, такие как символьные связи, именованные конвейеры, отображаемые в память файлы.

### 4.3.2. Иерархическая структура файловой системы

Пользователи обращаются к файлам по символьным именам. Однако способности человеческой памяти ограничивают количество имен объектов, к которым пользователь может обращаться по имени. Иерархическая организация пространства имен позволяет значительно расширить эти границы. Именно поэтому большинство файловых систем имеет иерархическую структуру, в которой уровни создаются за счет того, что каталог более низкого уровня может входить в каталог более высокого уровня (рис. 4.11).

Граф, описывающий иерархию каталогов, может быть деревом или сетью. Каталоги образуют дерево, если файлу разрешено входить только в один каталог (рис. 4.11, б), и сеть – если файл может входить сразу в несколько каталогов (рис. 4.11, в). Например, в MS-DOS и Windows каталоги образуют древовидную структуру, а в UNIX – сетевую. В древовидной структуре каждый файл является листом. Каталог самого верхнего уровня называется корневым каталогом, или корнем (root).

При такой организации пользователь освобожден от запоминания имен всех файлов, ему достаточно примерно представлять, к какой группе может быть отнесен тот или иной файл, чтобы путем последовательного просмотра каталогов найти его. Иерархическая структура удобна для многопользовательской работы: каждый пользователь со своими файлами локализуется в своем каталоге или поддереве каталогов, и вместе с тем все файлы в системе логически связаны.

Частным случаем иерархической структуры является одноуровневая организация, когда все файлы входят в один каталог (рис. 4.11, а).

Все типы файлов имеют символьные имена. В иерархически организованных файловых системах обычно используются три типа имен файлов: простые, составные и относительные.

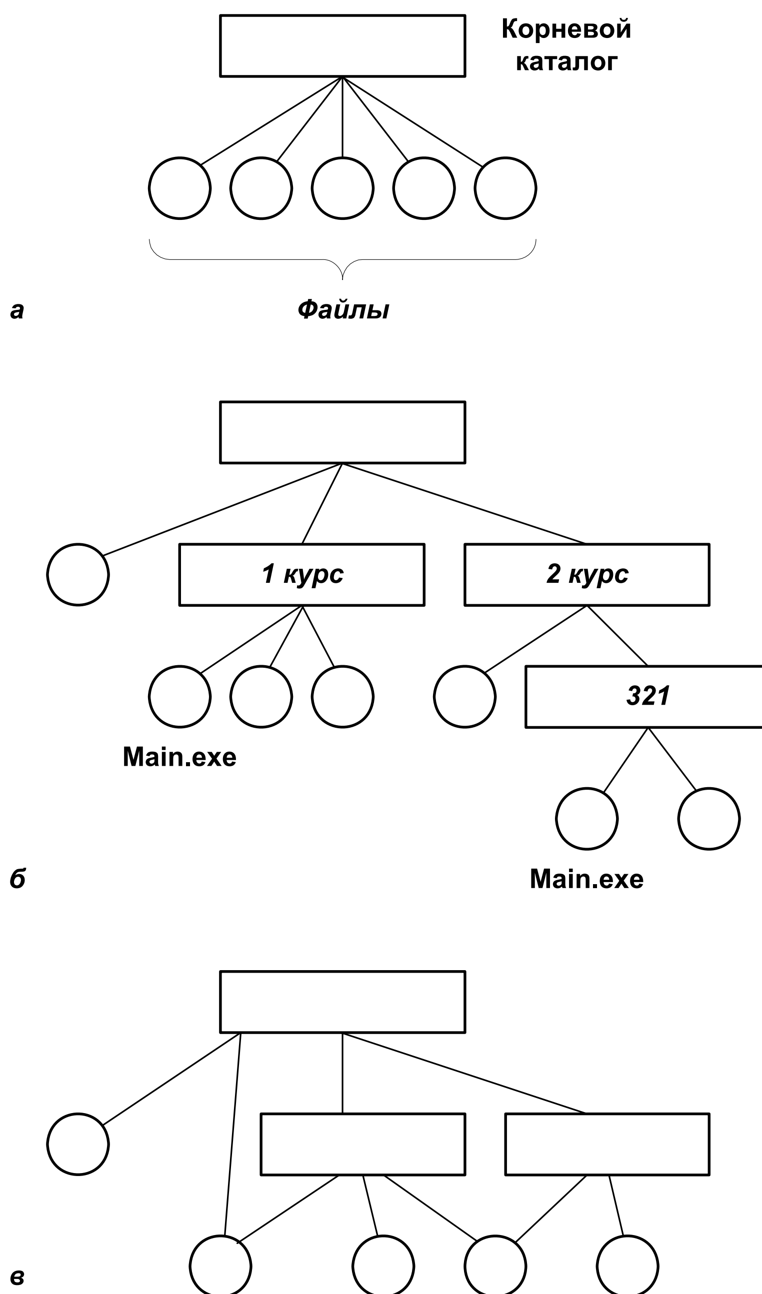



Рис. 4.11. Иерархия файловых систем

### 4.3.3. Имена файлов

Простое, или короткое, символьное имя идентифицирует файл в пределах одного каталога. Простые имена присваивают файлам пользователи и программисты, при этом они должны учитывать ограничения ОС как на номенклатуру символов, так и на длину имени. До сравнительно недавнего времени эти границы были весьма узкими. Так, в популярной файловой системе FAT длина имен ограничивались схемой 8.3 (8 символов – собственно имя, 3 символа – расширение имени), а в файловой системе s5, поддерживаемой многими версиями ОС UNIX, простое символьное имя не

могло содержать более 14 символов. Однако пользователю гораздо удобнее работать с длинными именами, поскольку они позволяют дать файлам легко запоминающиеся названия, ясно говорящие о том, что содержится в этом файле. Поэтому современные файловые системы, а также усовершенствованные варианты уже существовавших файловых систем, как правило, поддерживают длинные простые символьные имена файлов. Например, в файловых системах NTFS и FAT32, входящих в состав операционной системы Windows NT, имя файла может содержать до 255 символов.

Примеры простых имен файлов и каталогов:

Data
Учебные материалы
win.com
Proba.txt
_setup.dll
OS_0_01_Lek.doc
Иерархия файловых систем.vsd

В иерархических файловых системах разным файлам разрешено иметь одинаковые простые символьные имена при условии, что они принадлежат разным каталогам. То есть здесь работает схема "много файлов – одно простое имя". Для однозначной идентификации файла в таких системах используется так называемое полное имя.

Полное имя представляет собой цепочку простых символьных имен всех каталогов, через которые проходит путь от корня до данного файла. Таким образом, полное имя является составным, в котором простые имена отделены друг от друга принятым в ОС разделителем. Часто в качестве разделителя используется прямой или обратный слеш, при этом принято не указывать имя корневого каталога. На рис. 4.11, б два файла имеют простое имя main.exe, однако их составные имена /1 курс/main.exe и /2 курс/321/main.exe различаются.

В древовидной файловой системе между файлом и его полным именем имеется взаимно однозначное соответствие "один файл – одно полное имя". В файловых системах, имеющих сетевую структуру, файл может входить в несколько каталогов, а значит, иметь несколько полных имен; здесь справедливо соответствие "один файл – много полных имен". В обоих случаях файл однозначно идентифицируется полным именем.

Файл может быть идентифицирован также относительным именем. Относительное имя файла определяется через понятие "текущий каталог". Для каждого пользователя в каждый момент времени один из каталогов файловой системы является текущим, причем этот каталог выбирается самим пользователем по команде ОС. Файловая система фиксирует имя текущего каталога, чтобы затем использовать его как дополнение к относительным именам для образования полного имени файла. При использовании относительных имен пользователь идентифицирует файл цепочкой имен каталогов, через которые проходит маршрут от текущего каталога до

данного файла. Например, если текущим каталогом является каталог /2 курс, то относительное имя файла /2 курс/321/main.exe выглядит следующим образом: 321/main.exe.

В некоторых операционных системах разрешено присваивать одному и тому же файлу несколько простых имен, которые можно интерпретировать как псевдонимы. В этом случае, так же как в системе с сетевой структурой, устанавливается соответствие "один файл – много полных имен", так как каждому простому имени файла соответствует по крайней мере одно полное имя.

И хотя полное имя однозначно определяет файл, операционной системе проще работать с файлом, если между файлами и их именами имеется взаимно однозначное соответствие. С этой целью она присваивает файлу уникальное имя, так что справедливо соотношение "один файл – одно уникальное имя". Уникальное имя существует наряду с одним или несколькими символьными именами, присваиваемыми файлу пользователями или приложениями. Уникальное имя представляет собой числовой идентификатор и предназначено только для операционной системы. Примером такого уникального имени файла является номер индексного дескриптора в системе UNIX.

### 4.3.4. Монтирование

В общем случае вычислительная система может иметь несколько дисковых устройств. Даже типичный персональный компьютер обычно имеет один накопитель на жестком диске, один накопитель на гибких дисках и накопитель для компакт-дисков. Мощные же компьютеры, как правило, оснащены большим количеством дисковых накопителей, на которые устанавливаются пакеты дисков. Более того, даже одно физическое устройство с помощью средств операционной системы может быть представлено в виде нескольких логических устройств, в частности путем разбиения дискового пространства на разделы. Возникает вопрос, каким образом организовать хранение файлов в системе, имеющей несколько устройств внешней памяти?

Первое решение состоит в том, что на каждом из устройств размещается автономная файловая система, то есть файлы, находящиеся на этом устройстве, описываются деревом каталогов, никак не связанным с деревьями каталогов на других устройствах. В таком случае для однозначной идентификации файла пользователь наряду с составным символьным именем файла должен указывать идентификатор логического устройства. Примером такого автономного существования файловых систем является операционная система MS-DOS, в которой полное имя файла включает буквенный идентификатор логического диска. Так, при обращении к фай-

лу, расположенному на диске A, пользователь должен указать имя этого диска: A:\2 курс\321\main.exe.

Другим вариантом является такая организация хранения файлов, при которой пользователю предоставляется возможность объединять файловые системы, находящиеся на разных устройствах, в единую файловую систему, описываемую единым деревом каталогов. Такая операция называется монтированием. Рассмотрим, как осуществляется эта операция на примере ОС UNIX.

Среди всех имеющихся в системе логических дисковых устройств операционная система выделяет одно устройство, называемое системным. Пусть имеются две файловые системы, расположенные на разных логических дисках (рис. 4.12), причем один из дисков является системным.

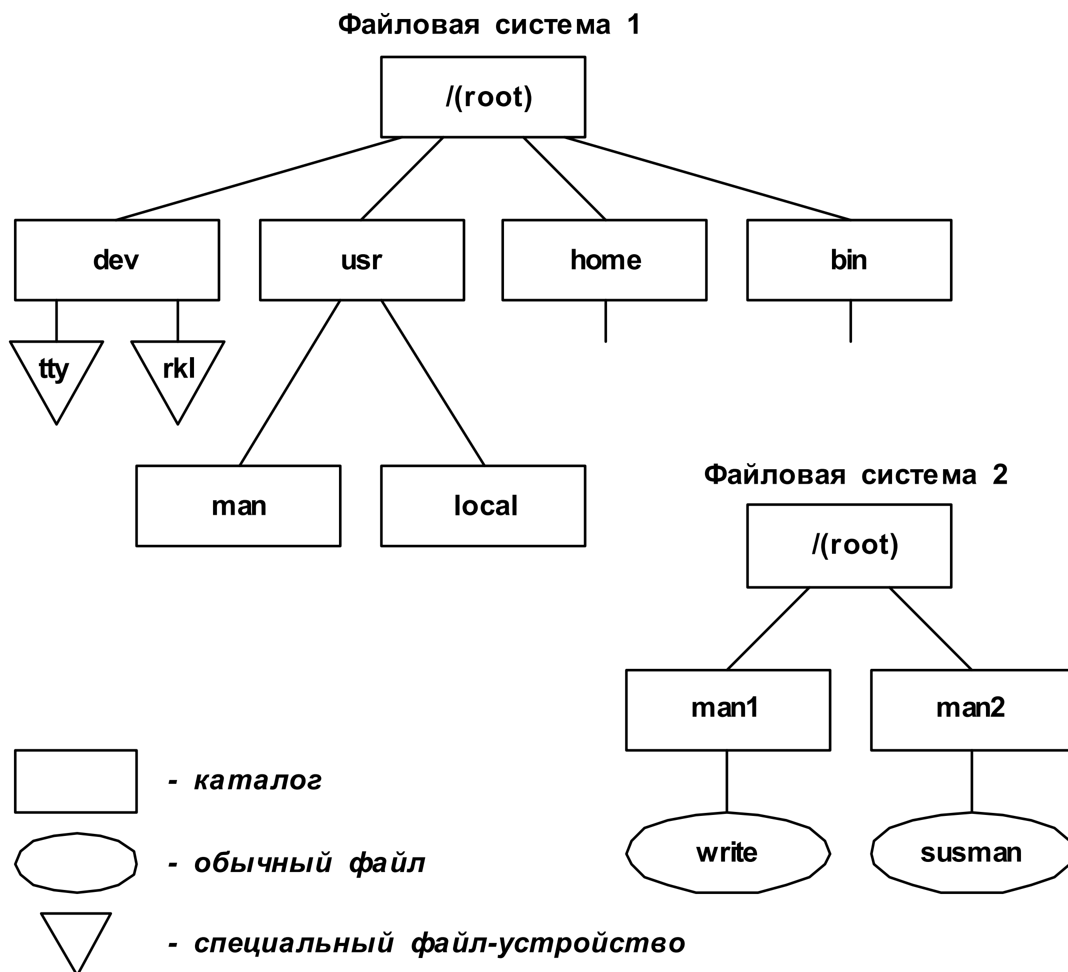



Рис. 4.12. Две файловые системы до монтирования

Файловая система, расположенная на системном диске, назначается корневой. Для связи иерархий файлов в корневой файловой системе выбирается некоторый существующий каталог, в данном примере – каталог man. После выполнения монтирования выбранный каталог man становится корневым каталогом второй файловой системы. Через этот каталог монти-

руемая файловая система подсоединяется как поддерево к общему дереву (рис. 4.13).

После монтирования общей файловой системы для пользователя нет логической разницы между корневой и смонтированной файловыми системами, в частности именование файлов производится так же, как если бы она с самого начала была единой.

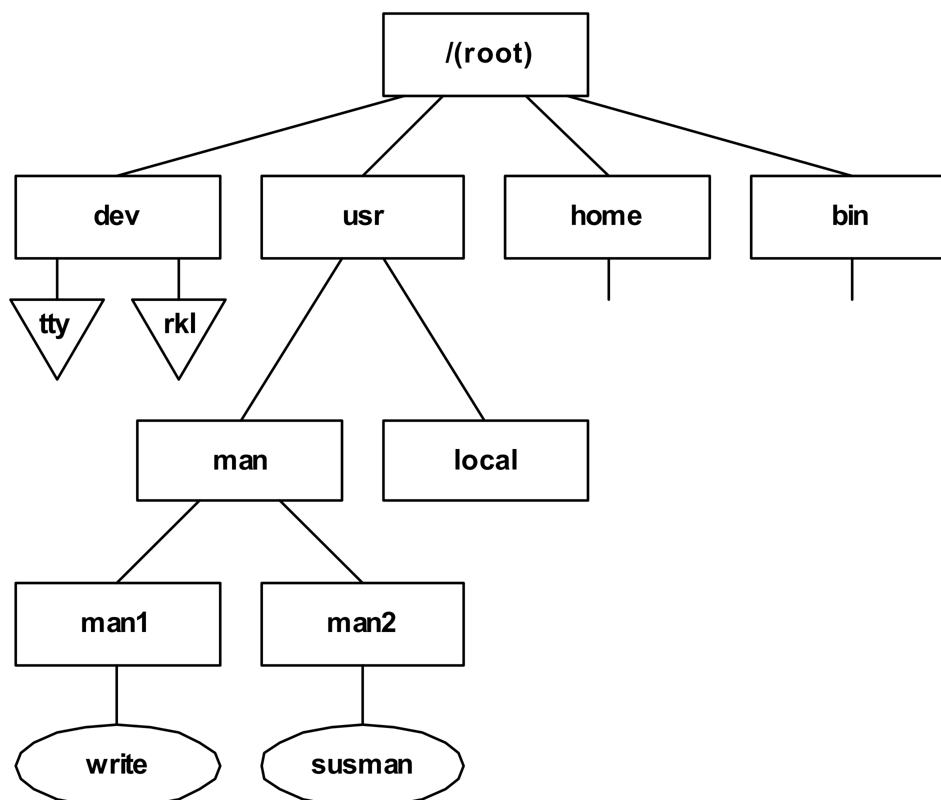



Рис. 4.13. Общая файловая система после монтирования

### 4.3.5. Атрибуты файлов

Понятие “файл” включает не только хранимые им данные и имя, но и атрибуты. Атрибуты – это информация, описывающая свойства файла. Примеры возможных атрибутов файла:

тип файла (обычный файл, каталог, специальный файл и т. п.);
владелец файла;
создатель файла;
пароль для доступа к файлу;
информация о разрешенных операциях доступа к файлу;
времена создания, последнего доступа и последнего изменения;
текущий размер файла;
максимальный размер файла;
признак “только для чтения”;

признак “скрытый файл”;
признак “системный файл”;
признак “архивный файл”;
признак “двоичный/символьный”;
признак “временный” (удалить после завершения процесса);
признак блокировки;
длина записи в файле;
указатель на ключевое поле в записи.

Набор атрибутов файла определяется спецификой файловой системы: в файловых системах разного типа для характеристики файлов могут использоваться разные наборы атрибутов. Например, в файловых системах, поддерживающих неструктурированные файлы, нет необходимости использовать три последних атрибута в приведенном списке, связанных со структуризацией файла. В однопользовательской ОС в наборе атрибутов будут отсутствовать характеристики, имеющие отношение к пользователям и защите, такие как владелец файла, создатель файла, пароль для доступа к файлу, информация о разрешенном доступе к файлу.

Пользователь может получать доступ к атрибутам, используя средства, предоставленные для этих целей файловой системой. Обычно разрешается читать значения любых атрибутов, а изменять – только некоторые. Например, пользователь может изменить права доступа к файлу (при условии, что он обладает необходимыми для этого полномочиями), но изменять дату создания или текущий размер файла ему не разрешается.

Значения атрибутов файлов могут непосредственно содержаться в каталогах, как это сделано в файловой системе MS-DOS (рис. 4.14, а). На рисунке представлена структура записи в каталоге, содержащая простое символьное имя и атрибуты файла. Здесь буквами обозначены признаки файла: R – только для чтения, A – архивный, H – скрытый, S – системный.

Рис. 4.14. Структура каталогов: а – структура записи каталога MS-DOS (32 байта), б – структура записи каталога ОС UNIX

Другим вариантом является размещение атрибутов в специальных таблицах, когда в каталогах содержатся только ссылки на эти таблицы. Такой подход реализован, например, в файловой системе *ufs* ОС UNIX. В этой файловой системе структура каталога очень простая. Запись о каждом файле содержит короткое символьное имя файла и указатель на индексный дескриптор файла, так называется в ufs таблица, в которой сосредоточены значения атрибутов файла (рис. 4.14, б).

В том и другом вариантах каталоги обеспечивают связь между именами файлов и собственно файлами. Однако подход, когда имя файла отделено от его атрибутов, делает систему более гибкой. Например, файл может быть легко включен сразу в несколько каталогов. Записи об этом файле в разных каталогах могут содержать разные простые имена, но в поле ссылки будет указан один и тот же номер индексного дескриптора.

### 4.3.6. Логическая организация файла

В общем случае данные, содержащиеся в файле, имеют некую логическую структуру. Эта структура является базой при разработке программы, предназначенной для обработки этих данных. Например, чтобы текст мог быть правильно выведен на экран, программа должна иметь возможность выделить отдельные слова, строки, абзацы и т. д. Признаками, отделяющими один структурный элемент от другого, могут служить определенные кодовые последовательности или просто известные программе значения смещений этих структурных элементов относительно начала файла. Поддержание структуры данных может быть либо целиком возложено на приложение, либо в той или иной степени эту работу может взять на себя файловая система.

В первом случае, когда все действия, связанные со структуризацией и интерпретацией содержимого файла целиком относятся к ведению приложения, файл представляется ФС неструктурированной последовательностью данных. Приложение формулирует запросы к файловой системе на ввод-вывод, используя общие для всех приложений системные средства, например, указывая смещение от начала файла и количество байт, которые необходимо считать или записать. Поступивший к приложению поток байт интерпретируется в соответствии с заложенной в программе логикой. Например, компилятор генерирует, а редактор связей воспринимает вполне определенный формат объектного модуля программы. При этом формат файла, в котором хранится объектный модуль, известен только этим программам. Подчеркнем, что интерпретация данных никак не связана с действительным способом их хранения в файловой системе.

Модель файла, в соответствии с которой содержимое файла представляется неструктурированной последовательностью (потоком) байт, стала популярной вместе с ОС UNIX, а теперь она широко используется в боль-

шинстве современных ОС, в том числе в MS-DOS, Windows NT/2000, NetWare. Неструктурированная модель файла позволяет легко организовать разделение файла между несколькими приложениями: разные приложения могут по-своему структурировать и интерпретировать данные, содержащиеся в файле.

Другая модель файла, которая применялась в ОС OS/360, DEC RSX и VMS, а в настоящее время используется достаточно редко, – это структурированный файл. В этом случае поддержание структуры файла поручается файловой системе. Файловая система видит файл как упорядоченную последовательность логических записей. Приложение может обращаться к ФС с запросами на ввод-вывод на уровне записей, например “считать запись 25 из файла FILE.DOC”. ФС должна обладать информацией о структуре файла, достаточной для того, чтобы выделить любую запись. ФС предоставляет приложению доступ к записи, а вся дальнейшая обработка данных, содержащихся в этой записи, выполняется приложением. Развитием этого подхода стали системы управления базами данных (СУБД), которые поддерживают не только сложную структуру данных, но и взаимосвязи между ними.

Логическая запись является наименьшим элементом данных, которым может оперировать программист при организации обмена с внешним устройством. Даже если физический обмен с устройством осуществляется большими единицами, операционная система должна обеспечивать программисту доступ к отдельной логической записи.

Файловая система может использовать два способа доступа к логическим записям: читать или записывать логические записи последовательно (последовательный доступ) или позиционировать файл на запись с указанным номером (прямой доступ).

Очевидно, что ОС не может поддерживать все возможные способы структурирования данных в файле, поэтому в тех ОС, в которых вообще существует поддержка логической структуризации файлов, она существует для небольшого числа широко распространенных схем логической организации файла.

К числу таких способов структуризации относится представление данных в виде записей, длина которых фиксирована в пределах файла (рис. 4.15, а). В таком случае доступ к $n$-й записи осуществляется либо путем последовательного чтения ($n$-1) предшествующих записей, либо прямо по адресу, вычисленному по ее порядковому номеру. Например, если $L$ – длина записи, то начальный адрес $n$-й записи равен $L \times n$. Заметим, что при такой логической организации размер записи фиксирован в пределах файла, а записи в различных файлах, принадлежащих одной и той же файловой системе, могут иметь различный размер.

| Индекс | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| Адрес | 21 | 201 | 315 | 661 | 670 | 715 |

Рис. 4.15. Способы логической организации файлов

Другой способ структуризации состоит в представлении данных в виде последовательности записей, размер которых изменяется в пределах одного файла. Если расположить значения длин записей так, как это показано на рис. 4.15, б, то для поиска нужной записи система должна последовательно считать все предшествующие записи. Вычислить адрес нужной записи по ее номеру при такой логической организации файла невозможно, а следовательно, не может быть применен более эффективный метод прямого доступа.

Файлы, доступ к записям которых осуществляется последовательно, по номерам позиций, называются неиндексированными, или последовательными.

Другим типом файлов являются индексированные файлы, они допускают более быстрый прямой доступ к отдельной логической записи. В индексированном файле (рис. 4.15, в) записи имеют одно или более ключевых (индексных) полей и могут адресоваться путем указания значений этих полей. Для быстрого поиска данных в индексированном файле предусматривается специальная индексная таблица, в которой значениям ключевых полей ставится в соответствие адрес внешней памяти. Этот ад-

рес может указывать либо непосредственно на искомую запись, либо на некоторую область внешней памяти, занимаемую несколькими записями, в число которых входит искомая запись. В последнем случае говорят, что файл имеет индексно-последовательную организацию, так как поиск включает два этапа: прямой доступ по индексу к указанной области диска, а затем последовательный просмотр записей в указанной области. Ведение индексных таблиц берет на себя файловая система. Понятно, что записи в индексированных файлах могут иметь произвольную длину.

Все вышесказанное в большей степени относится к обычным файлам, которые могут быть как структурированными, так и неструктурированными. Что же касается других типов файлов, то они обладают определенной структурой, известной файловой системе. Например, файловая система должна понимать структуру данных, хранящихся в файле-каталоге или файле типа “символьная связь”.

## 5. Физическая организация файловой системы

### 5.1. Структура магнитного диска

Представление пользователя о файловой системе как об иерархически организованном множестве информационных объектов имеет мало общего с порядком хранения файлов на диске. Файл, имеющий образ цельного, непрерывающегося набора байт, на самом деле очень часто разбросан “кусочками” по всему диску, причем это разбиение никак не связано с логической структурой файла, например, его отдельная логическая запись может быть расположена в несмежных секторах диска. Логически объединенные файлы из одного каталога совсем не обязаны соседствовать на диске. Принципы размещения файлов, каталогов и системной информации на реальном устройстве описываются физической организацией файловой системы. Очевидно, что разные файловые системы имеют разную физическую организацию.

Основным типом устройства, которое используется в современных вычислительных системах для хранения файлов, являются дисковые накопители. Эти устройства предназначены для считывания и записи данных на жесткие и гибкие магнитные диски. Жесткий диск состоит из одной или нескольких стеклянных или металлических пластин, каждая из которых покрыта с одной или двух сторон магнитным материалом. Таким образом, диск в общем случае состоит из пакета пластин (рис. 4.16). Однако обычно под термином “жесткий диск” понимают весь пакет магнитных дисков.

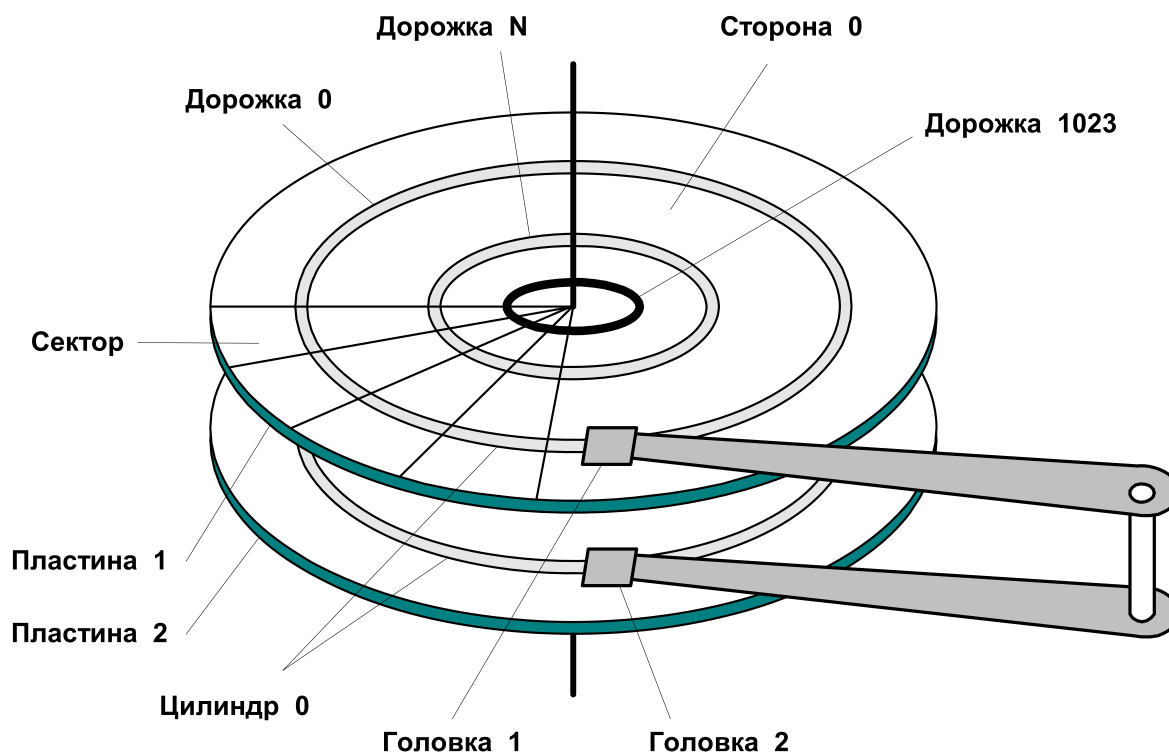



Рис. 4.16. Схема устройства жесткого диска

На каждой стороне каждой пластины размечены тонкие концентрические кольца – *дорожки* (traks), на которых хранятся данные. Количество дорожек зависит от типа диска. Нумерация дорожек начинается с 0 от внешнего края к центру диска. Головка может позиционироваться над заданной дорожкой. Головки перемещаются над поверхностью диска дискретными шагами, каждый шаг соответствует сдвигу на одну дорожку.

Совокупность дорожек одного радиуса на всех поверхностях всех пластин пакета называется *цилиндром* (cylinder). Каждая дорожка разбивается на фрагменты, называемые *секторами* (sectors), или *блоками* (blocks), так что все дорожки имеют равное число секторов, в которые можно максимально записать одно и то же число байт. Сектор имеет фиксированный для конкретной системы размер, выражающийся степенью двойки. Чаще всего размер сектора составляет 512 байт. Учитывая, что дорожки разного радиуса имеют одинаковое число секторов, плотность записи становится тем выше, чем ближе дорожка к центру. Вся совокупность физических секторов на винчестере представляет его неформатированную емкость.

Физический адрес сектора на диске определяется с помощью трех “координат”, то есть представляется триадой [c-h-s], где *c* – номер цилиндра (дорожки на поверхности диска, cylinder), *h* – номер рабочей поверхности диска (магнитной головки, head), а *s* – номер сектора на дорожке. Номер цилиндра *c* лежит в диапазоне 0..C-1, где C – количество цилиндров. Номер рабочей поверхности диска *h* принадлежит диапазону 0..H-1, где H – число магнитных головок в накопителе. Номер сектора на дорожке *s* указывается в диапазоне 1..S, где S – количество секторов на дорожке. На-

пример, триада [1-0-2] адресует сектор 2 на дорожке 0 (обычно верхняя рабочая поверхность) цилиндра 1.

Операционная система при работе с диском использует, как правило, собственную единицу дискового пространства, называемую *кластером* (cluster). Иногда кластер называют блоком (например, в ОС Unix), что может привести к терминологической путанице. Вообще, терминология, используемая при описании форматов дисков и файловых систем, зависит от аппаратной платформы (RISC, Wintel и т. п.) и операционной системы. Это нужно учитывать и трактовать термины в зависимости от контекста.

При создании файла место на диске ему выделяется кластерами. Например, если файл имеет размер 2560 байт, а размер кластера в файловой системе определен в 1024 байта, то файлу будет выделено на диске 3 кластера.

Дорожки и секторы создаются в результате выполнения процедуры физического, или низкоуровневого, форматирования диска, предшествующей использованию диска. Для определения границ блоков на диск записывается идентификационная информация. Низкоуровневый формат диска не зависит от типа операционной системы, которая этот диск будет использовать.

Разметку диска под конкретный тип файловой системы выполняют процедуры высокоуровневого, или логического, форматирования. При высокоуровневом форматировании определяется размер кластера и на диск записывается информация, необходимая для работы файловой системы, в том числе информация о доступном и неиспользуемом пространстве, о границах областей, отведенных под файлы и каталоги, информация о поврежденных областях. Кроме того, на диск записывается загрузчик операционной системы – небольшая программа, которая начинает процесс инициализации операционной системы после включения питания или рестарта компьютера.

Жесткий диск может быть разбит на несколько *разделов* (partition), которые, в принципе затем могут использоваться либо одной ОС, либо различными ОС. Причем самым главным является то, что на каждом разделе может быть организована своя файловая система. Однако для организации даже одной-единственной файловой системы необходимо определить, по крайней мере, один раздел.

Разделы диска могут быть двух типов – *primary* (обычно этот термин переводят как *первичный)* и *extended (расширенный)*. Максимальное число primary-разделов равно четырем. При этом на диске обязательно должен быть по крайней мере один primary-раздел.

Согласно спецификациям на одном жестком диске может быть только один *extended-раздел,* который, в свою очередь, может быть разделен на большое количество подразделов – *логических дисков* (*logical*). Во многих операционных системах используется термин “*том*” (*volume*). В разных ОС толкование этого термина имеет свои нюансы, но чаще всего он обо-

значает логическое устройство, отформатированное под конкретную файловую систему.

Один из primary-разделов должен быть *активным,* именно с него должна загружаться программа загрузки операционной системы, или так называемый *менеджер загрузки,* назначение которого – загрузить программу загрузки ОС из какого-нибудь другого раздела, и уже с ее помощью загружать операционную систему. Поскольку до загрузки ОС система управления файлами работать не может, то следует использовать для указания упомянутых загрузчиков исключительно абсолютные адреса в формате [c-h-s].

По физическому адресу [0-0-1] на винчестере располагается *главная загрузочная запись* (master boot record, MBR), содержащая *внесистемный загрузчик* (non-system bootstrap – NSB), а также *таблицу разделов* (partition table, PT). Эта запись занимает ровно один сектор, она размещается в памяти, начиная с адреса 0:7C00h, после чего управление передается коду, содержащемуся в этом самом первом секторе магнитного диска. Таким образом, в самом первом (стартовом) секторе физического жесткого диска находится не обычная запись boot record, как на дискете, а master boot record.

MBR является основным средством загрузки с жесткого диска, поддерживаемым BIOS. В MBR находятся три важных элемента:

1. Программа начальной загрузки (non-system bootstrap). Именно она запускается BIOS после успешной загрузки в память первого сектора с MBR и служит для поиска с помощью partition table активного раздела, копирования в оперативную память компьютера загрузчика system bootstrap из выбранного раздела и передачи ему управления, что позволяет осуществить загрузку ОС.

2. Таблица описания разделов диска (partition table). Располагается в MBR по смещению 0x1BE и занимает 64 байта.

3. Сигнатура MBR. Последние два байта MBR должны содержать число AA55h. По наличию этой сигнатуры BIOS проверяет, что первый блок был загружен успешно. Сигнатура эта выбрана не случайно. Ее успешная проверка позволяет установить, что все линии передачи данных могут передавать и нули, и единицы.

Таблица *partition table* описывает размещение и характеристики имеющихся на винчестере разделов. Можно сказать, что эта таблица разделов – одна из наиболее важных структур данных на жестком диске. Если эта таблица повреждена, то не только не будет загружаться операционная система (или одна из операционных систем, установленных на винчестере), но перестанут быть доступными и данные, расположенные на винчестере, особенно если жесткий диск был разбит на несколько разделов.

Упрощенно структура MBR представлена в табл. 4.1.

Таблица 4.1

Структура MBR

| Смещение (Offset) | Размер (Size) (байт) | Содержимое (Contents) |
|---|---|---|
| 0 | 446 | Программа анализа Partition Table и загрузки System Bootstrap с активного раздела жесткого диска |
| +1BEh | 16 | Partition 1 entry (Описатель раздела) |
| +1CEh | 16 | Partition 2 entry |
| +1DEh | 16 | Partition 3 entry |
| +1EEh | 16 | Partition 4 entry |
| +1FEh | 2 | Сигнатура (AA55h) |

Из рисунка видно, что в начале этого сектора располагается программа анализа таблицы разделов и чтения первого сектора из активного раздела диска. Сама таблица partition table располагается в конце MBR, и для описания каждого раздела в этой таблице отводится по 16 байтов. Первым байтом в элементе раздела идет флаг активности раздела boot indicator (0 – не активен, 128 (80H) – активен). Он служит для определения, является ли раздел системным загрузочным и есть ли необходимость производить загрузку операционной системы с него при старте компьютера. Активным может быть только один раздел. За флагом активности раздела следует байт номера головки, с которой начинается раздел. За ним следует два байта, означающие соответственно номер сектора и номер цилиндра загрузочного сектора, где располагается первый сектор загрузчика операционной системы. Затем следует кодовый идентификатор System ID (длиной в один байт), указывающий на принадлежность данного раздела к той или иной операционной системе и установке на нем соответствующей файловой системы. За байтом кода операционной системы расположен байт номера головки конца раздела, за которым идут два байта – номер сектора и номер цилиндра последнего сектора данного раздела. Номера сектора и номер цилиндра секторов в разделах занимают по 6 и 10 бит соответственно (рис. 4.17).

| Биты номера цилиндра | | | | | | | | | | Биты номера сектора | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

Рис. 4.17. Формат записи номера сектора и номера цилиндра

Вслед за сектором MBR размещаются собственно сами разделы (рис. 4.18).

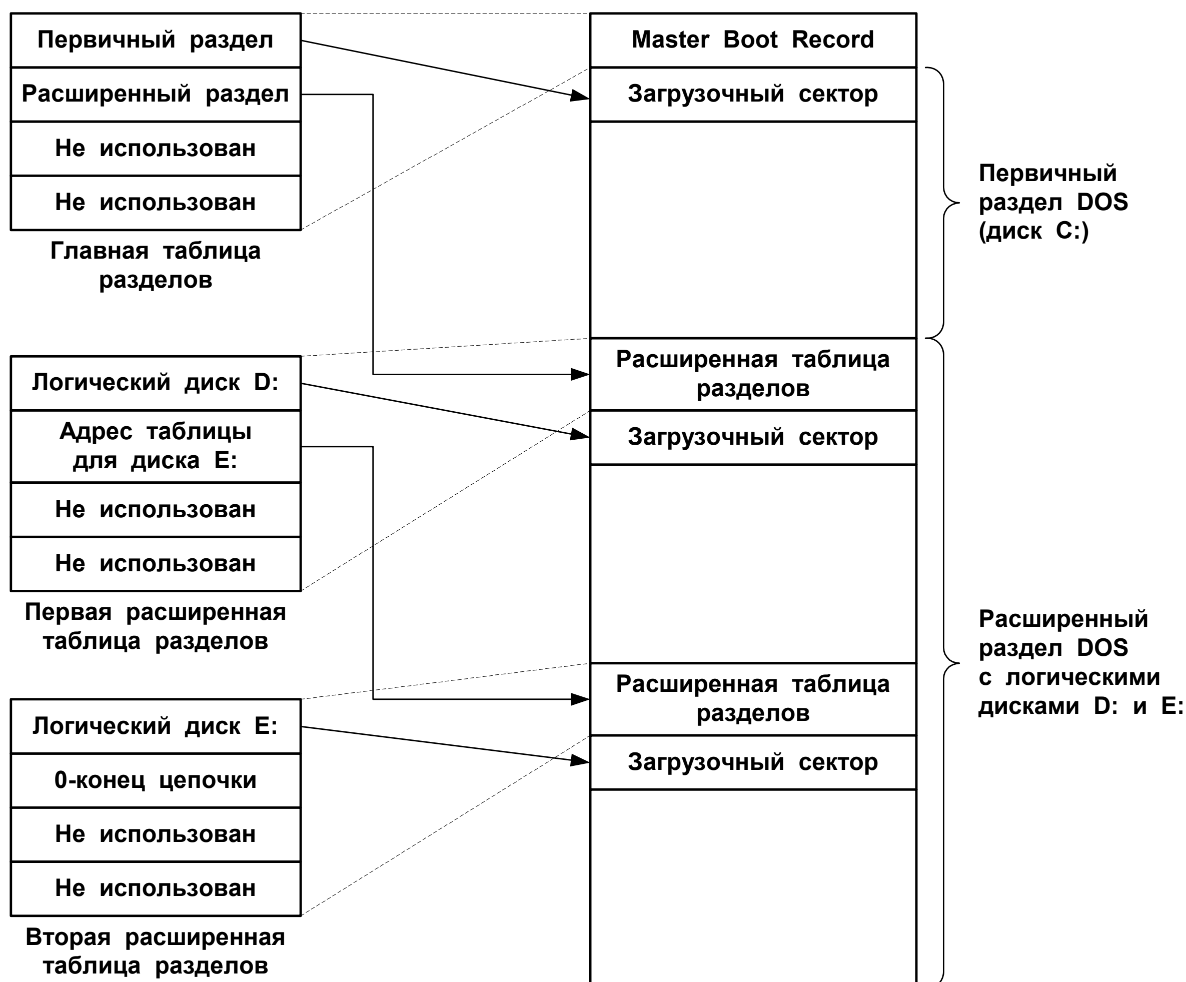



Рис. 4.18. Разбиение диска на разделы

В процессе начальной загрузки сектора MBR, содержащего таблицу partition table, работают программные модули BIOS. Начальная загрузка считается выполненной корректно только в том случае, когда таблица разделов содержит допустимую информацию.

В MS-DOS в первичном разделе может быть сформирован только один логический диск, а в расширенном – любое их количество. Каждый логический диск “управляется” своим логическим приводом. Каждому логическому диску на винчестере соответствует своя (относительная) *логическая* нумерация. Физическая же адресация жесткого диска сквозная.

Первичный раздел DOS включает только *системный логический диск* без каких-либо дополнительных информационных структур.

Расширенный раздел DOS содержит вторичную запись MBR (secondary MBR, SMBR), в состав которой вместо partition table входит таблица логического диска (LDT, logical disk table), ей аналогичная. Таблица LDT описывает размещение и характеристики раздела, содержащего единственный логический диск, а также может специфицировать следующую запись SMBR. Следовательно, если в расширенном разделе DOS создано *K* логических дисков, то он содержит *K* экземпляров SMBR, связанных в список.

Каждый элемент этого списка описывает соответствующий логический диск и ссылается (кроме последнего) на следующий элемент списка.

Утилиты, позволяющие разбить диск на разделы и тем самым сформировать как partition table, так и, возможно, списковую структуру из LDT (подобно тому, как это изображено на рис. 4.18), обычно называются FDISK (от form disk), хотя есть утилиты и с другим названием, умеющие выполнить разбиение диска. Напомним, что FDISK от MS-DOS позволяет создать только один primary-раздел и один extended, который, в свою очередь, предлагается разделить на несколько логических дисков. Аналогичные утилиты других ОС имеют существенно больше возможностей. Так, например, FDISK системы OS/2 позволяет создавать несколько primary-разделов, причем их можно выделять не только последовательно, начиная с первых цилиндров, но и с конца свободного дискового пространства. Это удобно, если нужно исключить из работы некоторый диапазон дискового пространства (например, из-за дефектов на поверхности магнитного диска).

В последнее время появилось большое количество утилит, которые предоставляют возможность более наглядно представить разбиение диска на разделы, поскольку в них используется графический интерфейс. Эти программы успешно и корректно работают с наиболее распространенными типами разделов (разделы под FAT, FAT32, NTFS). Однако созданы они в основном для работы в среде Win32API, что часто ограничивает возможность их использования. Одной из самых известных и мощных программ для работы с разделами жесткого диска является Partition Magic фирмы Power Quest.

## 5.2. Файловые системы современных операционных систем

### 5.2.1. Физическая организация и адресация файла

Важным компонентом физической организации файловой системы является физическая организация файла, то есть способ размещения файла на диске. Основными критериями эффективности физической организации файлов являются:

скорость доступа к данным;

объем адресной информации файла;

степень фрагментированности дискового пространства;

максимально возможный размер файла.

Непрерывное размещение – простейший вариант физической организации (рис. 4.19, *а*), при котором файлу предоставляется последовательность кластеров диска, образующих непрерывный участок дисковой памяти.

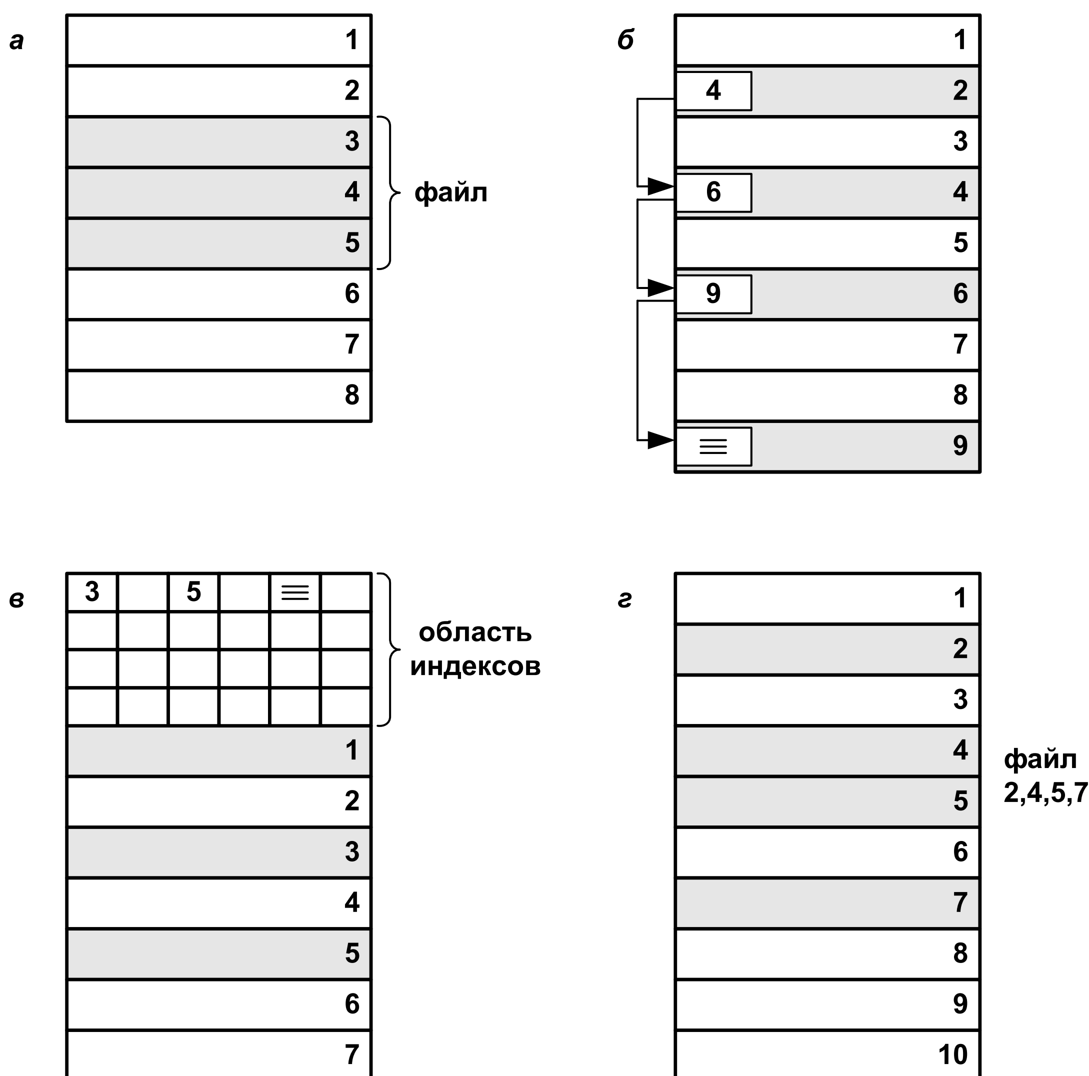



Рис. 4.19. Физическая организация файла: непрерывное размещение (а); связанный список кластеров (б); связанный список индексов (в); перечень номеров кластеров (г)

Основным достоинством этого метода является высокая скорость доступа, так как затраты на поиск и считывание кластеров файла минимальны. Также минимален объем адресной информации – достаточно хранить только номер первого кластера и объем файла. Данная физическая организация максимально возможный размер файла не ограничивает. Однако этот вариант имеет существенные недостатки, которые затрудняют его применимость на практике, несмотря на всю его логическую простоту. При более пристальном рассмотрении оказывается, что реализовать эту схему не так уж просто. Действительно, какого размера должна быть непрерывная область, выделяемая файлу, если файл при каждой модификации может увеличить свой размер? Еще более серьезной проблемой является фрагментация. Спустя некоторое время после создания файловой системы в результате выполнения многочисленных операций создания и удаления файлов пространство диска неминуемо превращается в “лоскут-

ное одеяло", включающее большое число свободных областей небольшого размера. Как всегда бывает при фрагментации, суммарный объем свободной памяти может быть очень большим, а выбрать место для размещения файла целиком невозможно. Поэтому на практике используются методы, в которых файл размещается в нескольких, в общем случае несмежных областях диска.

Следующий способ физической организации – размещение файла в виде связанного списка кластеров дисковой памяти (рис. 4.19, *б*). При таком способе в начале каждого кластера содержится указатель на следующий кластер. В этом случае адресная информация минимальна: расположение файла может быть задано одним числом – номером первого кластера. В отличие от предыдущего способа каждый кластер может быть присоединен к цепочке кластеров какого-либо файла, следовательно, фрагментация на уровне кластеров отсутствует. Файл может изменять свой размер во время своего существования, наращивая число кластеров. Недостатком является сложность реализации доступа к произвольно заданному месту файла – чтобы прочитать пятый по порядку кластер файла, необходимо последовательно прочитать четыре первых кластера, прослеживая цепочку номеров кластеров. Кроме того, при этом способе количество данных файла, содержащихся в одном кластере, не равно степени двойки (одно слово израсходовано на номер следующего кластера), а многие программы читают данные кластерами, размер которых равен степени двойки.

Популярным способом, применяемым, например, в файловой системе FAT, является использование связанного списка индексов (рис. 4.19, *в*). Этот способ является некоторой модификацией предыдущего. Файлу также выделяется память в виде связанного списка кластеров. Номер первого кластера запоминается в записи каталога, где хранятся характеристики этого файла. Остальная адресная информация отделена от кластеров файла. С каждым кластером диска связывается некоторый элемент – индекс. Индексы располагаются в отдельной области диска – в MS-DOS это таблица FAT (File Allocation Table), занимающая один кластер. Когда память свободна, все индексы имеют нулевое значение. Если некоторый кластер N назначен некоторому файлу, то индекс этого кластера становится равным либо номеру M следующего кластера данного файла, либо принимает специальное значение, являющееся признаком того, что этот кластер является для файла последним. Индекс же предыдущего кластера файла принимает значение N, указывая на вновь назначенный кластер.

При такой физической организации сохраняются все достоинства предыдущего способа: минимальность адресной информации, отсутствие фрагментации, отсутствие проблем при изменении размера. Кроме того, данный способ обладает дополнительными преимуществами. Во-первых, для доступа к произвольному кластеру файла не требуется последовательно считывать его кластеры, достаточно прочитать только секторы диска, содержащие таблицу индексов, отсчитать нужное количество кластеров

файла по цепочке и определить номер нужного кластера. Во-вторых, данные файла заполняют кластер целиком, а значит, имеют объем, равный степени двойки.

Еще один способ задания физического расположения файла заключается в простом перечислении номеров кластеров, занимаемых этим файлом (рис. 4.19, *г*). Этот перечень и служит адресом файла. Недостаток данного способа очевиден: длина адреса зависит от размера файла и для большого файла может составить значительную величину. Достоинством же является высокая скорость доступа к произвольному кластеру файла, так как здесь применяется прямая адресация, которая исключает просмотр цепочки указателей при поиске адреса произвольного кластера файла. Фрагментация на уровне кластеров в этом способе также отсутствует.

Последний подход с некоторыми модификациями используется в традиционных файловых системах ОС UNIX s5 и ufs. Для сокращения объема адресной информации прямой способ адресации сочетается с косвенным.

### 5.2.2. Файловая система FAT

Аббревиатура FAT (file allocation table) расшифровывается как “таблица размещения файлов”. Этот термин относится к линейной табличной структуре со сведениями о файлах – именами файлов, их атрибутами и другими данными, определяющими местонахождение файлов (или их фрагментов) в среде FAT. Элемент FAT определяет фактическую область диска, в которой хранится начало физического файла.

В файловой системе FAT логическое дисковое пространство любого логического диска делится на две области (рис. 4.20): *системную область* и *область данных*.

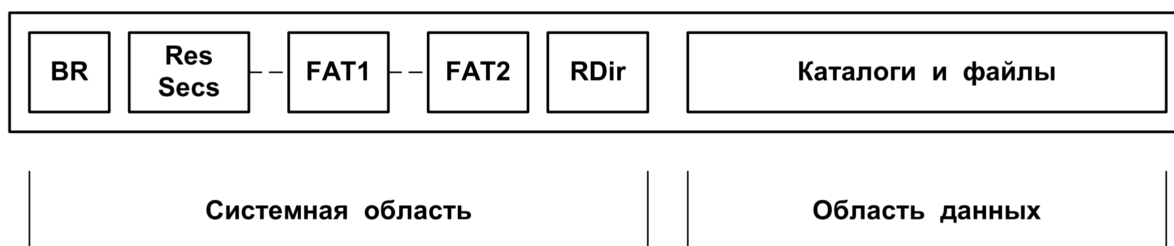



Рис. 4.20. Структура логического диска

Системная область логического диска создается и инициализируется при форматировании, а впоследствии обновляется при манипулировании файловой структурой. Область данных логического диска содержит файлы и каталоги, подчиненные корневому.

Системная область состоит из следующих компонентов, расположенных в логическом адресном пространстве подряд:

загрузочной записи (boot record, BR);

зарезервированных секторов (reserved sector, ResSecs);

таблицы размещения файлов (file allocation table, FAT);

корневого каталога (root directory, RDir).

Файловая система FAT поддерживает всего два типа файлов: обычный файл и каталог. Файловая система распределяет память только из области данных, причем использует в качестве минимальной единицы дискового пространства кластер.

*Кластер* представляет собой один или несколько смежных секторов в логическом дисковом адресном пространстве (точнее – только в области данных). В таблице FAT кластеры, принадлежащие одному файлу, связываются в цепочки. Для указания номера кластера в системе управления файлами FAT-16 используется 16-битовое слово, следовательно, можно иметь до $2^{16}$ = 65536 кластеров (с номерами от 0 до 65535).

Файл или каталог занимает целое число кластеров. Последний кластер при этом может быть задействован не полностью, что приведет к заметной потере дискового пространства при большом размере кластера. На дискетах кластер занимает один или два сектора, а на жестких дисках – в зависимости от объема раздела (табл. 4.2).

Таблица 4.2

Соотношения между размером раздела и размером кластеров в FAT16

| Емкость раздела, Мбайт | Количество секторов в кластере | Размер кластеров, Кбайт |
|---|---|---|
| 16-127 | 4 | 2 |
| 128-255 | 8 | 4 |
| 256-511 | 16 | 8 |
| 512-1023 | 32 | 16 |
| 1024-2047 | 64 | 32 |

Однако слишком большой размер кластера ведет к неэффективному использованию области данных, особенно в случае большого количества маленьких файлов. В среднем на каждый файл теряется около половины кластера. Из табл. 1 следует, что при размере кластера в 32 сектора (объем раздела – от 512 Мбайт до 1023 Мбайт), то есть 16 Кбайт, средняя величина потерь на файл составит 8 Кбайт, и при числе файлов в несколько тысяч потери могут составлять более 100 Мбайт. Поэтому в современных файловых системах (к ним, прежде всего, следует отнести HPFS, NTFS, FAT32 и некоторые другие, поддерживаемые ОС семейства UNIX) размеры кластеров ограничиваются (обычно – от 512 байт до 4 Кбайт). В FAT32 проблема решается за счет того, что собственно сама FAT в этой файловой системе может содержать до $2^{28}$ кластеров (в 32-битовом слове, используемом для представления номера кластера, фактически учитываются

только 28 разрядов). Также следует отметить, что системы управления файлами, созданные для Windows 9x и Windows NT, могут работать с разделами размером до 4 Гбайт, на которых установлена система FAT, тогда как DOS, естественно, с такими разделами работать не сможет.

*Таблица размещения файлов*

Таблица размещения файлов является очень важной информационной структурой. Она (как основная копия, так и резервная) состоит из массива индексных указателей, количество которых равно количеству кластеров области данных. Между кластерами и индексными указателями имеется взаимно однозначное соответствие – нулевой указатель соответствует нулевому кластеру и т. д.

Индексный указатель может принимать следующие значения, характеризующие состояние связанного с ним кластера:

кластер свободен (не используется);

кластер используется файлом и не является последним кластером файла; в этом случае индексный указатель содержит номер следующего кластера файла;

последний кластер файла;

дефектный кластер;

резервный кластер.

Таблица FAT является общей для всех файлов раздела. В исходном состоянии (после форматирования) все кластеры раздела свободны и все индексные указатели (кроме тех, которые соответствуют резервным и дефектным блокам) принимают значение “кластер свободен”. При размещении файла ОС просматривает FAT, начиная с начала, и ищет первый свободный индексный указатель. После его обнаружения в поле записи каталога “номер первого кластера” фиксируется номер этого указателя. В кластер с этим номером записываются данные файла, он становится первым кластером файла. Если файл умещается в одном кластере, то в указатель, соответствующий данному кластеру, заносится специальное значение “последний кластер файла”. Если же размер файла больше одного кластера, то ОС продолжает просмотр FAT и ищет следующий указатель на свободный кластер. После его обнаружения в предыдущий указатель заносится номер этого кластера, который теперь становится следующим кластером файла. Процесс повторяется до тех пор, пока не будут размещены все данные файла. Таким образом создается связный список всех кластеров файла.

Достаточно наглядно идея файловой системы с использованием таблицы размещения файлов FAT проиллюстрирована рис. 4.21. Из этого рисунка видно, что файл с именем MYFILE.TXT размещается, начиная с восьмого кластера. Всего файл MYFILE.TXT занимает 12 кластеров. Цепочка кластеров (chain) для нашего примера может быть записана следующим образом: 08, 09, 0A, 0B, 15, 16, 17, 19, 1A, 1B, 1C, 1D. Кластер с номером 18 помечен специальным кодом F7 как плохой (bad), он не может

быть использован для размещения данных. При форматировании обычно проверяется поверхность магнитного диска, и те секторы, при контрольном чтении с которых происходили ошибки, помечаются в FAT как плохие. Кластер 1D помечен кодом FF как конечный (последний в цепочке) кластер, принадлежащий данному файлу. Свободные (незанятые) кластеры помечаются кодом 00; при выделении нового кластера для записи файла берется первый свободный кластер.

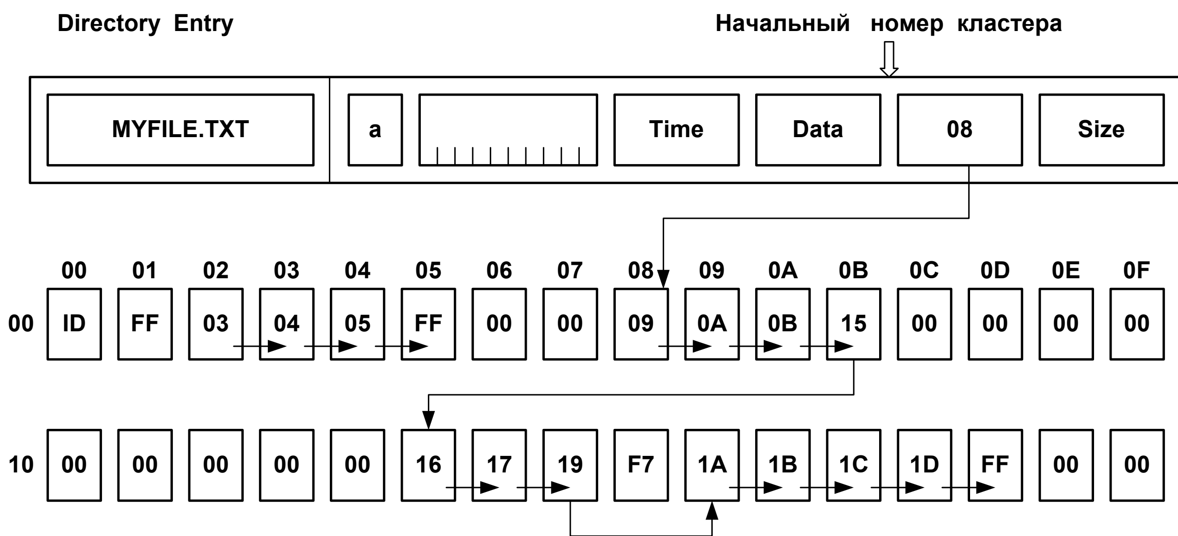



Рис. 4.21. Основная концепция FAT

Поскольку файлы на диске изменяются – удаляются, перемещаются, увеличиваются или уменьшаются, – то упомянутое правило выделения первого свободного кластера для новой порции данных приводит к *фрагментации* файлов, то есть данные одного файла могут располагаться не в смежных кластерах, а, порой, в очень удаленных друг от друга, образуя сложные цепочки. Естественно, что это приводит к существенному замедлению работы с файлами.

Так как FAT используется при доступе к диску очень интенсивно, она обычно загружается в ОЗУ (в буфера ввода/вывода или кэш) и остается там настолько долго, насколько это возможно.

В связи с чрезвычайной важностью FAT она обычно хранится в двух идентичных экземплярах, второй из которых непосредственно следует за первым. Обновляются копии FAT одновременно. Используется же только первый экземпляр. Если он по каким-либо причинам окажется разрушенным, то произойдет обращение ко второму экземпляру. Так, например, утилита проверки и восстановления файловой структуры ScanDisk из ОС Windows 9x при обнаружении несоответствия первичной и резервной копии FAT предлагает восстановить главную таблицу, используя данные из копии.

Упомянутый корневой каталог отличается от обычного каталога тем, что он, помимо размещения в фиксированном месте логического диска, еще имеет и фиксированное число элементов. Для каждого файла и каталога в файловой системе хранится информация в соответствии со структурой, изображенной в табл. 4.3.

*Файловые системы VFAT и FAT32*

Одной из важнейших характеристик исходной FAT было использование имен файлов формата “8.3”, в котором 8 символов отводится на указание имени файла и 3 символа – для расширения имени.

Таблица 4.3

Элемент каталога

| Размер поля данных, байт | Содержание поля |
|---|---|
| 11 | Имя файла или каталога |
| 1 | Атрибуты файла |
| 1 | Резервное поле |
| 3 | Время создания |
| 2 | Дата создания |
| 2 | Дата последнего доступа |
| 2 | Зарезервировано |
| 2 | Время последней модификации |
| 2 | Дата последней модификации |
| 2 | Номер начального кластера в FAT |
| 4 | Размер файла |

К стандартной FAT (имеется в виду прежде всего реализация FAT16) добавились еще две разновидности, используемые в широко распространенных операционных системах Microsoft (конкретно – в Windows 95 и Windows NT): VFAT (виртуальная FAT) и FAT32, используемая в одной из редакций ОС Windows 95 и Windows 98. Ныне эта файловая система (FAT32) поддерживается и такими ОС, как Windows Millennium Edition, и всеми ОС семейства Windows 2000. Имеются реализации систем управления файлами FAT32 для Windows NT и ОС Linux.

Файловая система VFAT впервые появилась в Windows for Workgroups 3.11 и была предназначена для выполнения файлового ввода/вывода в защищенном режиме. С выходом Windows 95 в VFAT добавилась поддержка длинных имен файлов (long file name, LFN). Тем не менее, VFAT сохраняет совместимость с исходным вариантом FAT; это означает, что наряду с длинными именами в ней поддерживаются имена формата “8.3”, а также существует специальный механизм для преобразования имен “8.3” в длинные имена, и наоборот. Именно файловая система VFAT поддерживается исходными версиями Windows 95, Windows NT 4. При работе с VFAT крайне важно использовать файловые утилиты, поддерживающие

VFAT вообще и длинные имена в частности. Дело в том, что более ранние файловые утилиты DOS запросто модифицируют то, что кажется им исходной структурой FAT. Это может привести к потере или порче длинных имен из таблицы FAT, поддерживаемой VFAT (или FAT32). Следовательно, для томов VFAT необходимо пользоваться файловыми утилитами, которые понимают и сохраняют файловую структуру VFAT.

В исходной версии Windows 95 основной файловой системой была 32-разрядная VFAT. VFAT может использовать 32-разрядные драйверы защищенного режима или 16-разрядные драйверы реального режима. При этом элементы FAT остаются 12- или 16-разрядными, поэтому на диске используется та же структура данных, что и в предыдущих реализациях FAT. VFAT обрабатывает все обращения к жесткому диску и использует 32-разрядный код для всех файловых операций с дисковыми томами.

Основными недостатками файловых систем FAT и VFAT являются большие потери на кластеризацию при больших размерах логического диска и ограничения на сам размер логического диска. Это привело к разработке новой реализации файловой системы с использованием той же идеи использования таблицы FAT. Поэтому в Microsoft Windows 95 OEM Service Release 2 (эта версия Windows 95 часто называется Windows 95 OSR2) на смену системе VFAT пришла файловая система FAT32. FAT32 является полностью самостоятельной 32-разрядной файловой системой и содержит многочисленные усовершенствования и дополнения по сравнению с предыдущими реализациями FAT.

Принципиальное отличие заключается в том, что FAT32 намного эффективнее расходует дисковое пространство. Прежде всего, система FAT32 использует кластеры меньшего размера по сравнению с предыдущими версиями, которые ограничивались 65 535 кластерами на том (соответственно, с увеличением размера диска приходилось увеличивать и размер кластеров). Следовательно, даже для дисков размером до 8 Гбайт FAT32 может использовать 4-килобайтные кластеры. В результате по сравнению с дисками FAT16 экономится значительное дисковое пространство (в среднем 10-15 %).

FAT32 также может перемещать корневой каталог и использовать резервную копию FAT вместо стандартной. Расширенная загрузочная запись FAT32 позволяет создавать копии критических структур данных; это повышает устойчивость дисков FAT32 к нарушениям структуры FAT по сравнению с предыдущими версиями. Корневой каталог в FAT32 представлен в виде обычной цепочки кластеров. Следовательно, корневой каталог может находиться в произвольном месте диска, что снимает действовавшее ранее ограничение на размер корневого каталога (512 элементов).

Windows 95 OSR2 и Windows 98 могут работать и с разделами VFAT, созданными Windows NT. То, что говорилось ранее об использовании файловых утилит VFAT с томами VFAT, относится и к FAT32. Поскольку

прежние утилиты FAT (для FAT32 в эту категорию входят обе файловые системы, FAT и VFAT) могут повредить или уничтожить важную служебную информацию, для томов FAT32 нельзя пользоваться никакими файловыми утилитами, кроме утилит FAT32.

Кроме повышения емкости FAT до величины в 4 Тбайт, файловая система FAT32 вносит ряд необходимых усовершенствований в структуру корневого каталога. Предыдущие реализации требовали, чтобы вся информация корневого каталога FAT находилась в одном дисковом кластере. При этом корневой каталог мог содержать не более 512 файлов. Необходимость представлять длинные имена и обеспечить совместимость с прежними версиями FAT привела разработчиков компании Microsoft к компромиссному решению: для представления длинного имени они стали использовать элементы каталога, в том числе и корневого.

### 5.2.3. Файловая система NTFS

В название файловой системы NTFS (New Technology File System) входят слова “New Technology”, то есть “новая технология”. Действительно, NTFS содержит ряд значительных усовершенствований и изменений, существенно отличающих ее от других файловых систем. С точки зрения пользователей, файлы по-прежнему хранятся в каталогах (часто называемых “папками” в среде Windows). Однако в NTFS, в отличие от FAT, работа на дисках большого объема происходит намного эффективнее; имеются средства для ограничения в доступе к файлам и каталогам, введены механизмы, существенно повышающие надежность файловой системы, сняты многие ограничения на максимальное количество дисковых секторов и/или кластеров.

*Основные возможности файловой системы NTFS*

При проектировании системы NTFS особое внимание было уделено следующим характеристикам:

*Надежность.* Высокопроизводительные компьютеры и системы совместного пользования (серверы) должны обладать повышенной надежностью, которая является ключевым элементом структуры и поведения NTFS. Одним из способов увеличения надежности является введение механизма транзакций, при котором осуществляется *журналирование* файловых операций;

*Расширенная функциональность.* NTFS проектировалась с учетом возможного расширения. В ней были воплощены многие дополнительные возможности – усовершенствованная отказоустойчивость, эмуляция других файловых систем, мощная модель безопасности, параллельная обработка потоков данных и создание файловых атрибутов, определяемых пользователем;

*Поддержка POSIX.* К числу базовых средств файловой системы POSIX относится необязательное использование имен файлов с учетом регистра, хранение времени последнего обращения к файлу и механизм так называемых “жестких ссылок” – альтернативных имен, позволяющих ссылаться на один и тот же файл по двум и более именам;

*Гибкость.* Модель распределения дискового пространства в NTFS отличается чрезвычайной гибкостью. Размер кластера может изменяться от 512 байт до 64 Кбайт; он представляет собой число, кратное внутреннему кванту распределения дискового пространства. NTFS также поддерживает длинные имена файлов, набор символов Unicode и альтернативные имена формата 8.3 для совместимости с FAT.

NTFS превосходно справляется с обработкой больших массивов данных и достаточно хорошо проявляет себя при работе с томами объемом от 300-400 Мбайт и выше. Максимально возможные размеры тома (и размеры файла) составляют 16 Эбайт (один экзабайт равен $2^{64}$, или приблизительно 16 000 млрд гигабайт). Количество файлов в корневом и некорневом каталогах не ограничено. Поскольку в основу структуры каталогов NTFS заложена эффективная структура данных, называемая “бинарным деревом”, время поиска файлов в NTFS (в отличие от систем на базе FAT) не связано линейной зависимостью с их количеством.

Система NTFS также обладает определенными средствами самовосстановления. NTFS поддерживает различные механизмы проверки целостности системы, включая ведение журналов транзакций, позволяющих воспроизвести файловые операции записи по специальному системному журналу.

Файловая система NTFS поддерживает объектную модель безопасности NT и рассматривает все тома, каталоги и файлы как самостоятельные объекты. NTFS обеспечивает безопасность на уровне файлов; это означает, что права доступа к томам, каталогам и файлам могут зависеть от учетной записи пользователя и тех групп, к которым он принадлежит. Каждый раз, когда пользователь обращается к объекту файловой системы, его права доступа проверяются по списку разрешений данного объекта. Если пользователь обладает достаточным уровнем прав, его запрос удовлетворяется; в противном случае запрос отклоняется. Эта модель безопасности применяется как при локальной регистрации пользователей на компьютерах с NT, так и при удаленных сетевых запросах.

*Структура тома с файловой системой NTFS*

Одним из основных понятий, используемых при работе с NTFS, является понятие *тома* (volume). Возможно также создание отказоустойчивого тома, занимающего несколько разделов, то есть использование RAID-технологии. Как и многие другие системы, NTFS делит все полезное дисковое пространство тома на кластеры и поддерживает размеры кластеров от 512 байт до 64 Кбайт; стандартом же считается кластер размером 2 или 4 Кбайт.

Все дисковое пространство в NTFS делится на две неравные части (рис. 4.22). Первые 12 % диска отводятся под так называемую MFT-зону – пространство, которое может занимать, увеличиваясь в размере, главный служебный *метафайл* MFT – специальный файл, главная системная структура данных, которая и позволяет определять местонахождение всех остальных файлов. Запись каких-либо данных в эту область невозможна. MFT-зона всегда держится пустой – это делается для того, чтобы самый главный, служебный файл (MFT) по возможности не фрагментировался при своем росте. Остальные 88 % тома представляют собой обычное пространство для хранения файлов.

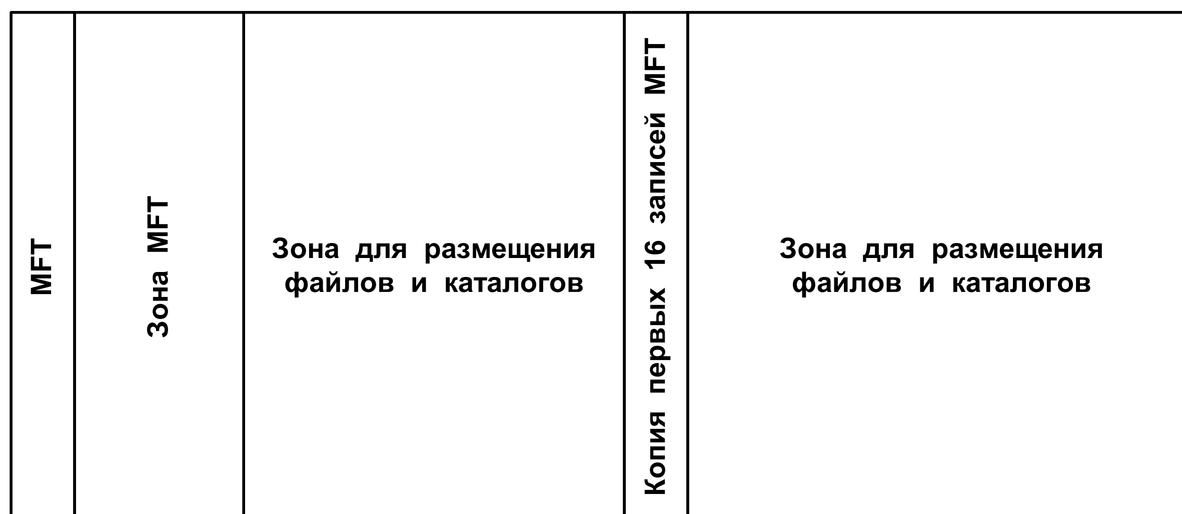



Рис. 4.22. Структура тома NTFS

MFT (master file table, общая таблица файлов) представляет собой централизованный каталог всех остальных файлов диска, в том числе и себя самого. MFT поделен на записи фиксированного размера в 1 Кбайт, и каждая запись соответствует какому-либо файлу (в общем смысле этого слова). Первые 16 файлов носят служебный характер и недоступны операционной системе – они называются *метафайлами,* причем самый первый метафайл – сам MFT. Эти первые 16 элементов MFT – единственная часть диска, имеющая строго фиксированное положение. Копия этих же 16 записей хранится в середине тома для надежности, поскольку они очень важны. Остальные части MFT-файла могут располагаться, как и любой другой файл, в произвольных местах диска – восстановить его положение можно с помощью его самого, “зацепившись” за самую основу – за первый элемент MFT.

Упомянутые первые 16 файлов NTFS (метафайлы) носят служебный характер; каждый из них отвечает за какой-либо аспект работы системы. Метафайлы находятся в корневом каталоге NTFS-тома. Все они начинаются с символа имени “$”, хотя получить какую-либо информацию о них стандартными средствами сложно. В табл. 4.4 приведены основные из-

вестные метафайлы и их назначение. Таким образом, можно узнать, например, сколько операционная система тратит на каталогизацию тома, посмотрев размер файла $MFT.

Таблица 4.4

Метафайлы NTFS

| Имя метафайла | Назначение метафайла |
|---|---|
| $MFT | Сам Master File Table |
| $MFTmirr | Копия первых 16 записей MFT, размещенная посередине тома |
| $LogFile | Файл поддержки операций журналирования |
| $Volume | Служебная информация – метка тома, версия файловой системы и т. д. |
| $AttrDef | Список стандартных атрибутов файлов на томе |
| $. | Корневой каталог |
| $Bitmap | Карта свободного места тома |
| $Boot | Загрузочный сектор (если раздел загрузочный) |
| $Quota | Файл, в котором записаны права пользователей на использование дискового пространства (этот файл начал работать лишь в Windows 2000 с системой NTFS 5.0) |
| $Upcase | Файл – таблица соответствия заглавных и прописных букв в именах файлов. В NTFS имена файлов записываются в Unicode (что составляет 65 тысяч различных символов) и искать большие и малые эквиваленты в данном случае – нетривиальная задача |

Итак, все файлы тома упоминаются в MFT. В этой структуре хранится вся информация о файлах, за исключением собственно данных. Имя файла, размер, положение на диске отдельных фрагментов и т. д. – все это хранится в соответствующей записи. Если для информации не хватает одной записи MFT, то используется несколько записей, причем не обязательно идущих подряд. Файлы могут иметь не очень большой размер. Тогда применяется довольно удачное решение: данные файла хранятся прямо в MFT, в оставшемся от основных данных месте в пределах одной записи MFT. Файлы, занимающие сотни байт, обычно не имеют своего “физического” воплощения в основной файловой области – все данные такого файла хранятся в одном месте, в MFT.

Файл в томе с NTFS идентифицируется так называемой файловой ссылкой (File Reference), которая представляется как 64-разрядное число. Файловая ссылка состоит из номера файла, который соответствует позиции его файловой записи в MFT, и номера последовательности. Последний увеличивается всякий раз, когда данная позиция в MFT используется

повторно, что позволяет файловой системе NTFS выполнять внутренние проверки целостности.

Каждый файл в NTFS представлен с помощью *потоков* (streams), то есть у него нет как таковых “просто данных”, а есть “потоки”. Для правильного понимания потока достаточно указать, что один из потоков и носит привычный нам смысл – данные файла. Но большинство атрибутов файла – это тоже потоки. Таким образом, получается, что базовая сущность у файла только одна – номер в MFT, а все остальное, включая и его потоки, – опционально. Данный подход может эффективно использоваться – например, файлу можно “прилепить” еще один поток, записав в него любые данные. В Windows 2000 таким образом записана информация об авторе и содержании файла (одна из закладок в свойствах файла, просматриваемых, например, из проводника). Интересно, что эти дополнительные потоки не видны стандартными средствами работы с файлами: наблюдаемый размер файла – это лишь размер основного потока, который содержит традиционные данные. Можно, к примеру, иметь файл нулевой длины, при стирании которого освободится 1 Гбайт свободного места – просто потому, что какая-нибудь хитрая программа или технология “прилепила” к нему дополнительный поток (альтернативные данные) такого большого размера. Но на самом деле в настоящее время потоки практически не используются, так что опасаться подобных ситуаций не следует, хотя гипотетически они возможны. Просто необходимо иметь в виду, что файл в NTFS – это более глубокое понятие, чем можно себе представить, просматривая каталоги диска.

Стандартные же атрибуты для файлов и каталогов в томе NTFS имеют фиксированные имена и коды типа, они перечислены в табл. 4.5.

Таблица 4.5

Атрибуты файлов в системе NTFS

| Системный атрибут | Описание атрибута |
|---|---|
| Стандартная информация о файле | Традиционные атрибуты Read Only, Hidden, Archive, System, отметки времени, включая время создания или последней модификации, число каталогов, ссылающихся на файл |
| Список атрибутов | Список атрибутов, из которых состоит файл, и файловая ссылка на файловую запись и MFT, в которой расположен каждый из атрибутов. Последний используется, если файлу необходимо более одной записи в MFT |

Окончание таблицы 2.1

| Системный атрибут | Описание атрибута |
|---|---|
| Имя файла | Имя файла в символах Unicode. Файл может иметь несколько атрибутов – имен файла, подобно тому как это имеет место в UNIX-системах. Это случается, когда имеется связь POSIX с данным файлом или если у файла есть автоматически сгенерированное имя в формате 8.3 |
| Дескриптор защиты | Структура данных защиты (ACL), предохраняющая файл от несанкционированного доступа. Атрибут “дескриптор защиты” определяет, кто владелец файла и кто имеет доступ к нему |
| Данные | Собственно данные файла, его содержимое. В NTFS у файла по умолчанию есть один безымянный атрибут данных, и он может иметь дополнительные именованные атрибуты данных. У каталога нет атрибута данных по умолчанию, но он может иметь необязательные именованные атрибуты данных |
| Корень индекса, размещение индекса, битовая карта (только для каталогов) | Атрибуты, используемые для индексов имен файлов в больших каталогах |
| Расширенные атрибуты HPFS | Атрибуты, используемые для реализации расширенных атрибутов HPFS для подсистемы OS/2 и OS/2-клиентов файл-серверов Windows NT |

Атрибуты файла в записях MFT расположены в порядке возрастания числовых значений кодов типа, причем некоторые типы атрибутов могут встречаться в записи более одного раза: например, если у файла есть несколько атрибутов данных или несколько имен. Обязательными для каждого файла в томе NTFS являются атрибут стандартной информации, атрибут имени файла, атрибут дескриптора защиты и атрибут данных. Остальные атрибуты могут встречаться при необходимости.

Имя файла в NTFS, в отличие от файловых систем FAT и HPFS, может содержать любые символы, включая полный набор национальных алфавитов, так как данные представлены в Unicode – 16-битном представлении, которое дает 65 535 разных символов. Максимальная длина имени файла в NTFS – 255 символов.

Большой вклад в эффективность файловой системы вносит организация каталога. Каталог в NTFS представляет собой специальный файл, храня-

щий ссылки на другие файлы и каталоги, создавая иерархическое строение данных на диске. Файл каталога поделен на блоки, каждый из которых содержит имя файла, базовые атрибуты и ссылку на элемент MFT, который уже предоставляет полную информацию об элементе каталога. Главный каталог диска – корневой – ничем не отличается от обычных каталогов, кроме специальной ссылки на него из начала метафайла MFT.

*Основные отличия FAT и NTFS*

Если говорить о накладных расходах на хранение служебной информации, FAT отличается от NTFS большей компактностью и меньшей сложностью. В большинстве томов FAT на хранение таблицы размещения, содержащей информацию обо всех файлах тома, расходуется менее 1 Мбайт. Столь низкие накладные расходы позволяют форматировать в FAT жесткие диски малого объема и флоппи-диски. В NTFS служебные данные занимают больше места, чем в FAT. Так каждый элемент каталога занимает 2 Кбайт. Однако это имеет и свои преимущества, так как содержимое файлов объемом 1500 байт и менее может полностью храниться в элементе каталога.

Система NTFS не может использоваться для форматирования флоппи-дисков. Не стоит пользоваться ею для форматирования разделов объемом менее 50-100 Мбайт. Относительно высокие накладные расходы приводят к тому, что для малых разделов служебные данные могут занимать до 25 % объема носителя.

Следующий критерий сравнения – размер файлов. Разделы FAT имеют объем до 2 Гбайт, VFAT – до 4 Гбайт и FAT32 – до 4 Тбайт. Тем не менее, из-за особенностей своего внутреннего строения разделы FAT лучше всего работают для разделов объемом 200 Мбайт и менее. Разделы NTFS могут достигать 16 Эбайт, однако в настоящее время из-за аппаратных и других системных причин размер файлов ограничивается 2 Тбайт.

Разделы FAT могут использоваться практически во всех операционных системах. За редкими исключениями, с разделами NTFS можно работать напрямую только из Windows NT, хотя и имеются для ряда ОС соответствующие реализации систем управления файлами для чтения файлов из томов NTFS. Так, например, утилита (драйвер) NTFSDOS позволяет читать данные NTFS на компьютере, загруженном в режиме MS-DOS. Однако полноценных реализаций для работы с NTFS вне системы Windows NT пока нет.

Разделы FAT не обеспечивают локальной безопасности. С другой стороны, разделы NTFS обеспечивают локальную безопасность как файлов, так и каталогов. Для разделов FAT могут устанавливаться общие права, связанные с общим доступом к каталогам в сети. Однако такая защита не помешает пользователю с локальным входом получить доступ к файлам своего компьютера. В отношении безопасности NTFS оказывается предпочтительным вариантом. Разделы NTFS могут запрещать или ограничивать доступ как удаленных, так и локальных пользователей. Следователь-

но, к защищенным файлам смогут обратиться лишь те пользователи, которым были предоставлены соответствующие права.

Windows NT содержит специальную утилиту CONVERT.EXE, которая преобразует тома FAT в эквивалентные тома NTFS, однако для обратного преобразования (из NTFS в FAT) подобных утилит не существует. Чтобы выполнить такое обратное преобразование, необходимо создать раздел FAT, скопировать в него файлы из раздела NTFS и затем удалить оригиналы. Важно при этом не забывать и о том, что при копировании файлов из NTFS в FAT теряются все атрибуты безопасности NTFS (напомним, что в FAT не предусмотрены средства для определения и последующего хранения этих атрибутов).

В последнее время появилось еще одно очень важное обстоятельство, связанное с тем, что объемы дисковых механизмов намного превысили максимально допустимый размер, приемлемый для FAT, – 8,4 Гбайт. Этот предел объясняется максимально возможными значениями в адресе сектора, для которого, как мы уже знаем, отводится всего 3 байта. Поэтому в подавляющем большинстве случаев при работе в среде Windows-систем используют либо FAT32, либо NTFS. Последняя, безусловно, лучше, но она не поддерживается в широко распространенных ОС Windows 9x.

# ГЛАВА 5

# ЗАЩИТА ПАМЯТИ И ДАННЫХ

## 1. Методы защиты памяти и данных

Основной целью защиты является обеспечение сохранности и предотвращение несанкционированного использования данных, хранящихся в машине.

ЭВМ, как правило, работают в мультипрограммном режиме, т.е. одновременно выполняется несколько программ, принадлежащих одному или нескольким разным заданиям. Поэтому необходимы специальные способы защиты оперативной памяти, запрещающие одной программе считывать или записывать данные в область памяти, отведенную другой программе. Защиту от влияния одних программ на другие называют защитой памяти.

Прикладные программы пользователей с помощью средств файловой системы обращаются к различным файлам данных, имеющих ограничения на использование этих файлов. В связи с этим существуют специальные способы защиты от несанкционированного использования файлов. Защита от несанкционированного использования файлов получила название защиты данных.

### 1.1. Защита памяти

Сущность защиты памяти заключается в проверке правильности формирования исполнительных адресов при каждом обращении к оперативной памяти и выработке необходимых сигналов по результатам проверки.

В современных ЭВМ защита памяти осуществляется в трех основных режимах: защита от записи, защита от считывания, защита от записи и от считывания. Управляющие и прикладные программы, как правило, защищают как от записи, так и от считывания. Таблицы констант, стандартные программы должны быть защищены только от записи.

Проверка правильности формирования исполнительных адресов осуществляется непосредственно в ходе вычислений и требует определенных

затрат времени. Чтобы эти затраты времени были небольшими, защита памяти организуется с использованием аппаратурных средств.

К простейшим способам защиты памяти, нашедшим применение в современных ЭВМ, относятся защита по граничным адресам и защита по ключам.

### 1.1.1. Защита по граничным адресам

Защита по граничным адресам основана на использовании граничных регистров, вводимых в состав процессора или канала ввода-вывода. Программе выделяется область памяти, которая задается двумя значениями $S_1$ и $S_2$ граничных адресов выделенной области (рис. 5.1).

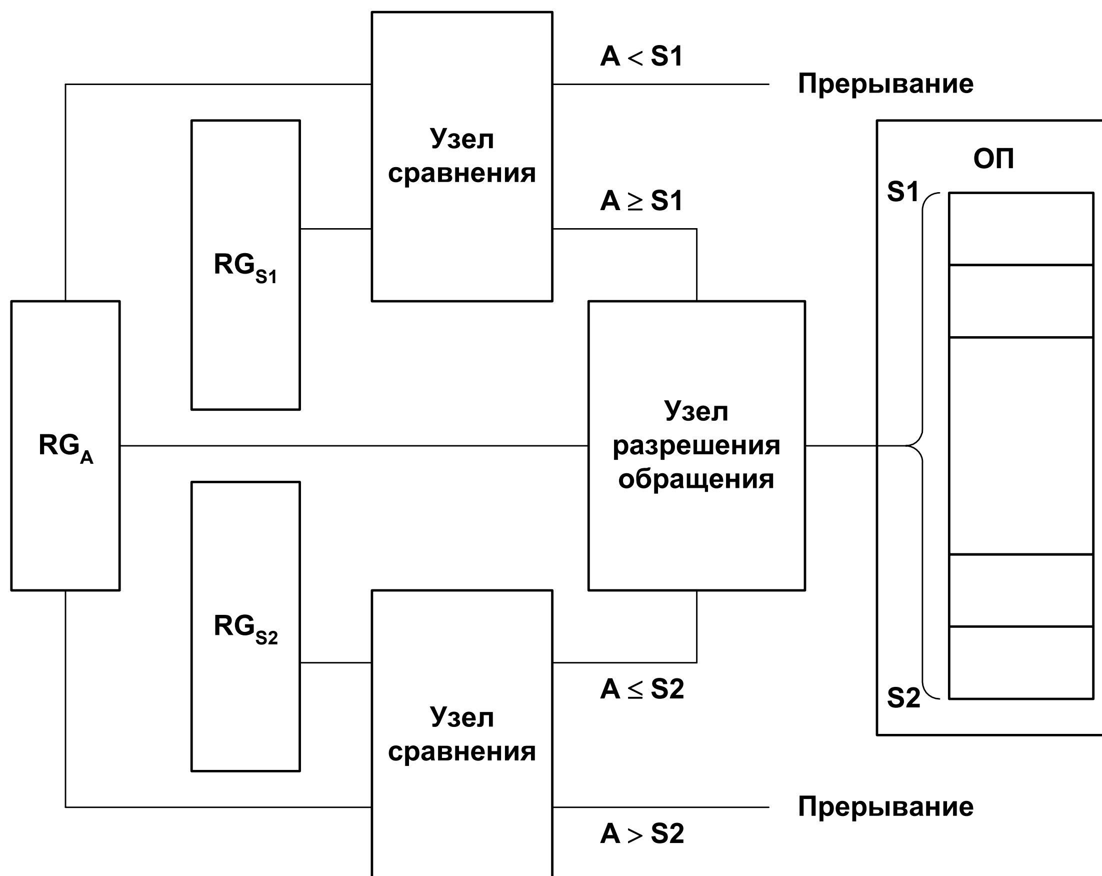



Рис. 5.1. Защита по граничным адресам

Эти значения заносятся в соответствующие регистры в момент инициирования программы. При обращении к выделенной области оперативной памяти каждый адрес обращения $A_i$ подвергается проверке на соблюдение следующих условий:

$$S1 \leq A_i \leq S2.$$

Если хотя бы одно из условий не выполняется, то возникает прерывание по нарушению защиты памяти и управление передается специальной программе, обрабатывающей нарушение защиты памяти.

Такой способ защиты обладает достоинством, заключающемся в том, что границы выделяемой области подвижны и ее размеры могут быть произвольны. Если же программа и данные будут находиться в различных областях памяти, то в этом случае для реализации защиты памяти потребуется две пары граничных регистров. Дальнейшее разделение программы и данных и размещение отдельных частей в различных областях оперативной памяти приводи к необходимости использования множеств пар граничных регистров.

### 1.1.2. Защита по ключам

Способ защиты по ключам позволяет организовать доступ программ к областям памяти, расположенным не подряд. Этот способ обеспечивает довольно простую организацию защиты памяти в условиях страничного или сегментного распределения памяти.

В соответствии с принципом защиты по ключам каждой физической странице оперативной памяти присваивается некоторое кодовое слово, называемое ключом памяти. Все множество ключей хранится в памяти ключей, представляющей собой регистровую память, которая защищена от доступа прикладных программ.

Запись в память ключей может производиться только супервизором.

Каждой программе, выполняемой в мультипрограммном режиме, присваивается код защиты, называемый ключом программы. Ключ программы входит в состав управляющих слов процессора и канала.

Перед выполнением программы супервизор записывает ключ защиты страницы памяти. Равный ключу программы, в соответствующую ячейку памяти ключей защиты. Адрес ячейки, в которую записывается ключ защиты страницы памяти, определяется по номеру страницы оперативной памяти в которой размещается данная программа. Если программа размещается в нескольких физических страницах памяти, то один и тот же ключ страницы памяти записывается сразу в нескольких ячейках памяти ключей защиты.

При работе процессора по этой программе ключ программы сравнивается с соответствующим ключом защиты страницы памяти. Несовпадение этих ключей приводит к блокировке обращения к оперативной памяти.

На рис. 5.2. показан вариант аппаратурной реализации защиты памяти при помощи ключей. При обращении к памяти исполнительный адрес $A_i$ поступает в регистр адреса $RG_A$. Группа старших разрядов кода исполнительного адреса, соответствующая номеру $N_C$ страницы, к которой производится обращение, используется как адрес для выборки ключа защиты из

существенно более быстрой памяти ключей защиты (ПКЗ). Ключ защиты памяти подается на вход узла сравнения (УС), где он сравнивается с ключом программы, находящимся в регистре управляющего слова процессора ($RG_{УСП}$). Совпадение ключей приводит к появлению сигнала "Обращение разрешено" (ОР), который разрешает выдачу адреса $A_i$ в оперативную память. Несовпадение приводит к выработке сигнала прерывания (Пр) по защите памяти.

Рис. 5.2. Защита по ключам

Ключ защиты обычно состоит из нескольких разрядов. Часть разрядов (как правило, четыре) представляют собственно ключ защиты, а остальные являются дополнительными и используются при организации вычислительного процесса. К дополнительным разрядам относятся: разряд режима обращения, разряд выборки и разряд изменения.

Выработка сигнала ОР производится с учетом режима обращения (запись или считывание), указываемого дополнительным разрядом ключа защиты (рис. 5.3), хранящимся на триггере режима обращения ($T_{PO}$).

Если, например, в этом разряде установлен нуль, то при несовпадении ключей обращение к оперативной памяти запрещается только при записи. Если же в этом разряде записана единица, то обращение к оперативной памяти в случае несовпадения ключей программы и защиты страницы запрещается как при записи, так и при считывании данных.

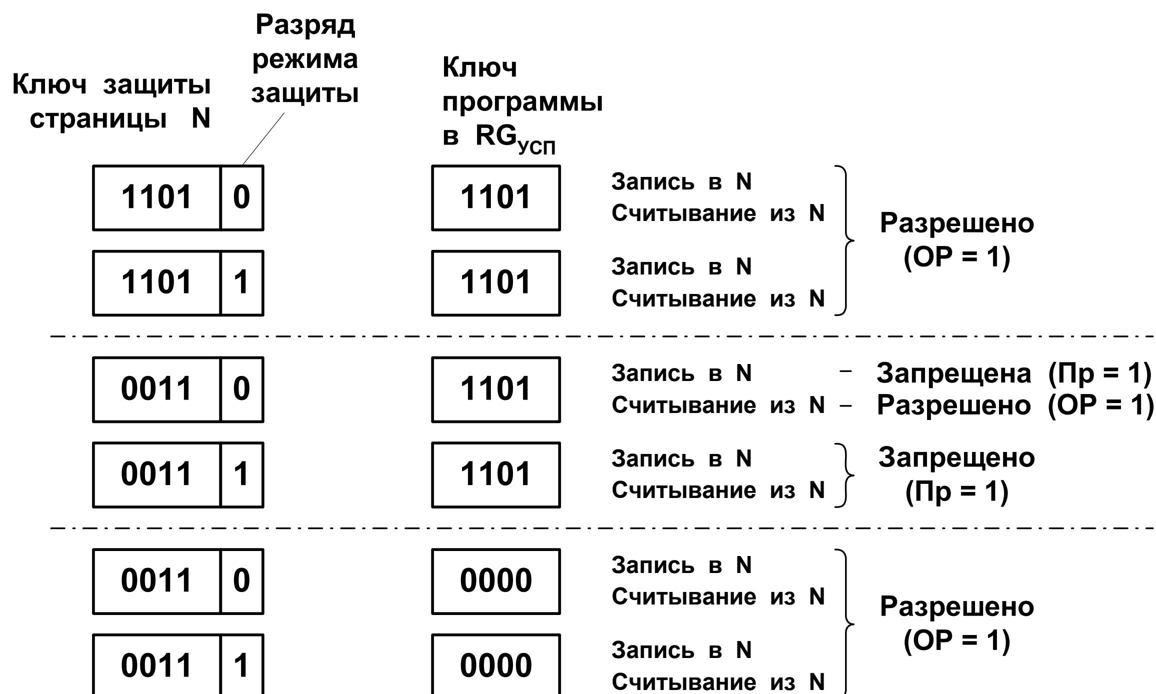



Рис. 5.3. Принцип формирования сигналов
разрешения обращения к памяти

Ключ программы супервизора имеет код, состоящий из одних нулей (нулевой ключ). При таком ключе разрешены обращения к любой странице памяти для записи и считывания независимо от значения разряда режима защиты. обращения также разрешены при любом ключе программы, если ключ защиты состоит из одних нулей (такой ключ обычно имеют страницы, хранящие стандартные подпрограммы). Для выявления нулевых ключей в состав системы защиты памяти может вводится специальный узел анализа.

Способ защиты по ключам применяется в ЕС ЭВМ.

В заключение следует отметить, что в случае использования в ЭВМ виртуальной памяти можно осуществлять защиту на уровне математических страниц. Причем защита памяти на уровне математических страниц не исключает одновременного применения защиты на уровне физических страниц.

## 1.2. Защита данных

Целью защиты данных является исключение несанкционированного доступа к файлам, при котором производится нарушение ограничений, наложенных на использование этих файлов.

Ограничения на использование файлов разделяются на два вида: *ограничение на доступ к файлу* и *ограничение на полномочия пользователей по отношению к файлу*. Ограничение на доступ определяет пользователей,

которым разрешено обращение к файлу. Это ограничение совершенно не определяет действий с файлом, которые может производить пользователь. По отношению к каждому файлу ограничения на доступ делят всех пользователей на две группы:

1) пользователи, которым доступен *i*-файл (множество $П^{Д}_{i}$);

2) пользователи, которым *i*-файл не доступен (множество $П^{НД}_{i}$).

Если множество $П^{НД}_{i}$ оказывается пустым ($П^{НД}_{i} = \varnothing$), то *i*-й файл не имеет ограничений на доступ и защищать его не имеет смысла. Множество $П^{Д}_{i}$ не может быть пустым, так как не имеет смысла хранить файлы, к которым не допущен ни один пользователь (при условии, что операционная система рассматривается как своеобразный пользователь).

Ограничения на полномочия определяют множество операций над файлом, разрешенных пользователю, имеющему доступ к файлу. Ограничения этого вида определены только для пользователей, удовлетворяющих ограничению по доступу.

Обращение пользователя к файлу считается санкционированным, если файл доступен данному пользователю и задаваемая пользователем операция соответствует его полномочиям. В противном случае обращение считается несанкционированным и запрещается.

Запись, содержащая ограничения на доступ к файлу и на полномочия по операциям с ним, называется *ордером защиты*. В общем случае ордер защиты состоит из кода доступности и кода полномочий, однако один из этих кодов может отсутствовать. Способы защиты данных во многом зависят от принятого способа задания доступности и полномочий.

### 1.2.1. Задание доступности и полномочий

Для определения доступности файла могут применяться различные методы. Практическую реализацию нашли методы с использованием специальных паролей, матрицы управления доступом и шифрования.

*Метод паролей* состоит в том, что в каждый ордер, связанный с защищаемым файлом, на место кода доступности записывается некоторый числовой, буквенный или буквенно-цифровой пароль. При обращении к файлу пользователь должен указать свой пароль. Файл считается доступным пользователю, если указанный им пароль совпадает с паролем ордера защиты.

*Метод с использованием матрицы управления доступом* заключается в том, что ордер защиты не имеет кода доступности. Но права доступа задаются для каждого пользователя по отношению ко всем файлам с помощью матрицы доступа, строки которой задают имена пользователей, а столбцы – имена файлов. При этом файл *j* считается доступным *i*-му пользователю, если элемент матрицы $Ф_{ij}$ ненулевой, если $Ф_{ij} = 0$, то *j*-й файл не доступен *i*-му пользователю.

*Метод шифрования* состоит в зашифровке данных файлов. Все пользователи имеют доступ к файлам, но только часть пользователей знает способ расшифровки данных. Дешифрование (при чтении) файла выполняется с помощью специального кодового ключа, которым обладает только пользователь, работающий с данным файлом.

Существующие методы задания полномочий различаются множеством разрешенных операций, для которых задаются эти полномочия. По существу задание полномочий сводится к кодированию прав пользователей на выполнение отдельных операций. Это кодирование производится с помощью двоичных переменных.

Если право пользователя на $i$-ую операцию обозначить $P_i$, то полномочия пользователя по отношению к множеству из $K$ операций можно задать вектором $P = (P_1, P_2, \ldots, P_K)$. Значения компонентов этого вектора определяются следующим образом:

$$P_i = \begin{cases} 1, \text{ если i - я операция над файлом разрешена;} \\ 0 \text{ - в противном случае} \end{cases}$$

В качестве разрешенных операций могут быть: чтение файла, загрузка файла для выполнения в качестве программы, запись в файл, уничтожение файла и т.п. Например, в ОС ЕС все множество разрешенных операций состоит из двух операций: записи и чтения.

Таким образом, все файлы доступны всем пользователям. Их либо не защищают вообще, если полномочия пользователей по отношению к ним не ограничены, либо они имеют по одному ордеру, содержащему вектор полномочий, одинаковый для всех пользователей. Например, трансляторы должны быть защищены от записи со стороны всех пользователей.

Индивидуальные файлы используются только одним пользователем либо группой пользователей, имеющих одинаковые полномочия. Индивидуальные файлы имеют по одному ордеру на каждый файл. Этот ордер включает как код доступности, так и код полномочий.

Групповые файлы используются несколькими пользователями с различными полномочиями. Каждый групповой файл имеет столько ордеров, сколько различных по полномочиям обращений он допускает. Эти ордера различаются как кодами доступности, так и кодами полномочий. Наличие ордеров с различными кодами доступности, но с одинаковыми кодами полномочий допустимо, но нецелесообразно. Ордера с одинаковым кодом доступности, но с различными кодами полномочий недопустимы, так как это порождает неопределенность процедур защиты.

### 1.2.2. Реализация защиты

Реализация защиты от несанкционированного обращения производится с помощью специального модуля файловой системы, обращение к которому происходит при открытии файла, нуждающегося в защите.

Файлы, требующие защиты, помечаются специальным признаком, называемым признаком защиты, который указывает на необходимость обращения к модулю защиты. Этот признак размещается в справочнике файлов. Кроме того, признак защиты $J_з$ дублируется в метке файла. Дублирование $J_з$ в метке файла при наличии его в справочнике файлов используется для обращения к модулю защиты, если обращение к файлу производится минуя справочник. Ордера защиты либо располагаются в справочнике файлов, либо организуются в виде специального файла ордеров.

С целью сокращения затрат времени на выполнение процедур защиты контроль доступности и контроль полномочий выполняются на разных этапах.

Проверка доступности файла осуществляется один раз при обращении к справочнику файлов для поиска на этапе его открытия. При обнаружении в элементе справочника признака защиты осуществляется обращение по признаку защиты (рис.5.4).

Если произошло обращение к модулю защиты, а признак защиты не равен единице, то это свидетельствует об ошибке в системных программах. Информация об этой ошибке должна быть немедленно выдана оператору ЭВМ.

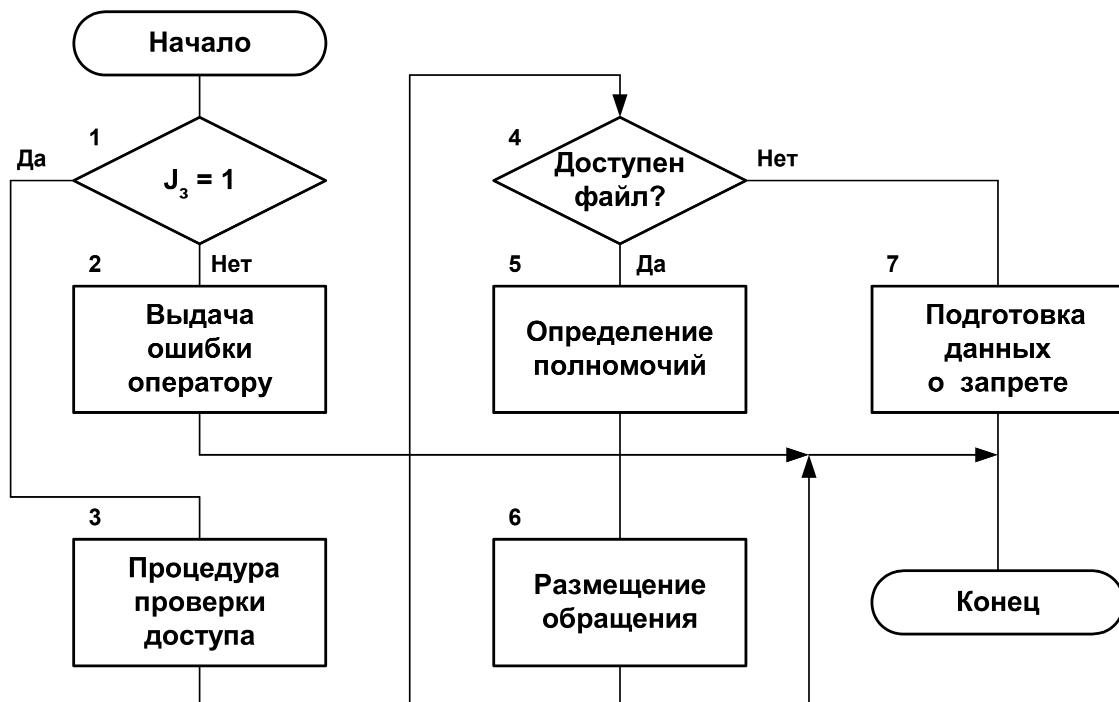



Рис.5.4. Алгоритм защиты

При правильном обращении производится проверка доступности файла с использованием ордеров защиты (символ 3 и 4). Для доступных файлов осуществляется выборка векторы полномочий (символ 5). Однако проверка полномочий на этом этапе не производится, так как еще не известно, какие операции будут выполняться над файлом. Проверка полномочий осуществляется при появлении требования на действия с файлом, когда будет задан вид операции.

После выборки полномочий операционная система санкционирует обращение к файлу (символ 6). При этом модулем защиты выполняются некоторые дополнительные действия. К таким действиям относятся: установка признака положительного завершения процедуры защиты для последующего контроля по признаку в метке файла; подсчет количества обращений к файлу и выдача пользователю даты последнего обращения к файлу для дополнительного контроля.

В случае установления недоступности файла формируется информация о попытке несанкционированного обращения к файлу (символ 7) и производится выход для последующего снятия задания.

В рассмотренном алгоритме защиты (рис. 5.4) символ 3 соответствует процедуре проверки доступности файла. Реализация этой процедуры с использованием различных методов имеет ряд особенностей.

Ордера защиты сами могут иметь дополнительную защиту. Защита ордеров осуществляется специальным кодом, хранящимся вместе с ордером. Эта защита делается для того, чтобы исключить возможность обращения другими программами к ордерам, кроме модуля защиты.

При реализации защиты по методу паролей ордера защиты, относящиеся к одному файлу, объединяются в список. Этот список размещается в соответствующем элементе справочника файлов. Признаком того, что некоторый элемент содержит список, является условие $J_Д = 1$.

Проверка доступности файла сводится к следующим действиям:

1) Переход на начало списка ордеров.

2) Последовательное сравнение пароля, заданного пользователем, с паролями ордеров защиты.

При совпадении паролей доступность файла считается установленной, выбирается вектор полномочий и контроль прекращается. Если список ордеров исчерпан, а совпадение паролей не произошло, то обращение считается некорректным. В этом случае пользователь может повторить пароль. Если и повторно пароль не совпадает с ордером, то фиксируется попытка несанкционированного обращения к файлу.

Этот метод обеспечивает наиболее простую процедуру контроля доступности файлов. Его недостатком является хранение пароля у пользователя и необходимость обмена им с операционной системой при каждом обращении к файлу, что способствует разглашению паролей. Кроме того, пользователю, имеющему доступ к нескольким файлам, необходимо хранить много паролей.

Особенность реализации метода с использованием матрицы управления доступом состоит в том, чтобы установить связи пользователя с файлами.

В методе паролей эта связь устанавливается с помощью кода доступности. Здесь этот код не используется, а ордера защиты содержат только коды полномочий.

Ордера защиты при реализации этого метода объединяются в матрицу полномочий, которая может одновременно использоваться и для управления доступом (рис. 5.5).

| Шифр пользователя | Код режима файла | | | | | |
|---|---|---|---|---|---|---|
| | $R_1$ | $R_2$ | • • • | $R_j$ | • • • | $R_m$ |
| $П_1$ | $P_{11}$ | $P_{12}$ | • • • | $P_{1j}$ | • • • | $P_{1m}$ |
| $П_2$ | $P_{21}$ | $P_{22}$ | • • • | $P_{2j}$ | • • • | $P_{2m}$ |
| ⋮ | ⋮ | ⋮ | • • • | ⋮ | • • • | ⋮ |
| $П_n$ | $P_{n1}$ | $P_{n2}$ | • • • | $P_{nj}$ | • • • | $P_{nm}$ |

Рис. 5.5. Матрица полномочий

В этом случае нет необходимости иметь отдельную матрицу управления доступом, поскольку недоступность файла может кодироваться нулевыми значениями компонентов вектора полномочий.

Строка такой матрицы задает коды полномочий пользователя на множестве всех файлов (ордер пользователя). Столбец – коды полномочий всех пользователей по отношению к одному файлу. Полномочия одного пользователя по отношению к одному файлу задаются вектором полномочий ($P_{ij}$), определенным на множестве допустимых операций с файлом.

Для сокращения объема матрицы полномочий и времени поиска в ней все файлы в зависимости от их доступности разбиваются на группы. Каждой группе присваивается определенный код режима ($R_j$). Тогда матрица будет определять полномочия пользователей по отношению не к каждому файлу, а к группам файлов, заданным кодами режима.

При реализации этого метода ордера защиты удобнее связывать не с файлами, а с пользователями. Для этого имя или шифр пользователя снабжается ссылкой на ордер пользователя. При допуске пользователя к работе с ЭВМ выбирается ссылка, которая является первым входом в матрицу полномочий.

Элементы справочника файлов, нуждающихся в защите, также снабжаются признаком защиты. Однако вместо ссылки на ордер они имеют только код режима, который является вторым входом в матрицу полномочий.

Проверка доступности файла производится следующим образом:

1. При обнаружении $J_j = 1$ выбирается код режима файла (R*).
2. По ссылке выбирается ордер пользователя.
3. В ордере пользователя отыскивается вектор полномочий, соответствующий режиму R*.
4. Вектор полномочий сравнивается с нулем.

При совпадении этого вектора с нулем обращение считается корректным.

Защита с помощью матрицы полномочий не связана с обменом паролями, что повышает надежность защиты. Однако этот метод требует сложной процедуры установления полномочий при образовании новых файлов и при допуске к машине новых пользователей. Этот метод может найти широкое применение в информационно-вычислительных системах (ИВС), где существует строгая иерархия пользователей и определены их полномочия к имеющимся в ИВС файлам.

*Возможности файловой системы NTFS по ограничению доступа к файлам и каталогам*

Благодаря наличию механизма расширенных атрибутов, в NTFS именно с их помощью реализованы ограничения в доступе к файлам и каталогам. Эти дополнительные атрибуты, использованные для ограничения в доступе к файловым объектам, назвали атрибутами безопасности. При каждом обращении к такому объекту сравнивается специальный список прав доступа, приписанный ему, со специальным системным идентификатором, несущим информацию о том, от имени кого осуществляется текущий запрос к файлу или каталогу. Если имеется в списке необходимое разрешение, то действие выполняется, в противном случае система сообщает об отказе.

Файловая система NTFS имеет следующие разрешения, которые могут быть приписаны любому файлу и/или каталогу, и которые называются индивидуальными разрешениями. Это разрешения *Read (прочитать), Write (записать), eXecute (выполнить), Delete (удалить), Change Permissions (изменить разрешения),* и *Take Ownership (стать владельцем).* Соответствующие этим разрешениям действия можно выполнять, только если для данного пользователя или для группы (к которой он принадлежит) имеется одноименное разрешение. Комбинации этих индивидуальных разрешений и определяют те действия, которые могут быть выполнены с файлом или каталогом (табл. 5.1).

Таблица 5.1

Соответствие стандартных и индивидуальных разрешений в NTFS

| Стандартные разрешения NTFS | Соответствующие им комбинации индивидуальных разрешений NTFS | |
|---|---|---|
| | Для каталогов | Для файлов |
| No access (нет доступа) | Нет никаких разрешений | Нет никаких разрешений |
| List (просмотр) | Read, eXecute | Нет никаких разрешений |
| Read (чтение) | Read, eXecute | Read, eXecute |
| Add (добавление) | Write, eXecute | Нет никаких разрешений |
| Add & Read (чтение и добавление) | Read, Write, eXecute | Read, eXecute |
| Change (изменение) | Read, Write, eXecute, Delete | Read, Write, eXecute, Delete |
| Full Control (полный доступ) | Все разрешения | Все разрешения |

Изначально всему диску, а значит, и файлам, которые на нем создаются, присвоены все индивидуальные разрешения для группы *Everyone* (все). То есть, любой пользователь, имея полный набор индивидуальных разрешений на файлы и каталоги, может изменять их по своему усмотрению, т.е. ограничивать других пользователей в правах доступа. Если изменить разрешения на каталог, то новые файлы, создаваемые в нем, будут получать и соответствующие разрешения: они будут наследовать разрешения своего родительского каталога.

Имеются так называемые стандартные разрешения, которые, по замыслу разработчиков, следует использовать для указания наиболее распространенных комбинаций индивидуальных разрешений. Поясним эти стандартные разрешения с помощью таблицы, расположенной на следующей странице. Помимо этих стандартных разрешений можно использовать специальные. Они определяются как явная комбинация индивидуальных разрешений.

Фактические разрешения для пользователя, которые он будет иметь на файл или каталог, определяются как сумма разрешений, которые он получает как член нескольких групп. У этого общего правила есть исключение. Разрешение *No Access* (нет доступа) имеет приоритет над остальными. Оно запрещает любой доступ к файлу или каталогу, даже если пользователю, как члену другой группы, дано необходимое разрешение. Можно сказать, что это стандартное разрешение означает не отсутствие разрешений, а наложение явного запрета, и что оно отменяет для пользователя или

группы все разрешения, установленные в остальных строках списка прав доступа.

Каталоги обычно обладают теми же разрешениями, что и находящиеся в них файлы и папки, хотя у каждого файла могут быть свои разрешения. Разрешения, которые имеются у файла, имеют приоритет над разрешениями, которые установлены на каталог, в котором находится этот файл. Например, если создается каталог внутри другого каталога, для которого администраторы обладают правом полного доступа, а пользователи – правом чтения, то новый каталог унаследует эти права. То же относится и к файлам, копируемым из другого каталога или перемещаемым из другого раздела NTFS.

Если каталог или файл перемещается в другой каталог того же раздела NTFS, то атрибуты безопасности не наследуются от нового каталога. Дело в том, что при перемещении файлов в границах одного раздела NTFS изменяется только указатель местонахождения объекта, а все остальные атрибуты (включая атрибуты безопасности) остаются без изменений.

Три следующих важных правила помогут определить состояние прав доступа при перемещении или копировании объектов NTFS:

при перемещении файлов в границах раздела NTFS сохраняются исходные права доступа;

при выполнении других операций (создании или копировании файлов, а также их перемещении между разделами NTFS) наследуются права доступа родительского каталога;

при перемещении файлов из раздела NTFS в раздел FAT все права NTFS теряются.

## 2. Система безопасности операционной системы

По мере компьютеризации общества в электронную форму переносится все больше и больше данных, конфиденциальных по своей природе: банковские счета и другая коммерческая информация, штабные документы и т. д. Проблема защиты пользовательских данных от нежелательного прочтения или модификации встает очень часто и в самых разнообразных ситуациях – от секретных баз данных Министерства обороны до архива личной переписки.

Совершенствование средств доступа к данным и их совместного использования всегда порождает и дополнительные возможности несанкционированного доступа. Наиболее ярким примером являются современные глобальные сети, которые предоставляют доступ к огромному богатству информационных сред. Эти же сети представляют серьезную угрозу безопасности подключающихся к сети организаций. Основная специфика угрозы заключается в том, что в таких сетях злоумышленнику значительно

легче обеспечить свою анонимность даже в случае обнаружения факта "взлома".

При рассмотрении безопасности информационных систем обычно выделяют две группы проблем: безопасность компьютера и сетевая безопасность. К *безопасности компьютера* относят все проблемы защиты данных, хранящихся и обрабатывающихся компьютером, который рассматривается как автономная система. Эти проблемы решаются средствами операционных систем и приложений, таких как базы данных, а также встроенными аппаратными средствами компьютера. Под *сетевой безопасностью* понимают все вопросы, связанные с взаимодействием устройств в сети, это прежде всего защита данных в момент их передачи по линиям связи и защита от несанкционированного удаленного доступа в сеть. И хотя подчас проблемы компьютерной и сетевой безопасности трудно отделить друг от друга, настолько тесно они связаны, совершенно очевидно, что сетевая безопасность имеет свою специфику.

Автономно работающий компьютер можно эффективно защитить от внешних покушений разнообразными способами, например, просто запереть на замок клавиатуру или снять жесткий накопитель и поместить его в сейф. Компьютер, работающий в сети, по определению не может полностью отгородиться от мира, он должен общаться с другими компьютерами, возможно, даже удаленными от него на большое расстояние, поэтому обеспечение безопасности в сети является задачей значительно более сложной. При работе в сети логический вход чужого пользователя в ваш компьютер является штатной ситуацией. Обеспечение безопасности в такой ситуации сводится к тому, чтобы сделать это проникновение контролируемым – каждому пользователю сети должны быть четко определены его права по доступу к информации, внешним устройствам и выполнению системных действий на каждом из компьютеров сети.

## 2.1. Основные понятия безопасности

### 2.1.1. Конфиденциальность, целостность и доступность данных

Безопасная информационная система – это система, которая, во-первых, защищает данные от несанкционированного доступа, во-вторых, всегда готова предоставить их своим пользователям, а в-третьих, надежно хранит информацию и гарантирует неизменность данных. Таким образом, безопасная система по определению обладает свойствами конфиденциальности, доступности и целостности.

*Конфиденциальность (confidentiality)* – гарантия того, что секретные данные будут доступны только тем пользователям, которым этот доступ разрешен (такие пользователи называются авторизованными).

*Доступность (availability)* – гарантия того, что авторизованные пользователи всегда получат доступ к данным.

*Целостность (integrity)* – гарантия сохранности данными правильных значений, которая обеспечивается запретом для неавторизованных пользователей каким-либо образом изменять, модифицировать, разрушать или создавать данные.

Требования безопасности могут меняться в зависимости от назначения системы, характера используемых данных и типа возможных угроз. Трудно представить систему, для которой были бы не важны свойства целостности и доступности, но свойство конфиденциальности не всегда является обязательным. Например, при публикации информации в Интернете на Web-сервере с целью сделать ее доступной для самого широкого круга людей, то конфиденциальность в данном случае не требуется. Однако требования целостности и доступности остаются актуальными.

Понятия конфиденциальности, доступности и целостности могут быть определены не только по отношению к информации, но и к другим ресурсам вычислительной сети, например внешним устройствам или приложениям. Существует множество системных ресурсов, возможность “незаконного” использования которых может привести к нарушению безопасности системы. Например, неограниченный доступ к устройству печати позволяет злоумышленнику получать копии распечатываемых документов, изменять параметры настройки, что может привести к изменению очередности работ и даже к выводу устройства из строя. Свойство конфиденциальности, примененное к устройству печати, можно интерпретировать так, что доступ к устройству имеют те и только те пользователи, которым этот доступ разрешен, причем они могут выполнять только те операции с устройством, которые для них определены. Свойство доступности устройства означает его готовность к использованию всякий раз, когда в этом возникает необходимость. А свойство целостности может быть определено как свойство неизменности параметров настройки данного устройства. Легальность использования сетевых устройств важна не только постольку, поскольку она влияет на безопасность данных. Устройства могут предоставлять различные услуги: распечатку текстов, отправку факсов, доступ в Интернет, электронную почту и т. п., незаконное потребление которых, наносящее материальный ущерб, также является нарушением безопасности системы.

Любое действие, которое направлено на нарушение конфиденциальности, целостности и/или доступности информации, а также на нелегальное использование других ресурсов сети, называется *угрозой.*

Реализованная угроза называется *атакой.*

*Риск* – это вероятностная оценка величины возможного ущерба, который может понести владелец информационного ресурса в результате успешно проведенной атаки. Значение риска тем выше, чем более уязвимой

является существующая система безопасности и чем выше вероятность реализации атаки.

### 2.1.2. Классификация угроз

Универсальной классификации угроз не существует, возможно, и потому, что нет предела творческим способностям человека, и каждый день применяются новые способы незаконного проникновения в сеть, разрабатываются новые средства мониторинга сетевого графика, появляются новые вирусы, находятся новые изъяны в существующих программных и аппаратных сетевых продуктах. В ответ на это разрабатываются все более изощренные средства защиты, которые ставят преграду на пути многих типов угроз, но затем сами становятся новыми объектами атак. Тем не менее можно сделать некоторые обобщения. Так, прежде всего угрозы могут быть разделены на умышленные и неумышленные.

*Неумышленные угрозы* вызываются ошибочными действиями лояльных сотрудников, становятся следствием их низкой квалификации или безответственности. Кроме того, к такому роду угроз относятся последствия ненадежной работы программных и аппаратных средств системы. Так, например, из-за отказа диска, контроллера диска или всего файлового сервера могут оказаться недоступными данные, критически важные для работы организации. Поэтому вопросы безопасности так тесно переплетаются с вопросами надежности, отказоустойчивости технических средств. Угрозы безопасности, которые вытекают из ненадежности работы программно-аппаратных средств, предотвращаются путем их совершенствования, использования резервирования на уровне аппаратуры (RAID-массивы, многопроцессорные компьютеры, источники бесперебойного питания, кластерные архитектуры) или на уровне массивов данных (тиражирование файлов, резервное копирование).

*Умышленные угрозы* могут ограничиваться либо пассивным чтением данных или мониторингом системы, либо включать в себя активные действия, например нарушение целостности и доступности информации, приведение в нерабочее состояние приложений и устройств. Так, умышленные угрозы возникают в результате деятельности хакеров и явно направлены на нанесение ущерба организации.

В вычислительных сетях можно выделить следующие типы умышленных угроз:

незаконное проникновение в один из компьютеров сети под видом легального пользователя;

системы с помощью программ-вирусов;

нелегальные действия легального пользователя;

“подслушивание” внутрисетевого графика.

*Незаконное проникновение* может быть реализовано через уязвимые места в системе безопасности с использованием недокументированных возможностей операционной системы. Эти возможности могут позволить злоумышленнику “обойти” стандартную процедуру, контролирующую вход в сеть.

Другим способом незаконного проникновения в сеть является использование “чужих” паролей, полученных путем подглядывания, расшифровки файла паролей, подбора паролей или получения пароля путем анализа сетевого графика. Особенно опасно проникновение злоумышленника под именем пользователя, наделенного большими полномочиями, например администратора сети. Для того чтобы завладеть паролем администратора, злоумышленник может попытаться войти в сеть под именем простого пользователя. Поэтому очень важно, чтобы все пользователи сети сохраняли свои пароли в тайне, а также выбирали их так, чтобы максимально затруднить угадывание.

Подбор паролей злоумышленник выполняет с использованием специальных программ, которые работают путем перебора слов из некоторого файла, содержащего большое количество слов. Содержимое файла-словаря формируется с учетом психологических особенностей человека, которые выражаются в том, что человек выбирает в качестве пароля легко запоминаемые слова или буквенные сочетания.

Возможности словарной атаки впервые были продемонстрированы в 1987 году молодым тогда студентом Робертом Моррисом. Разработанная им программа – “червь Морриса” – использовала словарную атаку и последующий вход в систему с подобранным паролем как один из основных способов размножения. Червь использовал для распространения и другие приемы, продемонстрировавшие не только специфические изъяны тогдашних ОС семейства Unix, но и несколько фундаментальных проблем компьютерной безопасности, поэтому мы еще несколько раз будем возвращаться к обсуждению этой программы.

Еще один способ получения пароля – это внедрение в чужой компьютер “троянского коня”. Так называют резидентную программу, работающую без ведома хозяина данного компьютера и выполняющую действия, заданные злоумышленником. В частности, такого рода программа может считывать коды пароля, вводимого пользователем во время логического входа в систему.

Программа-“троянский конь” всегда маскируется под какую-нибудь полезную утилиту или игру, а производит действия, разрушающие систему. По такому принципу действуют и *программы-вирусы,* отличительной особенностью которых является способность “заражать” другие файлы, внедряя в них свои собственные копии. Чаще всего вирусы поражают исполняемые файлы. Когда такой исполняемый код загружается в оперативную память для выполнения, вместе с ним получает возможность испол-

нить свои вредительские действия вирус. Вирусы могут привести к повреждению или даже полной утрате информации.

*Нелегальные действия легального пользователя* – этот тип угроз исходит от легальных пользователей сети, которые, используя свои полномочия, пытаются выполнять действия, выходящие за рамки их должностных обязанностей. Например, администратор сети имеет практически неограниченные права на доступ ко всем сетевым ресурсам. Однако в организации может быть информация, доступ к которой администратору сети запрещен. Для реализации этих ограничений могут быть предприняты специальные меры, такие, например, как шифрование данных, но и в этом случае администратор может попытаться получить доступ к ключу. Нелегальные действия может попытаться предпринять и обычный пользователь сети. Существующая статистика говорит о том, что едва ли не половина всех попыток нарушения безопасности системы исходит от сотрудников организации, которые как раз и являются легальными пользователями сети.

*“Подслушивание” внутрисетевого трафика* – это незаконный мониторинг сети, захват и анализ сетевых сообщений. Существует много доступных программных и аппаратных анализаторов трафика, которые делают эту задачу достаточно тривиальной. Еще более усложняется защита от этого типа угроз в сетях с глобальными связями. Глобальные связи, простирающиеся на десятки и тысячи километров, по своей природе являются менее защищенными, чем локальные связи (больше возможностей для прослушивания трафика, более удобная для злоумышленника позиция при проведении процедур аутентификации). Такая опасность одинаково присуща всем видам территориальных каналов связи и никак не зависит от того, используются собственные, арендуемые каналы или услуги общедоступных территориальных сетей, подобных Интернету.

### 2.1.3. Системный подход к обеспечению безопасности

Построение и поддержка безопасной системы требует системного подхода. В соответствии с этим подходом прежде всего необходимо осознать весь спектр возможных угроз для конкретной сети и для каждой из этих угроз продумать тактику ее отражения. В этой борьбе можно и нужно использовать самые разноплановые средства и приемы – морально-этические и законодательные, административные и психологические, защитные возможности программных и аппаратных средств сети.

К *физическим* средствам защиты относятся экранирование помещений для защиты от излучения, проверка поставляемой аппаратуры на соответствие ее спецификациям и отсутствие аппаратных “жучков”, средства наружного наблюдения, устройства, блокирующие физический доступ к отдельным блокам компьютера, различные замки и другое оборудование,

защищающие помещения, где находятся носители информации, от незаконного проникновения и т. п.

*Технические* средства информационной безопасности реализуются программным и аппаратным обеспечением вычислительных сетей. Такие средства, называемые также службами сетевой безопасности, решают самые разнообразные задачи по защите системы, например контроль доступа, включающий процедуры аутентификации и авторизации, аудит, шифрование информации, антивирусную защиту, контроль сетевого графика и много других задач. Технические средства безопасности могут быть либо встроены в программное и аппаратное обеспечение сети, либо реализованы в виде отдельных продуктов, созданных специально для решения проблем безопасности.

### 2.1.4. Политика безопасности

Важность и сложность проблемы обеспечения безопасности требует выработки *политики информационной безопасности,* которая подразумевает ответы на следующие вопросы:

1. Какую информацию защищать?
2. Какой ущерб понесет предприятие при потере или при раскрытии тех или иных данных?
3. Кто или что является возможным источником угрозы, какого рода атаки на безопасность системы могут быть предприняты?
4. Какие средства использовать для защиты каждого вида информации?

Специалисты, ответственные за безопасность системы, формируя политику безопасности, должны учитывать несколько базовых принципов. Одним из таких принципов является предоставление каждому сотруднику организации того *минимального уровня привилегий* на доступ к данным, который необходим ему для выполнения его должностных обязанностей. Учитывая, что большая часть нарушений в области безопасности предприятий исходит именно от собственных сотрудников, важно ввести четкие ограничения для всех пользователей сети, не наделяя их излишними возможностями.

Следующий принцип – использование *комплексного подхода к* обеспечению безопасности. Чтобы затруднить злоумышленнику доступ к данным, необходимо предусмотреть самые разные средства безопасности, начиная с организационно-административных запретов и кончая встроенными средствами сетевой аппаратуры. Административный запрет на работу в воскресные дни ставит потенциального нарушителя под визуальный контроль администратора и других пользователей, физические средства защиты (закрытые помещения, блокировочные ключи) ограничивают непосредственный контакт пользователя только приписанным ему компьютером, встроенные средства сетевой ОС (система аутентификации и автори-

зации) предотвращают вход в сеть нелегальных пользователей, а для легального пользователя ограничивают возможности только разрешенными для него операциями (подсистема аудита фиксирует его действия). Такая система защиты с многократным резервированием средств безопасности увеличивает вероятность сохранности данных.

Используя многоуровневую систему защиты, важно обеспечивать *баланс надежности защиты всех уровней.* Если в сети все сообщения шифруются, но ключи легкодоступны, то эффект от шифрования нулевой. Или если на компьютерах установлена файловая система, поддерживающая избирательный доступ на уровне отдельных файлов, но имеется возможность получить жесткий диск и установить его на другой машине, то все достоинства средств защиты файловой системы сводятся на нет.

Следующим универсальным принципом является использование средств, которые при отказе переходят в состояние *максимальной защиты.* Это касается самых различных средств безопасности. Если, например, автоматический пропускной пункт в какое-либо помещение ломается, то он должен фиксироваться в таком положении, чтобы ни один человек не мог пройти на защищаемую территорию. А если в сети имеется устройство, которое анализирует весь входной трафик и отбрасывает кадры с определенным, заранее заданным обратным адресом, то при отказе оно должно полностью блокировать вход в сеть. Неприемлемым следовало бы признать устройство, которое бы при отказе пропускало в сеть весь внешний трафик.

*Принцип единого контрольно-пропускного пункта* – весь входящий во внутреннюю сеть и выходящий во внешнюю сеть трафик должен проходить через единственный узел сети, например через межсетевой экран (firewall). Только это позволяет в достаточной степени контролировать трафик. В противном случае, когда в сети имеется множество пользовательских станций, имеющих независимый выход во внешнюю сеть, очень трудно скоординировать правила, ограничивающие права пользователей внутренней сети по доступу к серверам внешней сети и обратно – права внешних клиентов по доступу к ресурсам внутренней сети.

*Принцип баланса возможного ущерба от реализации угрозы и затрат на ее предотвращение.* Ни одна система безопасности не гарантирует защиту данных на уровне 100 %, поскольку является результатом компромисса между возможными рисками и возможными затратами. Определяя политику безопасности, администратор должен взвесить величину ущерба, которую может понести предприятие в результате нарушения защиты данных, и соотнести ее с величиной затрат, требуемых на обеспечение безопасности этих данных.

## 2.2. Базовые технологии безопасности

В разных программных и аппаратных продуктах, предназначенных для защиты данных, часто используются одинаковые подходы, приемы и технические решения. К таким базовым технологиям безопасности относятся аутентификация, авторизация, аудит и технология защищенного канала.

### 2.2.1. Шифрование

*Шифрование* – это краеугольный камень всех служб информационной безопасности, будь то система аутентификации или авторизации, средства создания защищенного канала или способ безопасного хранения данных.

Любая процедура шифрования, превращающая информацию из обычного "понятного" вида в "нечитабельный" зашифрованный вид, естественно, должна быть дополнена процедурой дешифрирования, которая, будучи примененной к зашифрованному тексту, снова приводит его в понятный вид. Пара процедур – шифрование и дешифрирование – называется *криптосистемой.*

Информацию, над которой выполняются функции шифрования и дешифрирования, будем условно называть "текст", учитывая, что это может быть также числовой массив или графические данные.

В современных алгоритмах шифрования предусматривается наличие параметра – *секретного ключа.* В криптографии принято правило Керкхоффа: "Стойкость шифра должна определяться только секретностью ключа". Так, все стандартные алгоритмы шифрования (например, DES, PGP) широко известны, их детальное описание содержится в легко доступных документах, но от этого их эффективность не снижается. Злоумышленнику может быть все известно об алгоритме шифрования, кроме секретного ключа (следует отметить, однако, что существует немало фирменных алгоритмов, описание которых не публикуется).

Алгоритм шифрования считается *раскрытым,* если найдена процедура, позволяющая подобрать ключ за реальное время. Сложность алгоритма раскрытия является одной из важных характеристик криптосистемы и называется *криптостойкостью.*

Существуют два класса криптосистем – симметричные и асимметричные. В симметричных схемах шифрования (классическая криптография) секретный ключ зашифровки совпадает с секретным ключом расшифровки. В асимметричных схемах шифрования (криптография с открытым ключом) открытый ключ зашифровки не совпадает с секретным ключом расшифровки.

<u>*Симметричные алгоритмы шифрования*</u>

На рис. 5.6 приведена классическая модель симметричной криптосистемы, теоретические основы которой впервые были изложены в 1949 го-

ду в работе Клода Шеннона. В данной модели три участника: отправитель, получатель, злоумышленник. Задача отправителя заключается в том, чтобы по открытому каналу передать некоторое сообщение в защищенном виде. Для этого он на ключе k зашифровывает открытый текст X и передает шифрованный текст Y. Задача получателя заключается в том, чтобы расшифровать Y и прочитать сообщение X. Предполагается, что отправитель имеет свой источник ключа. Сгенерированный ключ заранее по надежному каналу передается получателю. Задача злоумышленника заключается в перехвате и чтении передаваемых сообщений, а также в имитации ложных сообщений.

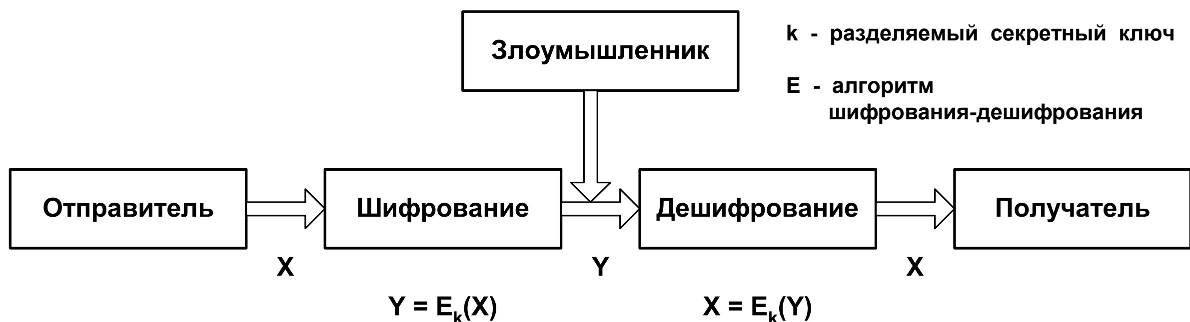



Рис. 5.6. Модель симметричного шифрования

Модель является универсальной – если зашифрованные данные хранятся в компьютере и никуда не передаются, отправитель и получатель совмещаются в одном лице, а в роли злоумышленника выступает некто, имеющий доступ к компьютеру в ваше отсутствие.

Наиболее популярным стандартным симметричным алгоритмом шифрования данных является *DES (Data Encryption Standard).* Алгоритм разработан фирмой IBM и в 1976 году был рекомендован Национальным бюро стандартов к использованию в открытых секторах экономики. Суть этого алгоритма заключается в следующем (рис. 5.7).

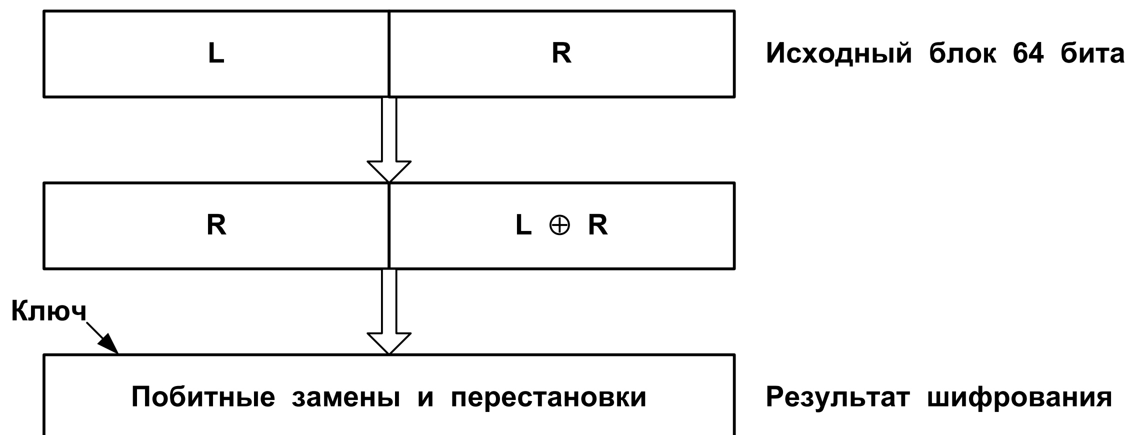



Рис. 5.7. Схема шифрования по алгоритму DES

Данные шифруются поблочно. Перед шифрованием любая форма представления данных преобразуется в числовую. Эти числа получают путем любой открытой процедуры преобразования блока текста в число. Например, ими могли бы быть значения двоичных чисел, полученных слиянием ASCII-кодов последовательных символов соответствующего блока текста. На вход шифрующей функции поступает блок данных размером 64 бита, он делится пополам на левую (L) и правую (R) части. На первом этапе на место левой части результирующего блока помещается правая часть исходного блока. Правая часть результирующего блока вычисляется как сумма по модулю 2 (операция XOR) левой и правой частей исходного блока. Затем на основе случайной двоичной последовательности по определенной схеме в полученном результате выполняются побитные замены и перестановки. Используемая двоичная последовательность, представляющая собой ключ данного алгоритма, имеет длину 64 бита, из которых 56 действительно случайны, а 8 предназначены для контроля ключа.

Вот уже в течение двух десятков лет алгоритм DES испытывается на стойкость. И хотя существуют примеры успешных попыток “взлома” данного алгоритма, в целом можно считать, что он выдержал испытания. Алгоритм DES широко используется в различных технологиях и продуктах безопасности информационных систем. Для того чтобы повысить криптостойкость алгоритма DES, иногда применяют его усиленный вариант, называемый “тройным DES”, который включает троекратное шифрование с использованием двух разных ключей. При этом можно считать, что длина ключа увеличивается с 56 бит до 112 бит, а значит криптостойкость алгоритма существенно повышается. Но за это приходится платить производительностью – “тройной DES” требует в три раза больше времени, чем “обычный” DES.

В симметричных алгоритмах главную проблему представляют ключи. Во-первых, криптостойкость многих симметричных алгоритмов зависит от качества ключа, это предъявляет повышенные требования к службе генерации ключей. Во-вторых, принципиальной является надежность канала передачи ключа второму участнику секретных переговоров. Проблема с ключами возникает даже в системе с двумя абонентами, а в системе с n абонентами, желающими обмениваться секретными данными по принципу “каждый с каждым”, потребуется $n \times (n-1)/2$ ключей, которые должны быть сгенерированы и распределены надежным образом. То есть количество ключей пропорционально квадрату количества абонентов, что при большом числе абонентов делает задачу чрезвычайно сложной. Несимметричные алгоритмы, основанные на использовании открытых ключей, снимают эту проблему.

*Несимметричные алгоритмы шифрования*

В середине 70-х двое ученых – Винфилд Диффи и Мартин Хеллман – описали принципы шифрования с открытыми ключами.

Особенность шифрования на основе открытых ключей состоит в том, что одновременно генерируется уникальная *пара* ключей, таких, что текст, зашифрованный одним ключом, может быть расшифрован только с использованием второго ключа и наоборот.

В модели криптосхемы с открытым ключом также три участника: отправитель, получатель, злоумышленник (рис. 5.8). Задача отправителя заключается в том, чтобы по открытому каналу связи передать некоторое сообщение в защищенном виде. Получатель генерирует на своей стороне два ключа: открытый E и закрытый D. Закрытый ключ D (часто называемый также личным ключом) абонент должен сохранять в защищенном месте, а открытый ключ E он может передать всем, с кем он хочет поддерживать защищенные отношения. Открытый ключ используется для шифрования текста, но расшифровать текст можно только с помощью закрытого ключа. Поэтому открытый ключ передается отправителю в незащищенном виде. Отправитель, используя открытый ключ получателя, шифрует сообщение X и передает его получателю. Получатель расшифровывает сообщение своим закрытым ключом D.

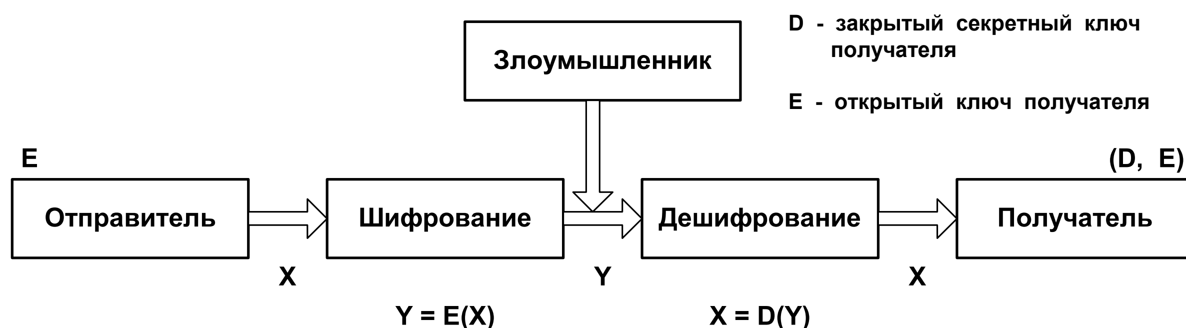
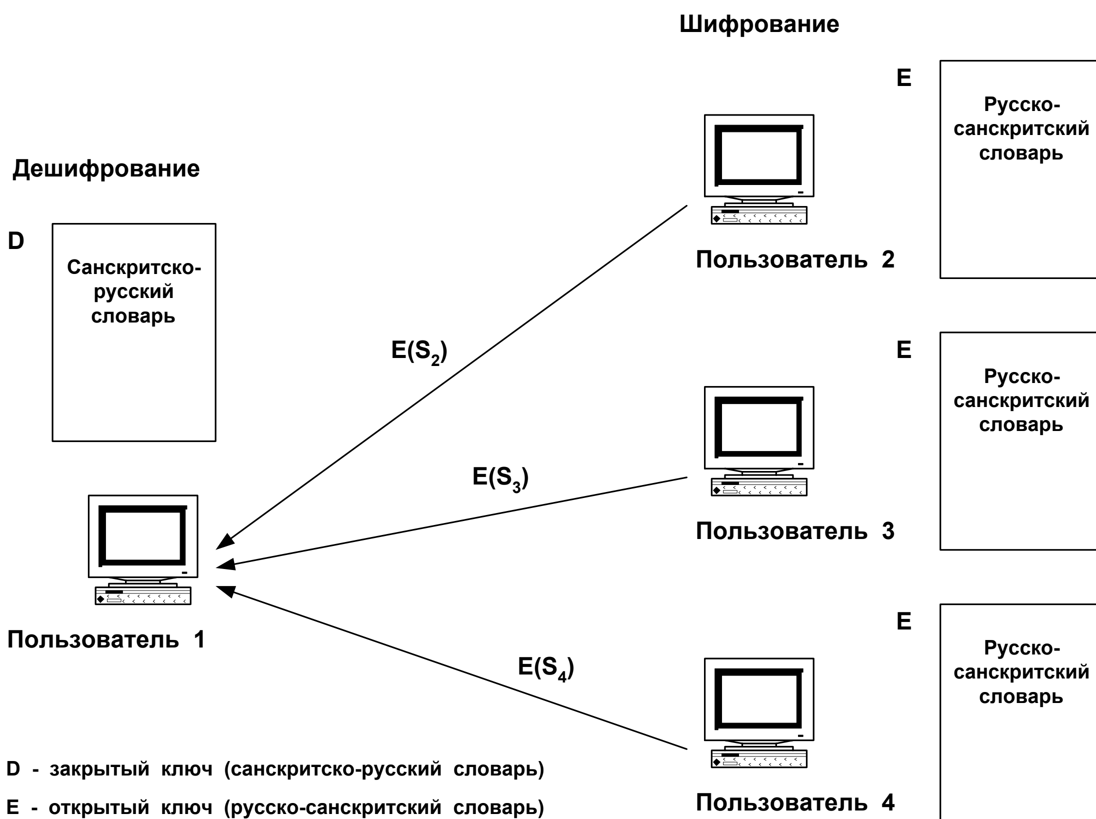


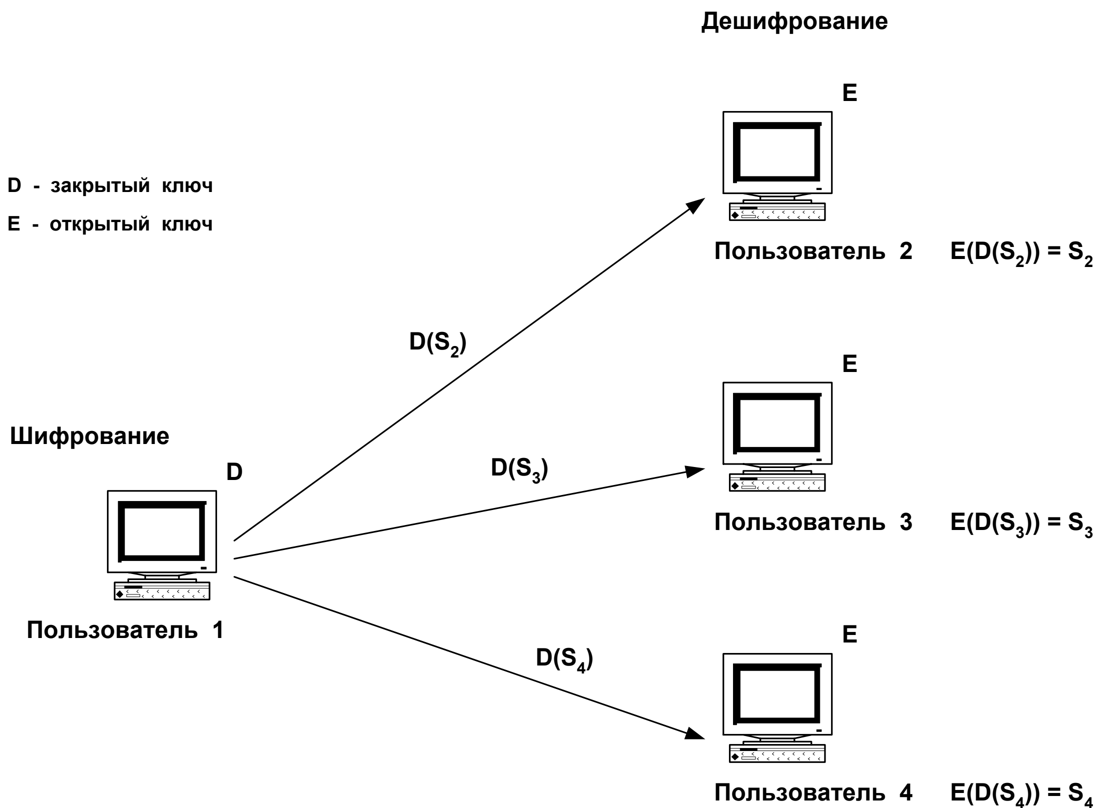

Рис. 5.8. Модель криптосхемы с открытым ключом

Очевидно, что числа, одно из которых используется для шифрования текста, а другое – для дешифрирования, не могут быть независимыми друг от друга, а значит, есть теоретическая возможность вычисления закрытого ключа по открытому, но это связано с огромным количеством вычислений, которые требуют соответственно огромного времени. Поясним принципиальную связь между закрытым и открытым ключами следующей аналогией (рис 5.9).

Пусть абонент 1 (рис. 5.9, *а)* решает вести секретную переписку со своими сотрудниками на малоизвестном языке, например санскрите. Для этого он обзаводится санскритско-русским словарем, а всем своим абонентам посылает, русско-санскритские словари. Каждый из них, пользуясь словарем, пишет сообщения на санскрите и посылает их абоненту 1, который переводит их на русский язык, пользуясь доступным только ему санскритско-русским словарем.




*а*




*б*

Рис. 5.9. Две схемы использования открытого и закрытого ключей

Очевидно, что здесь роль открытого ключа E играет русско-санскритский словарь, а роль закрытого ключа D – санскритско-русский словарь. Могут ли абоненты 2, 3 и 4 прочитать чужие сообщения $S_2$, $S_3$, $S_4$, которые посылает каждый из них абоненту 1? Вообще-то нет, так как, для этого им нужен санскритско-русский словарь, обладателем которого является только абонент 1. Но теоретическая возможность этого имеется, так как затратив массу времени, можно прямым перебором составить санскритско-русский словарь по русско-санскритскому словарю. Такая процедура, требующая больших временных затрат, является отдаленной аналогией восстановления закрытого ключа по открытому.

На рис. 5.9, *б* показана другая схема использования открытого и закрытого ключей, целью которой является подтверждение авторства (аутентификация или электронная подпись) посылаемого сообщения. В этом случае поток сообщений имеет обратное направление – от абонента 1, обладателя закрытого ключа D, к его корреспондентам, обладателям открытого ключа E. Если абонент 1 хочет аутентифицировать себя (поставить электронную подпись), то он шифрует известный текст своим закрытым ключом D и передает шифровку своим корреспондентам. Если им удается расшифровать текст открытым ключом абонента 1, то это доказывает, что текст был зашифрован его же закрытым ключом, а значит, именно он является автором этого сообщения. Заметим, что в этом случае сообщения $S_2$, $S_3$, $S_4$, адресованные разным абонентам, не являются секретными, так как все они – обладатели одного и того же открытого ключа, с помощью которого они могут расшифровывать все сообщения, поступающие от абонента 1.

Если же нужна взаимная аутентификация и двунаправленный секретный обмен сообщениями, то каждая из общающихся сторон генерирует собственную пару ключей и посылает открытый ключ своему корреспонденту.

Для того чтобы в сети все *n* абонентов имели возможность не только принимать зашифрованные сообщения, но и сами посылать таковые, каждый абонент должен обладать своей собственной парой ключей E и D. Всего в сети будет *2n* ключей: *n* открытых ключей для шифрования и *n* секретных ключей для дешифрирования. Таким образом решается проблема масштабируемости – квадратичная зависимость количества ключей от числа абонентов в симметричных алгоритмах заменяется линейной зависимостью в несимметричных алгоритмах. Исчезает и задача секретной доставки ключа. Злоумышленнику нет смысла стремиться завладеть открытым ключом, поскольку это не дает возможности расшифровывать текст или вычислить закрытый ключ.

Хотя информация об открытом ключе не является секретной, ее нужно защищать от подлогов, чтобы злоумышленник под именем легального пользователя не навязал свой открытый ключ, после чего с помощью своего закрытого ключа он может расшифровывать все сообщения, посылае-

мые легальному пользователю и отправлять свои сообщения от его имени. Проще всего было бы распространять списки, связывающие имена пользователей с их открытыми ключами широковещательно, путем публикаций в средствах массовой информации (бюллетени, специализированные журналы и т. п.). Однако при таком подходе мы снова, как и в случае с паролями, сталкиваемся с плохой масштабируемостью. Решением этой проблемы является технология цифровых сертификатов. Сертификат – это электронный документ, который связывает конкретного пользователя с конкретным ключом.

В настоящее время одним из наиболее популярных криптоалгоритмов с открытым ключом является криптоалгоритм *RSA*.

*Криптоалгоритм RSA*

В 1978 году трое ученых (Ривест, Шамир и Адлеман) разработали систему шифрования с открытыми ключами RSA (Rivest, Shamir, Adleman), полностью отвечающую всем принципам Диффи-Хеллмана. Этот метод состоит в следующем:

1. Случайно выбираются два очень больших простых числа p и q.

2. Вычисляются два произведения $n = p \times q$ и $m = (p - 1) \times (q - 1)$.

3. Выбирается случайное целое число E, не имеющее общих сомножителей с m.

4. Находится D, такое, что DE = 1 по модулю m.

5. Исходный текст, X, разбивается на блоки таким образом, чтобы $0 < X < n$.

6. Для шифрования сообщения необходимо вычислить $C = X^E$ по модулю n.

7. Для дешифрирования вычисляется $X = C^D$ по модулю n.

Таким образом, чтобы зашифровать сообщение, необходимо знать пару чисел (E, n), а чтобы дешифрировать – пару чисел (D, n). Первая пара – это открытый ключ, а вторая – закрытый.

Зная открытый ключ (E, n), можно вычислить значение закрытого ключа D. Необходимым промежуточным действием в этом преобразовании является нахождение чисел p и q, для чего нужно разложить на простые множители очень большое число n, а на это требуется очень много времени. Именно с огромной вычислительной сложностью разложения большого числа на простые множители связана высокая криптостойкость алгоритма RSA. В некоторых публикациях приводятся следующие оценки: для того чтобы найти разложение 200-значного числа, понадобится 4 миллиарда лет работы компьютера с быстродействием миллион операций в секунду. Однако следует учесть, что в настоящее время активно ведутся работы по совершенствованию методов разложения больших чисел, поэтому в алгоритме RSA стараются применять числа длиной более 200 десятичных разрядов.

Программная реализация криптоалгоритмов типа RSA значительно сложнее и менее производительна, чем реализация классических крипто-

алгоритмов типа DES. Вследствие сложности реализации операций модульной арифметики криптоалгоритм RSA часто используют только для шифрования небольших объемов информации, например для рассылки классических секретных ключей или в алгоритмах цифровой подписи, а основную часть пересылаемой информации шифруют с помощью симметричных алгоритмов.

*Односторонние функции шифрования*

Во многих базовых технологиях безопасности используется еще один прием шифрования – шифрование с помощью односторонней функции (one-way function), называемой также хэш-функцией (hash function), или дайджест-функцией (digest function).

Эта функция, примененная к шифруемым данным, дает в результате значение (дайджест), состоящее из фиксированного небольшого числа байт (рис. 5.10, *а*).

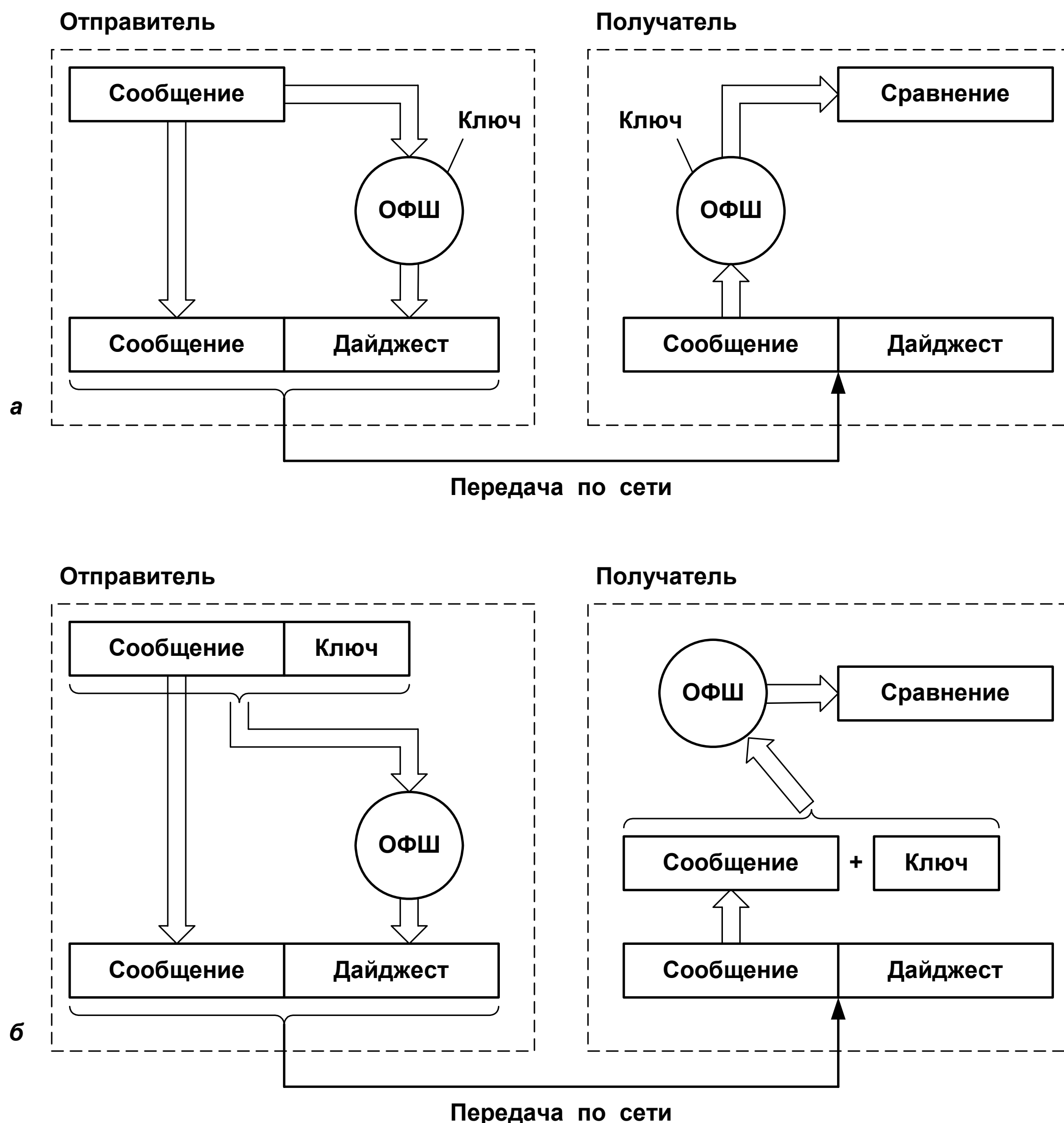



Рис. 5.10. Односторонние функции шифрования

Дайджест передается вместе с исходным сообщением. Получатель сообщения, зная, какая односторонняя функция шифрования (ОФШ) была применена для получения дайджеста, заново вычисляет его, используя незашифрованную часть сообщения. Если значения полученного и вычисленного дайджестов совпадают, то значит, содержимое сообщения не было подвергнуто никаким изменениям. Знание дайджеста не дает возможности восстановить исходное сообщение, но зато позволяет проверить целостность данных.

Дайджест является своего рода контрольной суммой для исходного сообщения. Однако имеется и существенное отличие. Использование контрольной суммы является средством проверки целостности передаваемых сообщений по ненадежным линиям связи. Это средство не направлено на борьбу со злоумышленниками, которым в такой ситуации ничто не мешает подменить сообщение, добавив к нему новое значение контрольной суммы. Получатель в таком случае не заметит никакой подмены.

В отличие от контрольной суммы при вычислении дайджеста требуются секретные ключи. В случае, если для получения дайджеста использовалась односторонняя функция с параметром, который известен только отправителю и получателю, любая модификация исходного сообщения будет немедленно обнаружена.

Для того чтобы в сети все *n* абонентов имели возможность не только принимать зашифрованные сообщения, но и сами посылать таковые, каждый абонент должен обладать своей собственной парой ключей E и D. Всего в сети будет *2n* ключей: *n* открытых ключей для шифрования и *n* секретных ключей для дешифрирования. Таким образом решается проблема масштабируемости – квадратичная зависимость количества ключей от числа абонентов в симметричных алгоритмах заменяется линейной зависимостью в несимметричных алгоритмах. Исчезает и задача секретной доставки ключа. Злоумышленнику нет смысла стремиться завладеть открытым ключом, поскольку это не дает возможности расшифровывать текст или вычислить закрытый ключ.

На рис. 5.10, *б* показан другой вариант использования односторонней функции шифрования для обеспечения целостности данных. В данном случае односторонняя функция не имеет параметра-ключа, но зато применяется не просто к сообщению, а к сообщению, дополненному секретным ключом. Получатель, извлекая исходное сообщение, также дополняет его тем же известным ему секретным ключом, после чего применяет к полученным данным одностороннюю функцию. Результат вычислений сравнивается с полученным по сети дайджестом.

Помимо обеспечения целостности сообщений дайджест может быть использован в качестве электронной подписи для аутентификации передаваемого документа.

Построение односторонних функций является трудной задачей. Такого рода функции должны удовлетворять двум условиям:

1) по дайджесту, вычисленному с помощью данной функции, невозможно каким-либо образом вычислить исходное сообщение;

2) должна отсутствовать возможность вычисления двух разных сообщений, для которых с помощью данной функции могли быть вычислены одинаковые дайджесты.

Наиболее популярной в системах безопасности в настоящее время является серия хэш-функций MD2, MD4, MD5. Все они генерируют дайджесты фиксированной длины 16 байт. Адаптированным вариантом MD4 является американский стандарт SHA, длина дайджеста в котором составляет 20 байт. Компания IBM поддерживает односторонние функции MDC2 и MDC4, основанные на алгоритме шифрования DES.

### 2.2.2. Аутентификация, авторизация, аудит

*Аутентификация*

*Аутентификация (authentication)* предотвращает доступ к сети нежелательных лиц и разрешает вход для легальных пользователей. Термин “аутентификация” в переводе с латинского означает “установление подлинности”. Аутентификацию следует отличать от идентификации. Идентификаторы пользователей используются в системе с теми же целями, что и идентификаторы любых других объектов, файлов, процессов, структур данных, но они не связаны непосредственно с обеспечением безопасности. Идентификация заключается в сообщении пользователем системе своего идентификатора, в то время как аутентификация – это процедура доказательства пользователем того, что он есть тот, за кого себя выдает, в частности, доказательство того, что именно ему принадлежит введенный им идентификатор.

В процедуре аутентификации участвуют две стороны: одна сторона доказывает свою аутентичность, предъявляя некоторые доказательства, а другая сторона – аутентификатор – проверяет эти доказательства и принимает решение. В качестве доказательства аутентичности используются самые разнообразные приемы:

1) аутентифицируемый может продемонстрировать знание некоего общего для обеих сторон секрета: слова (пароля) или факта (даты и места события, прозвища человека и т. п.);

2) аутентифицируемый может продемонстрировать, что он владеет неким уникальным предметом (физическим ключом), в качестве которого может выступать, например, электронная магнитная карта;

3) аутентифицируемый может доказать свою идентичность, используя собственные биохарактеристики: рисунок радужной оболочки глаза или отпечатки пальцев, которые предварительно были занесены в базу данных аутентификатора.

Сетевые службы аутентификации строятся на основе всех этих приемов, но чаще всего для доказательства идентичности пользователя используются пароли. Простота и логическая ясность механизмов аутентификации на основе паролей в какой-то степени компенсирует известные слабости паролей. Это, во-первых, возможность раскрытия и разгадывания паролей, а во-вторых, возможность “подслушивания” пароля путем анализа сетевого трафика. Для снижения уровня угрозы от раскрытия паролей администраторы сети, как правило, применяют встроенные программные средства для формирования политики назначения и использования паролей: задание максимального и минимального сроков действия пароля, хранение списка уже использованных паролей, управление поведением системы после нескольких неудачных попыток логического входа и т. п. Перехват паролей по сети можно предупредить путем их шифрования перед передачей в сеть.

Легальность пользователя может устанавливаться по отношению к различным системам. Так, работая в сети, пользователь может проходить процедуру аутентификации и как локальный пользователь, который претендует на использование ресурсов только данного компьютера, и как пользователь сети, который хочет получить доступ ко всем сетевым ресурсам. При локальной аутентификации пользователь вводит свои идентификатор и пароль, которые автономно обрабатываются операционной системой, установленной на данном компьютере. При логическом входе в сеть данные о пользователе (идентификатор и пароль) передаются на сервер, который хранит учетные записи обо всех пользователях сети. Многие приложения имеют свои средства определения, является ли пользователь законным. И тогда пользователю приходится проходить дополнительные этапы проверки.

В качестве объектов, требующих аутентификации, могут выступать не только пользователи, но и различные устройства, приложения, текстовая и другая информация. Так, например, пользователь, обращающийся с запросом к корпоративному серверу, должен доказать ему свою легальность, но он также должен убедиться сам, что ведет диалог действительно с сервером своей организации. Другими словами, сервер и клиент должны пройти процедуру взаимной аутентификации. Здесь мы имеем дело с аутентификацией на уровне приложений. При установлении сеанса связи между двумя устройствами также часто предусматриваются процедуры взаимной аутентификации на более низком, канальном уровне. Примером такой процедуры является аутентификация по протоколам PAP и CHAP, входящим в семейство протоколов PPP. Аутентификация данных означает доказательство целостности этих данных, а также того, что они поступили именно от того человека, который объявил об этом. Для этого используется механизм электронной подписи.

В вычислительных сетях процедуры аутентификации часто реализуются теми же программными средствами, что и процедуры авторизации. В

отличие от аутентификации, которая распознает легальных и нелегальных пользователей, система авторизации имеет дело только с легальными пользователями, которые уже успешно прошли процедуру аутентификации. Цель подсистем авторизации состоит в том, чтобы предоставить каждому легальному пользователю именно те виды доступа и к тем ресурсам, которые были для него определены администратором системы.

*Авторизация доступа*

Средства *авторизации (authorization)* контролируют доступ легальных пользователей к ресурсам системы, предоставляя каждому из них именно те права, которые ему были определены администратором. Кроме предоставления прав доступа пользователям к каталогам, файлам и принтерам система авторизации может контролировать возможность выполнения пользователями различных системных функций, таких как локальный доступ к серверу, установка системного времени, создание резервных копий данных, выключение сервера и т. п.

Система авторизации наделяет пользователя сети правами выполнять определенные действия над определенными ресурсами. Для этого могут быть использованы различные формы предоставления правил доступа, которые часто делят на два класса:

1) избирательный доступ;

2) мандатный доступ.

*Избирательные права доступа* реализуются в операционных системах универсального назначения. В наиболее распространенном варианте такого подхода определенные операции над определенным ресурсом разрешаются или запрещаются пользователям или группам пользователей, явно указанным своими *идентификаторами.* Например, пользователю, имеющему идентификатор User_T, может быть разрешено выполнять операции чтения и записи по отношению к файлу File1. Модификацией этого способа является использование для идентификации пользователей их *должностей,* или факта их принадлежности к персоналу того или иного производственного подразделения, или еще каких-либо других позиционирующих характеристик. Примером такого правила может служить следующее: файл бухгалтерской отчетности BUCH могут читать работники бухгалтерии и руководитель организации.

*Мандатный подход* к определению прав доступа заключается в том, что вся информация делится на уровни в зависимости от степени секретности, а все пользователи сети также делятся на группы, образующие иерархию в соответствии с *уровнем допуска к* этой информации. Такой подход используется в известном делении информации на информацию для служебного пользования, “секретно”, “совершенно секретно”. При этом пользователи этой информации в зависимости от определенного для них статуса получают различные формы допуска: первую, вторую или третью. В отличие от систем с избирательными правами доступа в системах с мандатным подходом пользователи в принципе не имеют возможности изме-

нить уровень доступности информации. Например, пользователь более высокого уровня не может разрешить читать данные из своего файла пользователю, относящемуся к более низкому уровню. Отсюда видно, что мандатный подход является более строгим, он в корне пресекает всякий волюнтаризм со стороны пользователя. Именно поэтому он часто используется в системах военного назначения.

Процедуры авторизации реализуются программными средствами, которые могут быть встроены в операционную систему или в приложение, а также могут поставляться в виде отдельных программных продуктов. При этом программные системы авторизации могут строиться на базе двух схем:

1) централизованная схема авторизации, базирующаяся на сервере;

2) децентрализованная схема, базирующаяся на рабочих станциях.

В первой схеме сервер управляет процессом предоставления ресурсов пользователю. Главная цель таких систем – реализовать “принцип единого входа”. В соответствии с централизованной схемой пользователь один раз логически входит в сеть и получает на все время работы некоторый набор разрешений по доступу к ресурсам сети. Система Kerberos с ее сервером безопасности и архитектурой клиент-сервер является наиболее известной системой этого типа. Системы TACACS и RADIUS, часто применяемые совместно с системами удаленного доступа, также реализуют этот подход.

При втором подходе рабочая станция сама является защищенной – средства защиты работают на каждой машине, и сервер не требуется. Рассмотрим работу системы, в которой не предусмотрена процедура однократного логического входа. Теоретически доступ к каждому приложению должен контролироваться средствами безопасности самого приложения или же средствами, существующими в той операционной среде, в которой оно работает. В корпоративной сети администратору придется отслеживать работу механизмов безопасности, используемых всеми типами приложений – электронной почтой, службой каталогов локальной сети, базами данных хостов и т. п. Когда администратору приходится добавлять или удалять пользователей, то часто требуется вручную конфигурировать доступ к каждой программе или системе.

В крупных сетях часто применяется комбинированный подход предоставления пользователю прав доступа к ресурсам сети: сервер удаленного доступа ограничивает доступ пользователя к подсетям или серверам корпоративной сети, то есть к укрупненным элементам сети, а каждый отдельный сервер сети сам по себе ограничивает доступ пользователя к своим внутренним ресурсам: разделяемым каталогам, принтерам или приложениям. Сервер удаленного доступа предоставляет доступ на основании имеющегося у него списка прав доступа пользователя (Access Control List, ACL), а каждый отдельный сервер сети предоставляет доступ к своим ресурсам на основании хранящегося у него списка прав доступа, например ACL файловой системы.

Подчеркнем, что системы аутентификации и авторизации совместно выполняют одну задачу, поэтому необходимо предъявлять одинаковый уровень требований к системам авторизации и аутентификации. Ненадежность одного звена здесь не может быть компенсирована высоким качеством другого звена. Если при аутентификации используются пароли, то требуются чрезвычайные меры по их защите. Однажды украденный пароль открывает двери ко всем приложениям и данным, к которым пользователь с этим паролем имел легальный доступ.

*Аудит*

*Аудит (auditing)* – фиксация в системном журнале событий, связанных с доступом к защищаемым системным ресурсам. Подсистема аудита современных ОС позволяет дифференцированно задавать перечень интересующих администратора событий с помощью удобного графического интерфейса. Средства учета и наблюдения обеспечивают возможность обнаружить и зафиксировать важные события, связанные с безопасностью, или любые попытки создать, получить доступ или удалить системные ресурсы. Аудит используется для того, чтобы засекать даже неудачные попытки “взлома” системы.

Учет и наблюдение означает способность системы безопасности “шпионить” за выбранными объектами и их пользователями и выдавать сообщения тревоги, когда кто-нибудь пытается читать или модифицировать системный файл. Если кто-то пытается выполнить действия, определенные системой безопасности для отслеживания, то система аудита пишет сообщение в журнал регистрации, идентифицируя пользователя. Системный менеджер может создавать отчеты о безопасности, которые содержат информацию из журнала регистрации. Для “сверхбезопасных” систем предусматриваются аудио- и видеосигналы тревоги, устанавливаемые на машинах администраторов, отвечающих за безопасность.

Поскольку никакая система безопасности не гарантирует защиту на уровне 100 %, то последним рубежом в борьбе с нарушениями оказывается система аудита. Действительно, после того как злоумышленнику удалось провести успешную атаку, пострадавшей стороне не остается ничего другого, как обратиться к службе аудита. Если при настройке службы аудита были правильно заданы события, которые требуется отслеживать, то подробный анализ записей в журнале может дать много полезной информации. Эта информация, возможно, позволит найти злоумышленника или по крайней мере предотвратить повторение подобных атак путем устранения уязвимых мест в системе защиты.

### 2.2.3. Технология защищенного канала

Как уже было сказано, задачу защиты данных можно разделить на две подзадачи: защиту данных внутри компьютера и защиту данных в процессе их передачи из одного компьютера в другой. Для обеспечения безопасности данных при их передаче по публичным сетям используются различные технологии защищенного канала.

*Технология защищенного канала* призвана обеспечивать безопасность передачи данных по открытой транспортной сети, например по Интернету. Защищенный канал подразумевает выполнение трех основных функций:

1) взаимную аутентификацию абонентов при установлении соединения, которая может быть выполнена, например, путем обмена паролями;

2) защиту передаваемых по каналу сообщений от несанкционированного доступа, например, путем шифрования;

3) подтверждение целостности поступающих по каналу сообщений, например, путем передачи одновременно с сообщением его дайджеста.

Совокупность защищенных каналов, созданных предприятием в публичной сети для объединения своих филиалов, часто называют *виртуальной частной сетью* (Virtual Private Network, VPN).

Существуют разные реализации технологии защищенного канала, которые, в частности, могут работать на разных уровнях модели OSI. Так, функции популярного протокола SSL соответствуют *представительному* уровню модели OSI. Новая версия *сетевого* протокола IP предусматривает все функции – взаимную аутентификацию, шифрование и обеспечение целостности, – которые по определению свойственны защищенному каналу, а протокол туннелирования PPTP защищает данные на *канальном* уровне.

В зависимости от места расположения программного обеспечения защищенного канала различают две схемы его образования:

1) схему с конечными узлами, взаимодействующими через публичную сеть (рис. 5.11, *а);*

2) схему с оборудованием поставщика услуг публичной сети, расположенным на границе между частной и публичной сетями (рис. 5.11, *б).*

В первом случае защищенный канал образуется программными средствами, установленными на двух удаленных компьютерах, принадлежащих двум разным локальным сетям одной организации и связанных между собой через публичную сеть. Преимуществом этого подхода является полная защищенность канала вдоль всего пути следования, а также возможность использования любых протоколов создания защищенных каналов, лишь бы на конечных точках канала поддерживался один и тот же протокол.

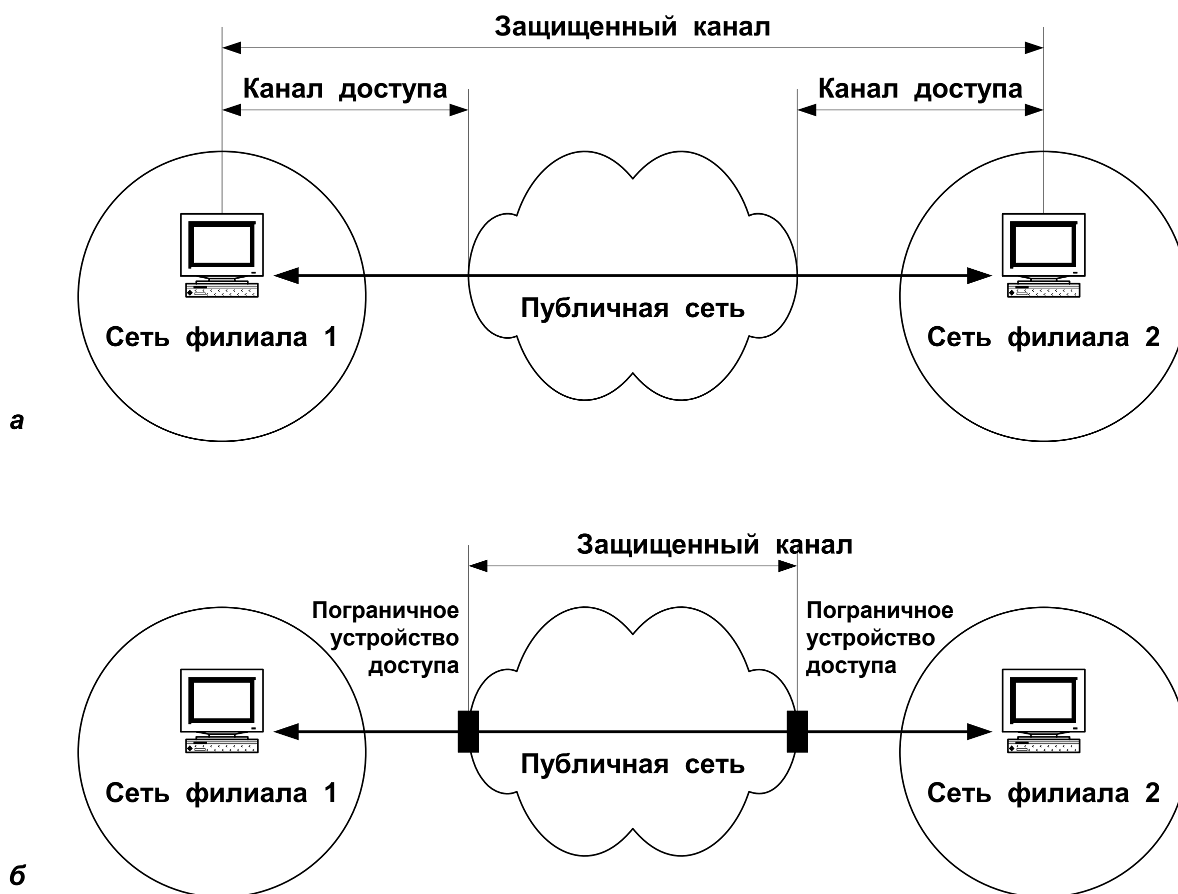



Рис. 5.11. Два способа образования защищенного канала

Недостатки заключаются в избыточности и децентрализованности решения. Избыточность состоит в том, что вряд ли стоит создавать защищенный канал на всем пути прохождения данных: уязвимыми для злоумышленников обычно являются сети с коммутацией пакетов, а не каналы телефонной сети или выделенные каналы, через которые локальные сети подключены к территориальной сети. Поэтому защиту каналов доступа к публичной сети можно считать избыточной. Децентрализация заключается в том, что для каждого компьютера, которому требуется предоставить услуги защищенного канала, необходимо отдельно устанавливать, конфигурировать и администрировать программные средства защиты данных. Подключение каждого нового компьютера к защищенному каналу требует выполнения этих трудоемких работ заново.

Во втором случае клиенты и серверы не участвуют в создании защищенного канала – он прокладывается только внутри публичной сети с коммутацией пакетов, например внутри Интернета. Канал может быть проложен, например, между сервером удаленного доступа поставщика услуг публичной сети и пограничным маршрутизатором корпоративной сети. Это хорошо масштабируемое решение, управляемое централизованно как администратором корпоративной сети, так и администратором сети поставщика услуг. Для компьютеров корпоративной сети канал прозрачен

– программное обеспечение этих конечных узлов остается без изменений. Такой гибкий подход позволяет легко образовывать новые каналы защищенного взаимодействия между компьютерами независимо от их места расположения. Реализация этого подхода сложнее – нужен стандартный протокол образования защищенного канала, требуется установка у всех поставщиков услуг программного обеспечения, поддерживающего такой протокол, необходима поддержка протокола производителями пограничного коммуникационного оборудования. Однако вариант, когда все заботы по поддержанию защищенного канала берет на себя поставщик услуг публичной сети, оставляет сомнения в надежности защиты: во-первых, незащищенными оказываются каналы доступа к публичной сети, во-вторых, потребитель услуг чувствует себя в полной зависимости от надежности поставщика услуг. И тем не менее, специалисты прогнозируют, что именно вторая схема в ближайшем будущем станет основной в построении защищенных каналов.

# 3. Аутентификация в современных операционных системах

## 3.1. Аутентификация пользователей

### 3.1.1. Сетевая аутентификация на основе многоразового пароля

В соответствии с базовым принципом “единого входа”, когда пользователю достаточно один раз пройти процедуру аутентификации, чтобы получить доступ ко всем сетевым ресурсам, в современных операционных системах предусматриваются централизованные службы аутентификации. Такая служба поддерживается одним из серверов сети и использует для своей работы базу данных, в которой хранятся учетные данные (иногда называемые бюджетами) о пользователях сети. Учетные данные содержат наряду с другой информацией идентификаторы и пароли пользователей. Упрощенно схема аутентификации в сети выглядит следующим образом. Когда пользователь осуществляет логический вход в сеть, он набирает на клавиатуре своего компьютера свои идентификатор и пароль. Эти данные используются службой аутентификации – в централизованной базе данных, хранящейся на сервере, по идентификатору пользователя находится соответствующая запись, из нее извлекается пароль и сравнивается с тем, который ввел пользователь. Если они совпадают, то аутентификация считается успешной, пользователь получает легальный статус и те права, которые определены для него системой авторизации.

Однако такая упрощенная схема имеет большой изъян. А именно при передаче пароля с клиентского компьютера на сервер, выполняющий про-

цедуру аутентификации, этот пароль может быть перехвачен злоумышленником. Поэтому в разных операционных системах применяются разные приемы, чтобы избежать передачи пароля по сети в незащищенном виде. Рассмотрим, как решается эта проблема в популярной сетевой ОС Windows NT.

В основе концепции сетевой безопасности Windows NT лежит понятие домена. Домен – это совокупность пользователей, серверов и рабочих станций, учетная информация о которых централизованно хранится в общей базе данных, называемой базой SAM (Security Accounts Manager database). Над этой базой данных реализована служба Directory Services, которая, как и любая централизованная справочная служба, устраняет дублирование учетных данных в нескольких компьютерах и сокращает число рутинных операций по администрированию. Одной из функций службы Directory Services является аутентификация пользователей. Служба Directory Services построена в архитектуре клиент-сервер. Каждый пользователь при логическом входе в сеть вызывает клиентскую часть службы, которая передает запрос на аутентификацию и поддерживает диалог с серверной частью.

Аутентификация пользователей домена выполняется на основе их паролей, хранящихся в зашифрованном виде в базе SAM (рис. 5.12). Пароли зашифровываются с помощью односторонней функции при занесении их в базу данных во время процедуры создания учетной записи для нового пользователя. Введем обозначение для этой односторонней функции – ОФШ1. Таким образом, пароль P хранится в базе данных SAM в виде дайджеста *d*(*P*). (Напомним, что знание дайджеста не позволяет восстановить исходное сообщение.)

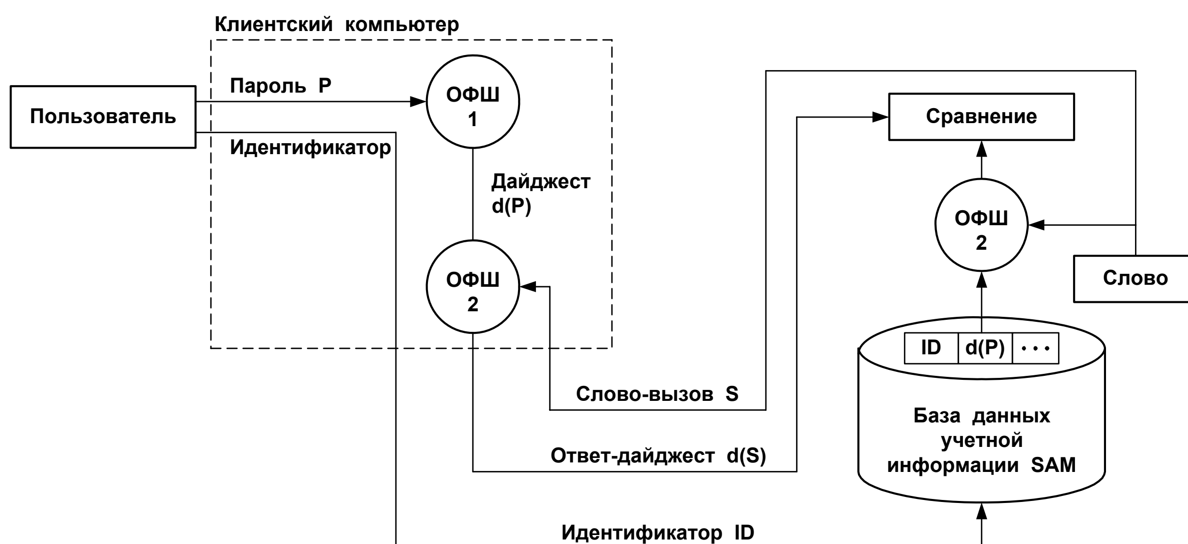



Рис. 5.11. Схема сетевой аутентификации на основе многоразового пароля

При логическом входе пользователь локально вводит в свой компьютер имя-идентификатор (ID) и пароль *P*. Клиентская часть подсистемы аутентификации, получив эти данные, передает запрос по сети на сервер, хранящий базу SAM. В этом запросе в открытом виде содержится идентификатор пользователя ID*,* но пароль не передается в сеть ни в каком виде.

К паролю на клиентской станции применяется та же односторонняя функция ОФШ1, которая была использована при записи пароля в базу данных SAM, то есть динамически вычисляется дайджест пароля $d(P)$.

В ответ на поступивший запрос серверная часть службы аутентификации генерирует случайное число S случайной длины, называемое словом-вызовом (challenge). Это слово передается по сети с сервера на клиентскую станцию пользователя. К слову-вызову на клиентской стороне применяется односторонняя функция шифрования ОФШ2. В отличие от функции ОФШ1 функция ОФШ2 является параметрической и получает в качестве параметра дайджест пароля $d(P)$. Полученный в результате ответ $d(S)$ передается по сети на сервер SAM.

Параллельно этому на сервере слово-вызов S аналогично шифруется с помощью той же односторонней функции ОФШ2 и дайджеста пароля пользователя d(P), извлеченного из базы SAM, а затем сравнивается с ответом, переданным клиентской станцией. При совпадении результатов считается, что аутентификация прошла успешно. Таким образом, при логическом входе пользователя пароли в сети Windows NT никогда не передаются по каналам связи.

Заметим также, что при каждом запросе на аутентификацию генерируется новое слово-вызов, так что перехват ответа $d(S)$ клиентского компьютера не может быть использован в ходе другой процедуры аутентификации.

### 3.1.2. Аутентификация с использованием одноразового пароля

Алгоритмы аутентификации, основанные на многоразовых паролях, не очень надежны. Пароли можно подсмотреть или просто украсть. Более надежными оказываются схемы, использующие *одноразовые пароли.* С другой стороны, одноразовые пароли намного дешевле и проще биометрических систем аутентификации, таких как сканеры сетчатки глаза или отпечатков пальцев. Все это делает системы, основанные на одноразовых паролях, очень перспективными. Следует иметь в виду, что, как правило, системы аутентификации на основе одноразовых паролей рассчитаны на проверку только удаленных, а не локальных пользователей.

Генерация одноразовых паролей может выполняться либо программно, либо аппаратно. Некоторые реализации аппаратных устройств доступа на основе одноразовых паролей представляют собой миниатюрные устройства со встроенным микропроцессором, похожие на обычные пластиковые

карточки, используемые для доступа к банкоматам. Такие карточки, часто называемые *аппаратными ключами,* могут иметь клавиатуру и маленькое дисплейное окно. Аппаратные ключи могут быть также реализованы в виде присоединяемого к разъему устройства, которое располагается между компьютером и модемом, или в виде карты (гибкого диска), вставляемой в дисковод компьютера.

Существуют и программные реализации средств аутентификации на основе одноразовых паролей *(программные ключи).* Программные ключи размещаются на сменном магнитном диске в виде обычной программы, важной частью которой является генератор одноразовых паролей. Применение программных ключей и присоединяемых к компьютеру карточек связано с некоторым риском, так как пользователи часто забывают гибкие диски в машине или не отсоединяют карточки от ноутбуков.

Независимо от того, какую реализацию системы аутентификации на основе одноразовых паролей выбирает пользователь, он, как и в системах аутентификации с использованием многоразовых паролей, сообщает системе свой идентификатор, однако вместо того, чтобы вводить каждый раз один и тот же пароль, он указывает последовательность цифр, сообщаемую ему аппаратным или программным ключом. Через определенный небольшой период времени генерируется другая последовательность – новый пароль. Аутентификационный сервер проверяет введенную последовательность и разрешает пользователю осуществить логический вход. Аутентификационный сервер может представлять собой отдельное устройство, выделенный компьютер или же программу, выполняемую на обычном сервере.

Рассмотрим подробнее две схемы, основанные на использовании аппаратных ключей.

*Синхронизация по времени*

Механизм аутентификации в значительной степени зависит от производителя. Одним из наиболее популярных механизмов является схема, разработанная компанией Security Dynamics (рис. 5.13). Схема синхронизации основана на алгоритме, который через определенный интервал времени (изменяемый при желании администратором сети) генерирует случайное число. Алгоритм использует два параметра:

1) секретный ключ, представляющий собой 64-битное число, уникально назначаемое каждому пользователю и хранящееся одновременно в аппаратном ключе и в базе данных аутентификационного сервера;

2) значение текущего времени.

Когда удаленный пользователь пытается совершить логический вход в сеть, то ему предлагается ввести его личный персональный номер (PIN), состоящий из 4 десятичных цифр, а также 6 цифр случайного числа, отображаемого в тот момент на дисплее аппаратного ключа. На основе PIN-кода сервер извлекает из базы данных информацию о пользователе, а именно его секретный ключ.

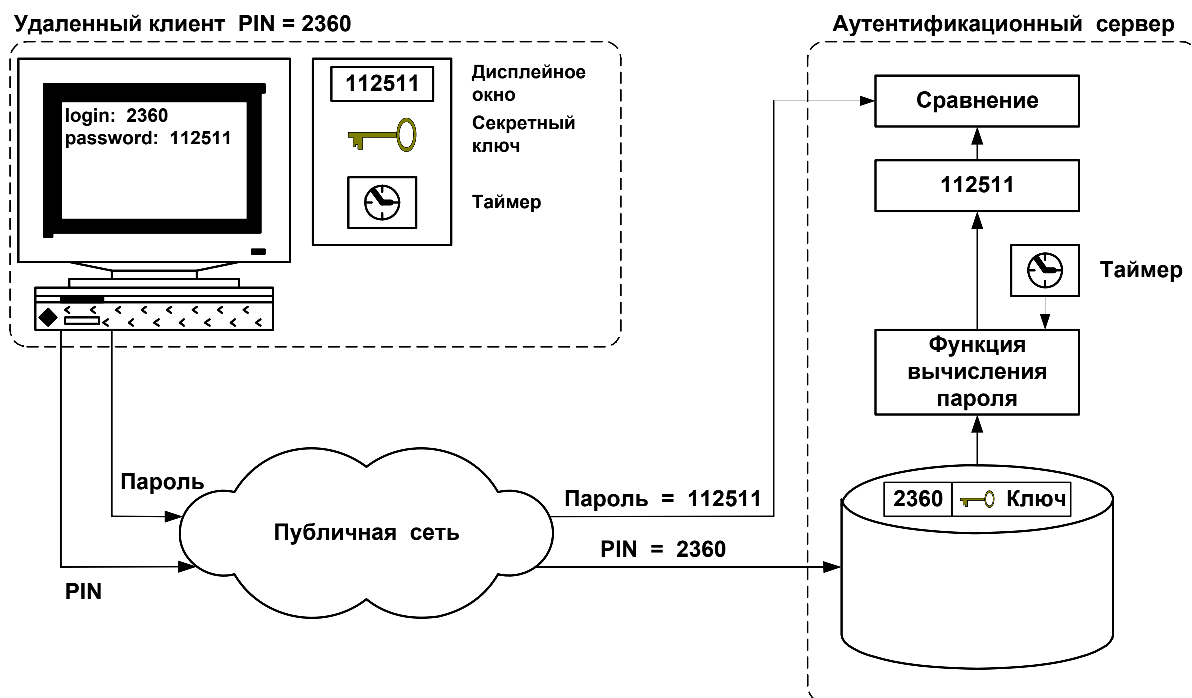



Рис. 5.13. Аутентификация, основанная на временной синхронизации

Затем сервер выполняет алгоритм генерации случайного числа, используя в качестве параметров найденный секретный ключ и значение текущего времени, и проверяет, совпадает ли сгенерированное число с числом, которое ввел пользователь. Если они совпадают, то пользователю разрешается логический вход.

Потенциальной проблемой этой схемы является временная синхронизация сервера и аппаратного ключа (ясно, что вопрос согласования часовых поясов решается просто). Гораздо сложнее обстоит дело с постепенным рассогласованием внутренних часов сервера и аппаратного ключа, тем более что потенциально аппаратный ключ может работать несколько лет. Компания Security Dynamics решает эту проблему двумя способами. Во-первых, при производстве аппаратного ключа измеряется отклонение частоты его таймера от номинала. Далее эта величина учитывается в виде параметра алгоритма сервера. Во-вторых, сервер отслеживает коды, генерируемые конкретным аппаратным ключом, и если таймер данного ключа постоянно спешит или отстает, то сервер динамически подстраивается под него.

Существует еще одна проблема, связанная со схемой временной синхронизации. Случайное число, генерируемое аппаратным ключом, является достоверным паролем в течение определенного интервала времени. Теоретически возможно, что очень проворный хакер может перехватить PIN-код и случайное число и использовать их для доступа к какому-либо серверу сети.

*Схема с использованием слова-вызова*

Другая схема применения аппаратных ключей основана на идее, очень сходной с рассмотренной выше идеей сетевой аутентификации. В том и другом случаях используется слово-вызов. Такая схема получила название “запрос-ответ”. Когда пользователь пытается осуществить логический вход, то аутентификационный сервер передает ему запрос в виде случайного числа (рис. 5.14). Аппаратный ключ пользователя зашифровывает это случайное число, используя алгоритм DES и секретный ключ пользователя. Секретный ключ пользователя хранится в базе данных сервера и в памяти аппаратного ключа. В зашифрованном виде слово-вызов возвращается на сервер. Сервер, в свою очередь, также зашифровывает сгенерированное им самим случайное число с помощью алгоритма DES и того же секретного ключа пользователя, а затем сравнивает результат с числом, полученным от аппаратного ключа. Как и в методе временной синхронизации, в случае совпадения этих двух чисел пользователю разрешается вход в сеть.

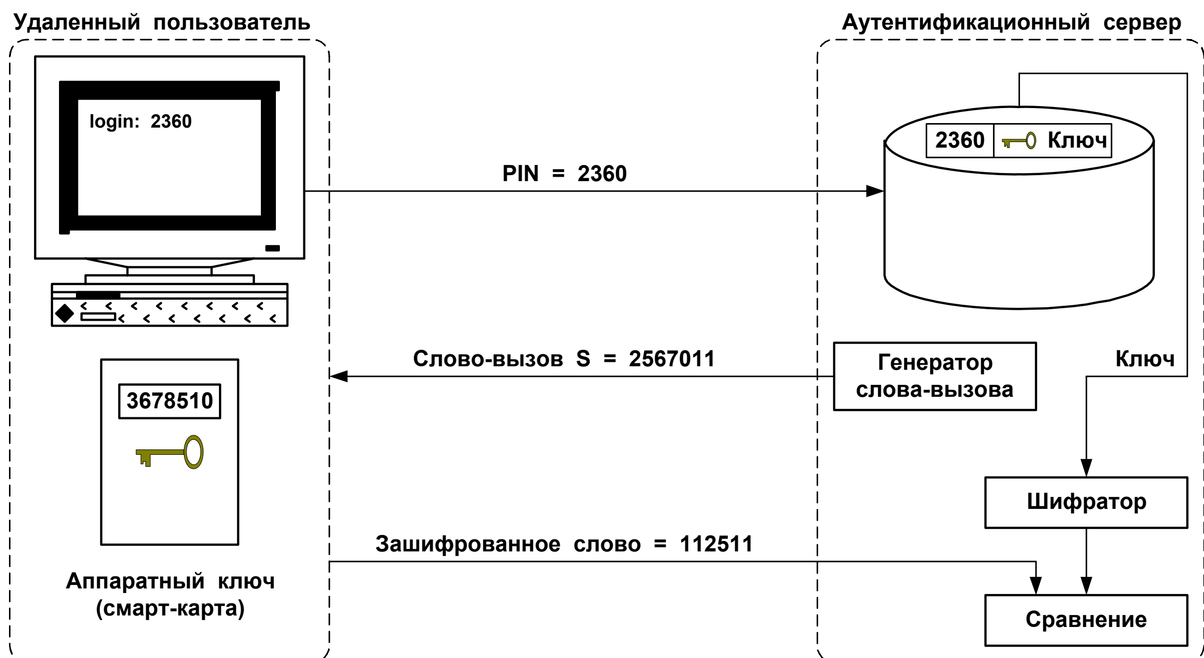



Рис. 5.14. Аутентификация по схеме “запрос-ответ”

### 3.1.3. Аутентификация на основе сертификатов

Механизм слова-вызова имеет свои ограничения – он обычно требует наличия компьютера на каждом конце соединения, так как аппаратный ключ должен иметь возможность как получать, так и отправлять информацию. А схема временной синхронизации позволяет ограничиться простым терминалом или факсом. В этом случае пользователи могут даже

вводить свой пароль с телефонной клавиатуры, когда звонят в сеть для получения голосовой почты.

Схема “запрос-ответ” уступает схеме временной синхронизации по простоте использования. Для логического входа по схеме временной синхронизации пользователю достаточно набрать 10 цифр. Схемы же “запрос-ответ” могут потребовать от пользователя выполнения большего числа ручных действий. В некоторых схемах “запрос-ответ” пользователь должен сам вводить секретный ключ, а затем набирать на клавиатуре компьютера полученное с помощью аппаратного ключа зашифрованное слово-вызов. В некоторых случаях пользователь должен вторично совершить логический вход в коммуникационный сервер уже после аутентификации.

Аутентификация с применением цифровых сертификатов является альтернативой использованию паролей и представляется естественным решением в условиях, когда число пользователей сети (пусть и потенциальных) измеряется миллионами. В таких обстоятельствах процедура предварительной регистрации пользователей, связанная с назначением и хранением их паролей, становится крайне обременительной, опасной, а иногда и просто нереализуемой. При использовании сертификатов сеть, которая дает пользователю доступ к своим ресурсам, не хранит никакой информации о своих пользователях – они ее предоставляют сами в своих запросах в виде сертификатов, удостоверяющих личность пользователей. Сертификаты выдаются специальными уполномоченными организациями – *центрами сертификации (Certificate Authority, CA).* Поэтому задача хранения секретной информации (закрытых ключей) возлагается на самих пользователей, что делает это решение гораздо более масштабируемым, чем вариант с использованием централизованной базы паролей.

*Схема использования сертификатов*

Аутентификация личности на основе сертификатов происходит примерно так же, как на проходной большого предприятия. Вахтер пропускает людей на территорию на основании пропуска, который содержит фотографию и подпись сотрудника, удостоверенных печатью предприятия и подписью лица, выдавшего пропуск. Сертификат является аналогом пропуска и выдается по запросам специальными сертифицирующими центрами при выполнении определенных условий.

Сертификат представляет собой электронную форму, в которой содержится следующая информация:

1) открытый ключ владельца данного сертификата;

2) сведения о владельце сертификата, такие, например, как имя, адрес электронной почты, наименование организации, в которой он работает, и т. п.;

3) наименование сертифицирующей организации, выдавшей данный сертификат.

Кроме того, сертификат содержит электронную подпись сертифицирующей организации – зашифрованные закрытым ключом этой организации данные, содержащиеся в сертификате.

Использование сертификатов основано на предположении, что сертифицирующих организаций немного и их открытые ключи могут быть всем известны каким-либо способом, например, из публикаций в журналах.

Когда пользователь хочет подтвердить свою личность, он предъявляет свой сертификат в двух формах – открытой (то есть такой, в которой он получил его в сертифицирующей организации) и зашифрованной с применением своего закрытого ключа. Сторона, проводящая аутентификацию, берет из открытого сертификата открытый ключ пользователя и расшифровывает с помощью него зашифрованный сертификат. Совпадение результата с открытым сертификатом подтверждает факт, что предъявитель действительно является владельцем закрытого ключа, парного с указанным открытым.

Затем с помощью известного открытого ключа указанной в сертификате организации проводится расшифровка подписи этой организации в сертификате. Если в результате получается тот же сертификат с тем же именем пользователя и его открытым ключом – значит, он действительно прошел регистрацию в сертификационном центре, является тем, за кого себя выдает, и указанный в сертификате открытый ключ действительно принадлежит ему.

Сертификаты можно использовать не только для аутентификации, но и для предоставления избирательных прав доступа. Для этого в сертификат могут вводиться дополнительные поля, в которых указывается принадлежность его владельцев той или иной категории пользователей. Эта категория назначается сертифицирующей организацией в зависимости от условий, на которых выдается сертификат. Например, организация, поставляющая через Интернет на коммерческой основе информацию, может выдавать сертификаты определенной категории пользователям, оплатившим годовую подписку на некоторый бюллетень, а Web-сервер будет предоставлять доступ к страницам бюллетеня только пользователям, предъявившим сертификат данной категории.

Подчеркнем тесную связь открытых ключей с сертификатами. Сертификат является не только удостоверением личности, но и удостоверением принадлежности открытого ключа. Цифровой сертификат устанавливает и гарантирует соответствие между открытым ключом и его владельцем. Это предотвращает угрозу подмены открытого ключа. Если некоторому абоненту поступает открытый ключ в составе сертификата, то он может быть уверен, что этот открытый ключ гарантированно принадлежит отправителю, адрес и другие сведения о котором содержатся в этом сертификате.

При использовании сертификатов отпадает необходимость хранить на серверах корпораций списки пользователей с их паролями, вместо этого достаточно иметь на сервере список имен и открытых ключей сертифици-

рующих организаций. Может также понадобиться некоторый механизм отображений категорий владельцев сертификатов на традиционные группы пользователей для того, чтобы можно было использовать в неизменном виде механизмы управления избирательным доступом большинства операционных систем или приложений.

*Инфраструктура с открытыми ключами*

Несмотря на активное использование технологии цифровых сертификатов во многих системах безопасности, эта технология еще не решила целый ряд серьезных проблем. Это прежде всего поддержание базы данных о выпущенных сертификатах. Сертификат выдается не навсегда, а на некоторый вполне определенный срок. По истечении срока годности сертификат должен либо обновляться, либо аннулироваться. Кроме того, необходимо предусмотреть возможность досрочного прекращения полномочий сертификата. Все заинтересованные участники информационного процесса должны быть вовремя оповещены о том, что некоторый сертификат уже не действителен. Для этого сертифицирующая организация должна оперативно поддерживать список аннулированных сертификатов.

Имеется также ряд проблем, связанных с тем, что сертифицирующие организации существуют не в единственном числе. Все они выпускают сертификаты, но даже если эти сертификаты соответствуют единому стандарту (сейчас это, как правило, стандарт X.509), все равно остаются нерешенными многие вопросы. Все ли сертифицирующие центры заслуживают доверия? Каким образом можно проверить полномочия того или иного сертифицирующего центра? Можно ли создать иерархию сертифицирующих центров, когда сертифицирующий центр, стоящий выше, мог бы сертифицировать центры, расположенные ниже по иерархии? Как организовать совместное использование сертификатов, выпущенных разными сертифицирующими организациями?

Для решения упомянутых выше и многих других проблем, возникающих в системах, использующих технологии шифрования с открытыми ключами, оказывается необходимым комплекс программных средств и методик, называемый *инфраструктурой с открытыми ключами (Public Key Infrastructure, PKI)*. Информационные системы больших предприятий нуждаются в специальных средствах администрирования и управления цифровыми сертификатами, парами открытых/закрытых ключей, а также приложениями, функционирующими в среде с открытыми ключами.

В настоящее время любой пользователь имеет возможность, загрузив широко доступное программное обеспечение, абсолютно бесконтрольно сгенерировать себе пару открытый/закрытый ключ. Затем он может также совершенно независимо от администрации вести шифрованную переписку со своими внешними абонентами. Такая “свобода” пользователя часто не соответствует принятой на предприятии политике безопасности. Для обеспечения более надежной защиты корпоративной информации желательно реализовать централизованную службу генерации и распределения

ключей. Для администрации предприятия важно иметь возможность получить копии закрытых ключей каждого пользователя сети, чтобы в случае увольнения пользователя или потери пользователем его закрытого ключа сохранить доступ к зашифрованным данным этого пользователя. В противном случае резко ухудшается одна из трех характеристик безопасной системы – доступность данных.

Процедура, позволяющая получать копии закрытых ключей, называется *восстановлением ключей.* Вопрос, включать ли в продукты безопасности средства восстановления ключей, в последние годы приобрел политический оттенок. В Соединенных Штатах Америки прошли бурные дебаты, тему которых можно примерно сформулировать так: обладает ли правительство правом иметь доступ к любой частной информации при условии, что на это есть постановление суда?

И хотя в такой широкой постановке проблема восстановления ключей все еще не решена, необходимость наличия средств восстановления в корпоративных продуктах ни у кого не вызывает никаких сомнений. Принцип доступности данных не должен нарушаться из-за волюнтаризма сотрудников, монопольно владеющих своими закрытыми ключами. Ключ может быть восстановлен при выполнении некоторых условий, которые должны быть четко определены в политике безопасности предприятия.

Как только принимается решение о включении в систему безопасности средств восстановления, возникает вопрос, как же быть с надежностью защиты данных, как обеспечить пользователю уверенность в том, что его закрытый ключ не используется с какими-либо другими целями, отличными от резервирования? Некоторую уверенность в секретности хранения закрытых ключей может дать технология *депонирования ключей.* Депонирование ключей – это предоставление закрытых ключей на хранение третьей стороне, надежность которой не вызывает сомнений. Этой третьей стороной может быть правительственная организация или группа уполномоченных на это сотрудников предприятия, которым оказывается полное доверие.

## 3.2. Аутентификация информации

Под аутентификацией информации в компьютерных системах понимают установление подлинности данных, полученных по сети, исключительно на основе информации, содержащейся в полученном сообщении.

Если конечной целью шифрования информации является обеспечение защиты от несанкционированного ознакомления с этой информацией, то конечной целью аутентификации информации является обеспечение защиты участников информационного обмена от навязывания ложной информации. Концепция аутентификации в широком смысле предусматри-

вает установление подлинности информации как при условии наличия взаимного доверия между участниками обмена, так и при его отсутствии.

В компьютерных системах выделяют два вида аутентификации информации:

1) аутентификация хранящихся массивов данных и программ – установление того факта, что данные не подвергались модификации;

2) аутентификация сообщений – установление подлинности полученного сообщения, в том числе решение вопроса об авторстве этого сообщения и установление факта приема.

### 3.2.1. Цифровая подпись

Для решения задачи аутентификации информации используется концепция цифровой (или электронной) подписи. Согласно терминологии, утвержденной Международной организацией по стандартизации (ISO), под термином “цифровая подпись” понимаются методы, позволяющие устанавливать подлинность автора сообщения (документа) при возникновении спора относительно авторства этого сообщения. Основная область применения цифровой подписи – это финансовые документы, сопровождающие электронные сделки, документы, фиксирующие международные договоренности и т. п. До настоящего времени наиболее часто для построения схемы цифровой подписи использовался алгоритм RSA (рис. 5.15). В основе этого алгоритма лежит концепция Диффи-Хеллмана. Она заключается в том, что каждый пользователь сети имеет свой закрытый ключ, необходимый для формирования подписи; соответствующий этому секретному ключу открытый ключ, предназначенный для проверки подписи, известен всем другим пользователям сети.

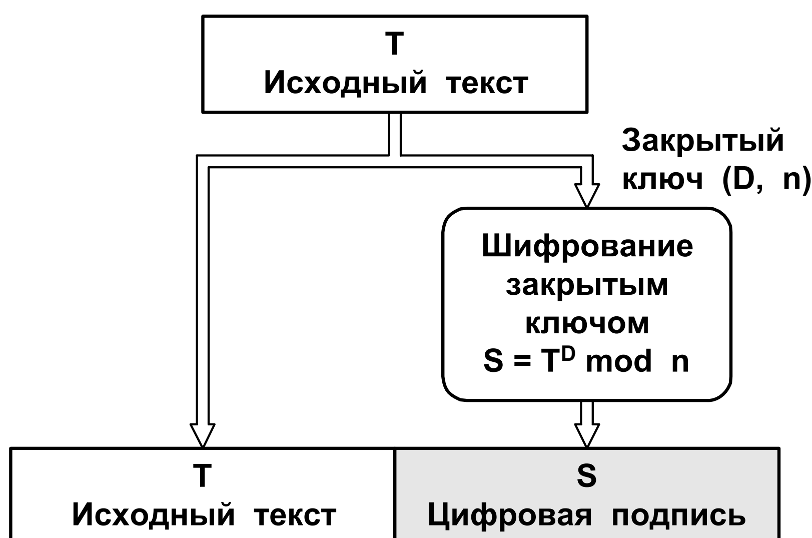



Рис. 5.15 Схема формирования цифровой подписи по алгоритму RSA

Подписанное сообщение состоит из двух частей: незашифрованной части, в которой содержится исходный текст T, и зашифрованной части, представляющей собой цифровую подпись. Цифровая подпись S вычисляется с использованием закрытого ключа (D, n) по формуле

$$S = T^D \bmod n.$$

Сообщение посылается в виде пары (T, S). Каждый пользователь, имеющий соответствующий открытый ключ (E, n), получив сообщение, отделяет открытую часть T, расшифровывает цифровую подпись S и проверяет равенство

$$T = S^E \bmod n.$$

Если результат расшифровки цифровой подписи совпадает с открытой частью сообщения, то считается, что документ подлинный, не претерпел никаких изменений в процессе передачи, а автором его является именно тот человек, который передал свой открытый ключ получателю. Если сообщение снабжено цифровой подписью, то получатель может быть уверен, что оно не было изменено или подделано по пути. Такие схемы аутентификации называются асимметричными. К недостаткам данного алгоритма можно отнести то, что длина подписи в этом случае равна длине сообщения, что не всегда удобно.

Цифровые подписи применяются к тексту до того, как он шифруется. Если помимо снабжения текста электронного документа цифровой подписью надо обеспечить его конфиденциальность, то вначале к тексту применяют цифровую подпись, а затем шифруют все вместе: и текст, и цифровую подпись (рис. 5.16).

Другие методы цифровой подписи основаны на формировании соответствующей сообщению контрольной комбинации с помощью классических алгоритмов типа DES. Учитывая более высокую производительность алгоритма DES по сравнению с алгоритмом RSA, он более эффективен для подтверждения аутентичности больших объемов информации. А для коротких сообщений типа платежных поручений или квитанций подтверждения приема, наверное, лучше подходит алгоритм RSA.

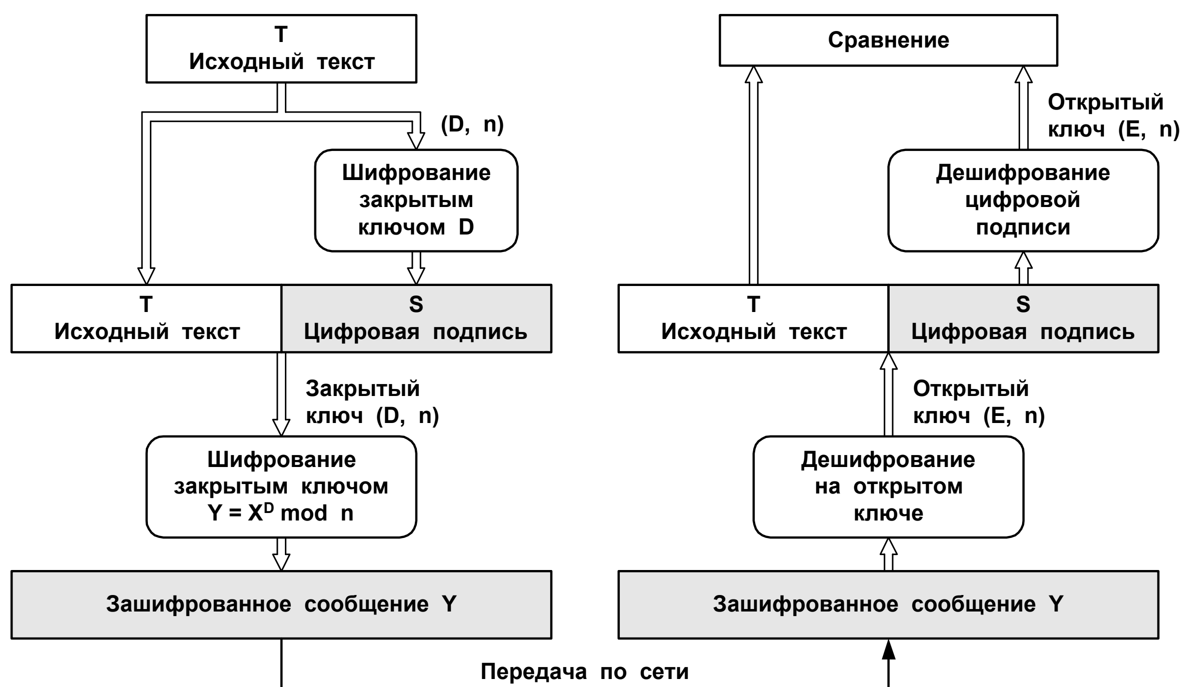



Рис. 5.16. Обеспечение конфиденциальности документа с цифровой подписью

### 3.2.2. Аутентификация программных кодов

Компания Microsoft разработала средства для доказательства аутентичности программных кодов, распространяемых через Интернет. Пользователю важно иметь доказательства, что программа, которую он загрузил с какого-либо сервера, действительно содержит коды, разработанные определенной компанией. Протоколы защищенного канала типа SSL помочь здесь не могут, так как позволяют удостоверить только аутентичность сервера. Microsoft разработала технологию аутентикода (Authenticode), суть которой состоит в следующем.

Организация, желающая подтвердить свое авторство на программу, должна встроить в распространяемый код так называемый подписывающий блок (рис. 5.17). Этот блок состоит из двух частей. Первая часть – сертификат этой организации, полученный обычным образом от какого-либо сертифицирующего центра. Вторую часть образует зашифрованный дайджест, полученный в результате применения односторонней функции к распространяемому коду. Шифрование дайджеста выполняется с помощью закрытого ключа организации.

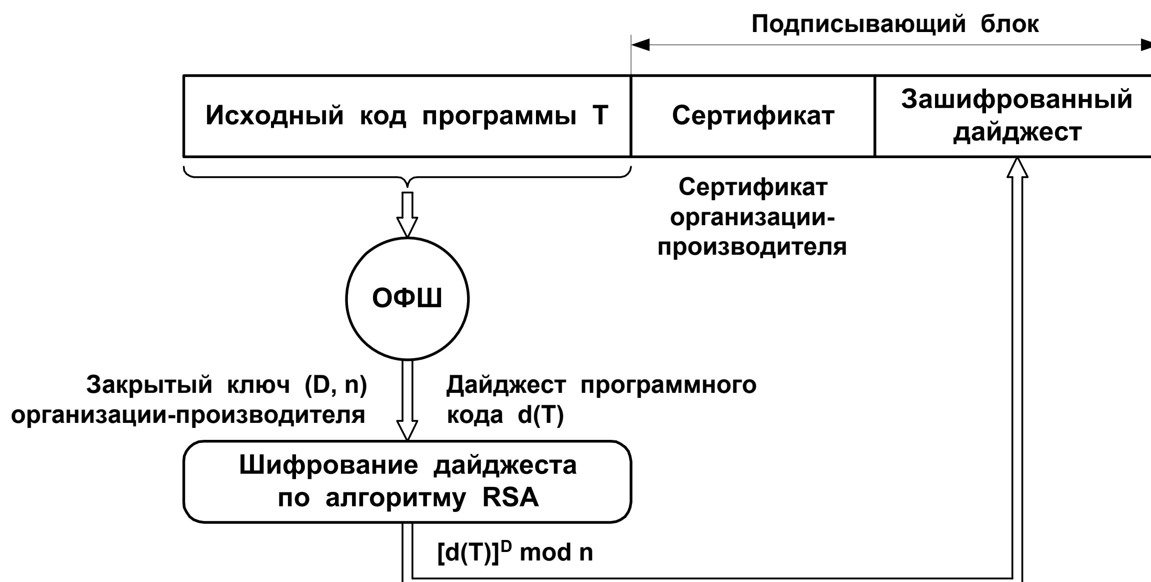



Рис. 5.17. Схема получения аутентикода

## ЗАКЛЮЧЕНИЕ

В заключение изучения курса “Операционные системы” рассмотрим существующие семейства операционных систем и охарактеризуем основные операционные системы, применяемые в настоящее время.

### MVS, OS/390, z/OS

Первые две ОС этого семейства вышли в 1966 г., вскоре после анонса аппаратной архитектуры System/360. Это были PCP (Primary Control program – первичная управляющая программа) и DOS/360 (Disk Operating System). Архитектура обеих систем была типична для вычислительных систем второго поколения – это были пакетные мониторы, рассчитанные на работу одной прикладной программы без защиты памяти (первые компьютеры серии System/360 не имели диспетчеров памяти).

В 1967 г. были выпущены версии PCP: MVT (Multiprogramming with a Variable number of Tasks – многопрограммная [система] с переменным числом задач) и MFT (Multiprogramming with a Fixed number of Tasks – то же, но с фиксированным числом задач). Обе системы реализовывали вытесняющую многозадачность в едином адресном пространстве – MFT использовала разделы памяти, а MVT – относительную загрузку с базовым регистром. Позднее, к MVT была добавлена подсистема работы с несколькими терминалами в режиме разделения времени TSO (Timesharing Option – возможность разделения времени), ASP (Asymmetric Multiprocessing System – асимметричная многопроцессорность) и ряд других прикладных подсистем.

В 1972 г., после появления машин System/370 с диспетчером памяти, была выпущена переходная система OS/SVS (Single Virtual Storage – единая виртуальная память), которая позволяла использовать страничную подкачку, но не защиту заданий друг от друга. Наконец, в 1974 г. была выпущена MVS (Multiple Virtual Storage – множественная виртуальная память), которая предоставляла каждой задаче собственное виртуальное адресное пространство объемом до 2 Гбайт (в System/360 и первых моделях System/370 адрес был 24-разрядным). Большая часть дополнительных подсистем MVT была включена в стандартную поставку MVS.

MVS воплотила все наиболее прогрессивные функции и архитектурные концепции своего времени. MVS была основным продуктом IBM вплоть до середины 90-х, когда вышла новая версия системы – OS/390.

В OS/390 основные архитектурные принципы MVS не подверглись пересмотру. Новшества заключались в добавлении следующих возможностей:

поддержка многомашинных кластеров (Parallel Sysplex);

развитие сетевых средств равноправного (peer-to-peer) взаимодействия;

поддержка сетевых протоколов семейства TCP/IP (ранее взаимодействие с сетями TCP/IP и NETBIOS осуществлялось посредством включения в состав вычислительного комплекса модуля с процессором x86 под управлением OS/2);

совместимость с системами семейства Unix (в 1998 г.OS/390 прошла набор тестов Unix 95 консорциума X/Open и получила право называться UNIX).

В 1999 г., в связи с началом выпуска 64-разрядного семейства компьютеров z900, вышла 64-разрядная версия системы z/OS.

Системы под управлением OS/390 и z/OS применяются главным образом в качестве серверов транзакций и СУБД масштаба предприятия и составляют становой хребет вычислительных систем большинства крупных компаний.

## Семейство Unix

Обширное и бурно развивающееся семейство Unix оказало огромное идейное влияние на развитие операционных систем в 80-е и 90-е годы XX столетия.

Применения систем семейства крайне разнообразны, начиная от встраиваемых приложений реального времени, включая графические рабочие станции для САПР и геоинформационных систем, и заканчивая серверами класса предприятия и массивно параллельными суперкомпьютерами. Некоторые важные рыночные ниши, например передачу почты и другие структурные сервисы Internet, системы семейства занимают практически монопольно.

Первые версии UNIX были рассчитаны на машины без диспетчера памяти. Процессы загружались в единое адресное пространство. Ядро системы размещалось в нижних адресах ОЗУ, начиная с адреса 0, и называлось *сегментом реентерабельных процедур.* Реентерабельность обеспечивалась переустановкой стека в момент системного вызова и запретом переключения задач на все время исполнения модулей ядра. На машинах с базовой адресацией выполнялось перемещение образов процессов по памяти и сброс образа процесса на диск (задачный своппинг).

В 1973 г. одна из дочерних компаний AT&T (ведущей разработку UNIX), Western Electric, дала разрешение на использование UNIX в некоммерческих целях. Началось распространение системы в университетах США. Наибольший вклад в распространение и развитие университетской версии системы внес университет Беркли, в котором было создано специальное подразделение – BSD (Berkeley Software Distribution}.

В BSD UNIX было включено множество ценных нововведений, таких, как:

сегментная (на старших моделях PDP-11) и страничная (на VAX-11/780) виртуальная память;

раздельные адресные пространства процессов и выделенное адресное пространство ядра;

абсолютные загрузочные модули формата a.out;

примитивная форма разделяемых библиотек;

усовершенствования механизма обработки сигналов;

управление сессиями и заданиями в пределах сессии.

Самое важное нововведение было сделано в начале 80-х, когда в рамках работ по проекту DARPA сетевое программное обеспечение ARPANet было перенесено с TOPS/20 на BSD Unix. Вскоре сетевой стек BSD стал референтной реализацией (реализация, на совместимость с которой тестируют все остальные) того, что ныне известно как семейство протоколов TCP/IP.

В 1987 г. AT&T выпустила версию UNIX System V Release 3, включавшая в себя асинхронные драйверы последовательных устройств (STREAMS), универсальный API для доступа к сетевым протоколам (TLI), средства межпроцессного взаимодействия (семафоры, очереди сообщений и сегменты разделяемой памяти).

UNIX System V Release 4 вышла на рынок в 1989 г. под названием UNIX SVR4. Микроядерная система обеспечивала полную бинарную совместимость с SVR3, бинарную же совместимость с 16- и 32-разрядными Xenix на процессоре x86, и совместимость на уровне исходных текстов с BSD Unix v4.3. Заявленная цель консолидации всех основных ветвей Unix в единой системе была полностью достигнута. Sun Microsystems приступила к переводу своих пользователей на Sun OS 5.x (ныне известна как Solaris), основанную на ядре SVR4.

Версия SVR4 была этапной – она включала в себя следующие компоненты:

многопоточное микроядро;

класс планирования реального времени (процессы с этим классом планирования имеют приоритет выше, чем нити ядра);

новый формат загрузочного модуля ELF (Executable and Linking Format), обеспечивавший удобную работу с разделяемыми и динамическими библиотеками;

динамическое подключение и отключение областей свопинга;

динамическую загрузку и выгрузку модулей ядра;

многопоточность в пределах одного процесса (так называемые LWP (Light Weight Processes – легкие процессы));

псевдофайловую систему /proc, обеспечивающую контролируемый доступ к адресным пространствам других процессов и структурам данных ядра;

оптимизирующий компилятор ANSI C, по качеству кода не уступающий GNU C.

На всем протяжении 90-х, архитектура ядра не подверглась существенным изменениям. Как и MVS полутора десятилетиями раньше, UNIX достиг совершенства в своем роде и нуждается не в новой архитектуре, а только в оптимизации существующего кода (ядро SVR4 несколько тяжеловато по сравнению с монолитными ядрами BSD и Linux) и развитии отдельных подсистем.

*Linux*

В 1991 г. Л. Торвальдс, в тот момент – студент университета Хельсинки, приступил к разработке того, что ныне известно как Linux – полноценной операционной системы. В 1992 г. была выпущена первая публичная версия системы. Вышедшее в 1997 г. ядро Linux 2.0 имело вполне приемлемую по стандартам коммерческих ОС надежность и почти все наиболее прогрессивные черты других Unix-систем:

загрузочные модули и разделяемые библиотеки формата ELF;

псевдофайловую систему /proc;

динамическое подключение и отключение своп-файлов;

длинные файлы (64-разрядные – длина файла и смещение в нем);

многопоточность в пределах одного процесса (POSIX thread library);

поддержку симметричной многопроцессорности;

динамическую загрузку и выгрузку модулей ядра;

стек TCP/IP, совместимый с BSD 4.4, с поддержкой IPSec, фильтрации пакетов и др.;

SysV IPC;

бинарную совместимость с UNIX System V на процессорах x86 (iBCS – Intel Binary Compatibility Standard) и, позднее, на SPARC и MIPS;

поддержку задач реального времени (класс планирования реального времени в монолитном Linux невозможен; такие задачи загружаются как модули ядра).

Linux и другие подобные ему операционные системы были изначально разработаны для использования несколькими людьми на одной и той же машине, и для защиты этих людей друг от друга. Интерфейс пользователя в Linux отделен от ядра, привилегированной основы операционной системы. И это ядро тщательно защищено от модифицирования обычными программами. Вот почему Linux не имеет вирусов.

Ядро Linux быстро развивается и еще не достигло той степени “зрелости” и стабильности, которая характерна для SVR4 и ветвей BSD. В част-

ности, поэтому среднее количество опасных ошибок, обнаруживаемых в системе за фиксированный интервал времени, существенно выше, чем в двух указанных ОС; производительность отдельных подсистем также оставляет желать лучшего. Однако положение довольно быстро улучшается и, по-видимому, в обозримом будущем Linux может стать одним из технологических лидеров отрасли.

## Семейство CP/M

Родоначальником семейства является дисковая операционная система CP/M (Control Program/Monitor) фирмы Digital Research. Первая версия системы была разработана в 1974 г. для использования в инструментальных микропроцессорных системах на основе микропроцессоров 18080 и i8085.

CP/M была первой ОС для машин такого рода, обеспечившей возможность использования гибких дисков, поэтому она быстро приобрела огромную популярность и стала стандартом де-факто для микрокомпьютеров. Система была перенесена практически на все 8- и 16-разрядные и многие 32-разрядные микропроцессоры манчестерской архитектуры. Появившиеся в конце 70-х персональные компьютеры обычно также были ориентированы на использование CP/M. В начале 80-х были реализованы многозадачная и сетевая версии CP/M. Появилось также немало клонов системы, программно совместимых с ней и в целом аналогичных по архитектуре.

В 1981 г. фирма IBM анонсировала персональный компьютер IBM PC на основе 16-разрядного процессора i8088. Первоначально предполагалось, что в качестве ОС для этого компьютера будет использоваться CP/M, однако представителям IBM не удалось достичь приемлемых условий соглашения с Digital Research. Фирма Microsoft переделала загрузчик и дисковую подсистему QDOS для работы с IBM PC и использования сервисов ПЗУ этого компьютера (эти сервисы также называются BIOS, хотя имеют довольно мало общего с BIOS CP/M), и предложила результат фирме IBM. К версии 3.30 MS DOS (такое название получила новая система) уже накопила очень много отличий от оригинальной CP/M. В качестве файловой системы была использована изобретенная лично Б. Гейтсом для применения в интерпретаторе BASIC ФС FAT. Со времен DOS 3.30 архитектура системы не подверглась сколько-нибудь заметным изменениям. Так, DOS 7, входящая в состав Windows 98/ME в качестве вторичного загрузчика, отличается от 3.30 только поддержкой файловой системы FAT32.

Ближе к середине 90-х стало очевидно, что дни DOS как платформы сочтены.

*Windows NT/2000/XP*

Наработки Microsoft по OS/2 New Technology были в 1993 г. выпущены на рынок под названием Windows NT. Версии 3.x и 4.0 этой системы обеспечивали совместимость с 16-разрядными приложениями для OS/2 1.x в отдельной подсистеме, без возможности обращаться из 16-разрядных приложений к 32-разрядным DLL и наоборот.

Наиболее важной заимствованной концепцией была журнальная файловая система NTFS, представляющая собой любопытный гибрид HPFS (основной ФС OS/2) и FCS2 (основной ФС VAX/VMS). Это заимствование следует признать довольно удачным.

Гораздо менее удачным было заимствование своеобразной стратегии управления рабочими множествами процессов в ОЗУ, используемой в VMS: разработчики Microsoft устранили из этой стратегии одно из ключевых понятий, квоту размера рабочего множества. В результате получилась система, практически не способная воспользоваться преимуществами страничной подкачки, потому что даже небольшая нехватка оперативной памяти приводит к резкому падению производительности из-за неспособности системы сбалансировать потребности приложений и дискового кэша.

Еще одна ключевая для понимания архитектуры Win32 концепция была введена по настоятельным просьбам разработчиков графических приложений для Apple Macintosh. Речь идет о *системном реестре (system registry),* централизованной базе данных, в которой все модули системы, стандартные утилиты и прикладные программы хранят все, что считают нужным сохранить.

Системный реестр впервые был реализован в Mac OS. Эта система имеет довольно простое ядро и небогатый набор системных настроек, поэтому реестр Mac OS в основном содержит настройки прикладных программ и в такой форме вполне терпим. Напротив, довольно сложная многопользовательская Windows NT, поддерживающая широкий спектр внешних устройств, нуждается в большом объеме конфигурационных данных для самой системы. Характер обращений к разным частям этих данных сильно различается – некоторые, например, нужны только при загрузке системы, а изменению подлежат только при изменении аппаратной конфигурации. Другие же меняются при каждом открытии нового окна пользовательской программой. Относительная ценность этих данных также различается очень резко: искажение некоторых может привести к невозможности загрузить систему или к потере пользователями доступа к ней, некоторые другие можно было бы и не хранить вовсе. В Windows NT этот концептуальный недостаток усугубляется недостатками реализации – реестр не имеет адекватных средств резервного копирования и восстановления (при фатальных повреждениях реестра Microsoft рекомендует переустановку системы) и фактически лишен средств самоконтроля и диагностики.

С момента выхода версии 3.51 и до настоящего времени архитектура системы не подвергалась ни пересмотру, ни сколько-нибудь существенному развитию. Наибольшее количество новшеств было введено в Windows 2000, когда в состав системы была включена служба каталогов Active Directory и ряд мелких улучшений. В частности, было исправлено множество мелких, но раздражающих недостатков стека TCP/IP версии 4.0. В Windows XP были введены средства импорта реестра систем линии Windows 95/98/ME, сокеты типа RAW и новая схема защиты от неавторизованного копирования. При слабо изменяющемся ядре, развивался преимущественно пользовательский интерфейс.

Главными недостатками Windows NT в качестве сервера являются неэффективная стратегия управления памятью и беспрецедентное для промышленно эксплуатируемого программного продукта количество проблем с безопасностью. Для сервера, исполняющего стабильную смесь приложений, первый недостаток не очень критичен – многие серверные приложения (особенно серверы СУБД) при старте занимают всю доступную память и практически не запрашивают ее в процессе работы. Благодаря этому система может стабилизировать свой динамический кэш и обеспечивать в стационарном режиме приемлемую производительность.

Второй недостаток является критически важным и одного его – в идеальном мире – было бы достаточно, чтобы Windows NT не использовалась в качестве сервера ни при каких обстоятельствах.

*Windows 95/98/ME*

В первой половине 90-х годов XX столетия практически всем разработчикам и техническим специалистам было очевидно, что MS и DR DOS доживают последние дни: они не удовлетворяли запросам пользователей практически ни по одному из параметров: приложения требовали больших объемов памяти и перехода к 32-разрядной архитектуре, пользователям требовались большая надежность, многозадачность, более развитые сетевые средства. Напротив, преимущества DOS, такие, как небольшая потребность в памяти, становились все менее и менее критичными.

Основным препятствием на пути перехода пользователей на другие платформы было требование совместимости с существующими приложениями и драйверами нестандартных внешних устройств для DOS. Наилучшим образом удовлетворяла этому требованию IBM OS/2, в виртуальной машине которой можно было запустить не только практически любое приложение DOS, но и использовать многие модули ядра DOS, в том числе – загружая в разных виртуальных машинах разные версии ДОС и разные наборы драйверов. Однако высокие требования этой системы к ресурсам и ориентированная на корпоративных пользователей схема лицензирования приводили к тому, что система не получила большого распространения на массовом рынке.

В 1992-1993 гг. Microsoft занялась разработкой системы, которая должна была заполнить перспективную рыночную нишу “многозадачной ДОС защищенного режима”.

Новая система, получившая название Windows 95, вышла на рынок в 1995 г. Система с самого начала задумывалась как переходная, предназначенная для облегчения перевода пользовательской базы DOS на Windows NT, однако прошло не менее 4-5 лет, прежде чем совместимость с приложениями DOS перестала быть решающим параметром при выборе ОС для настольного компьютера. За это время успело выйти несколько версий “переходной” системы (OSR2, 98, 98SE, Millennium Edition) и даже после выхода XP Microsoft еще не готова объявить о прекращении поддержки этой линии ОС.

# ЛИТЕРАТУРА

1. Гордеев А. В., Молчанов А. Ю. Системное программное обеспечение: Учебник. – СПб.: Питер, 2002.

2. Олифер В. Г., Олифер Н. А. Сетевые операционные системы: Учебник. – СПб.: Питер, 2002.

3. Соловьев Г. Н., Никитин В. Д. Операционные системы ЭВМ. Учебное пособие для студентов ВУЗов, обучающихся по специальностям “Электронные вычислительные машины, системы, комплексы и сети”, “Автоматизированные системы обработки информации и управления”. – М.: Высшая школа, 1989.

4. Иртегов Д. В. Введение в операционные системы. – СПб.: БХВ-Петербург, 2002.

5. Лопашинов П. М., Домников А. В. Операционные системы и системное программирование. Конспект лекций. Часть 1. Принципы построения операционных систем. Смоленск. Изд-во ВА ПВО СВ РФ. 1996.

6. Под ред. Ганитулина А. Х. Основы построения цифровых электронных вычислительных машин комплексов вооружения ПВО. Учебник. Пушкин. Изд-во ПВУРЭ ПВО. 1986.

7. Каган Б. М. Электронные вычислительные машины и системы: учебное пособие для ВУЗов. – 2-е изд. М.: Энергоатомиздат, 1985.

8. Расширенное техническое руководство по Windows NT Workstation 4.0. В двух книгах. М.: СК Пресс, 1998.

9. Андреев А. Г. и др. Microsoft Windows 2000 Professional. Русская версия / Под общ. ред. А. Н. Чекмарева и Д. Б. Вишнякова. – Спб.: БХВ-Петербург, 2003.