Министерство образования Республики Беларусь

Учреждение образования
«Полоцкий государственный университет»

**О. Н. ТРАВКИН**

# СИСТЕМНОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ

Учебно-методический комплекс для студентов специальностей
1-40 01 01 «Программное обеспечение информационных технологий»,
1-40 02 01 «Вычислительные машины, системы и сети»

Новополоцк
ПГУ
2009

УДК 004.45(075.8)
ББК 32.97я73
Т65

Рекомендовано к изданию методической комиссией
радиотехнического факультета в качестве учебно-методического
комплекса (протокол № 5 от 23.05.2008)

РЕЦЕНЗЕНТЫ:

зам начальника Новополоцкого ГУЭС А. П. Сушко;
канд. техн. наук, проректор по информатизации УО «ПГУ» Д. О. Глухов

**Травкин, О. Н.**

Т65 Системное программное обеспечение : учеб.-метод. комплекс / О. Н. Травкин. – Новополоцк : ПГУ, 2009. – 224 с.

ISBN 978-985-418-871-3.

Включает программу лекционных и лабораторных занятий, нормы оценки знаний студентов по итогам изучения курса. Предлагается базовый конспект лекций, вопросы и задания для самопроверки в конце каждой темы. Приведен список литературных источников для самостоятельной учебной работы студентов.

**УДК 004.45(075.8)**
**ББК 32.97я73**

**ISBN 978-985-418-871-3**

## СОДЕРЖАНИЕ

# ВВЕДЕНИЕ

## Цель и задачи дисциплины

Цель преподавания дисциплины состоит в получении комплекса знаний о принципах работы операционных систем (ОС) различных типов, концепциях их построения и внутренней архитектуре. Данная цель достигается путем изучения общих принципов построения операционных систем, которые справедливы для большинства ОС, а также фундаментальных концепций, положенных в их основу.

Главной задачей курса является приобретение знаний, способствующих пониманию студентами принципов функционирования и реализации системного программного обеспечения на архитектурном уровне, приобретение навыков и знаний, необходимых для его эксплуатации, приобретение знаний о его видах и разновидностях, а также приобретение навыков его разработки и внедрения.

В результате изучения дисциплины студенты должны знать:

- историю появления и развития операционных систем;
- назначение и функции операционной системы;
- виды архитектур операционных систем;
- понятие о процессах и потоках;
- виды памяти и принципы управления памятью;
- механизмы ввода-вывода и файловые системы;
- особенности реализации механизма мультипрограммирования в процессорах Pentium.

## Структура дисциплины

Согласно учебному плану, курс «Системное программное обеспечение» изучается студентами на 3 курсе (1 семестр), рассчитан на 64 аудиторных часа, включает в себя следующие виды занятий:

- 32 часа лекций;
- 32 часа лабораторных работ.

Далее представлено распределение курса по видам аудиторных занятий по разделам и темам.

# ЛЕКЦИОННЫЙ КУРС

| Наименования разделов и тем лекций и их содержание | Количество часов |
|---|---|
| 1 | 2 |
| **Введение в курс «Системное программное обеспечение»** | |
| Содержание дисциплины и ее взаимосвязь с другими дисциплинами. | 2 |
| **Раздел 1. Эволюция операционных систем** | |
| Появление первых операционных систем. Появление мультипрограммных операционных систем для мэйнфреймов. Операционные системы и глобальные сети. Операционные системы миникомпьютеров и первые локальные сети. Развитие операционных систем в 80-е годы XX в. Особенности современного этапа развития операционных систем. | 2 |
| **Раздел 2. Назначение и функции операционной системы** | |
| Операционные системы для автономного компьютера. ОС как виртуальная машина. ОС как система управления ресурсами. Функциональные компоненты ОС автономного компьютера. Управление процессами. Управление памятью. Управление файлами и внешними устройствами. Защита данных и администрирование. Интерфейс прикладного программирования. Пользовательский интерфейс. Сетевые операционные системы. Сетевые и распределенные ОС. Два значения термина «сетевая ОС». Функциональные компоненты сетевой ОС. Сетевые службы и сетевые сервисы. Встроенные сетевые службы и сетевые оболочки. Одноранговые и серверные сетевые ОС. Операционные системы в одноранговых сетях. Операционные системы в сетях с выделенными серверами. Требования к современным ОС. | 4 |
| **Раздел 3. Архитектура операционной системы** | |
| Ядро и вспомогательные модули ОС. Ядро в привилегированном режиме. Многослойная структура ОС. Аппаратная зависимость и переносимость ОС. Машинно-зависимые компоненты ОС. Переносимость ОС. Микроядерная архитектура. Коцепция микроядерной архитектуры. Преимущества и недостатки микроядерной архитектуры. Совместимость и множественные прикладные среды. Двоичная совместимость и совместимость исходных текстов. Трансляция библиотек. Способы реализации прикладных программных сред. | 6 |
| **Раздел 4. Процессы и потоки** | |
| Мультипрограммирование. Мультипрограммирование в системах пакетной обработки. Мультипрограммирование в системах разделения времени. Мультипрограммирование в системах реального времени. Мультипроцессорная обработка. Планирование процессов и потоков. Понятия «процесс» и «поток». Создание процессов и потоков. Планирование и диспетчеризация потоков. Состояния потока. Вытесняющие и невытесняющие алгоритмы планирования. Алгоритмы планирования, основанные на квантовании. Алгоритмы планирования, основанные на приоритетах. Смешанные алгоритмы планирования. Планирование в системах реального времени. Моменты перепланировки. | 6 |

| 1 | 2 |
|---|---|
| **Раздел 5. Управление памятью** | |
| Функции ОС по управлению памятью. Типы адресов. Алгоритмы распределения памяти. Распределение памяти фиксированными разделами. Распределение памяти динамическими разделами. Перемещаемые разделы. Свопинг и виртуальная память. Страничное распределение. Сегментное распределение. Сегментно-страничное распределение. Разделяемые сегменты памяти. Кэширование данных. Иерархия запоминающих устройств. Кэш-память. Принцип действия кэш-памяти. Проблема согласования данных. Способы отображения основной памяти на кэш. Схемы выполнения запросов в системах с кэш-памятью. | 6 |
| **Раздел 6. Файловая система** | |
| Логическая организация файловой системы. Цели и задачи файловой системы. Типы файлов. Иерархическая структура файловой системы. Имена файлов. Монтирование. Атрибуты файлов. Логическая организация файла. Физическая организация NTFS. Структура тома NTFS. Структура файлов NTFS. Каталоги NTFS. Файловые операции. Два способа организации файловых операций. Открытие файла. Обмен данными с файлом. Блокировки файлов. Стандартные файлы ввода и вывода, перенаправление вывода. Контроль доступа к файлам. Доступ к файлам как частный случай доступа к разделяемым ресурсам. Механизм контроля доступа. Организация контроля доступа в ОС UNIX. Организация контроля доступа в ОС Windows NT. | 6 |
| **ВСЕГО:** | **32** |

## ЛАБОРАТОРНЫЕ ЗАНЯТИЯ

| **Наименование лабораторной работы** | **Количество часов** |
|---|---|
| 1. Создание и компиляция программного кода в среде WASM32. Создание простейшего консольного приложения | 4 |
| 2. Создание окна Windows | 4 |
| 3. Отображение текста в окне | 4 |
| 4. Работа с клавиатурой в Windows | 4 |
| 5. Работа с мышью в Windows | 4 |
| 6. Главное меню | 4 |
| 7. Дочерние окна | 4 |
| 8. Работа с памятью и файлами | 4 |
| **ВСЕГО:** | **32** |

## Оценка знаний студентов

Для оценки работы и знаний студентов в результате изучения курса «Системное программное обеспечение» используется накопительная система. Результирующая оценка выставляется по сумме баллов, набранных студентом в течение всего учебного семестра, а также в результате выходного итогового контроля – экзамена.

Для получения аттестации необходимо выполнить все предшествующие аттестации лабораторные работы.

## Распределение баллов по видам занятий

| **Вид занятий** | **Форма оценки активности студента** | **Максимальное количество баллов по каждой форме оценки** | **Максимальное количество баллов по каждому виду занятий** |
|---|---|---|---|
| Лабораторные занятия | Защита работы в течение отведенного на нее занятия | 5 | 25 × 8 = 200 |
| | Защита работы в течение семестра | 20 | |
| Экзамен | Качество ответов на экзаменационные вопросы | 100 | 100 |

Дополнительные баллы предусматриваются за выполнение задач повышенной сложности (до 100 баллов). Для получения аттестации студент должен сдать все предшествующие лабораторные работы.

## Шкала выставления итоговой оценки

| **Оценка** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| **Сумма баллов** | 0 – 70 | 71 – 140 | 141 – 210 | 211 – 225 | 226 – 240 | 241 – 255 | 256 – 270 | 271 – 285 | 286 – 300 | 320 и более |

Очевидно, что для получения минимальной положительной оценки – 4 балла, студент должен набрать 211 баллов, для чего требуется:

– выполнить все лабораторные работы – минимум 200 баллов;

– получить 11 баллов при сдаче экзамена.

Для получения высшей оценки – 10 баллов – студенту необходимо будет проявить способность самостоятельно и творчески решать задачи повышенной сложности.

# ТЕМА 1. ЭВОЛЮЦИЯ ОПЕРАЦИОННЫХ СИСТЕМ

Цель изучения темы – приобретение студентами общих понятий об эволюции операционных систем, о первых вычислительных системах, этапах развития элементной базы для вычислительных систем, а также стадиях развития персональных компьютеров и операционных систем для них.

В результате изучения темы студенты должны:

– иметь представление об истории операционных систем;

– знать основные этапы развития элементной базы для вычислительных систем;

– знать основные свойства современных операционных систем;

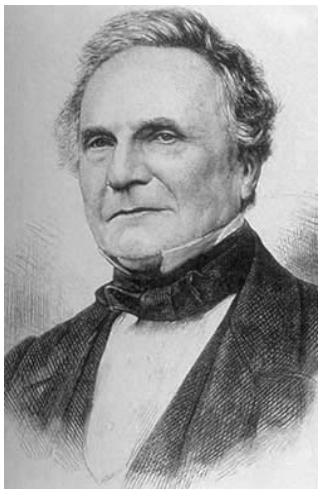
– иметь представление об эволюции концепции мультипрограммирования на протяжении всей истории операционных систем;

– иметь представление о влиянии Интернета на развитие операционных систем.

## Содержание темы

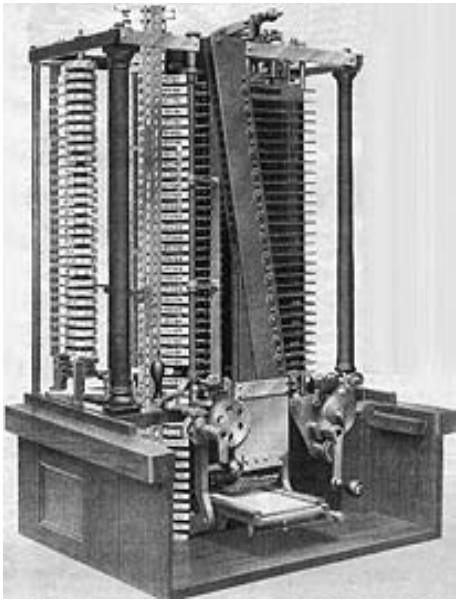
1. Появление первых операционных систем.
2. Появление мультипрограммных операционных систем для мэйнфреймов.
3. Операционные системы и глобальные сети.
4. Операционные системы мини-компьютеров и первые локальные сети.
5. Развитие операционных систем в 80-е годы XX в.
6. Особенности современного этапа развития операционных систем.
7. Вопросы и задания для самопроверки.

## 1.1. Появление первых операционных систем



Идея компьютера была предложена английским математиком Чарльзом Бэбиджем (Charles Babbage) в середине девятнадцатого века. Его механическая «аналитическая машина» так и не смогла по-настоящему заработать, потому что технологии того времени не соответствовали требованиям, необходимым для изготовления нужных деталей точной механики. Конечно, никакой речи об операционной системе для этого «компьютера» не шло.

Рис. 1.1. Чарльз Бэббидж



Настоящее рождение цифровых вычислительных машин произошло вскоре после окончания Второй мировой войны. В середине 40-х гг. XX в. были созданы первые ламповые вычислительные устройства. В то время одна и та же группа людей участвовала и в проектировании, и в эксплуатации, и в программировании вычислительной машины. Это была скорее научно-исследовательская работа в области вычислительной техники, а не использование компьютеров в качестве инструмента решения каких-либо практических задач из других прикладных областей.

Рис. 1.2. Вычислительная машина Бэббиджа

Программирование осуществлялось исключительно на машинном языке. Не было никакого системного программного обеспечения, кроме библиотек математических и служебных подпрограмм, которые программист мог использовать для того, чтобы не писать каждый раз коды, вычисляющие значение какой-либо математической функции или управляющие стандартным устройством ввода-вывода. Операционные системы все еще не появились, все задачи организации вычислительного процесса решались вручную каждым программистом с пульта управления, который представлял собой примитивное устройство ввода-вывода, состоящее из кнопок, переключателей и индикаторов.

С середины 50-х годов XX в. начался новый период в развитии вычислительной техники, связанный с появлением новой технической базы – полупроводниковых элементов. Возросло быстродействие процессоров, увеличились объемы оперативной и внешней памяти. Компьютеры стали более надежными, теперь они могли непрерывно работать настолько долго, чтобы на них можно было возложить выполнение действительно практически важных задач.

Наряду с совершенствованием аппаратуры, заметный прогресс наблюдался также в области автоматизации программирования и организации вычислительных работ. В эти годы появились первые алгоритмические языки, и таким образом к библиотекам математических и служебных подпрограмм добавился новый тип системного программного обеспечения – трансляторы.

Выполнение каждой программы стало включать большое количество вспомогательных работ: загрузка нужного транслятора (АЛГОЛ, ФОР-

ТРАН, КОБОЛ и т. п.), запуск транслятора и получение результирующей программы в машинных кодах, связывание программы с библиотечными подпрограммами, загрузка программы в оперативную память, запуск программы, вывод результатов на периферийное устройство. Для организации эффективного совместного использования трансляторов, библиотечных программ и загрузчиков в штат многих вычислительных центров были введены должности операторов, профессионально выполнявших работу по организации вычислительного процесса для всех пользователей этого центра.

Но как бы быстро и надежно ни работали операторы, они никак не могли состязаться в производительности с работой устройств компьютера. Бóльшую часть времени процессор простаивал в ожидании, пока оператор запустит очередную задачу. А поскольку процессор представлял собой весьма дорогое устройство, то низкая эффективность его использования означала низкую эффективность использования компьютера в целом. Для решения этой проблемы были разработаны первые системы пакетной обработки, которые автоматизировали всю последовательность действий оператора по организации вычислительного процесса.

Ранние *системы пакетной обработки* явились прообразом современных операционных систем и стали первыми системными программами, предназначенными не для обработки данных, а для управления вычислительным процессом. В ходе реализации систем пакетной обработки был разработан формализованный *язык управления заданиями*, с помощью которого программист сообщал системе и оператору, какие действия и в какой последовательности он хочет выполнить на вычислительной машине. Типовой набор директив обычно включал признак начала отдельной работы, вызов транслятора, вызов загрузчика, признаки начала и конца исходных данных.

Оператор составлял *пакет заданий*, которые в дальнейшем без его участия последовательно запускались на выполнение управляющей программой – *монитором.* Кроме того, монитор был способен самостоятельно обрабатывать наиболее часто встречающиеся при работе пользовательских программ аварийные ситуации, такие как отсутствие исходных данных, переполнение регистров, деление на ноль, обращение к несуществующей области памяти и т. д. Пакет обычно представлял собой набор перфокарт, но для ускорения работы мог переноситься на более удобный и емкий носитель, например, на магнитную ленту или магнитный диск. Сама программа-монитор в первых реализациях также хранилась на перфокартах или перфоленте, а в более поздних – на магнитной ленте и магнитных дисках.

Ранние системы пакетной обработки значительно сократили затраты времени на вспомогательные действия по организации вычислительного

процесса, а значит, был сделан еще один шаг по повышению эффективности использования компьютеров. Однако при этом программисты-пользователи лишились непосредственного доступа к компьютеру, что снижало эффективность их работы: внесение любого исправления требовало значительно больше времени, чем при интерактивной работе за пультом машины.

## 1.2. Появление мультипрограммных операционных систем для мэйнфреймов

Следующий важный период развития операционных систем относится к 1965 – 1975 гг. В это время в технической базе вычислительных машин произошел переход от отдельных полупроводниковых элементов типа транзисторов к интегральным микросхемам, что открыло путь к появлению следующего поколения компьютеров. Большие функциональные возможности интегральных схем сделали возможным реализацию на практике сложных компьютерных архитектур, таких, например, как IBM / 360.

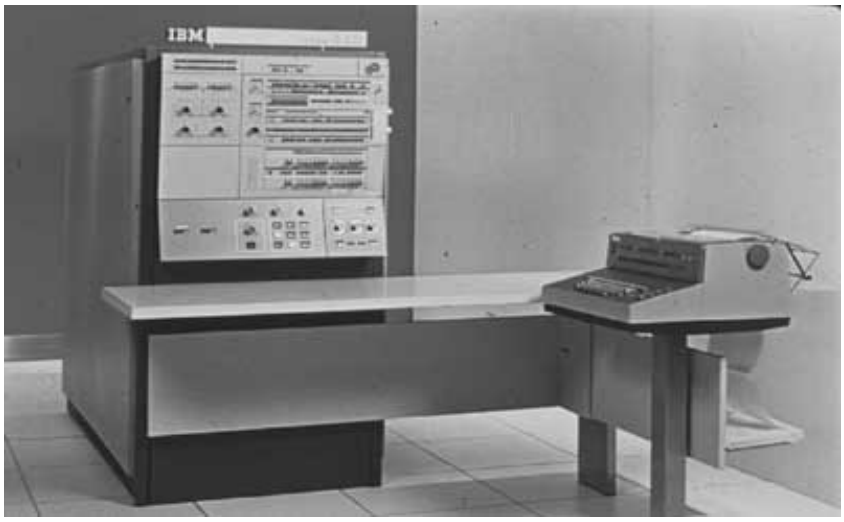



Рис. 1.3. ЭВМ IBM / 360

В этот период были реализованы практически все основные механизмы, используемые в современных ОС: мультипрограммирование, мультипроцессирование, поддержка многотерминального многопользовательского режима, виртуальная память, файловые системы, разграничение доступа и сетевая работа. В эти годы начинается расцвет системного программирования. Из направления прикладной математики, представляющего интерес для узкого круга специалистов, системное программирование превращается в отрасль индустрии, оказывающую непосредственное влияние на практическую деятельность миллионов людей.

Революционным событием данного этапа явилась промышленная реализация мультипрограммирования (в виде концепции и экспериментальных систем этот способ организации вычислений существовал уже около десяти лет). В условиях резко возросших возможностей компьютера по обработке и хранению данных выполнение только одной программы в каждый момент времени оказалось крайне неэффективным. Решением стало *мультипрограммирование* – способ организации вычислительного процесса, при котором в памяти компьютера находилось одновременно несколько программ, попеременно выполняющихся на одном процессоре. Эти усовершенствования значительно улучшили эффективность вычислительной системы: вычислительный процесс теперь мог производиться почти постоянно, а не менее половины времени работы компьютера, как это было раньше.

Мультипрограммирование было реализовано в двух вариантах – в системах пакетной обработки и разделения времени.

Мультипрограммные системы *пакетной обработки* так же, как и их однопрограммные предшественники, имели своей целью обеспечение максимальной загрузки аппаратуры компьютера, однако решали эту задачу более эффективно. В мультипрограммном пакетном режиме процессор не простаивал, пока одна программа выполняла операцию ввода-вывода (как это происходило при последовательном выполнении программ в системах ранней пакетной обработки), а переключался на другую, готовую к выполнению программу. В результате достигалась сбалансированная загрузка всех устройств компьютера, а следовательно, увеличивалось число задач, решаемых за единицу времени. В мультипрограммных системах пакетной обработки пользователь по-прежнему был лишен возможности интерактивно взаимодействовать со своими программами. Чтобы хотя бы частично вернуть пользователям ощущение непосредственного взаимодействия с компьютером, был разработан другой вариант мультипрограммных систем – *системы разделения времени*. Этот вариант рассчитан на *многотерминальные системы*, когда каждый пользователь работает за своим терминалом. В числе первых операционных систем разделения времени, разработанных в середине 60-х гг. XX в., были TSS / 360 (компания IBM), CTSS и MULTICS (Массачусетский технологический институт совместно с Bell Labs и компанией General Electric). Вариант мультипрограммирования, применяемый в системах разделения времени, был нацелен на создание для каждого отдельного пользователя иллюзии единоличного владения вычислительной машиной за счет периодического выделения каждой программе своей доли процессорного времени. В системах разделения времени эффективность

использования оборудования ниже, чем в системах пакетной обработки, что явилось платой за удобства работы пользователя.

Многотерминальный режим использовался не только в системах разделения времени, но и в системах пакетной обработки. При этом не только оператор, но и все пользователи получали возможность формировать свои задания и управлять их выполнением со своего терминала. Такие операционные системы получили название *систем удаленного ввода заданий*. Терминальные комплексы могли располагаться на большом расстоянии от процессорных стоек, соединяясь с ними с помощью различных глобальных связей – модемных соединений телефонных сетей или выделенных каналов. Для поддержания удаленной работы терминалов в операционных системах появились специальные программные модули, реализующие различные (в то время, как правило, нестандартные) протоколы связи. Такие вычислительные системы с удаленными терминалами, сохраняя централизованный характер обработки данных, в какой-то степени являлись прообразом современных сетей, а соответствующее системное программное обеспечение – прообразом сетевых операционных систем.

К этому времени можно наблюдать существенное изменение в распределении функций между аппаратными и программными средствами компьютера. Операционные системы становились неотъемлемыми элементами компьютеров, играя роль «продолжения» аппаратуры. В первых вычислительных машинах программист, напрямую взаимодействуя с аппаратурой, мог выполнить загрузку программных кодов, используя пультовые переключатели и лампочки индикаторов, а затем вручную запустить программу на выполнение, нажав кнопку «пуск». В компьютерах 60-х гг. XX в. большую часть действий по организации вычислительного процесса взяла на себя операционная система. В большинстве современных компьютеров не предусмотрено даже теоретической возможности выполнения какой-либо вычислительной работы без участия операционной системы. После включения питания автоматически происходит поиск, загрузка и запуск операционной системы, а в случае ее отсутствия компьютер просто останавливается.

Реализация мультипрограммирования потребовала внесения очень важных изменений в аппаратуру компьютера, непосредственно направленных на поддержку нового способа организации вычислительного процесса. При разделении ресурсов компьютера между программами необходимо обеспечить быстрое переключение процессора с одной программы на другую, а также надежно защитить коды и данные одной программы от непреднамеренной или преднамеренной порчи другой программой. В процессорах появились привилегированный и пользовательский режимы ра-

боты, специальные регистры для быстрого переключения с одной программы на другую, средства защиты областей памяти, а также развитая система прерываний.

В привилегированном режиме, предназначенном для работы программных модулей операционной системы, процессор мог выполнять все команды, в том числе и те из них, которые позволяли осуществлять распределение и защиту ресурсов компьютера. Программам, работающим в пользовательском режиме, некоторые команды процессора были недоступны. Таким образом, только ОС могла управлять аппаратными средствами и исполнять роль монитора и арбитра для пользовательских программ, которые выполнялись в непривилегированном, пользовательском режиме.

Система прерываний позволяла синхронизировать работу различных устройств компьютера, работающих параллельно и асинхронно, таких как каналы ввода-вывода, диски, принтеры и т. п. *Аппаратная поддержка операционных систем* стала с тех пор неотъемлемым свойством практически любых компьютерных систем, включая персональные компьютеры.


Рис. 1.4. ЭВМ PDP-11

Еще одной важной тенденцией этого периода является создание *семейств программно-совместимых машин* и *операционных систем* для них. Примерами семейств программно-совместимых машин, построенных на интегральных микросхемах, являются серии машин IBM / 360 и IBM / 370 (аналоги этих семейств советского производства – машины серии ЕС), PDP-11 (советские аналоги – СМ-3, СМ-4, СМ-1420). Вскоре идея программно-совместимых машин стала общепризнанной.

Программная совместимость требовала и совместимости операционных систем. Однако такая совместимость подразумевает возможность работы на больших и малых вычислительных системах, с большим и малым количеством разнообразной периферии, в коммерческой области и в области научных исследований. Операционные системы, построенные с намерением соответствовать всем этим противоречивым требованиям, оказались чрезвычайно сложными. Они состояли из многих миллионов ассемблерных строк, написанных тысячами программистов, и содержали тысячи

ошибок, вызывающих нескончаемый поток исправлений. Операционные системы этого поколения были очень дорогими. Так, разработка OS / 360, объем кода для которой составил 8 Мбайт, стоила компании IBM 80 млн долларов.

Однако, несмотря на впечатляющие размеры и множество проблем, OS / 360 и другие подобные ей операционные системы этого поколения действительно соответствовали большинству требований потребителей. За это десятилетие был сделан огромный шаг вперед и заложен прочный фундамент для создания современных операционных систем.

### 1.3. Операционные системы и глобальные сети

В начале 70-х годов XX в. появились первые *сетевые операционные системы*, которые, в отличие от многотерминальных ОС, позволяли не только рассредоточить пользователей, но и организовать распределенное хранение и обработку данных между несколькими компьютерами, связанными электрическими связями. Любая сетевая операционная система, с одной стороны, выполняет все функции локальной операционной системы, а с другой – обладает некоторыми дополнительными средствами, позволяющими ей взаимодействовать по сети с операционными системами других компьютеров. Программные модули, реализующие сетевые функции, появлялись в операционных системах постепенно, по мере развития сетевых технологий, аппаратной базы компьютеров и возникновения новых задач, требующих сетевой обработки.

Хотя теоретические работы по созданию концепций сетевого взаимодействия велись почти с самого появления вычислительных машин, значимые практические результаты по объединению компьютеров в сети были получены в конце 60-х гг. XX в., когда с помощью глобальных связей и техники коммутации пакетов удалось реализовать взаимодействие машин класса мэйнфреймов и суперкомпьютеров. Эти дорогостоящие компьютеры часто хранили уникальные данные и программы, доступ к которым необходимо было обеспечить широкому кругу пользователей, находившихся в различных городах на значительном расстоянии от вычислительных центров.

В 1969 г. Министерство обороны США инициировало работы по объединению суперкомпьютеров оборонных и научно-исследовательских центров в единую сеть. Эта сеть получила название ARPANET и явилась отправной точкой для создания самой известной ныне глобальной сети – Интернета. Сеть ARPANET объединяла компьютеры разных типов, работавшие под управлением различных ОС с добавленными модулями, реализующими коммуникационные протоколы, общие для всех компьютеров сети.

В 1974 г. компания IBM объявила о создании собственной сетевой архитектуры для своих мэйнфреймов, получившей название SNA (System Network Architecture). Эта многоуровневая архитектура, во многом подобная стандартной модели OSI, появившейся несколько позже, обеспечивала взаимодействие типа «терминал-терминал», «терминал-компьютер» и «компьютер-компьютер» по глобальным связям. Нижние уровни архитектуры были реализованы специализированными аппаратными средствами, наиболее важным из которых является процессор телеобработки. Функции верхних уровней SNA выполнялись программными модулями. Один из них составлял основу программного обеспечения процессора телеобработки. Другие модули работали на центральном процессоре в составе стандартной операционной системы IBM для мэйнфреймов.

В это же время в Европе велись активные работы по созданию и стандартизации сетей X.25. Эти сети с коммутацией пакетов не были привязаны к какой-либо конкретной операционной системе. После получения статуса международного стандарта в 1974 г. протоколы X.25 стали поддерживаться многими операционными системами. С 1980 г. компания IBM включила поддержку протоколов X.25 в архитектуру SNA и в свои операционные системы.

## 1.4. Операционные системы мини-компьютеров и первые локальные сети

К середине 70-х гг. XX в. наряду с мэйнфреймами широкое распространение получили мини-компьютеры, такие как PDP-11, Nova, HP. Мини-компьютеры первыми использовали преимущества больших интегральных схем, позволявших реализовать достаточно мощные функции при сравнительно невысокой стоимости компьютера.

Архитектура мини-компьютеров была значительно упрощена по сравнению с мэйнфреймами, что нашло отражение и в их операционных системах. Многие функции мультипрограммных многопользовательских ОС мэйнфреймов были усечены, учитывая ограниченность ресурсов мини-компьютеров. Операционные системы мини-компьютеров часто стали делать специализированными, например, только для управления в реальном времени (ОС JT-11 для мини-компьютеров PDP-11) или только для поддержания режима разделения времени (RSX-11M для тех же компьютеров). Эти операционные системы не всегда были многопользовательскими, что во многих случаях оправдывалось невысокой стоимостью компьютеров.

Важной вехой в истории мини-компьютеров и вообще в истории операционных систем явилось создание *ОС UNIX*. Первоначально эта ОС предназначалась для поддержания режима разделения времени в мини-компьютере PDP-7. С середины 70-х гг. XX в. началось массовое использование ОС UNIX. К этому времени программный код для UNIX был на 90 % написан на языке высокого уровня С. Широкое распространение эффективных С-компиляторов сделало UNIX уникальной для того времени ОС, обладающей возможностью сравнительно легкого переноса на различные типы компьютеров. Поскольку эта ОС поставлялась вместе с исходными кодами, то она стала первой открытой ОС, которую могли совершенствовать простые пользователи-энтузиасты. Хотя UNIX была первоначально разработана для мини-компьютеров, гибкость, элегантность, мощные функциональные возможности и открытость позволили ей занять прочные позиции во всех классах компьютеров: суперкомпьютерах, мэйнфреймах, мини-компьютерах, серверах и рабочих станциях на базе RISC-процессоров, персональных компьютерах.

Доступность мини-компьютеров и вследствие этого их распространенность на предприятиях послужили мощным стимулом для создания *локальных сетей*. Предприятие могло позволить себе иметь несколько мини-компьютеров, находящихся в одном здании или даже в одной комнате. Естественно, возникала потребность в обмене информацией между ними и в совместном использовании более дорогого периферийного оборудования.

Первые локальные сети строились с помощью нестандартного коммуникационного оборудования, в простейшем случае – путем прямого соединения последовательных портов компьютеров. Программное обеспечение также было нестандартным и реализовывалось в виде пользовательских приложений. Первое сетевое приложение для ОС UNIX – программа UUCP (UNIX-to-UNIX Copy program) – появилась в 1976 г. и начала распространяться с версией 7 AT&T UNIX с 1978 г. Эта программа позволяла копировать файлы с одного компьютера на другой в пределах локальной сети через различные аппаратные интерфейсы – RS-232, токовую петлю и т. п., а кроме того, могла работать через глобальные связи, например, модемные.

### 1.5. Развитие операционных систем в 80-е годы XX века

К наиболее важным событиям этого десятилетия можно отнести разработку стека TCP / IP, становление Интернета, стандартизацию технологий локальных сетей, появление персональных компьютеров и операционных систем для них.

Рабочий вариант стека протоколов TCP / IP был создан в конце 70-х гг. XX в. Этот стек представлял собой набор общих протоколов для разнородной вычислительной среды и предназначался для связи экспериментальной сети ARPANET с другими «сателлитными» сетями. В 1983 г. стек протоколов TCP / IP был принят Министерством обороны США в качестве военного стандарта. Переходу компьютеров сети ARPANET на стек TCP / IP способствовала его реализация для операционной системы BSD UNIX. С этого времени началось совместное существование UNIX и протоколов TCP / IP, а практически все многочисленные версии Unix стали сетевыми.

Внедрение протоколов TCP / IP в ARPANET придало этой сети все основные черты, отличающие современный Интернет. В 1983 г. сеть ARPANET была разделена на две части: MILNET, поддерживающую военные ведомства США, и новую ARPANET. Для обозначения составной сети ARPANET и MILNET стало использоваться название *Internet,* которое в русском языке со временем и с легкой руки локализаторов Microsoft превратилось в *Интернет*. Интернет стал отличным полигоном для испытаний многих сетевых операционных систем, позволившим проверить в реальных условиях возможности их взаимодействия, степень масштабируемости, способность работы при экстремальной нагрузке, создаваемой сотнями и тысячами пользователей. Стек протоколов TCP / IP также ждала завидная судьба. Независимость от производителей, гибкость и эффективность, доказанные успешной работой в Интернете, а также открытость и доступность стандартов сделали протоколы TCP / IP не только главным транспортным механизмом Интернета, но и основным стеком большинства сетевых операционных систем.

Все десятилетие было отмечено постоянным появлением новых, все более совершенных версий ОС UNIX. Среди них были и фирменные версии UNIX: SunOS, HP-UX, Irix, AIX и многие другие, в которых производители компьютеров адаптировали код ядра и системных утилит для своей аппаратуры. Разнообразие версий породило проблему их совместимости, которую периодически пытались решить различные организации. В результате были приняты стандарты POSIX и XPG, определяющие интерфейсы ОС для приложений, а специальное подразделение компании AT&T выпустило несколько версий UNIX System III и UNIX System V, призванных консолидировать разработчиков на уровне кода ядра.

Начало 80-х гг. XX в. связано с еще одним знаменательным для истории операционных систем событием – появлением *персональных компьютеров*. С точки зрения архитектуры персональные компьютеры ничем не отличались от класса мини-компьютеров типа PDP-11, но их стоимость

была существенно ниже. Если мини-компьютер позволил иметь собственную вычислительную машину отделу предприятия или университету, то персональный компьютер дал такую возможность отдельному человеку. Компьютеры стали широко использоваться неспециалистами, что потребовало разработки «дружественного» программного обеспечения, и предоставление этих «дружественных» функций стало прямой обязанностью операционных систем. Персональные компьютеры послужили также мощным катализатором роста локальных сетей, создав для этого отличную материальную основу в виде десятков и сотен компьютеров, принадлежащих одному предприятию и расположенных в пределах одного здания. В результате поддержка сетевых функций стала для ОС персональных компьютеров необходимым условием.

Однако и дружественный интерфейс, и сетевые функции появились у операционных систем персональных компьютеров не сразу. Первая версия наиболее популярной операционной системы раннего этапа развития персональных компьютеров – MS-DOS компании Microsoft – была лишена этих возможностей. Это была однопрограммная однопользовательская ОС с интерфейсом командной строки, способная стартовать с дискеты. Основной задачей для нее было управление файлами, расположенными на гибких и жестких дисках в UNIX-подобной иерархической файловой системе, а также поочередный запуск программ. MS-DOS не была защищена от программ пользователя, так как процессор Intel 8088 не поддерживал привилегированного режима. Разработчики первых персональных компьютеров считали, что при индивидуальном использовании компьютера и ограниченных возможностях аппаратуры нет смысла в поддержке мультипрограммирования, поэтому в процессоре не было предусмотрено привилегированного режима и других механизмов поддержки мультипрограммных систем.

Недостающие функции для MS-DOS и подобных ей ОС компенсировались внешними программами, предоставлявшими пользователю удобный графический интерфейс (например, Norton Commander) или средства тонкого управления дисками (например, PC Tools). Наибольшее влияние на развитие программного обеспечения для персональных компьютеров оказала операционная среда Windows компании Microsoft, представлявшая собой надстройку над MS-DOS.

Сетевые функции также реализовывались в основном сетевыми оболочками, работавшими поверх ОС. При сетевой работе всегда необходимо поддерживать многопользовательский режим, при котором один пользователь интерактивный, а остальные получают доступ к ресурсам компьютера по сети. В таком случае от операционной системы требуется хотя бы неко-

торый минимум функциональной поддержки многопользовательского режима. История сетевых средств MS-DOS началась с версии 3.1. Эта версия MS-DOS добавила к файловой системе необходимые средства блокировки файлов и записей, которые позволили более чем одному пользователю иметь доступ к файлу. Пользуясь этими функциями, сетевые оболочки могли обеспечить разделение файлов между сетевыми пользователями.

Вместе с выпуском версии MS-DOS 3.1 в 1984 г. компания Microsoft также выпустила продукт, называемый Microsoft Networks, который обычно неформально называют MS-NET. Некоторые концепции, заложенные в MS-NET, такие как введение в структуру базовых сетевых компонентов – редиректора и сетевого сервера, успешно перешли в более поздние сетевые продукты Microsoft: LAN Manager, Windows for Workgroups, а затем и в Windows NT.

Сетевые оболочки для персональных компьютеров выпускали и другие компании: IBM, Artisoft, Performance Technology и другие.

Иной путь выбрала компания Novell. Она изначально сделала ставку на разработку операционной системы со встроенными сетевыми функциями и добилась на этом пути значительных успехов. Ее сетевые операционные системы NetWare на долгое время стали эталоном производительности, надежности и защищенности для локальных сетей.

Первая сетевая операционная система компании Novell появилась на рынке в 1983 г. и называлась OS-Net. Эта ОС предназначалась для сетей, имевших звездообразную топологию, центральным элементом которых был специализированный компьютер на базе микропроцессора Motorola 68000. Немного позже, когда фирма IBM выпустила персональные компьютеры PC XT, компания Novell разработала новый продукт – NetWare 86, рассчитанный на архитектуру микропроцессоров семейства Intel 8088.

С первой версии ОС NetWare распространялась как операционная система для центрального сервера локальной сети, которая за счет специализации на выполнении функций файл-сервера обеспечивает максимально возможную для данного класса компьютеров скорость удаленного доступа к файлам и повышенную безопасность данных. За высокую производительность пользователи сетей Novell NetWare расплачиваются стоимостью: выделенный файл-сервер не может использоваться в качестве рабочей станции, а его специализированная ОС имеет весьма специфический прикладной программный интерфейс (API), что требует от разработчиков особых знаний, специального опыта и приложения значительных усилий.

В отличие от Novell, большинство других компаний развивали сетевые средства для персональных компьютеров в рамках операционных сис-

тем с универсальным интерфейсом API, т. е. операционных систем общего назначения. Такие системы по мере развития аппаратных платформ персональных компьютеров стали все больше приобретать черты операционных систем мини-компьютеров.

В 1987 г. в результате совместных усилий Microsoft и IBM появилась первая многозадачная операционная система для персональных компьютеров с процессором Intel 80286, в полной мере использующая возможности защищенного режима – OS / 2. Эта хорошо продуманная система поддерживала вытесняющую многозадачность, виртуальную память, графический пользовательский интерфейс (не с первой версии) и виртуальную машину для выполнения DOS-приложений. Фактически она выходила за пределы простой многозадачности с концепцией распараллеливания отдельных процессов, получившей название *многопоточности.*

OS / 2 с ее развитыми функциями многозадачности и файловой системой HPFS со встроенными средствами многопользовательской защиты оказалась хорошей платформой для построения локальных сетей персональных компьютеров. Наибольшее распространение получили сетевые оболочки LAN Manager компании Microsoft и LAN Server компании IBM, разработанные этими компаниями на основе одного базового кода. Эти оболочки уступали по производительности файловому серверу NetWare и потребляли больше аппаратных ресурсов, но имели важные достоинства: они позволяли, во-первых, выполнять на сервере любые программы, разработанные для OS / 2, MS-DOS и Windows, а во-вторых, использовать компьютер, на котором они работали, в качестве рабочей станции.

Сетевые разработки компаний Microsoft и IBM привели к появлению NetBIOS – очень популярного транспортного протокола и одновременно интерфейса прикладного программирования для локальных сетей, получившего применение практически во всех сетевых операционных системах для персональных компьютеров. Этот протокол и сегодня применяется для создания небольших локальных сетей.

Не очень удачная рыночная судьба OS / 2 не позволила системам LAN Manager и LAN Server захватить заметную долю рынка, но принципы работы этих сетевых систем во многом нашли свое воплощение в более удачной операционной системе 90-х гг. XX в. – Microsoft Windows NT, содержащей встроенные сетевые компоненты, некоторые из которых имеют приставку LM – от LAN Manager.

В 80-е гг. XX в.были приняты основные стандарты на коммуникационные технологии для локальных сетей: в 1980 г. – Ethernet, в 1985 г. – Token Ring, в конце 80-х гг. XX в. – FDDI. Это позволило обеспечить совмес-

тимость сетевых операционных систем на нижних уровнях, а также стандартизовать интерфейс ОС с драйверами сетевых адаптеров.

Для персональных компьютеров применялись не только специально разработанные для них операционные системы, подобные MS-DOS, NetWare и OS / 2, но и адаптировались уже существующие ОС. Появление процессоров Intel 80286 и особенно 80386 с поддержкой мультипрограммирования позволило перенести на платформу персональных компьютеров ОС UNIX. Наиболее известной системой этого типа была версия UNIX компании Santa Cruz Operation (SCO UNIX), а также разработанная в начале 90-х гг. XX в. Линусом Торвальдсом операционная система LINUX.

## 1.6. Особенности современного этапа развития операционных систем

В 90-е гг. XX в. практически все операционные системы, занимающие заметное место на рынке, стали *сетевыми*. Сетевые функции сегодня встраиваются в ядро ОС, являясь ее неотъемлемой частью. Операционные системы получили средства для работы со всеми основными технологиями локальных (Ethernet, Fast Ethernet, Gigabit Ethernet, Token Ring, FDDI, ATM) и глобальных (X.25, frame relay, ISDN, ATM) сетей, а также средства для создания составных сетей (IP, IPX, AppleTalk, RIP, OSPF, NLSP). В операционных системах используются средства мультиплексирования нескольких стеков протоколов, за счет которых компьютеры могут поддерживать одновременную сетевую работу с разнородными клиентами и серверами. Появились специализированные ОС, которые предназначены исключительно для выполнения коммуникационных задач. Например, сетевая операционная система IOS компании Cisco Systems, работающая в маршрутизаторах, организует в мультипрограммном режиме выполнение набора программ, каждая из которых реализует один из коммуникационных протоколов.

Во второй половине 90-х гг. XX в. все производители операционных систем резко усилили поддержку средств работы с Интернетом. Исключение составили производители UNIX-систем, в которых эта поддержка всегда была существенной. Кроме самого стека TCP / IP, в комплект поставки начали включать утилиты, реализующие такие популярные сервисы Интернета, как telnet, ftp, DNS и Web. Влияние Интернета проявилось и в том, что компьютер превратился из чисто вычислительного устройства в средство коммуникаций с развитыми вычислительными возможностями.

Особое внимание в течение всего последнего десятилетия уделялось *корпоративным* сетевым операционным системам. Их дальнейшее разви-

тие представляет одну из наиболее важных задач и в обозримом будущем. Корпоративная операционная система отличается способностью устойчиво работать в крупных сетях, которые характерны для больших предприятий, имеющих отделения в десятках городов и, возможно, в разных странах. Таким сетям органически присуща высокая степень гетерогенности программных и аппаратных средств, поэтому корпоративная ОС должна беспроблемно взаимодействовать с операционными системами разных типов и работать на различных аппаратных платформах. К настоящему времени достаточно явно определилась тройка лидеров в классе корпоративных ОС – это Novell NetWare 4.x и 5.0, Microsoft Windows NT 4.0 и Windows 2000, а также UNIX и LINUX-системы различных производителей аппаратных платформ.

Для корпоративной ОС очень важно наличие средств централизованного администрирования и управления, позволяющих в единой базе данных хранить учетные записи о десятках тысяч пользователей, компьютеров, коммуникационных устройств и модулей программного обеспечения, имеющихся в корпоративной сети. В современных ОС средства централизованного администрирования обычно базируются на единой *справочной службе*. Первой успешной реализацией справочной службы корпоративного масштаба была система StreetTalk компании Banyan. К настоящему времени наибольшее признание получила справочная служба NDS компании Novell, выпущенная впервые в 1993 г. для первой корпоративной версии NetWare 4.0. Роль централизованной справочной службы настолько велика, что именно по качеству справочной службы оценивают пригодность операционной системы для работы в корпоративном масштабе. Длительная задержка выпуска Windows NT 2000 во многом была связана с созданием для этой ОС масштабируемой справочной службы Active Directory, без которой этому семейству ОС трудно было претендовать на звание истинно корпоративной ОС.

Создание многофункциональной масштабируемой справочной службы является стратегическим направлением эволюции ОС. От успехов этого направления во многом зависит и дальнейшее развитие Интернета. Такая служба нужна для превращения Интернета в предсказуемую и управляемую систему, например, для обеспечения требуемого качества обслуживания трафика пользователей, поддержки крупных распределенных приложений, построения эффективной почтовой системы и т. п.

На современном этапе развития операционных систем на передний план вышли средства обеспечения *безопасности*. Это связано с возрастанием ценности информации, обрабатываемой компьютерами, а также с по-

вышением уровня угроз, существующих при передаче данных по сетям, особенно по публичным, таким как Интернет. Многие операционные системы обладают сегодня развитыми средствами защиты информации, основанными на шифрации данных, аутентификации и авторизации.

Современным операционным системам присуща *многоплатформенность*, т. е. способность работать на совершенно различных типах компьютеров. Многие операционные системы имеют специальные версии для поддержки кластерных архитектур, обеспечивающих высокую производительность и отказоустойчивость. Исключением пока является ОС NetWare, все версии которой разработаны для платформы Intel, а реализации функций NetWare в виде оболочки для других ОС, например NetWare for AIX, успеха не имели.

В последние годы получила дальнейшее развитие долговременная тенденция повышения *удобства работы* человека с компьютером. Эффективность работы человека становится основным фактором, определяющим эффективность вычислительной системы в целом. Усилия человека не должны тратиться на настройку параметров вычислительного процесса, как это происходило в ОС предыдущих поколений. Например, в системах пакетной обработки для мэйнфреймов каждый пользователь должен был с помощью языка управления заданиями определить большое количество параметров, относящихся к организации вычислительных процессов в компьютере. Так, для системы OS / 360 язык управления заданиями JCL предусматривал возможность определения пользователем более 40 параметров, среди которых были приоритет задания, требования к основной памяти, предельное время выполнения задания, перечень используемых устройств ввода-вывода и режимы их работы.

Современная операционная система берет на себя выполнение задачи выбора параметров операционной среды, используя для этой цели различные адаптивные алгоритмы. Например, тайм-ауты в коммуникационных протоколах часто определяются в зависимости от условий работы сети. Распределение оперативной памяти между процессами осуществляется автоматически с помощью механизмов виртуальной памяти в зависимости от активности этих процессов и информации о частоте использования ими той или иной страницы. Мгновенные приоритеты процессов определяются динамически в зависимости от предыстории, включающей, например, время нахождения процесса в очереди, процент использования выделенного кванта времени, интенсивность ввода-вывода и т. п. Даже в процессе установки большинство ОС предлагают режим выбора параметров по умолчанию, который гарантирует пусть не оптимальное, но всегда приемлемое качество работы систем.

Постоянно повышается удобство интерактивной работы с компьютером путем включения в операционную систему развитых графических интерфейсов, использующих наряду с графикой звук и видеоизображение. Это особенно важно для превращения компьютера в терминал новой публичной сети, которой постепенно становится Интернет, так как для массового пользователя терминал должен быть почти таким же понятным и удобным, как телефонный аппарат. Пользовательский интерфейс операционной системы становится все более интеллектуальным, направляя действия человека в типовых ситуациях и принимая за него рутинные решения.

Уровень удобств в использования ресурсов, которые сегодня предоставляют пользователям, администраторам и разработчикам приложений операционные системы изолированных компьютеров, для сетевых операционных систем является только заманчивой перспективой. Пока пользователи и администраторы сети тратят значительное время на попытки выяснить, где находится тот или иной ресурс, разработчики сетевых приложений прилагают много усилий для определения местоположения данных и программных модулей в сети. Операционные системы будущего должны обеспечить высокий уровень прозрачности сетевых ресурсов, взяв на себя задачу организации распределенных вычислений, превратив сеть в виртуальный компьютер. Именно этот смысл вкладывают в лаконичный лозунг «Сеть – это компьютер» специалисты компании Sun, но для превращения лозунга в жизнь разработчикам операционных систем нужно пройти еще немалый путь.

## 1.7. Вопросы и задания для самопроверки

1. Какие события в развитии технической базы вычислительных машин стали вехами в истории операционных систем?

2. В чем состояло принципиальное отличие первых мониторов пакетной обработки от уже существовавших к этому времени системных обрабатывающих программ – трансляторов, загрузчиков, компоновщиков, библиотек процедур?

3. Может ли компьютер работать без операционной системы?

4. Как эволюционировало отношение к концепции мультипрограммирования на протяжении всей истории ОС?

5. Какое влияние на развитие ОС оказал Интернет?

6. Чем объясняется особое место ОС UNIX в истории операционных систем?

7. Опишите историю сетевых ОС.

8. В чем состоят современные тенденции развития ОС?

# ТЕМА 2. НАЗНАЧЕНИЕ И ФУНКЦИИ ОПЕРАЦИОННОЙ СИСТЕМЫ

Цель изучения темы – приобретение студентами сведений о функциях операционных систем, областях применения, аппаратных платформах и методах реализации.

В результате изучения темы студенты должны:

– знать основные функции операционной системы;

– иметь представление о следующих понятиях: виртуальная машина, ресурсы вычислительной системы, процессы, интерфейс прикладного программирования.

## Содержание темы

1. Операционные системы для автономного компьютера.
1.1. Операционная система как виртуальная машина.
1.2. Операционная система как система управления ресурсами.
2. Функциональные компоненты операционной системы автономного компьютера.
2.1. Управление процессами.
2.2. Управление памятью.
2.3. Управление файлами и внешними устройствами.
2.4. Защита данных и администрирование.
2.5. Интерфейс прикладного программирования.
2.6. Пользовательский интерфейс.
3. Сетевые операционные системы.
3.1. Сетевые и распределенные операционные системы.
3.2. Два значения термина «сетевая операционная система».
3.3. Функциональные компоненты сетевой операционной системы.
3.4. Сетевые службы и сетевые сервисы.
3.5. Встроенные сетевые службы и сетевые оболочки.
4. Одноранговые и серверные сетевые операционные системы.
4.1. Операционные системы в одноранговых сетях.
4.2. Операционные системы в сетях с выделенными серверами.
5. Требования к современным операционным системам.
6. Вопросы и задания для самопроверки.

## 2.1. Операционные системы для автономного компьютера

Сегодня существует большое количество разных типов операционных систем, отличающихся областями применения, аппаратными платформами и методами реализации. Естественно, это обуславливает и значительные функциональные различия этих ОС. Даже у конкретной операционной системы набор выполняемых функций зачастую определить не так просто: та функция, которая сегодня выполняется внешним по отношению к ОС компонентом, завтра может стать ее неотъемлемой частью и наоборот. Поэтому при изучении операционных систем очень важно из всего многообразия выделить свойства, присущие всем операционным системам как классу продуктов.

Операционная система компьютера представляет собой комплекс взаимосвязанных программ, который действует как интерфейс между приложениями и пользователями с одной стороны и аппаратурой компьютера с другой стороны. В соответствии с этим определением, ОС выполняет две группы функций:

– предоставление пользователю или программисту вместо реальной аппаратуры компьютера расширенной виртуальной машины, с которой удобнее работать и которую легче программировать;

– повышение эффективности использования компьютера путем рационального управления его ресурсами в соответствии с некоторым критерием.

### 2.1.1. Операционная система как виртуальная машина

Для того чтобы успешно решать свои задачи, современный пользователь или даже прикладной программист может обойтись без досконального знания аппаратного устройства компьютера. Ему не обязательно быть в курсе того, как функционируют различные электронные блоки и электромеханические узлы компьютера. Более того, очень часто пользователь может не знать даже системы команд процессора. Пользователь-программист привык иметь дело с мощными высокоуровневыми функциями, которые ему предоставляет операционная система.

Так, например, при работе с диском программисту, пишущему приложение для работы под управлением ОС, или конечному пользователю ОС достаточно представлять его в виде некоторого набора файлов, каждый из которых имеет имя. Последовательность действий при работе с файлом заключается в его открытии, выполнении одной или нескольких операций чтения или записи и, наконец, закрытии файла. Такие частности, как используемая при записи частотная модуляция или текущее состояние двига-

теля механизма перемещения магнитных головок чтения / записи, не должны занимать программиста. Именно операционная система скрывает от программиста большую часть особенностей аппаратуры и предоставляет возможность простой и удобной работы с требуемыми файлами.

В результате реальная машина, способная выполнять только небольшой набор элементарных действий, определяемых системой команд, превращается в виртуальную машину, выполняющую широкий набор гораздо более мощных функций. Виртуальная машина тоже управляется командами, но это уже команды другого, более высокого уровня: удалить файл с определенным именем, запустить выполнение некоторой прикладной программы, повысить приоритет задачи, вывести текст из файла на печать. Таким образом, назначение ОС состоит в предоставлении пользователю / программисту некоторой расширенной виртуальной машины, которую легче программировать и с которой легче работать, чем непосредственно с аппаратурой, составляющей реальный компьютер или реальную сеть.

### 2.1.2. Операционная система как система управления ресурсами

Операционная система не только предоставляет пользователям и программистам удобный интерфейс к аппаратным средствам компьютера, но и является механизмом, распределяющим ресурсы компьютера.

К числу основных ресурсов современных вычислительных систем могут быть отнесены процессоры, основная память, таймеры, наборы данных, диски, накопители на магнитных лентах, принтеры, сетевые устройства и некоторые другие. Ресурсы распределяются между процессами. *Процесс* (*задача*) представляет собой базовое понятие большинства современных ОС и чаще всего кратко определяется как программа в стадии выполнения. Программа – это статический объект, представляющий собой файл с кодами и данными. Процесс – это динамический объект, который возникает в операционной системе после того, как пользователь или сама операционная система решает «запустить программу на выполнение», т. е. создать новую единицу вычислительной работы. Например, ОС может создать процесс в ответ на команду пользователя run prgl.exe, где prgl.exe – это имя файла, в котором хранится код программы.

Управление ресурсами вычислительной системы с целью наиболее эффективного их использования является назначением операционной системы. Например, мультипрограммная операционная система организует одновременное выполнение сразу нескольких процессов на одном компьютере, поочередно переключая процессор с одного процесса на другой,

исключая простои процессора, вызываемые обращениями процессов к вводу-выводу. ОС также отслеживает и разрешает конфликты, возникающие при обращении нескольких процессов к одному и тому же устройству ввода-вывода или к одним и тем же данным.

Критерии эффективности, в соответствии с которыми ОС организует управление ресурсами компьютера, могут быть различными. Например, в одних системах важен такой критерий, как пропускная способность вычислительной системы, в других – время ее реакции. Соответственно выбранному критерию эффективности операционные системы по-разному организуют вычислительный процесс.

Управление ресурсами включает решение следующих общих, не зависящих от типа ресурса задач:

– планирование ресурса, т. е. определение, какому процессу, когда и в каком количестве (если ресурс может выделяться частями) следует выделить данный ресурс;

– удовлетворение запросов на ресурсы;

– отслеживание состояния и учет использования ресурса, т. е. поддержание оперативной информации о том, занят или свободен ресурс и какая доля ресурса уже распределена;

– разрешение конфликтов между процессами.

Для решения этих общих задач управления ресурсами разные ОС используют различные алгоритмы, особенности которых в конечном счете и определяют облик ОС в целом, включая характеристики производительности, область применения и даже пользовательский интерфейс. Например, применяемый алгоритм управления процессором в значительной степени определяет, может ли ОС использоваться как система разделения времени, система пакетной обработки или система реального времени.

Задача организации эффективного совместного использования ресурсов несколькими процессами является весьма сложной, и сложность эта порождается в основном случайным характером возникновения запросов на потребление ресурсов. В мультипрограммной системе образуются очереди заявок от одновременно выполняемых программ к разделяемым ресурсам компьютера: процессору, странице памяти, принтеру, диску. Операционная система организует обслуживание этих очередей по разным алгоритмам: в порядке поступления, на основе приоритетов, кругового обслуживания и т. д. Анализ и определение оптимальных дисциплин обслуживания заявок является предметом специальной области прикладной математики – теории массового обслуживания. Эта теория иногда использует-

ся для оценки эффективности тех или иных алгоритмов управления очередями в операционных системах. Очень часто в ОС реализуются и эмпирические алгоритмы обслуживания очередей, прошедшие проверку практикой.

Таким образом, управление ресурсами составляет важную часть функций любой операционной системы, в особенности мультипрограммной. В отличие от функций расширенной машины, большинство функций управления ресурсами выполняются операционной системой автоматически и прикладному программисту недоступны.

## 2.2. Функциональные компоненты операционной системы автономного компьютера

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.

### 2.2.1. 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 вво-

да-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 процессов 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: 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, т. е. распределяет процессорное время между несколькими одновременно существующими в системе процессами, занимается созданием и уничтожением процессов, обеспечивает процессы необходимыми системными ресурсами, поддерживает синхронизацию процессов, а также обеспечивает взаимодействие между процессами.

### 2.2.2. 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 (в одних случаях память выделяется процессу в виде одной непрерывной области, а в других – в виде нескольких несмежных областей), степенью свободы границы областей (она может быть жестко зафиксирована на все время существования процесса или же динамически перемещаться при выделении процессу дополнительных объемов памяти). 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.

### 2.2.3. 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 специальных устройств ввода-вывода, таких как, например, аналого-цифровые преобразователи, может насчитывать сотни моделей. Эти модели могут существенно отличаться набором и последовательностью 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, например, модемом U-1496E компании ZyXEL, или же группой устройств определенного типа, например, любыми Hayes-совместимыми модемами. 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. От наличия подходящих драйверов во многом зависит успех операционной системы на рынке (например, отсутствие многих необходимых драйверов внешних устройств было одной из причин низкой популярности OS / 2).

Созданием драйверов устройств занимаются как разработчики конкретной ОС, так и специалисты компаний, выпускающих внешние устройст-

ва. 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 ОС.

Прикладные программисты могут пользоваться интерфейсом драйверов при разработке своих программ, но это не очень удобно: такой интерфейс обычно представляет собой низкоуровневые операции, обремененные большим количеством деталей.

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. Здесь мы опять имеем дело со свойством операционной системы подменять реальную аппаратуру удобными для пользователя и программиста абстракциями.

### 2.2.4. 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 администратор 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 фай-

лам. 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, т. е. мультипроцессирование, когда система продолжает работу при отказе одного из процессоров, хотя и с меньшей производительностью. Необходимо отметить, что многие ОС используют мультипроцессорную конфигурацию компьютера только для ускорения работы и при отказе одного из процессоров прекращают работу.

Поддержка отказоустойчивости также входит в обязанности системного администратора. В состав ОС обычно входят утилиты, позволяющие администратору выполнять регулярные операции резервного копирования для обеспечения быстрого восстановления важных данных.

### 2.2.5. Интерфейс прикладного программирования

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 определен-

ному 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, которым приложение получает услуги операционной системы, очень похож на вызов подпрограмм. Информация, нужная ОС и состоящая обычно из идентификатора команды и данных, помещается в определенное место памяти, в регистры и / или стек. Затем управление передается операционной системе, которая выполняет требуемую функцию и возвращает результаты через память, регистры или стеки. Если операция проведена неуспешно, то результат включает индикацию ошибки.

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.

### 2.2.6. 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.

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 интерак-

тивном 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.

## 2.3. Сетевые операционные системы

Операционная система компьютерной сети во многом аналогична ОС автономного компьютера: она также представляет собой комплекс взаимосвязанных программ, который обеспечивает удобство работы пользователям и программистам путем предоставления им некоторой виртуальной вычислительной системы и реализует эффективный способ разделения ресурсов между множеством выполняемых в сети процессов.

Компьютерная сеть – это набор компьютеров, связанных коммуникационной системой и снабженных соответствующим программным обеспечением, позволяющим пользователям сети получать доступ к ресурсам этого набора компьютеров. Сеть могут образовывать компьютеры разных типов: небольшие микропроцессоры, рабочие станции, мини-компьютеры, персональные компьютеры или суперкомпьютеры. Коммуникационная система может включать кабели, повторители, коммутаторы, маршрутизаторы и другие устройства, обеспечивающие передачу сообщений между любой парой компьютеров сети. Компьютерная сеть позволяет пользователю работать со своим компьютером как с автономным и добавляет к этому возможность доступа к информационным и аппаратным ресурсам других компьютеров сети.

При организации сетевой работы ОС играет роль интерфейса, экранирующего от пользователя все детали низкоуровневых программно-аппаратных средств сети. Например, вместо числовых адресов компьютеров сети, таких как MAC-адрес и IP-адрес, ОС компьютерной сети позволяет оперировать удобными для запоминания символьными именами. В результате в представлении пользователя сеть с ее множеством сложных и запутанных реальных деталей превращается в достаточно понятный набор разделяемых ресурсов.

### 2.3.1. Сетевые и распределенные операционные системы

В зависимости от того, какой виртуальный образ создает операционная система для того, чтобы подменить им реальную аппаратуру компьютерной сети, различают сетевые ОС и распределенные ОС.

Сетевая ОС предоставляет пользователю некую виртуальную вычислительную систему, работать с которой гораздо проще, чем с реальной сетевой аппаратурой. В то же время эта виртуальная система не полностью скрывает распределенную природу своего реального прототипа, т. е. является виртуальной сетью.

При использовании ресурсов компьютеров сети пользователь сетевой ОС всегда помнит, что он имеет дело с сетевыми ресурсами и что для доступа к ним нужно выполнить некоторые особые операции, например, отобразить удаленный разделяемый каталог на вымышленную локальную букву дисковода или поставить перед именем каталога еще и имя компьютера, на котором тот расположен. Пользователи сетевой ОС обычно должны быть в курсе того, где хранятся их файлы, и должны использовать явные команды передачи файлов для перемещения файлов с одной машины на другую.

Работая в среде сетевой ОС, пользователь хотя и может запустить задание на любой машине компьютерной сети, всегда знает, на какой машине выполняется его задание. По умолчанию пользовательское задание выполняется на той машине, на которой пользователь сделал логический вход. Если же он хочет выполнить задание на другой машине, то ему нужно либо выполнить логический вход в эту машину, используя команду типа remote login, либо ввести специальную команду удаленного выполнения, в которой он должен указать информацию, идентифицирующую удаленный компьютер.

Магистральным направлением развития сетевых операционных систем является достижение как можно более высокой степени прозрачности сетевых ресурсов. В идеальном случае сетевая ОС должна представить пользователю сетевые ресурсы в виде ресурсов единой централизованной виртуальной машины. Для такой операционной системы используют специальное название – распределенная ОС или истинно распределенная ОС.

Распределенная ОС, динамически и автоматически распределяя работы по различным машинам системы для обработки, заставляет набор сетевых машин работать как виртуальный унипроцессор. Пользователь распределенной ОС, вообще говоря, не имеет сведений о том, на какой машине выполняется его работа.

Распределенная ОС существует как единая операционная система в масштабах вычислительной системы. Каждый компьютер сети, работаю-

щей под управлением распределенной ОС, выполняет часть функций этой глобальной ОС. Распределенная ОС объединяет все компьютеры сети в том смысле, что они работают в тесной кооперации друг с другом для эффективного использования всех ресурсов компьютерной сети.

### 2.3.2. Два значения термина «сетевая операционная система»

В настоящее время практически все сетевые операционные системы еще очень далеки от идеала истинной распределенности. Степень автономности каждого компьютера в сети, работающей под управлением сетевой ОС, значительно выше по сравнению с компьютерами, работающими под управлением распределенной ОС.

В результате сетевая ОС может рассматриваться как набор операционных систем отдельных компьютеров, составляющих сеть. На разных компьютерах сети могут использоваться одинаковые или разные ОС. Например, на всех компьютерах сети может работать одна и та же ОС UNIX. Более реалистичным вариантом является сеть, в которой работают разные ОС, например, часть компьютеров работает под управлением UNIX, часть – под управлением NetWare, а остальные – под управлением Windows NT и Windows 98. Все эти операционные системы функционируют независимо друг от друга в том смысле, что каждая из них принимает независимые решения о создании и завершении своих собственных процессов и управлении локальными ресурсами. Но в любом случае операционные системы компьютеров, работающих в сети, должны включать взаимно согласованный набор коммуникационных протоколов для организации взаимодействия процессов, выполняющихся на разных компьютерах сети, и разделения ресурсов этих компьютеров между пользователями сети.

Если операционная система отдельного компьютера позволяет ему работать в сети, т. е. предоставлять свои ресурсы в общее пользование и / или потреблять ресурсы других компьютеров сети, то такая операционная система отдельного компьютера также называется сетевой ОС.

Таким образом, термин «сетевая операционная система» используется в двух значениях: во-первых, как совокупность ОС всех компьютеров сети и, во-вторых, как операционная система отдельного компьютера, способного работать в сети. Исходя из этого определения следует, что такие операционные системы, как, например, Windows NT, NetWare, Solaris, HP-UX, являются сетевыми, поскольку все они обладают средствами, которые позволяют их пользователям работать в сети.

### 2.3.3. Функциональные компоненты сетевой операционной системы

На рис. 2.1 показаны основные функциональные компоненты сетевой операционной системы:

– *средства управления локальными ресурсами* компьютера реализуют все функции ОС автономного компьютера (распределение оперативной памяти между процессами, планирование и диспетчеризацию процессов, управление процессорами в мультипроцессорных машинах, управление внешней памятью, интерфейс с пользователем и т. д.);

– в *сетевых средствах*, в свою очередь, можно выделить три компонента:

а) средства предоставления локальных ресурсов и услуг в общее пользование – *серверная часть ОС*;

б) средства запроса доступа к удаленным ресурсам и услугам – *клиентская часть ОС*;

в) *транспортные средства ОС*, которые совместно с коммуникационной системой обеспечивают передачу сообщений между компьютерами сети.

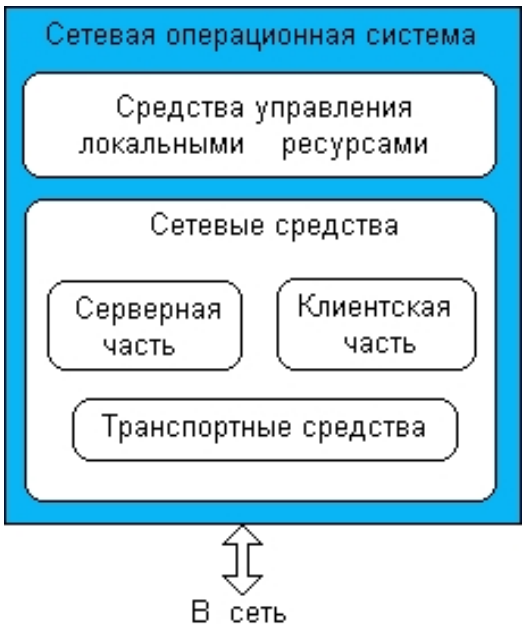



Рис. 2.1. Функциональные компоненты сетевой ОС

Упрощенно работа сетевой ОС происходит следующим образом. Предположим, что пользователь компьютера А решил разместить свой файл на диске другого компьютера сети – компьютера В. Для этого он набирает на клавиатуре соответствующую команду и нажимает клавишу Enter. Программный модуль ОС, отвечающий за интерфейс с пользователем, принимает эту команду и передает ее клиентской части ОС компьютера А.

Клиентская часть ОС не может получить непосредственный доступ к ресурсам другого компьютера – в данном случае к дискам и файлам компьютера В. Она может только «попросить» об этом серверную часть ОС, работающую на том компьютере, которому принадлежат эти ресурсы. Эти «просьбы» выражаются в виде *сообщений*, передаваемых по сети. Сообщения могут содержать не только команды на выполнение некоторых действий, но и собственно данные, например, содержимое некоторого файла.

Управляют передачей сообщений между клиентской и серверными частями по коммуникационной системе сети транспортные средства ОС. Эти средства выполняют такие функции, как формирование сообщений, разбиение сообщения на части (пакеты, кадры), преобразование имен компьютеров в числовые адреса, организацию надежной доставки сообщений, определение маршрута в сложной сети и т. д. Правила взаимодействия компьютеров при передаче сообщений по сети фиксируются в *коммуникационных протоколах*, таких как Ethernet, Token Ring, IP, IPX и пр. Чтобы два компьютера смогли обмениваться сообщениями по сети, транспортные средства их ОС должны поддерживать некоторый общий набор коммуникационных протоколов. Коммуникационные протоколы переносят сообщения клиентских и серверных частей ОС по сети, не вникая в их содержание.

На стороне компьютера В, на диске которого пользователь хочет разместить свой файл, должна работать серверная часть ОС, постоянно ожидающая прихода запросов из сети на удаленный доступ к ресурсам этого компьютера. Серверная часть, приняв запрос из сети, обращается к локальному диску и записывает в один из его каталогов указанный файл. Конечно, для выполнения этих действий требуется не одно, а целая серия сообщений, переносящих между компьютерами команды ОС и части передаваемого файла.

Очень удобной и полезной функцией клиентской части ОС является способность отличить запрос к удаленному файлу от запроса к локальному файлу. Если клиентская часть ОС умеет это делать, то приложения не должны заботиться о том, с локальным или удаленным файлом они работают, – клиентская программа сама распознает и перенаправляет (*redirect*) запрос к удаленной машине. Отсюда и название, часто используемое для клиентской части сетевой ОС, – *редиректор*. Иногда функции распознавания выделяются в отдельный программный модуль, в этом случае редиректором называют не всю клиентскую часть, а только этот модуль.

Клиентские части сетевых ОС выполняют также преобразование форматов запросов к ресурсам. Они принимают запросы от приложений на доступ к сетевым ресурсам в локальной форме, т. е. в форме, принятой в локальной части ОС. В сеть же запрос передается клиентской частью в

другой форме, соответствующей требованиям серверной части ОС, работающей на компьютере, где расположен требуемый ресурс. Клиентская часть также осуществляет прием ответов от серверной части и преобразование их в локальный формат, так что для приложения выполнение локальных и удаленных запросов неразличимо.

### 2.3.4. Сетевые службы и сетевые сервисы

Совокупность серверной и клиентской частей ОС, предоставляющих доступ к конкретному типу ресурса компьютера через сеть, называется *сетевой службой*. В приведенном выше примере клиентская и серверная части ОС, совместно обеспечивающие доступ через сеть к файловой системе компьютера, образуют файловую службу.

Говорят, что сетевая служба предоставляет пользователям сети некоторый набор *услуг*. Эти услуги иногда называют также *сетевым сервисом* (от англоязычного термина «service»). Необходимо отметить, что этот термин в технической литературе переводится и как «сервис», и как «услуга», и как «служба». Хотя указанные термины иногда используются как синонимы, следует иметь в виду, что в некоторых случаях различие в значениях этих терминов носит принципиальный характер. Далее в тексте под «службой» мы будем понимать сетевой компонент, который реализует некоторый набор услуг, а под «сервисом» – описание того набора услуг, который предоставляется данной службой. Таким образом, сервис – это интерфейс между потребителем услуг и поставщиком услуг (службой).

Каждая служба связана с определенным типом сетевых ресурсов и / или определенным способом доступа к этим ресурсам. Например, служба печати обеспечивает доступ пользователей сети к разделяемым принтерам сети и предоставляет сервис печати, а почтовая служба предоставляет доступ к информационному ресурсу сети – электронным письмам. Способом доступа к ресурсам отличается, например, служба удаленного доступа: она предоставляет пользователям компьютерной сети доступ ко всем ее ресурсам через коммутируемые телефонные каналы. Для получения удаленного доступа к конкретному ресурсу, например, к принтеру, служба удаленного доступа взаимодействует со службой печати. Наиболее важными для пользователей сетевых ОС являются файловая служба и служба печати.

Среди сетевых служб можно выделить такие, которые ориентированы не на простого пользователя, а на администратора. Такие службы используются для организации работы сети. Например, служба Bindery операционной системы Novell NetWare 3.x позволяет администратору вести базу данных о сетевых пользователях компьютера, на котором работает эта

ОС. Более прогрессивным является подход с созданием централизованной справочной службы, или, по-другому, службы каталогов, которая предназначена для ведения базы данных не только обо всех пользователях сети, но и обо всех ее программных и аппаратных компонентах. В качестве примеров службы каталогов часто приводятся NDS компании Novell и StreetTalk компании Banyan. Другими примерами сетевых служб, предоставляющих сервис администратору, являются: служба мониторинга сети, позволяющая захватывать и анализировать сетевой трафик; служба безопасности, в функции которой может входить, в частности, выполнение процедуры логического входа с проверкой пароля; служба резервного копирования и архивирования.

От того, насколько богатый набор услуг предлагает операционная система конечным пользователям, приложениям и администраторам сети, зависит ее позиция в общем ряду сетевых ОС.

Сетевые службы по своей природе являются клиент-серверными системами. Поскольку при реализации любого сетевого сервиса естественно возникает источник запросов (клиент) и исполнитель запросов (сервер), то и любая сетевая служба содержит в своем составе две несимметричные части – клиентскую и серверную (рис. 2.2). Сетевая служба может быть представлена в операционной системе либо обеими (клиентской и серверной) частями, либо только одной из них.

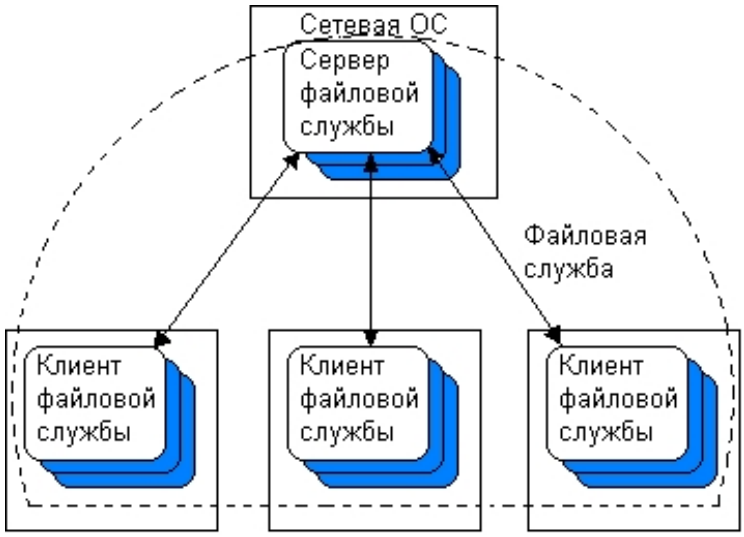



Рис. 2.2. Клиент-серверная природа сетевых служб

Обычно говорят, что сервер предоставляет свои ресурсы клиенту, а клиент ими пользуется. Необходимо отметить, что при предоставлении сетевой службой некоторой услуги используются ресурсы не только сервера,

но и клиента. Клиент может затрачивать значительную часть своих ресурсов (дискового пространства, процессорного времени и т. п.) на поддержание работы сетевой службы. Например, при реализации почтовой службы на диске клиента может храниться локальная копия базы данных, содержащей его переписку. В этом случае клиент выполняет большую работу при формировании сообщений в различных форматах, в том числе и сложном мультимедийном, поддерживает ведение адресной книги и выполняет еще много различных вспомогательных работ.

Принципиальной же разницей между клиентом и сервером является то, что инициатором выполнения работы сетевой службой всегда выступает клиент, а сервер находится в режиме пассивного ожидания запросов. Например, почтовый сервер осуществляет доставку почты на компьютер пользователя только при поступлении запроса от почтового клиента.

Обычно взаимодействие между клиентской и серверной частями стандартизуется, так что один тип сервера может быть рассчитан на работу с клиентами разного типа, реализованными различными способами и, может быть, разными производителями. Единственное условие для этого – клиенты и сервер должны поддерживать общий стандартный протокол взаимодействия.

### 2.3.5. Встроенные сетевые службы и сетевые оболочки

На практике сложилось несколько подходов к построению сетевых операционных систем, различающихся глубиной внедрения сетевых служб в операционную систему (рис. 2.3):

– сетевые службы глубоко *встроены* в ОС;

– сетевые службы объединены в виде некоторого набора – *оболочки*;

– сетевые службы производятся и поставляются в виде *отдельного продукта*.

Первые сетевые ОС представляли собой совокупность уже существующей локальной ОС и надстроенной над ней сетевой оболочки. При этом в локальную ОС встраивался минимум сетевых функций, необходимых для работы сетевой оболочки, которая выполняла основные сетевые функции.

Однако в дальнейшем разработчики сетевых ОС посчитали более эффективным подход, при котором сетевая ОС с самого начала работы над ней задумывается и проектируется специально для работы в сети. Сетевые функции этих ОС глубоко встраиваются в основные модули системы, что обеспечивает ее логическую стройность, простоту эксплуатации и модификации, а также высокую производительность. Важно, что при таком подходе отсутствует избыточность. Если все сетевые службы хорошо ин-

тегрированы, т. е. рассматриваются как неотъемлемые части ОС, то все внутренние механизмы такой операционной системы могут быть оптимизированы для выполнения сетевых функций. Например, ОС Windows NT компании Microsoft за счет встроенности сетевых средств обеспечивает более высокие показатели производительности и защищенности информации по сравнению с сетевой ОС LAN Manager той же компании, являющейся надстройкой над локальной операционной системой OS / 2. Другими примерами сетевых ОС со встроенными сетевыми службами являются все современные версии UNIX, NetWare, OS / 2 Warp.

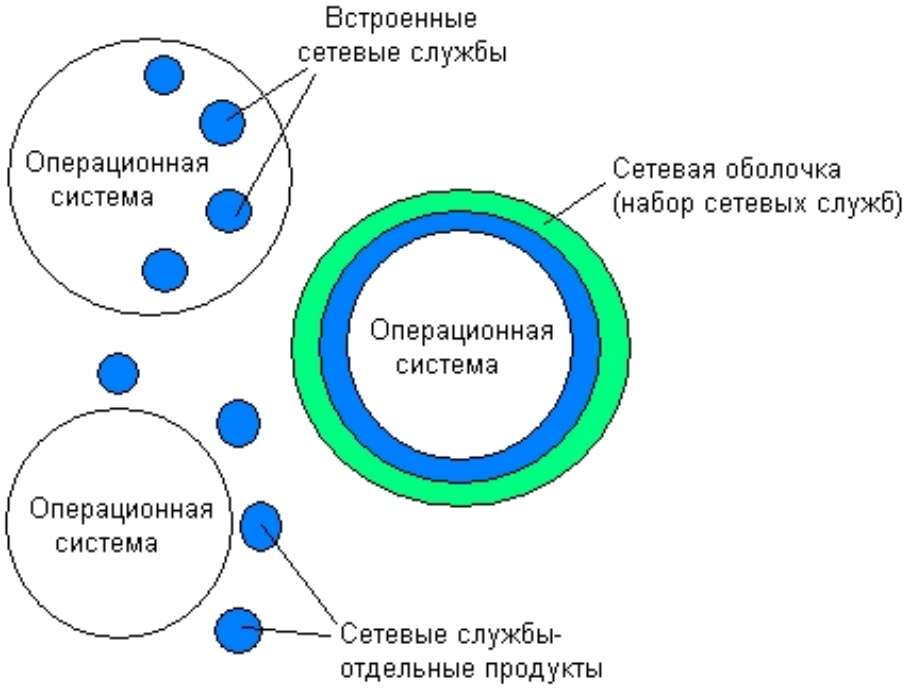



Рис. 2.3. Варианты построения сетевых ОС

Другой вариант реализации сетевых служб – объединение их в виде некоторого набора (оболочки), при этом все службы такого набора должны быть согласованы между собой, т. е. в своей работе они могут обращаться друг к другу, могут иметь в своем составе общие компоненты, например, общую подсистему аутентификации пользователей или единый пользовательский интерфейс. Для работы оболочки необходимо наличие некоторой локальной операционной системы, которая бы выполняла обычные функции, необходимые для управления аппаратурой компьютера и в среде которой выполнялись бы сетевые службы, составляющие эту оболочку. Оболочка представляет собой самостоятельный программный продукт и, как всякий продукт, имеет название, номер версии и другие соответствующие

характеристики. В качестве примеров сетевой оболочки можно указать, в частности, LAN Server и LAN Manager.

Одна и та же оболочка может предназначаться для работы над совершенно разными операционными системами. В таких случаях оболочка должна строиться с учетом специфики той операционной системы, над которой она будет работать. Так, LAN Server, например, существует в различных вариантах – для работы над операционными системами VAX VMS, VM, OS/400, AIX, OS / 2.

Сетевые оболочки часто подразделяются на клиентские и серверные. Оболочка, которая преимущественно содержит клиентские части сетевых служб, называется клиентской. Например, типичным набором программного обеспечения рабочей станции в сети NetWare является система MS-DOS с установленной над ней клиентской оболочкой NetWare, состоящей из клиентских частей файловой службы и службы печати, а также компонента, поддерживающего пользовательский интерфейс.

Серверная сетевая оболочка, примерами которой могут служить те же LAN Server и LAN Manager, а также NetWare for UNIX, File and Print Services for NetWare, ориентирована на выполнение серверных функций. Серверная оболочка как минимум содержит серверные компоненты двух основных сетевых служб – файловой службы и службы печати. Именно такой набор серверных компонентов реализован в упомянутых выше продуктах NetWare for UNIX и File and Print Services for NetWare. Некоторые же оболочки содержат настолько широкий набор сетевых служб, что их называют сетевыми операционными системами. Так, ни один обзор сетевых операционных систем не будет достаточно полным, если в нем отсутствует информация о LAN Server, LAN Manager, ENS, являющихся сетевыми оболочками. Таким образом, термин «сетевая операционная система» приобретает еще одно значение – набор сетевых служб, способных согласованно работать в общей операционной среде.

С одним типом ресурсов могут быть связаны разные службы, отличающиеся протоколом взаимодействия клиентских и серверных частей. Так, например, встроенная файловая служба Windows NT реализует протокол SMB, используемый во всех ОС компании Microsoft, а дополнительная файловая служба, входящая в состав оболочки File and Print Services for NetWare для Windows NT, работает по протоколу NCP, «родному» для сетей NetWare. Кроме того, в стандартную поставку Windows NT входит сервер FTP, предоставляющий услуги файлового сервера для UNIX-систем. Ничто не мешает приобрести и установить для работы в среде

Windows NT и другие файловые службы, такие, например, как NFS, кстати, имеющей несколько реализаций, выполненных разными фирмами. Наличие нескольких видов файловых услуг позволяет работать в сети приложениям, разработанным для разных операционных систем.

Сетевые оболочки создаются как для локальных, так и для сетевых операционных систем. Например, сетевая оболочка ENS (Enterprise Network Services), содержащая базовый набор сетевых служб операционной системы Banyan VINES, может работать над сетевыми ОС UNIX и NetWare. Конечно, для каждой из этих операционных систем требуется собственный вариант ENS.

Существует и третий способ реализации сетевой службы – в виде отдельного продукта. Например, сервер удаленного управления WinFrame – продукт компании Citrix – предназначен для работы в среде Windows NT. Он дополняет возможности встроенного в Windows NT сервера удаленного доступа Remote Access Server. Аналогичную службу удаленного доступа для NetWare также можно приобрести отдельно, купив программный продукт NetWare Connect.

С течением времени сетевая служба может получить разные формы реализации. Так, например, компания Novell планирует поставлять справочную службу NDS, первоначально встроенную в сетевую ОС NetWare, для других ОС. Для этого служба NDS будет переписана в виде отдельных продуктов, каждый из которых будет учитывать специфику соответствующей ОС. Уже имеются версии NDS для работы в средах SCO UNIX и HP-UX, Solaris 2.5 и Windows NT. А справочная служба StreetTalk уже давно существует и в виде встроенного модуля сетевой ОС Bayan Vines, и в составе оболочки ENS, и в виде отдельного продукта для различных операционных систем.

## 2.4. Одноранговые и серверные сетевые операционные системы

В зависимости от того, как распределены функции между компьютерами сети, они могут выступать в трех разных ролях:

– компьютер, занимающийся исключительно обслуживанием запросов других компьютеров, играет роль *выделенного сервера* сети;

– компьютер, обращающийся с запросами к ресурсам другой машины, исполняет роль *клиентского узла*;

– компьютер, совмещающий функции клиента и сервера, является *одноранговым узлом*.

Очевидно, что сеть не может состоять только из клиентских или только из серверных узлов. Сеть, оправдывающая свое назначение и обеспечивающая взаимодействие компьютеров, может быть построена по одной из трех следующих схем:

– сеть на основе одноранговых узлов – *одноранговая сеть*;

– сеть на основе клиентов и серверов – *сеть с выделенными серверами*;

– сеть, включающая узлы всех типов, – *гибридная сеть*.

Каждая из этих схем обладает своими достоинствами и недостатками, определяющими области их применения.

### 2.4.1. Операционные системы в одноранговых сетях

В одноранговых сетях (рис. 2.4) все компьютеры равны в возможностях доступа к ресурсам друг друга. Каждый пользователь может по своему желанию объявить какой-либо ресурс своего компьютера разделяемым, после чего другие пользователи могут его использовать. В одноранговых сетях на всех компьютерах устанавливается такая операционная система, которая предоставляет всем компьютерам в сети *потенциально* равные возможности. Сетевые операционные системы такого типа называются *одноранговыми ОС*. Очевидно, что одноранговые ОС должны включать как серверные, так и клиентские компоненты сетевых служб (на рис. 2.4 они соответственно обозначены буквами С и К). Примерами одноранговых ОС могут служить LANtastic, Personal Ware, Windows for Workgroups, Windows NT Workstation, Windows 95 / 98.

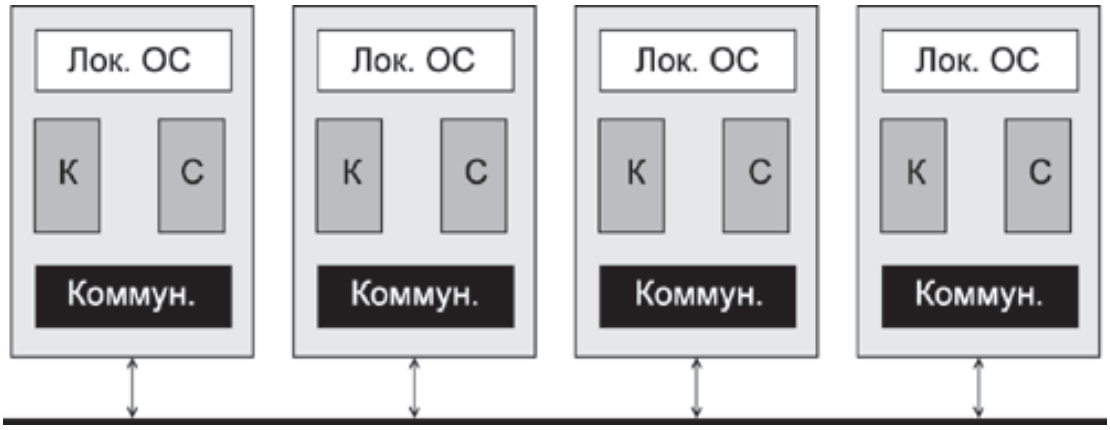



Рис. 2.4. Одноранговая сеть

При потенциальном равноправии всех компьютеров в одноранговой сети часто возникает функциональная несимметричность. Обычно в сети имеются пользователи, которые не желают предоставлять свои ресурсы в

совместное пользование. В таком случае серверные возможности их операционных систем не активизируются, и компьютеры выполняют функцию «чистых» клиентов (на рисунке неиспользуемые компоненты ОС изображены затемненными).

В то же время администратор может закрепить за некоторыми компьютерами сети только функции обслуживания запросов остальных компьютеров, превратив их таким образом в «чистые» серверы, за которыми не работают пользователи. В такой конфигурации одноранговые сети становятся похожи на сети с выделенными серверами, но это только внешняя схожесть: между этими двумя типами сетей остается существенное внутреннее различие. Изначально в одноранговых сетях специализация ОС не зависит от того, какую функциональную роль выполняет компьютер – клиента или сервера. Изменение роли компьютера в одноранговой сети достигается за счет того, что функции серверной или клиентской частей просто не используются.

Одноранговые сети проще в организации и эксплуатации, по этой схеме организуется работа в небольших сетях, в которых количество компьютеров не превышает 10 – 20. В этом случае нет необходимости в применении централизованных средств администрирования: нескольким пользователям нетрудно договориться между собой о перечне разделяемых ресурсов и паролях доступа к ним. Однако в больших сетях средства централизованного администрирования, хранения и обработки данных, а особенно защиты данных становятся необходимыми, и такие возможности легче обеспечить в сетях с выделенными серверами.

### 2.4.2. Операционные системы в сетях с выделенными серверами

В сетях с выделенными серверами (рис. 2.5) используются специальные варианты сетевых ОС, которые оптимизированы для работы в роли серверов и называются *серверными ОС*. Пользовательские компьютеры в этих сетях работают под управлением *клиентских ОС*.

Специализация операционной системы для работы в качестве сервера является естественным способом повышения эффективности серверных операций. А необходимость такого повышения часто ощущается весьма остро, особенно в крупной сети. При существовании в сети сотен или даже тысяч пользователей интенсивность запросов к совместно используемым ресурсам может быть очень большой, и сервер должен справляться с этим потоком запросов без значительных задержек. Очевидным решением этой проблемы является использование в качестве сервера компьютера с мощной аппаратной платформой и операционной системой, оптимизированной для серверных функций.

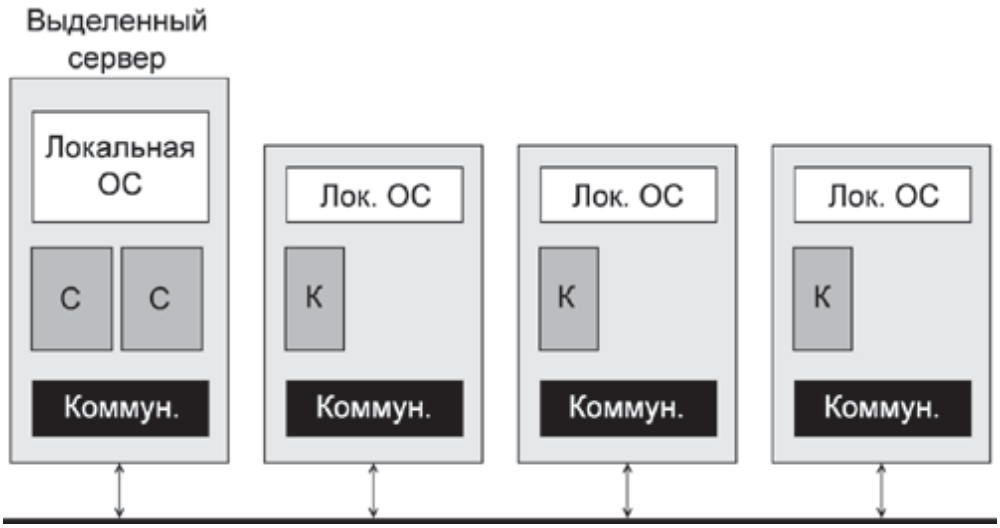



Рис. 2.5. Сеть с выделенными серверами

Чем меньше функций выполняет ОС, тем более эффективно можно их реализовать, поэтому для оптимизации серверных операций разработчики ОС вынуждены ущемлять некоторые другие ее функции, иногда вплоть до полного их отбрасывания. Одним из ярких примеров такого подхода является серверная ОС NetWare. Ее разработчики поставили перед собой цель оптимизировать выполнение файлового сервиса и сервиса печати. Для этого они полностью исключили из системы многие элементы, важные для универсальной ОС, в частности, графический интерфейс пользователя, поддержку универсальных приложений, защиту приложений мультипрограммного режима друг от друга, механизм виртуальной памяти. Все это позволило добиться уникальной скорости файлового доступа и вывело эту операционную систему в лидеры серверных ОС на долгое время.

Однако слишком узкая специализация некоторых серверных ОС является одновременно и их слабой стороной. Так, отсутствие в NetWare универсального интерфейса программирования и средств защиты приложений не позволяет использовать ее в качестве среды для выполнения приложений, приводит к необходимости включения в сеть других серверных ОС, когда требуется выполнение функций, отличных от файлового сервиса и сервиса печати.

Поэтому разработчики многих серверных операционных систем отказываются от функциональной ограниченности и включают в состав серверных ОС все компоненты, позволяющие использовать их в качестве универсального сервера и даже в качестве клиентской ОС. Такие серверные ОС снабжаются развитым графическим пользовательским интерфейсом и поддерживают универсальный API. Это сближает их с одноранго-

выми операционными системами, но существует несколько отличий, которые оправдывают отнесение их к классу серверных ОС:

– поддержка мощных аппаратных платформ, в том числе мультипроцессорных;

– поддержка большого числа одновременно выполняемых процессов и сетевых соединений;

– включение в состав ОС компонентов централизованного администрирования сети (например, справочной службы или службы аутентификации и авторизации пользователей сети);

– более широкий набор сетевых служб.

Клиентские операционные системы в сетях с выделенными серверами обычно освобождены от серверных функций, что значительно упрощает их организацию. Разработчики клиентских ОС уделяют основное внимание пользовательскому интерфейсу и клиентским частям сетевых служб. Наиболее простые клиентские ОС поддерживают только базовые сетевые службы – обычно файловую службу и службу печати. В то же время существуют так называемые универсальные клиенты, которые поддерживают широкий набор клиентских частей, позволяющих им работать практически со всеми серверами сети.

Многие компании, разрабатывающие сетевые ОС, выпускают два варианта одной и той же операционной системы. Один вариант предназначен для работы в качестве серверной ОС, а другой – в качестве клиентской. Эти варианты чаще всего основаны на одном и том же базовом коде, но отличаются набором служб и утилит, а также параметрами конфигурации, некоторые из которых устанавливаются по умолчанию и не поддаются изменению.

Например, операционная система Windows NT выпускается в варианте для рабочей станции – Windows NT Workstation – и в варианте для выделенного сервера – Windows NT Server. Оба эти варианта операционной системы включают клиентские и серверные части многих сетевых служб.

Так, ОС Windows NT Workstation, кроме выполнения функций сетевого клиента, может предоставлять сетевым пользователям файловый сервис, сервис печати, сервис удаленного доступа и другие сервисы, а следовательно, может служить основой для одноранговой сети. С другой стороны, ОС Windows NT Server содержит все необходимые средства, которые позволяют использовать компьютер под ее управлением в качестве клиентской рабочей станции. Под управлением ОС Windows NT Server имеется возможность локально запускать прикладные программы, которые мо-

гут потребовать выполнения клиентских функций ОС при появлении запросов к ресурсам других компьютеров сети. Windows NT Server имеет такой же развитый графический интерфейс, как и Windows NT Workstation, что позволяет с равным успехом использовать эти ОС для интерактивной работы пользователя или администратора.

Однако вариант Windows NT Server имеет больше возможностей для предоставления ресурсов своего компьютера другим пользователям сети, так как поддерживает более широкий набор функций, большее количество одновременных соединений с клиентами, централизованное управление сетью, более развитые средства защиты. Поэтому имеет смысл применять Windows NT Server в качестве ОС для выделенных серверов, а не клиентских компьютеров.

В больших сетях наряду с отношениями «клиент-сервер» сохраняется необходимость и в одноранговых связях, поэтому такие сети чаще всего строятся по гибридной схеме (рис. 2.6).

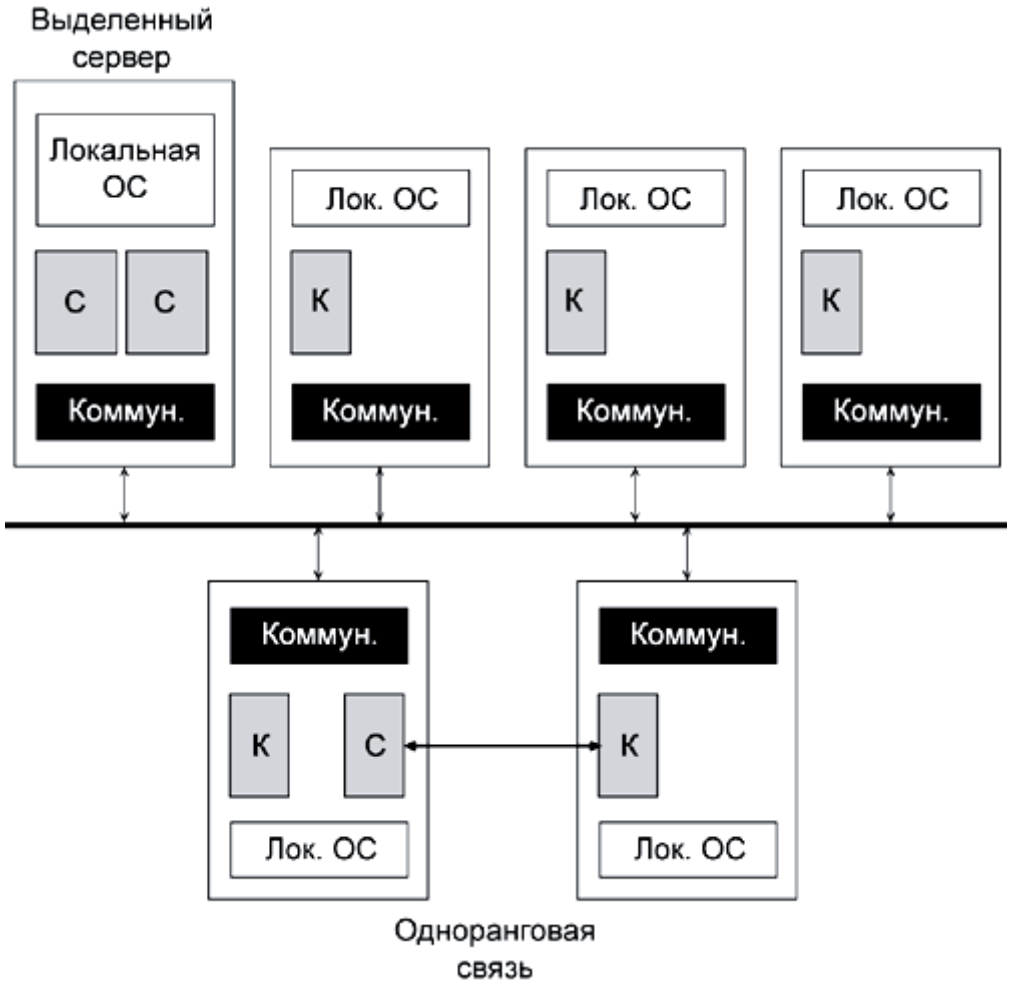



Рис. 2.6. Гибридная сеть

## 2.5. Требования к современным операционным системам

Главным требованием, предъявляемым к операционной системе, является выполнение ею основных функций эффективного управления ресурсами и обеспечение удобного интерфейса для пользователя и прикладных программ. Современная ОС, как правило, должна поддерживать мультипрограммную обработку, виртуальную память, свопинг, многооконный графический интерфейс пользователя, а также выполнять многие другие необходимые функции и услуги. Кроме этих требований функциональной полноты, к операционным системам предъявляются не менее важные эксплуатационные требования, которые перечислены ниже.

– *Расширяемость.* В то время как аппаратная часть компьютера устаревает за несколько лет, полезная жизнь операционных систем может измеряться десятилетиями. Примером может служить ОС UNIX. Поэтому операционные системы всегда эволюционно изменяются со временем, и эти изменения более значимы, чем изменения аппаратных средств. Изменения ОС обычно заключаются в приобретении ею новых свойств, например, поддержке новых типов внешних устройств или новых сетевых технологий. Если код ОС написан таким образом, что дополнения и изменения могут вноситься без нарушения целостности системы, такую ОС называют расширяемой. Расширяемость достигается за счет модульной структуры ОС, при которой программы строятся из набора отдельных модулей, взаимодействующих только через функциональный интерфейс.

– *Переносимость*. В идеале код ОС должен легко переноситься с процессора одного типа на процессор другого типа и с аппаратной платформы, которые различаются не только типом процессора, но и способом организации всей аппаратуры компьютера, одного типа на аппаратную платформу другого типа. Переносимые ОС имеют несколько вариантов реализации для разных платформ. Такое свойство ОС называют также *многоплатформенностью.*

– *Совместимость.* Существует несколько «долгоживущих» популярных операционных систем (разновидности UNIX, MS-DOS, Windows 3.x, Windows NT, OS / 2), для которых наработана широкая номенклатура приложений. Некоторые из них пользуются широкой популярностью. Поэтому для пользователя, переходящего по тем или иным причинам с одной ОС на другую, очень привлекательна возможность запуска в новой операционной системе привычного приложения. Если ОС имеет средства для выполнения прикладных программ, написанных для других операционных систем, то про нее говорят, что она обладает совместимостью с этими ОС. Следует различать совместимость на уровне двоичных

кодов и совместимость на уровне исходных текстов. Понятие совместимости включает также поддержку пользовательских интерфейсов других ОС.

– *Надежность* и *отказоустойчивость*. Система должна быть защищена как от внутренних, так и от внешних ошибок, сбоев и отказов. Ее действия должны быть всегда предсказуемыми, а приложения не должны иметь возможности наносить вред ОС. Надежность и отказоустойчивость ОС прежде всего определяются архитектурными решениями, положенными в ее основу, а также качеством ее реализации (отлаженностью кода). Кроме того, важно, включает ли ОС программную поддержку аппаратных средств обеспечения отказоустойчивости, таких, например, как дисковые массивы или источники бесперебойного питания.

– *Безопасность*. Современная ОС должна защищать данные и другие ресурсы вычислительной системы от несанкционированного доступа. Чтобы ОС обладала свойством безопасности, она должна как минимум иметь в своем составе средства аутентификации – определения легальности пользователей, авторизации – предоставления легальным пользователям дифференцированных прав доступа к ресурсам, аудита – фиксации всех подозрительных для безопасности системы событий. Свойство безопасности особенно важно для сетевых ОС. В таких ОС к задаче контроля доступа добавляется задача защиты данных, передаваемых по сети.

– *Производительность*. Операционная система должна обладать настолько хорошим быстродействием и временем реакции, насколько это позволяет аппаратная платформа. На производительность ОС влияет много факторов, среди которых основными являются архитектура ОС, многообразие функций, качество программирования кода, возможность исполнения ОС на высокопроизводительной (многопроцессорной) платформе.

## 2.6. Вопросы и задания для самопроверки

1. Поясните определение операционной системы как расширенной машины.

2. В соответствии с определением ОС, ее главными функциями являются предоставление удобств пользователю и эффективное управление ресурсами компьютера. Какая их этих двух функций должна была доминировать в мультипрограммных ОС времен IBM / 360? В первых ОС для персональных компьютеров?

3. В чем состоит отличие виртуальных машин, предоставляемых операционной системой простому пользователю и прикладному программисту?

4. Сравните интерфейс прикладного программиста с операционной системой и интерфейс системного программиста с реальной аппаратурой.

Что можно сказать о разнообразии и мощности интерфейсных функций, имеющихся в распоряжении каждого из них?

5. Назовите абстрактно сформулированные задачи ОС по управлению любым типом ресурса. Конкретизируйте эти задачи применительно к процессору, памяти, внешним устройствам.

6. Вставьте пропущенные определения: «Пользователю ... ОС не требуется знать, на каком из компьютеров сети хранятся файлы, с которыми он работает, а пользователю ... ОС эти сведения обычно необходимы».

7. Какие из утверждений верны?

а) сетевая операционная система – это совокупность операционных систем всех компьютеров сети;

б) сетевая операционная система – это операционная система отдельного компьютера, способного работать в сети;

в) сетевая операционная система – это набор сетевых служб, выполненный в виде оболочки.

8. Какой минимум функциональных возможностей надо добавить к локальной ОС, чтобы она стала сетевой?

9. Перечислите основные сетевые службы. Какие из них, как правило, встроены в операционную систему?

10. Какие из утверждений верны?

а) редиректор – клиентская часть сетевой службы;

б) редиректор – модуль, входящий в состав клиентской части сетевой службы, распознающий и перенаправляющий запросы к нужному сетевому серверу или локальной ОС.

11. Поясните значение следующих терминов применительно к сетевым ОС: «сервис», «сервер», «клиент», «служба», «оболочка», «услуга», «редиректор». Какие из них употребляются как синонимы?

12. Может ли сетевая оболочка работать над сетевой ОС?

13. В каких случаях может оказаться полезным наличие сразу нескольких серверных (клиентских) частей файловых служб?

14. Какие из следующих утверждений верны?

а) ОС выделенного сервера никогда не содержит клиентских частей сетевых служб;

б) в одноранговых ОС всегда имеются и клиентские, и серверные части сетевых служб;

в) в сетях с выделенными серверами могут поддерживаться одноранговые связи.

15. Может ли выделенный сервер обращаться с запросами к ресурсам клиентских станций?

16. Приведите примеры одноранговых ОС и ОС с выделенным сервером.

# ТЕМА 3. АРХИТЕКТУРА ОПЕРАЦИОННОЙ СИСТЕМЫ

Цель изучения темы – приобретение студентами понятий об общей архитектуре операционных систем, об их основных модулях, а также об универсальных подходах к структурированию операционных систем.

В результате изучения темы студенты должны:

– иметь представление о ядре операционной системы и режимах работы отдельных модулей;

– знать основные средства аппаратной поддержки операционной системы;

– иметь представление о микроядерной архитектуре операционных систем, ее преимуществах и недостатках;

– иметь представление о концепции множественных прикладных средств и ее назначении.

**Содержание темы**

1. Ядро и вспомогательные модули ОС.
2. Ядро в привилегированном режиме.
3. Многослойная структура ОС.
4. Аппаратная зависимость и переносимость ОС.
4.1. Типовые средства аппаратной поддержки ОС.
4.2. Машинно-зависимые компоненты ОС.
4.3. Переносимость операционной системы.
5. Микроядерная архитектура.
5.1. Концепция микроядерной архитектуры.
5.2. Преимущества и недостатки микроядерной архитектуры.
6. Совместимость и множественные прикладные среды.
6.1. Двоичная совместимость и совместимость исходных текстов.
6.2. Трансляция библиотек.
6.3. Способы реализации прикладных программных сред.
7. Вопросы и задания для самопроверки.

## 3.1. Ядро и вспомогательные модули операционной системы

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

– ядро – модули, выполняющие основные функции ОС;

– 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, т. е. 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, т. е. в виде исполняемых модулей стандартного для данной ОС формата, то часто бывает очень сложно провести четкую грань между операционной системой и приложениями.

Решение о том, является ли какая-либо программа частью ОС или нет, принимает производитель ОС (рис. 3.1). Среди многих факторов, способных повлиять на это решение, немаловажными являются перспективы того, будет ли программа иметь массовый спрос у потенциальных пользователей данной ОС.

Некоторая программа может существовать определенное время как пользовательское приложение, а потом стать частью ОС, или наоборот. Ярким примером такого изменения статуса программы является Web-браузер компании Microsoft, который сначала поставлялся как отдельное приложение, затем стал частью операционных систем Windows NT 4.0 и Windows 95 / 98, а сегодня существует большая вероятность того, что по решению суда этот браузер снова превратится в самостоятельное приложение.


Рис. 3.1. Нечеткость границы между ОС и приложениями

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;

– программы предоставления пользователю дополнительных услуг – специальный вариант пользовательского интерфейса, калькулятор, 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. 3.2).

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.

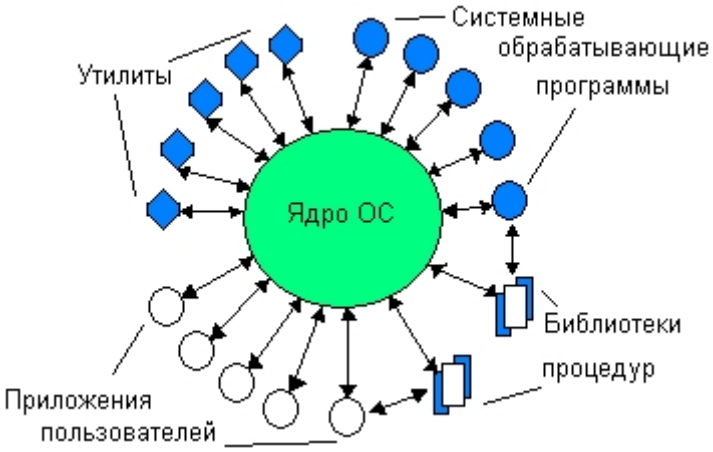



Рис. 3.2. Взаимодействие между ядром и вспомогательными модулями ОС

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

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

### 3.2. Ядро в привилегированном режиме

Для надежного управления ходом выполнения приложений операционная система должна иметь определенные привилегии по отношению к приложениям. В противном случае 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 режим (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 (рис. 3.3).

Так как ядро выполняет все основные функции ОС, то чаще всего именно ядро становится той частью ОС, которая работает в привилегированном режиме. Иногда это свойство – работа в привилегированном режиме – служит основным определением понятия «ядро».

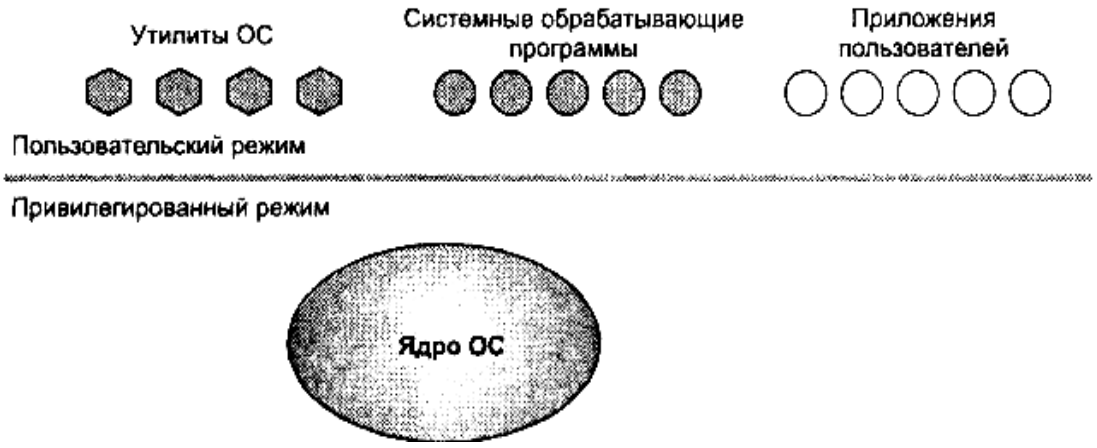



Рис. 3.3. Архитектура ОС с ядром в привилегированном режиме

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 от-

носится 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. Важно, что условия разрешения выполнения критичных инструкций находятся под полным контролем ОС и этот контроль обеспечивается за счет набора инструкций, безусловно запрещенных для пользовательского режима.

Аналогичным образом обеспечиваются привилегии ОС при доступе к памяти. Например, выполнение инструкции доступа к памяти для приложения разрешается, если инструкция обращается к области памяти, отведенной данному приложению операционной системой, и запрещается при обращении к областям памяти, занимаемым ОС или другими приложениями. Полный контроль ОС над доступом к памяти достигается за счет того, что инструкция или инструкции конфигурирования механизмов защиты памяти (например, изменения ключей защиты памяти в мэйнфреймах IBM или указателя таблицы дескрипторов памяти в процессорах Pentium) разрешается выполнять только в привилегированном режиме.

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. 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.

В некоторых случаях разработчики ОС отступают от этого классического варианта архитектуры, организуя работу ядра и приложений в одном и том же режиме. Так, известная специализированная операционная система NetWare компании Novell использует привилегированный режим процессоров Intel x86/ Pentium как для работы ядра, так и для работы своих специфических приложений – загружаемых модулей NLM (рис. 3.4). При таком построении ОС обращения приложений к ядру выполняются быстрее, так как нет переключения режимов, однако при этом отсутствует надежная аппаратная защита памяти, занимаемой модулями ОС, от некорректно работающего

приложения. Разработчики NetWare пошли на такое потенциальное снижение надежности своей операционной системы, поскольку ограниченный набор ее специализированных приложений позволяет компенсировать этот архитектурный недостаток за счет тщательной отладки каждого приложения.

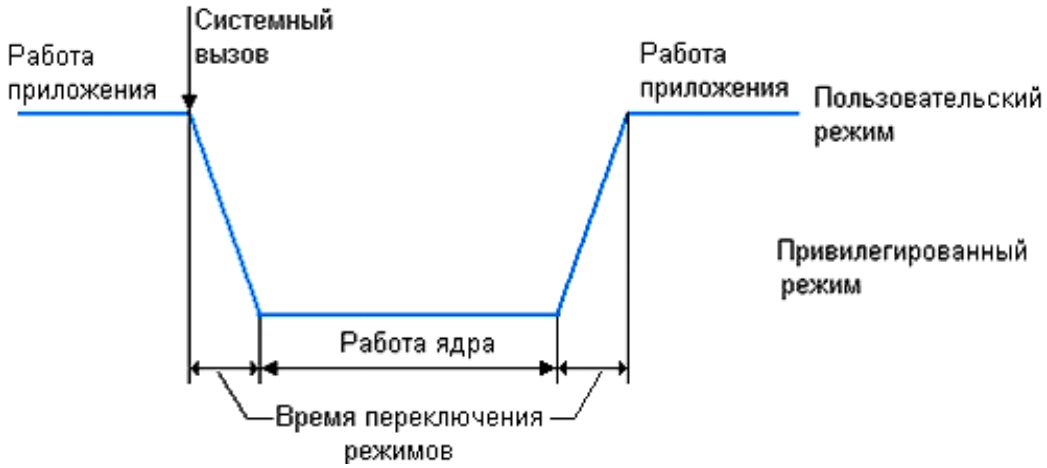



Рис. 3.4. Упрощенная архитектура ОС NetWare

В одном режиме работают также ядро и приложения тех операционных систем, которые разработаны для процессоров, вообще не поддерживающих привилегированного режима работы. Наиболее популярным процессором такого типа был процессор Intel 8088 / 86, послуживший основой для персональных компьютеров компании IBM. Операционная система MS-DOS, разработанная компанией Microsoft для этих компьютеров, состояла из двух модулей msdos.sys и io.sys, составлявших ядро системы, к которым с системными вызовами обращались командный интерпретатор command.com, системные утилиты и приложения. Некорректно написанные приложения вполне могли разрушить основные модули MS-DOS, что иногда и происходило, но область использования MS-DOS (и многих подобных ей ранних операционных систем для персональных компьютеров, таких как MSX, CP / M) и не предъявляла высоких требований к надежности ОС.

### 3.3. 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, со-

ставляют верхний слой системы. 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.

Многослойный подход является универсальным и эффективным способом декомпозиции сложных систем любого типа, в том числе и программных (рис. 3.5). 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 либо выполнить свою работу самостоятельно, либо обратиться к другому модулю своего слоя, либо обратиться за помощью к нижележащему слою через межслойный интерфейс.


Рис. 3.5. Трехслойная схема вычислительной системы

Такая организация системы имеет много достоинств. Она существенно упрощает разработку системы, так как позволяет сначала определить сверху вниз функции слоев и межслойные интерфейсы, а затем при детальной реализации постепенно наращивать мощность функций слоев, двигаясь снизу вверх. Кроме того, 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 (рис. 3.6).

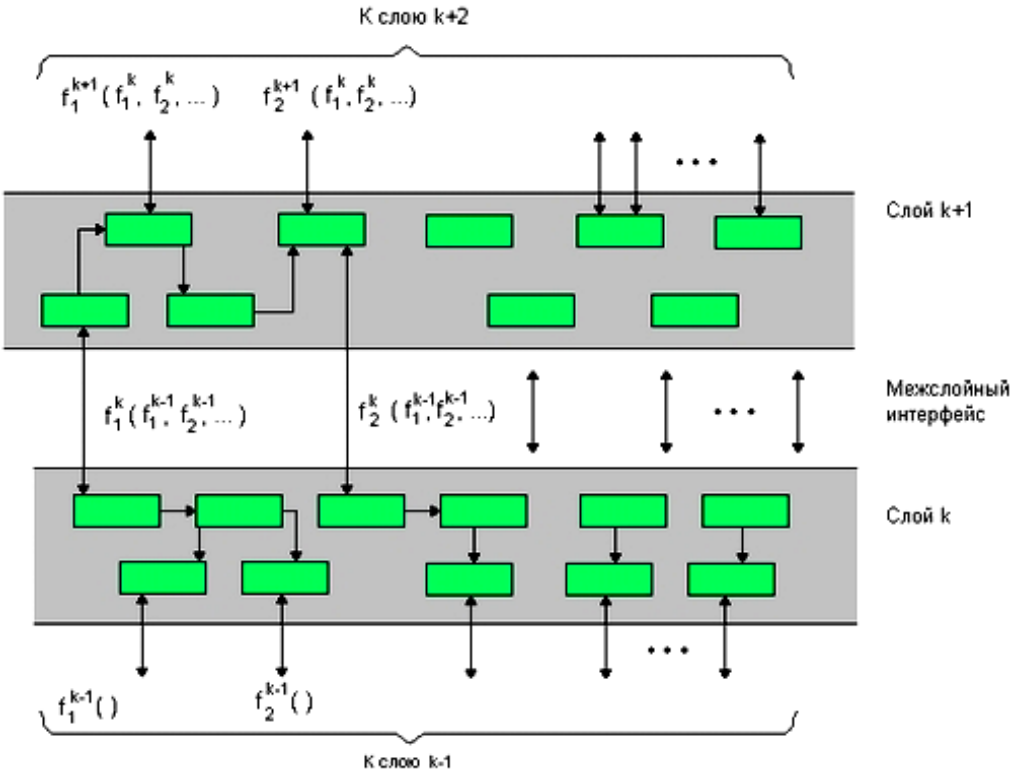



Рис. 3.6. Концепция многослойного взаимодействия

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

Ядро может состоять из следующих слоев.

– *Средства аппаратной поддержки ОС*. До сих пор об операционной системе говорилось как о комплексе программ, но, вообще говоря, часть функций ОС может выполняться и аппаратными средствами. Поэтому иногда можно встретить определение операционной системы как совокупности программных и аппаратных средств. К операционной системе относят, естественно, не все аппаратные устройства компьютера, а только средства аппаратной поддержки ОС, т. е. 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. Например, решение о том, что в данный момент нужно прервать выполнение текущего процесса А и начать выполнение процесса В, принимается менеджером процессов на вышележащем слое, а слою базовых механизмов передается только директива о том, что нужно выполнить переключение с контекста текущего процесса на контекст процесса В.

– *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. Разбиение на менеджеры может быть и несколько иным, например, менеджер файловой системы иногда объединяют с менеджером ввода-вывода, а функции управления доступом пользователей к системе в целом и ее отдельным объектам поручают отдельному менеджеру безопасности. 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. Менеджер должен отслеживать интенсивность обращений к страницам, время пребывания их в памяти, состояния процессов, использующих данные, и многие другие параметры, на основании которых он время от времени принимает решения о том, какие страницы необходимо выгрузить и какие загрузить. 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, предоставляют доступ к ресурсам системы в удобной и компактной форме, без указания деталей их физического расположения. Например, в операционной системе UNIX с помощью системного вызова fd = open("/doc/a.txt". 0_RDONLY) приложение открывает файл a.txt, хранящийся в каталоге /doc, а с помощью системного вызова read(fd, buffer, count) читает из этого файла в область своего адресного пространства, имеющую имя buffer, некоторое количество байт. Для осуществления таких комплексных действий системные вызовы обычно обращаются за помощью к функциям слоя менеджеров ресурсов, причем для выполнения одного системного вызова может понадобиться несколько таких обращений.

Приведенное разбиение ядра ОС на слои является достаточно условным (рис. 3.7). 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 базовые механизмы оформляются в отдельный слой. В этом случае менеджеры ресурсов не только планируют использование ресурсов, но и самостоятельно реализуют свои планы.


Рис. 3.7. Многослойная структура ядра ОС

Возможна и противоположная картина, когда ядро состоит из бóльшего 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.

Способ взаимодействия слоев в реальной ОС также может отклоняться от описанной выше схемы. Для ускорения работы ядра в некоторых случаях происходит непосредственное обращение с верхнего слоя к функциям нижних слоев, минуя промежуточные. Типичным примером такого «неправильного» взаимодействия является начальная стадия обработки системного вызова. На многих аппаратных платформах для реализации системного вызова используется инструкция программного прерывания. Этим приложение фактически вызывает модуль первичной обработки прерываний, который находится в слое базовых механизмов, а уже этот модуль вызывает нужную функцию из слоя системных вызовов. Сами функции системных вызовов также иногда нарушают субординацию иерархических слоев, обращаясь прямо к базовым механизмам ядра.

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 формализованно в гораздо большей степени.

### 3.4. Аппаратная зависимость и переносимость операционной системы

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 выде-

лить достаточно компактный слой машинно-зависимых компонентов ядра и сделать остальные слои ОС общими для разных аппаратных платформ.

### 3.4.1. Типовые средства аппаратной поддержки операционной системы

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:

- средства поддержки привилегированного режима;
- средства трансляции адресов;
- средства переключения процессов;
- система прерываний;
- системный таймер;
- 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 для 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 операций (т. е. 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.

Переключение контекста выполняется по определенным командам процессора, например, по команде перехода на новую задачу. Такая команда вызывает автоматическую загрузку данных из сохраненного контекста в регистры процессора, после чего процесс продолжается с прерванного ранее места.

*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, не синхронизированного с циклом работы процессора. Примерами таких событий может служить завершение операции ввода-вывода внешним устройством (например, запись блока данных контроллером диска), некорректное завершение арифметической операции (например, переполнение регистра), истечение интервала астрономического времени. При возникновении условий прерывания его источник (контроллер внешнего устройства, таймер, арифметический блок процессора и т. п.) генерирует определенный электрический сигнал. Этот сигнал прерывает выполнение процессором последовательности команд, задаваемой исполняемым кодом, и вызывает автоматический переход на заранее определенную процедуру, называемую *процедурой обработки прерываний*. В большинстве моделей процессоров отрабатываемый аппаратурой переход на процедуру обработки прерываний сопровождается заменой слова состояния машины (или даже всего контекста процесса), что позволяет одновременно с переходом по нужному адресу выполнить переход в привилегированный режим. После завершения обработки прерывания обычно происходит возврат к исполнению прерванного кода.

Прерывания играют важнейшую роль в работе любой операционной системы, являясь ее движущей силой. Действительно, бóльшая часть дей-

ствий ОС инициируется прерываниями различного типа. Даже системные вызовы от приложений выполняются на многих аппаратных платформах с помощью специальной инструкции прерывания, вызывающей переход к выполнению соответствующих процедур ядра (например, инструкция int в процессорах Intel или SVC в мэйнфреймах IBM).

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

### 3.4.2. Машинно-зависимые компоненты операционной системы

Одна и та же операционная система не может без каких-либо изменений устанавливаться на компьютерах, отличающихся типом процессора и / или 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.

Как показывает опыт разработки операционных систем, 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. Такая локализация машинно-зависимых модулей существенно упрощает перенос операционной системы на другую аппаратную платформу.

Объем машинно-зависимых компонентов ОС зависит от того, насколько велики отличия в аппаратных платформах, для которых разрабатывается ОС. Например, ОС, построенная на 32-битовых адресах, для переноса на машину с 16-битовыми адресами должна быть практически переписана заново. Одно из наиболее очевидных отличий – несовпадение системы команд процессоров – преодолевается достаточно просто. Операционная система программируется на языке высокого уровня, а затем соответствующим компилятором вырабатывается код для конкретного типа процессора. Однако во многих случаях различия в организации аппаратуры компьютера лежат гораздо глубже и преодолеть их таким способом не удается. Например, однопроцессорный и двухпроцессорный компьютеры требуют применения в ОС совершенно разных алгоритмов распределения процессорного времени. Аналогично отсутствие аппаратной поддержки виртуальной памяти приводит к принципиальному различию в реализации подсистемы управления памятью. В таких случаях не обойтись без внесения в код операционной системы специфики аппаратной платформы, для которой эта ОС предназначается.

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, т. е. 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. Эта система содержит драйверы для всех устройств, входящих в базовую конфигурацию компьютера: жестких и гибких дисков, клавиатуры, дисплея и т. д. Перечисленные 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.

### 3.4.3. 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. Чтобы обеспечить свойство мобильности ОС, разработчики должны следовать следующим правилам.

– 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. Программа, написанная на ассемблере, является переносимой только в тех случаях, когда перенос операционной системы планируется на компьютер, обладающий той же системой команд. В остальных случаях ассемблер используется только для тех непереносимых частей системы, которые должны непосредственно взаимодействовать с аппаратурой (например, обработчик прерываний),

или для частей, которые требуют максимальной скорости (например, целочисленная арифметика повышенной точности).

– 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. Для уменьшения аппаратной зависимости разработчики ОС должны также исключить возможность использования по умолчанию стандартных конфигураций аппаратуры или их характеристик. Аппаратно-зависимые параметры можно определить в программно-задаваемых данных абстрактного типа. Для осуществления всех необходимых действий по управлению аппаратурой, представленной этими параметрами, должен быть написан набор аппаратно-зависимых функций. Каждый раз, когда какому-либо модулю ОС требуется выполнить некоторое действие, связанное с аппаратурой, он манипулирует абстрактными данными, используя соответствующую функцию из имеющегося набора. Когда ОС переносится, то изменяются только эти данные и функции, которые ими манипулируют. Например, в ОС Windows NT диспетчер прерываний преобразует аппаратные уровни прерываний конкретного типа процессора в стандартный набор уровней прерываний IRQL, с которыми работают остальные модули операционной системы. Поэтому при переносе Windows NT на новую платформу нужно переписать, в частности, те коды диспетчера прерываний, которые занимаются отображением уровней прерывания на абстрактные уровни IRQL, а модули ОС, которые пользуются этими абстрактными уровнями, изменений не потребуют.

– Аппаратно-зависимый код должен быть надежно изолирован в нескольких модулях, а не распределяться 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 оформляться в виде компактных модулей, которые могут быть заменены аналогичными модулями для других процессоров. Для снятия платформенной зависимости, возникающей из-за различий между компьютерами разных производителей, построенными на одном и том же процессоре (например, MIPS R4000), должен быть введен хорошо локализованный программный слой машинно-зависимых функций.

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, лежащие 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 (рис. 3.8).

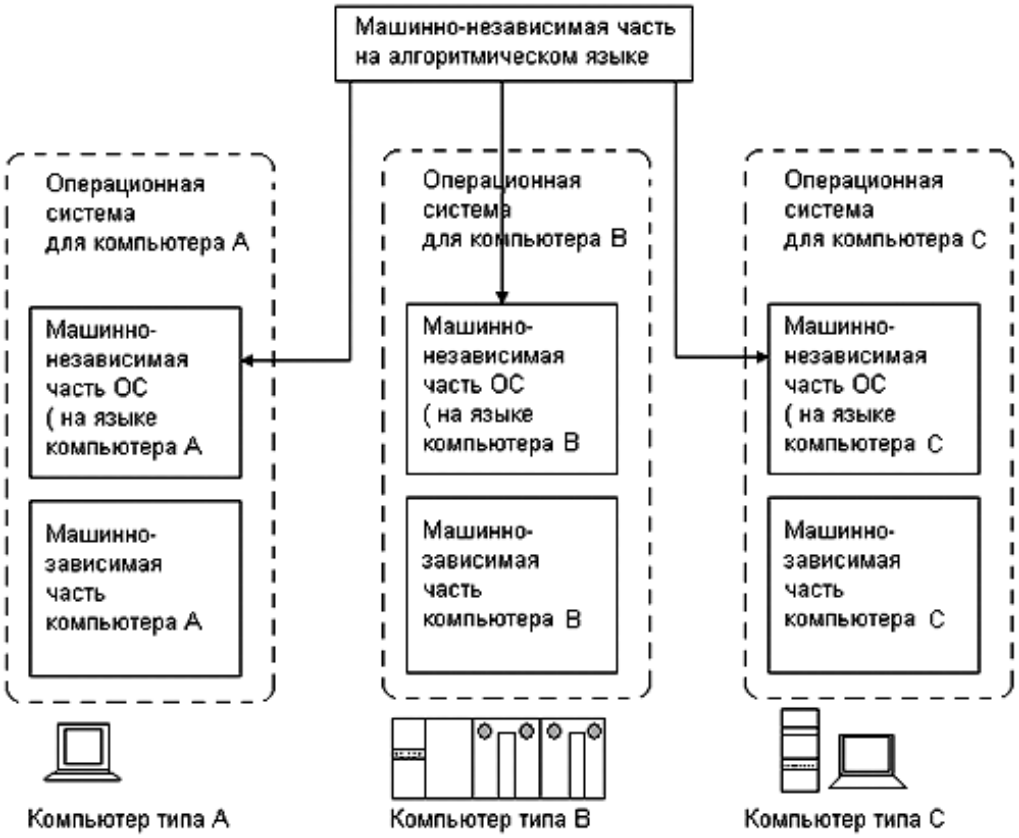



Рис. 3.8. Перенос операционной системы на разные аппаратные платформы

## 3.5. Микроядерная архитектура

### 3.5.1. Концепция микроядерной архитектуры

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*. 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, загрузке или чтению регистров устройств. Набор функций микроядра обычно соответствует функциям слоя базовых механизмов обычного ядра. Такие функции операционной системы практически невозможно выполнить в пространстве пользователя.

Все остальные, более высокоуровневые функции ядра оформляются в виде приложений, работающих в пользовательском режиме (рис. 3.9). 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.

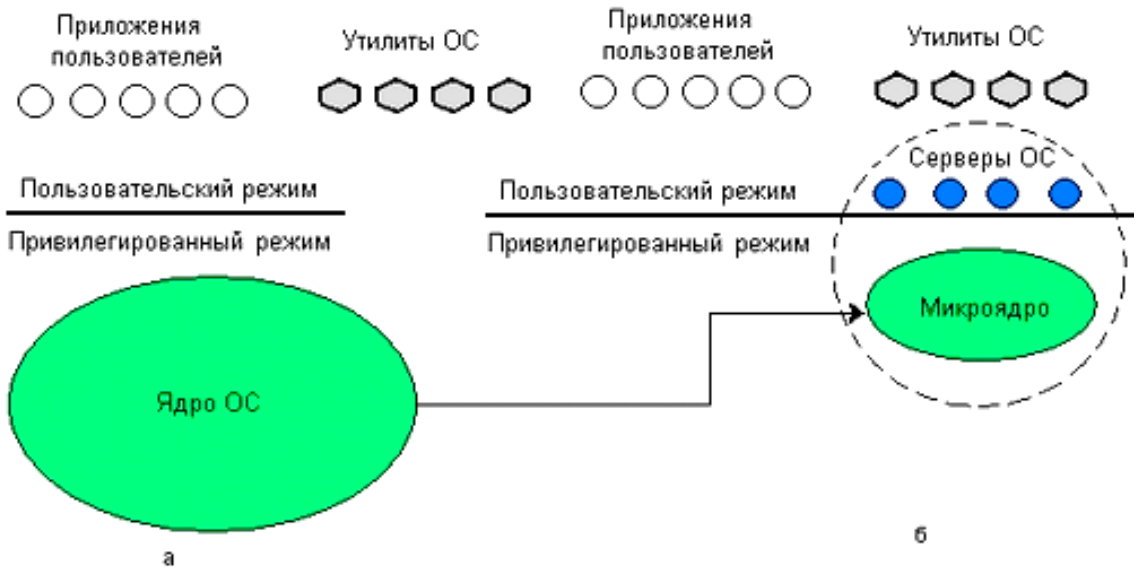



Рис. 3.9. Перенос основного объема функций ядра в пользовательское пространство

Совсем другая ситуация возникает, когда в форме приложения оформляется часть операционной системы. 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. 3.10). 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, которое выполняется 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.

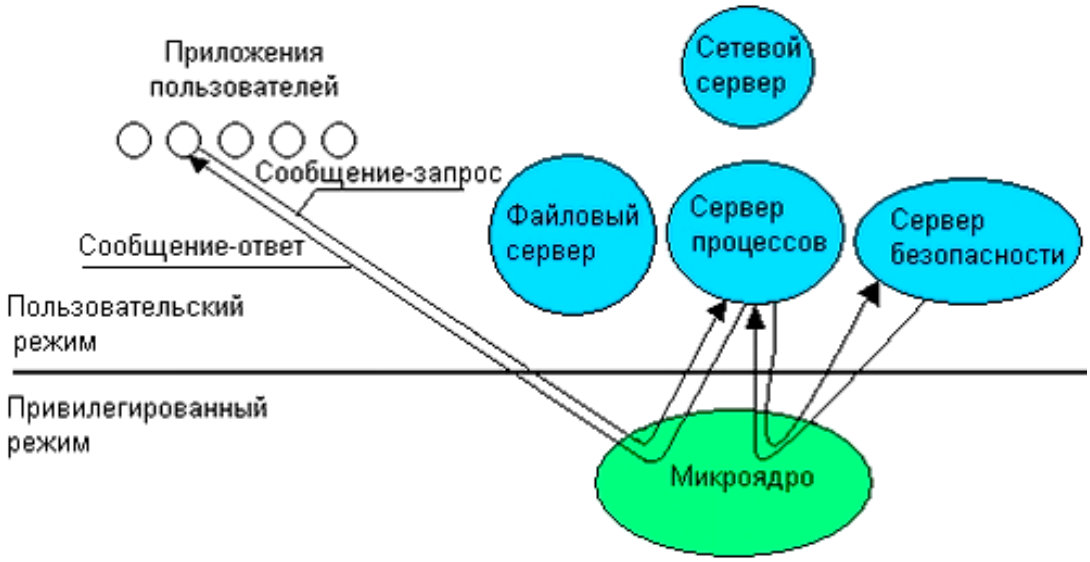



Рис. 3.10. Реализация системного вызова в микроядерной архитектуре

### 3.5.2. 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 изменения логически сгруппированы.

*Расширяемость* свойственна 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. Например, не всем пользователям нужны средства безопасности или поддержки распределенных вычислений, а удаление их из традиционного ядра чаще всего невозможно. Обычно традиционные операционные системы позволяют динамически добавлять в ядро или удалять из него только драйверы внешних устройств. Ввиду частых изменений в конфигурации подключенных к компьютеру внешних устройств подсистема ввода-вывода ядра допускает загрузку и выгрузку драйверов «на ходу», но для этого она разрабатывается особым образом (например, среда STREAMS в UNIX или менеджер ввода-вывода в Windows NT). 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 в ходе работы не нужные больше серверы обычными для остановки приложений средствами.

Использование микроядерной модели повышает *надежность* ОС. Каждый сервер выполняется в виде отдельного процесса в своей собственной области памяти и таким образом защищен от других серверов операционной системы в отличие от традиционной ОС, где все модули ядра могут влиять друг на друга. Если отдельный сервер терпит крах, то он может быть перезапущен без остановки 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 уменьшение объема 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: локальный транспорт просто заменяется на сетевой.

*Производительность*. При классической организации ОС (рис. 3.11, *а*) выполнение системного вызова сопровождается двумя переключениями режимов, а при микроядерной организации (рис. 3.11, *б*) – четырьмя. Та-

ким 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.

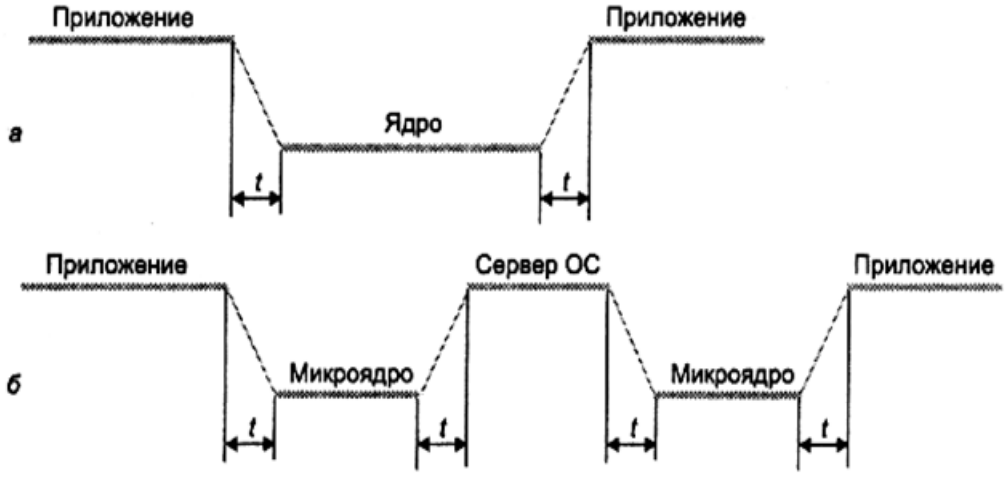



Рис. 3.11. Смена режимов при выполнении системного вызова

Главная проблема, с которой сталкиваются разработчики операционной системы, решившие применить микроядерный подход, – какие модули включать в микроядро, а какие выносить в пользовательское пространство. В идеальном варианте микроядро может состоять только из средств передачи сообщений, средств взаимодействия с аппаратурой, в т. ч. средств доступа к механизмам привилегированной защиты. Однако многие разработчики не всегда жестко придерживаются принципа минимизации функций ядра, часто жертвуя этим показателем ради повышения производительности. В результате различные 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.

### 3.6. Совместимость и множественные прикладные среды

В то время как многие архитектурные особенности операционных систем непосредственно касаются только системных программистов, концепция *множественных прикладных сред* непосредственно связана с нуждами конечных пользователей – возможностью операционной системы выполнять приложения, написанные для других операционных систем. Такое свойство операционной системы называется *совместимостью.*

### 3.6.1. Двоичная совместимость и совместимость исходных текстов

Необходимо различать совместимость на двоичном уровне и совместимость на уровне исходных текстов. Приложения обычно хранятся в ОС в виде исполняемых файлов, содержащих двоичные образы кодов и данных. Двоичная совместимость достигается в том случае, когда можно взять исполняемую программу и запустить ее на выполнение в среде другой ОС.

Совместимость на уровне исходных текстов требует наличия соответствующего компилятора в составе программного обеспечения компьютера, на котором предполагается выполнять данное приложение, а также совместимости на уровне библиотек и системных вызовов. При этом необходима перекомпиляция имеющихся исходных текстов в новый исполняемый модуль. Совместимость на уровне исходных текстов важна в основном для разработчиков приложений, в распоряжении которых эти исходные тексты всегда имеются. Но для конечных пользователей практическое значение имеет только двоичная совместимость, так как только в этом случае они могут использовать в различных операционных средах и на различных машинах один и тот же коммерческий продукт, поставляемый в виде двоичного исполняемого кода. Для пользователя, купившего в свое время пакет для MS-DOS (например, Lotus 1-2-3), важно, чтобы он мог запускать этот пакет без каких-либо изменений и на своей новой машине, работающей под управлением, например, Windows NT. Обладает ли новая ОС двоичной совместимостью или совместимостью исходных текстов с существующими операционными системами, зависит от многих факторов. Самый главный из них – архитектура процессора, на котором работает новая ОС. Если процессор использует тот же набор команд, возможно, с некоторыми добавлениями, и тот же диапазон адресов, то двоичная совместимость может быть достигнута довольно просто. Для этого достаточно соблюдать следующие условия:

– вызовы функций API, содержащих приложение, должны поддерживаться данной ОС;

– внутренняя структура исполняемого файла приложения должна соответствовать структуре исполняемых файлов данной ОС.

Гораздо сложнее достичь двоичной совместимости операционным системам, которые предназначены для выполнения на процессорах, имеющих разные архитектуры. Помимо соблюдения приведенных выше условий, необходимо организовать *эмуляцию* двоичного кода.

### 3.6.2. Трансляция библиотек

Выходом в таких случаях является использование так называемых *прикладных программных сред.* Одной из составляющих, которые форми-

руют прикладную программную среду, является набор функций интерфейса прикладного программирования API, которые операционная система предоставляет своим приложениям. Для сокращения времени на выполнение чужих программ прикладные среды имитируют обращения к библиотечным функциям.

Эффективность этого подхода связана с тем, что большинство сегодняшних программ работают под управлением GUI (графических интерфейсов пользователя) типа Windows, Mac или UNIX Motif, при этом приложения тратят большую часть времени, производя некоторые хорошо предсказуемые действия. Они непрерывно выполняют вызовы библиотек GUI для манипулирования окнами и для других связанных с GUI действий. Сегодня в типичных программах 60 – 80 % времени тратится на выполнение функций GUI и других библиотечных вызовов ОС. Именно это свойство приложений позволяет прикладным средам *компенсировать большие затраты времени, потраченные на покомандное эмулиро*вание программы. Тщательно спроектированная программная прикладная среда имеет в своем составе библиотеки, имитирующие внутренние библиотеки GUI, но написанные на «родном» коде. Таким образом достигается существенное ускорение выполнения программ с API другой операционной системы. Иногда такой подход называют трансляцией для того, чтобы отличать его от более медленного процесса эмулирования кода по одной команде за один раз. Например, для Windows-программы, работающей на Macintosh, при интерпретации команд процессора Intel 80x86 производительность может быть очень низкой. Но когда производится вызов функции GUI открытия окна, то модуль ОС, реализующий прикладную среду Windows, может перехватить этот вызов и перенаправить его на перекомпилированную для процессора Motorola 680x0 подпрограмму открытия окна. В результате на таких участках кода скорость работы программы может достичь (а возможно, и превзойти) скорость работы на «родном» процессоре.

Чтобы программа, написанная для одной ОС, могла быть выполнена в рамках другой ОС, недостаточно лишь обеспечить совместимость API. Концепции, положенные в основу разных ОС, могут входить в противоречие друг с другом. Например, в одной операционной системе приложению может быть разрешено непосредственно управлять устройствами ввода-вывода, в другой эти действия являются прерогативой ОС. Каждая операционная система имеет собственные механизмы защиты ресурсов, алгоритмы обработки ошибок и исключительных ситуаций, особую структуру процесса и схему управления памятью, собственную семантику доступа к файлам и графический пользовательский интерфейс. Для обеспечения совместимости

необходимо организовать бесконфликтное сосуществование в рамках одной ОС нескольких способов управления ресурсами компьютера.

### 3.6.3. Способы реализации прикладных программных сред

Создание полноценной прикладной среды, полностью совместимой со средой другой операционной системы, является достаточно сложной задачей, тесно связанной со структурой операционной системы. Существуют различные варианты построения множественных прикладных сред, отличающиеся как особенностями архитектурных решений, так и функциональными возможностями, обеспечивающими различную степень переносимости приложений.

Во многих версиях ОС UNIX транслятор прикладных сред реализуется в виде обычного приложения. В операционных системах, построенных с использованием микроядерной концепции, таких как, например, Windows NT или Workplace OS, прикладные среды выполняются в виде серверов пользовательского режима. А в OS / 2 с ее более простой архитектурой средства организации прикладных сред встроены глубоко в операционную систему.

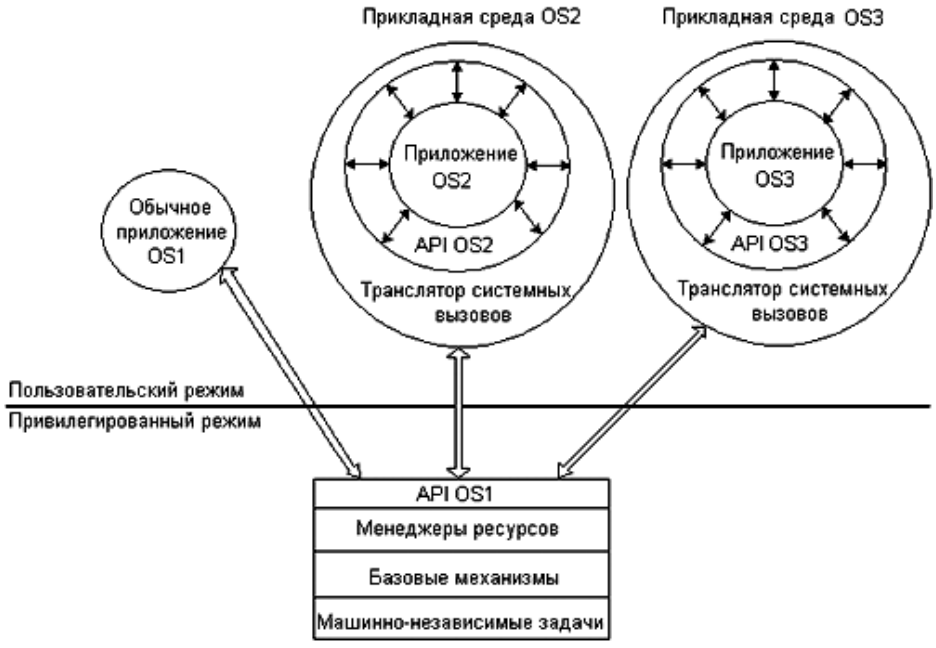



Рис. 3.12. Прикладные программные среды, транслирующие системные вызовы

Один из наиболее очевидных вариантов реализации множественных прикладных сред основывается на стандартной многоуровневой структуре ОС. На рис. 3.12 операционная система OS1 поддерживает,

кроме «родных» приложений, приложения операционных систем OS2 и OS3. Для этого в ее составе имеются специальные приложения – прикладные программные среды, – которые транслируют интерфейсы «чужих» операционных систем API OS2 и API OS3 в интерфейс «родной» операционной системы API OS1. Так, например, если бы в качестве OS2 выступала операционная система UNIX, а в качестве OS1 – OS / 2, для выполнения системного вызова создания процесса forkO в UNIX-приложении программная среда должна была бы обратиться к ядру операционной системы OS / 2 с системным вызовом DosExecPgmO.

К сожалению, поведение почти всех функций, составляющих API одной ОС, как правило, существенно отличается от поведения соответствующих функций другой ОС. Например, чтобы функция создания процесса в OS / 2 DosExecPgmO полностью соответствовала функции создания процесса forkO в UNIX-подобных системах, ее нужно было бы изменить, чтобы она поддерживала возможность копирования адресного пространства родительского процесса в пространство процесса-потомка. При нормальном поведении этой функции память процесса-потомка инициализируется на основе данных нового исполняемого файла.

В другом варианте реализации множественных прикладных сред операционная система имеет несколько равноправных прикладных программных интерфейсов. В приведенном на рис. 3.13 примере операционная система поддерживает приложения, написанные для OS1, OS2 и OS3. Для этого непосредственно в пространстве ядра системы размещены прикладные программные интерфейсы всех этих ОС: API OS1, API OS2 и API OS3.

В этом варианте функции уровня API обращаются к функциям нижележащего уровня ОС, которые должны поддерживать все три в общем случае несовместимые прикладные среды. В разных ОС по-разному осуществляется управление системным временем, используется разный формат времени дня, на основании собственных алгоритмов разделяется процессорное время и т. д. Функции каждого API реализуются ядром с учетом специфики соответствующей ОС, даже если они имеют аналогичное назначение. Например, как уже было сказано, функция создания процесса работает по-разному для приложений UNIX и OS / 2. При завершении процесса ядру также необходимо определять, к какой ОС относится данный процесс. Если этот процесс был создан по запросу UNIX-приложения, то в ходе его завершения ядро должно послать родительскому процессу сигнал, как это делается в ОС UNIX. А по завершении процесса OS / 2, созданного с параметром EXEC_SYNC, ядро должно отметить, что идентификатор

процесса не может быть повторно использован другим процессом OS / 2. Для того чтобы ядро могло выбрать нужный вариант реализации системного вызова, каждый процесс должен передавать в ядро набор идентифицирующих характеристик.

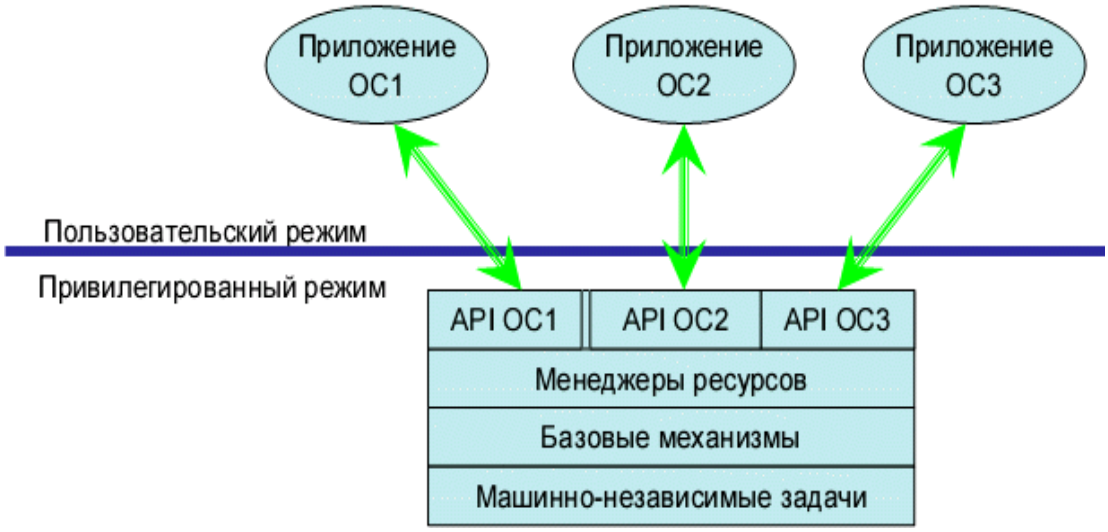



Рис. 3.13. Реализация совместимости на основе нескольких равноправных API

Еще один способ построения множественных прикладных сред основан на микроядерном подходе. При этом важно отделить базовые, общие для всех прикладных сред механизмы операционной системы от специфических для каждой из прикладных сред высокоуровневых функций, решающих стратегические задачи.

В соответствии с микроядерной архитектурой все функции ОС реализуются микроядром и серверами пользовательского режима. Важно, что каждая прикладная среда оформляется в виде отдельного сервера пользовательского режима и не включает базовых механизмов (рис. 3.14). Приложения, используя API, обращаются с системными вызовами к соответствующей прикладной среде через микроядро. Прикладная среда обрабатывает запрос, выполняет его, при необходимости обращаясь за помощью к базовым функциям микроядра, и отсылает результат приложению. В ходе выполнения запроса прикладная среда, в свою очередь, обращается к базовым механизмам ОС, реализуемым микроядром и другими серверами ОС.

Такому подходу к конструированию множественных прикладных сред присущи все достоинства и недостатки микроядерной архитектуры, в частности:

– можно добавлять и исключать прикладные среды, что является следствием расширяемости микроядерных ОС;

– надежность и стабильность выражаются в том, что при отказе одной из прикладных сред все остальные сохраняют работоспособность;

– низкая производительность микроядерных ОС сказывается на скорости работы прикладных сред, а значит, и на скорости выполнения приложений.

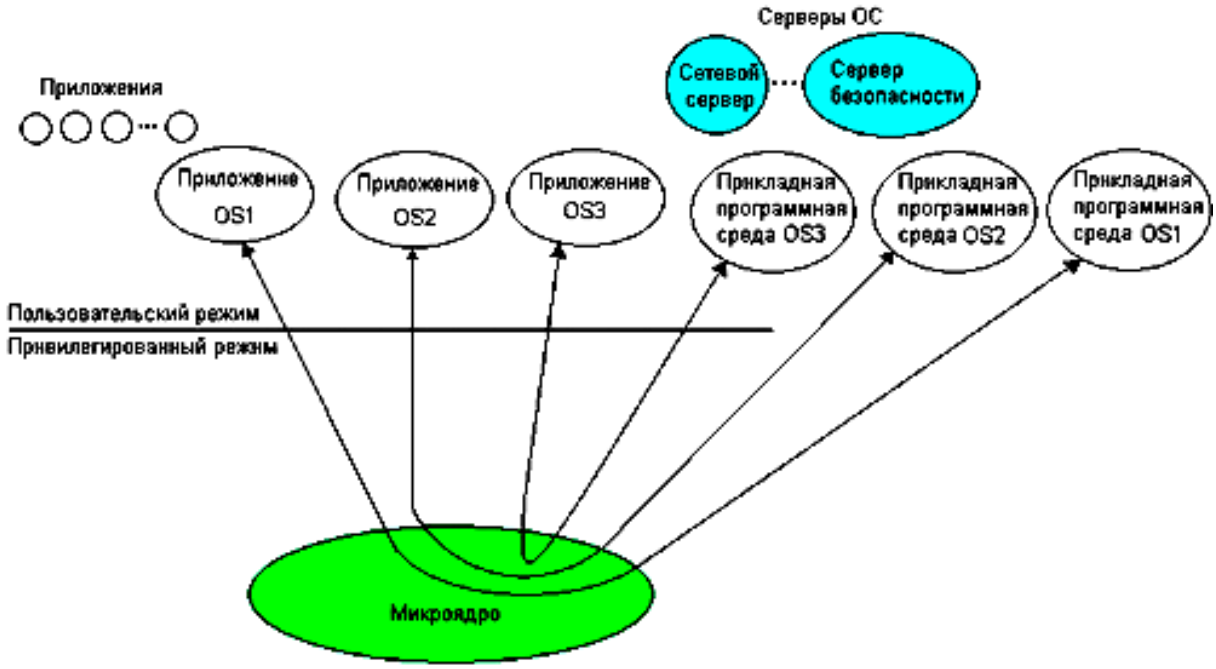



Рис. 3.14. Микроядерный подход к реализации множественных прикладных сред

Создание в рамках одной операционной системы нескольких прикладных сред для выполнения приложений различных ОС представляет собой путь, который позволяет иметь единственную версию программы и переносить ее между операционными системами. Множественные прикладные среды обеспечивают совместимость на двоичном уровне данной ОС с приложениями, написанными для других ОС. В результате пользователи получают большую свободу выбора операционных систем и более легкий доступ к качественному программному обеспечению.

## 3.7. Вопросы и задания для самопроверки

1. Какие из приведенных ниже терминов являются синонимами?
а) привилегированный режим;
б) защищенный режим;
в) режим супервизора;
г) пользовательский режим;

д) реальный режим;

е) режим ядра.

2. Можно ли, анализируя двоичный код программы, сделать вывод о невозможности ее выполнения в пользовательском режиме?

3. В чем состоят отличия работы процессора в привилегированном и пользовательском режимах?

4. В идеале микроядерная архитектура ОС требует размещения в микроядре только тех компонентов ОС, которые не могут выполняться в пользовательском режиме. Что заставляет разработчиков операционных систем отходить от этого принципа и расширять ядро за счет перенесения в него функций, которые могли бы быть реализованы в виде процессов-серверов?

5. Какие этапы включает разработка варианта мобильной ОС для новой аппаратной платформы?

6. Опишите порядок взаимодействия приложений с ОС, имеющей микроядерную архитектуру.

7. Какими этапами отличается выполнение системного вызова в микроядерной ОС и ОС с монолитным ядром?

8. Может ли программа, эмулируемая на «чужом» процессоре, выполняться быстрее, чем на «родном»?

# ТЕМА 4. ПРОЦЕССЫ И ПОТОКИ

Цель изучения темы – приобретение студентами знаний об организации рационального использования аппаратных и информационных ресурсов операционной системой, принципах работы процессов и потоков, а также методах их планирования и диспетчеризации.

В результате изучения темы студенты должны:

– иметь понятие о мультипрограммировании, знать отличия и критерии оптимизации систем пакетной обработки, систем разделения времени и систем реального времени;

– знать основные принципы мультипроцессорной обработки;

– знать основные виды алгоритмов планирования потоков;

– иметь представление о моментах перепланировки потоков.

## Содержание темы

1. Мультипрограммирование.
1.1. Мультипрограммирование в системах пакетной обработки.
1.2. Мультипрограммирование в системах разделения времени.
1.3. Мультипрограммирование в системах реального времени.
2. Мультипроцессорная обработка.
3. Планирование процессов и потоков.
3.1. Понятия «процесс» и «поток».
3.2. Создание процессов и потоков.
3.3. Планирование и диспетчеризация потоков.
3.4. Состояния потока.
3.5. Вытесняющие и невытесняющие алгоритмы планирования.
3.6. Алгоритмы планирования, основанные на квантовании.
3.7. Алгоритмы планирования, основанные на приоритетах.
3.8. Смешанные алгоритмы планирования.
3.9. Планирование в системах реального времени.
4. Моменты перепланировки.
5. Вопросы и задания для самопроверки.

## 4.1. Мультипрограммирование

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

граммы 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 за 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.

### 4.1.1. 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 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 обоснованным 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.

Системы пакетной обработки предназначались для решения задач в основном вычислительного характера, не требующих быстрого получения результатов. Главной целью и критерием эффективности систем пакетной

обработки является максимальная пропускная способность, т. е. решение максимального числа задач за единицу времени.

Для достижения этой цели в системах пакетной обработки используется следующая схема функционирования: в начале работы формируется пакет заданий, каждое задание содержит требование к системным ресурсам; из этого пакета заданий формируется мультипрограммная смесь, т. е. множество одновременно выполняемых задач. Для одновременного выполнения выбирают 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, т. е. выбирается «выгодное» задание. Следовательно, в вычислительных системах, работающих под управлением пакетных ОС, невозможно гарантировать выполнение того или иного задания в течение определенного периода времени.

Рассмотрим более детально совмещение во времени операций ввода-вывода и вычислений.

Такое совмещение может достигаться разными способами. Один из них характерен для компьютеров, имеющих специализированный процессор ввода-вывода. В компьютерах класса мэйнфреймов такие процессоры называют каналами. Обычно канал имеет систему команд, отличающуюся от системы команд центрального процессора. Эти команды специально предназначены для управления внешними устройствами, например, «проверить состояние устройства», «установить магнитную головку», «установить начало листа», «напечатать строку». Канальные программы могут храниться в той же оперативной памяти, что и программы центрального процессора. В системе команд центрального процессора предусматривается специальная инструкция, с помощью которой каналу передаются параметры и указания на то, какую программу ввода-вывода он должен выполнить. С этого момента центральный процессор и канал могут работать параллельно (рис. 4.1, *а*).

Другой способ совмещения вычислений с операциями ввода-вывода реализуется в компьютерах, в которых внешние устройства управляются не процессором ввода-вывода, а контроллерами. Каждое внешнее устройство или группа внешних устройств одного типа имеет свой собственный контроллер, который автономно отрабатывает команды, поступающие от центрального процессора. При этом контроллер и центральный процессор работают асинхронно. Поскольку многие внешние устройства включают электромеханические узлы, контроллер выполняет свои команды управле-

ния устройствами существенно медленнее, чем центральный процессор свои. Это обстоятельство используется для организации параллельного выполнения вычислений и операций ввода-вывода. В промежутке между передачей команд контроллеру центральный процессор может выполнять вычисления (рис. 4.1, *б*). Контроллер может сообщить центральному процессору о том, что он готов принять следующую команду, сигналом прерывания, либо центральный процессор узнает об этом, периодически опрашивая состояние контроллера.

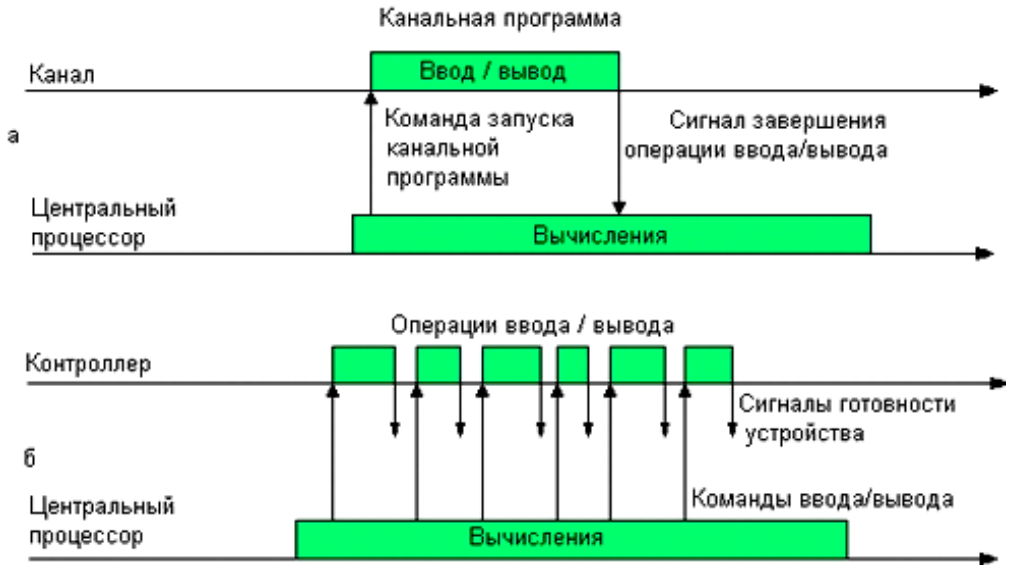



Рис. 4.1. Параллельное выполнение вычислений и операций ввода-вывода

Максимальный эффект ускорения достигается при наиболее полном перекрытии вычислений и ввода-вывода. Рассмотрим случай, когда процессор выполняет только одну задачу. В этой ситуации степень ускорения зависит от природы данной задачи и от того, насколько тщательно был выявлен возможный параллелизм при ее программировании. В задачах, где преобладают либо вычисления, либо ввод-вывод, ускорение почти отсутствует. Параллелизм в рамках одной задачи невозможен также, когда для продолжения вычислений необходимо полное завершение операции ввода-вывода, например, когда дальнейшие вычисления зависят от вводимых данных. В таких случаях неизбежны простои центрального процессора или канала.

Если в системе выполняется одновременно несколько задач, появляется возможность совмещения вычислений одной задачи с вводом-выводом другой. Пока одна задача ожидает какого-либо события (таким событием в мультипрограммной системе может быть не только завершение ввода-вывода, но и, например, наступление определенного момента времени, разблокирование файла или загрузка с диска недостающей стра-

ницы программы), процессор не простаивает, как это происходит при последовательном выполнении программ, а выполняет другую задачу.

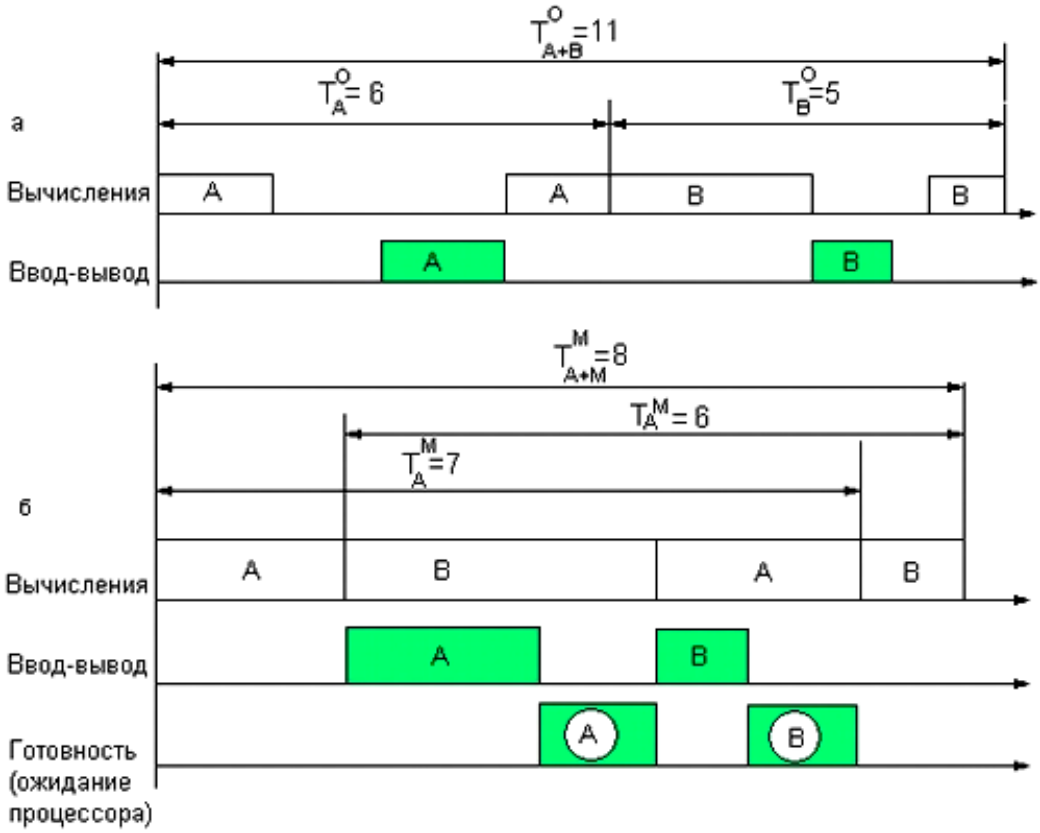



Рис. 4.2: *а* – время выполнения двух задач: в однопрограммной системе, *б* – в мультипрограммной системе

Общее время выполнения смеси задач часто оказывается меньше, чем их суммарное время последовательного выполнения (рис. 4.2, *а*). Однако выполнение отдельной задачи в мультипрограммном режиме может занять больше времени, чем при монопольном выделении процессора этой задаче. При совместном использовании процессора в системе могут возникать ситуации, когда задача готова выполняться, но процессор занят выполнением другой задачи. В таких случаях задача, завершившая ввод-вывод, вынуждена ждать освобождения процессора, и это удлиняет срок ее выполнения. Так, из рис. 4.2 видно, что в однопрограммном режиме задача А выполняется за 6 единиц времени, а в мультипрограммном – за 7. Задача В также вместо 5 единиц времени выполняется за 6. Но зато время выполнения обеих задач в мультипрограммном режиме составляет всего 8 единиц, что на 3 единицы меньше, чем при последовательном выполнении.

В системах пакетной обработки переключение процессора с выполнения одной задачи на выполнение другой происходит по инициативе са-

мой активной задачи, например, когда она отказывается от процессора из-за необходимости выполнить операцию ввода-вывода. Поэтому существует высокая вероятность того, что одна задача может надолго занять процессор и выполнение интерактивных задач станет невозможным. Взаимодействие пользователя с вычислительной машиной, на которой установлена система пакетной обработки, сводится к тому, что он приносит задание, отдает его диспетчеру-оператору, а в конце дня после выполнения всего пакета заданий получает результат. Очевидно, что такой порядок повышает эффективность функционирования аппаратуры, но снижает эффективность работы пользователя.

### 4.1.2. Мультипрограммирование в системах разделения времени

Повышение удобства и эффективности работы пользователя является целью другого способа мультипрограммирования – разделения времени. В *системах разделения времени* пользователю или пользователям предоставляется возможность интерактивной работы сразу с несколькими приложениями. Для этого каждое приложение должно регулярно получать возможность «общения» с пользователем. Понятно, что в пакетных системах возможности диалога пользователя с приложением весьма ограниченны. В системах разделения времени данная проблема решается за счет того, что 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 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.

Системы разделения времени обладают меньшей пропускной способностью, чем системы пакетной обработки, так как на выполнение принимается каждая запущенная пользователем задача, а не та, которая являет-

ся приоритетной. 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.

### 4.1.3. Мультипрограммирование в системах реального времени

Еще одна разновидность мультипрограммирования используется в *системах реального времени,* предназначенных для управления от компьютера различными техническими объектами (например, станком, спутником, научной экспериментальной установкой и т. п.) или технологическими процессами (например, гальванической линией, доменным процессом и т. п.). Во всех этих случаях существует предельно допустимое время, в течение которого должна быть выполнена управляющая объектом программа. В противном случае может произойти авария: спутник выйдет из зоны видимости, экспериментальные данные, поступающие с датчиков, будут потеряны, толщина гальванического покрытия не будет соответствовать норме. Таким образом, критерием эффективности является способность выдерживать заранее заданные интервалы времени между запуском программы и получением результата. Это время называется временем реакции системы, а соответствующее свойство системы – реактивностью. Требования ко времени реакции зависят от специфики управляемого процесса. Контроллер робота может требовать от встроенного компьютера ответ в течение менее 1 мс, в то время как при моделировании полета может быть приемлем ответ в течение 40 мс.

В системах реального времени мультипрограммная смесь представляет собой фиксированный набор заранее разработанных программ, а выбор программы на выполнение осуществляется по прерываниям исходя из текущего состояния объекта 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. Ко многим ситуациям в системах управления неприменимы статистические аргументы о малой вероятности возникновения пиковой нагрузки, основанные на том, что вероятность одновременного возникновения большого количества независимых событий невелика. Например, в системе управления атомной электростанцией в случае возникновения крупной аварии атомного реактора многие аварийные датчики сработают одновременно и создадут коррелированную нагрузку. 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.

## 4.2. Мультипроцессорная обработка

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

Концепция мультипроцессирования не нова, она известна с 70-х гг. XX в., но до середины 80-х гг. XX в. доступных многопроцессорных систем не существовало. Однако к настоящему времени стало обычным включение нескольких процессоров в архитектуру даже персонального компьютера. Более того, многопроцессорность теперь является одним из необходимых требований, которые предъявляются к компьютерам, используемым в качестве центрального сервера более-менее крупной сети.

Не следует путать *мультипроцессорную* обработку с *мультипрограммной*. 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. В этом случае несколько задач выполняется попеременно на одном процессоре, создавая лишь видимость параллельного выполнения. А в мультипроцессорных системах *несколько задач* выполняется 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. Так, например, возникает необходимость планировать процессы не для одного, а для нескольких процессоров, что гораздо сложнее. Сложность заключается 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.

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 *масштабируемой 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. Таким образом, 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 только в системах с симметричной мультипроцессорной архитектурой. 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 потребности в них 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.

## 4.3. Планирование процессов и потоков

Одной из основных подсистем мультипрограммной ОС, непосредственно влияющей на функционирование вычислительной машины, является подсистема управления процессами и потоками, которая занимается их созданием и уничтожением, поддерживает взаимодействие между ними, а также распределяет процессорное время между несколькими одновременно существующими в системе процессами и потоками.

Подсистема управления процессами и потоками ответственна за *обеспечение процессов необходимыми ресурсами*. В памяти ОС поддерживаются специальные информационные структуры, записывающие, какие ресурсы выделены каждому процессу. Ресурсы могут быть назначены процессу в единоличное пользование или совместно с другими процессами. Некоторые из ресурсов выделяются процессу при его создании, а некоторые – динамически по запросам во время выполнения. Ресурсы могут быть приписаны процессу на все время его жизни или только на определенный период. При выполнении этих функций подсистема управления процессами взаимодействует с другими подсистемами ОС, ответственными за управление ресурсами, такими как подсистема управления памятью, подсистема ввода-вывода, файловая система.

Когда в системе одновременно выполняется несколько независимых задач, то возникают дополнительные проблемы. Хотя потоки возникают и выполняются асинхронно, у них может возникнуть необходимость во взаимодействии, например, при обмене данными. Согласование скоростей потоков также очень важно для предотвращения эффекта «гонок» (когда несколько потоков пытаются изменить один и тот же файл), взаимных блокировок или других коллизий, которые возникают при совместном использовании ресурсов. *Синхронизация* потоков является одной из важных функций подсистемы управления процессами и потоками.

Каждый раз, когда процесс завершается, ОС предпринимает шаги, чтобы ликвидировать последствия его пребывания в системе. Подсистема управления процессами закрывает все файлы, с которыми работал процесс, освобождает области оперативной памяти, отведенные под коды, данные и системные информационные структуры процесса. Выполняется коррекция всевозможных очередей ОС и списков ресурсов, в которых имелись ссылки на завершаемый процесс.

### 4.3.1. Понятия «процесс» и «поток»

Чтобы поддерживать мультипрограммирование, ОС должна определить и оформить для себя те внутренние единицы работы, между ко-

торыми 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 мелких работ, для обозначения которых используют термины *«поток»,* или *«нить»*.

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, предоставить доступ к устройствам ввода-вывода, например, к последовательному порту для получения данных по подключенному к этому порту модему, и т. д. В ходе выполнения задачи 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 получили свое название благодаря тому, что они представляют собой последовательности (потоки выполнения) команд.

В простейшем случае процесс состоит из одного потока; именно таким образом трактовалось понятие «процесс» до середины 80-х гг. XX в. (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», т. е. 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 и 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.

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, где отсутствует понятие потока, возникают проблемы при организации параллельных вычислений в рамках процесса. 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 использо-

вание 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 данном случае все 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 выделяется собственное виртуальное адресное пространство, физическая память, закрепляются устройства ввода-вывода и т. п.

Из 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. Процессорное время распределяется между потоками операционной системой. Процессу ОС назначает адресное пространство и набор ресурсов, которые совместно используются всеми его потоками.

Создание потоков требует от ОС меньших накладных расходов, чем создание процессов. В отличие от процессов, которые принадлежат разным 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 нескольких 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 создавать как отдельные процессы. 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 важно не столько параллельное выполнение, сколько ясность программы.

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

### 4.3.2. Создание процессов и потоков

Создать процесс прежде всего означает создать *описатель процесса*, в качестве которого выступает одна или несколько информационных структур, содержащих все сведения о процессе, необходимые операцион-

ной 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 знаменует собой 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 каждого процесса ОС порождает 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 выбора 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. 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.

### 4.3.3. Планирование и диспетчеризация потоков

На протяжении существования процесса выполнение его потоков может быть многократно прервано и продолжено. Переход от выполнения одного потока к другому осуществляется в результате планирования и диспетчеризации. Работа по определению того, в какой момент необходимо прервать выполнение текущего активного потока и какому потоку предоставить возможность выполняться, называется *планированием.* Планирование потоков осуществляется на основе информации, хранящейся в описателях процессов и потоков. При планировании могут приниматься во внимание такие факторы, как приоритет потоков, время их ожидания в очереди, накопленное время выполнения, интенсивность обращений к вводу-выводу и др. Выполнение потоков планируется независимо от того, принадлежат ли они одному или разным процессам. Так, например, после выполнения потока некоторого процесса ОС может выбрать для выполнения другой поток того же процесса или же назначить к выполнению поток другого процесса.

Планирование потоков, по сути, включает в себя решение двух задач:

- определение момента времени для смены текущего активного потока;
- выбор потока из очереди готовых потоков.

Существует множество различных алгоритмов планирования потоков, по-своему решающих каждую из приведенных выше задач. Алгоритмы планирования могут преследовать различные цели и обеспечивать разное качество мультипрограммирования. Например, в одном случае выбирается такой алгоритм планирования, при котором гарантируется, что ни один поток / процесс не будет занимать процессор дольше определенного времени, в другом случае целью является максимально быстрое выполнение «коротких» задач, а в третьем случае преимущественное право занять процессор

получают потоки интерактивных приложений. Именно особенности реализации планирования потоков в наибольшей степени определяют специфику операционной системы, в частности, является ли она системой пакетной обработки, системой разделения времени или системой реального времени.

В большинстве операционных систем универсального назначения планирование осуществляется *динамически* (on-line), т. е. решения принимаются во время работы системы на основе анализа текущей ситуации. Операционная система работает в условиях неопределенности: потоки и процессы появляются в случайные моменты времени и так же непредсказуемо завершаются. Динамические планировщики могут гибко приспосабливаться к изменяющейся ситуации и не используют никаких предположений о мультипрограммной смеси. Для того чтобы оперативно найти в условиях такой неопределенности оптимальный в некотором смысле порядок выполнения задач, операционная система должна затрачивать значительные усилия.

Другой тип планирования – статический – может быть использован в специализированных системах, в которых весь набор одновременно выполняемых задач определен заранее, например, в системах реального времени. *Планировщик* называется *статическим* или *предварительным*, если он принимает решения о планировании не во время работы системы, а заранее (off-line). Соотношение между динамическим и статическим планировщиками аналогично соотношению между диспетчером железной дороги, который пропускает поезда строго по предварительно составленному расписанию, и регулировщиком на перекрестке автомобильных дорог, не оснащенном светофорами, который решает, какую машину остановить, а какую пропустить, в зависимости от ситуации на перекрестке.

Результатом работы статического планировщика является таблица, называемая расписанием, в которой указывается, какому потоку / процессу, когда и на какое время должен быть предоставлен процессор. Для построения расписания планировщику нужны как можно более полные предварительные сведения о характеристиках набора задач, например, о максимальном времени выполнения каждой задачи, ограничениях предшествования, ограничениях по взаимному исключению, предельным срокам и т. д.

После того как расписание готово, оно может использоваться операционной системой для переключения потоков и процессов. При этом накладные расходы ОС на исполнение расписания оказываются значительно меньшими, чем при динамическом планировании, и сводятся лишь к диспетчеризации потоков / процессов.

*Диспетчеризация* заключается в реализации найденного в результате планирования (динамического или статистического) решения, т. е. в пере-

ключении процессора с одного потока на другой. Прежде чем прервать выполнение потока, ОС запоминает его контекст, чтобы впоследствии использовать эту информацию для возобновления выполнения данного потока. Контекст отражает, во-первых, состояние аппаратуры компьютера в момент прерывания потока: значение счетчика команд, содержимое регистров общего назначения, режим работы процессора, флаги, маски прерываний и другие параметры. Во-вторых, контекст включает параметры операционной среды, а именно: ссылки на открытые файлы, данные о незавершенных операциях ввода-вывода, коды ошибок выполняемых данным потоком системных вызовов и т. д.

Диспетчеризация сводится к следующим операциям:

– сохранение контекста текущего потока, который требуется сменить;

– загрузка контекста нового потока, выбранного в результате планирования;

– запуск нового потока на выполнение.

Поскольку операция переключения контекстов существенно влияет на производительность вычислительной системы, программные модули ОС выполняют диспетчеризацию потоков совместно с аппаратными средствами процессора.

В контексте потока можно выделить часть, общую для всех потоков данного процесса (ссылки на открытые файлы), и часть, относящуюся только к данному потоку (содержимое регистров, счетчик команд, режим процессора). Например, в среде NetWare 4.x различаются три вида контекстов: глобальный контекст (контекст процесса), контекст группы потоков и контекст отдельного потока. Соотношение между данными этих контекстов напоминает соотношение глобальных и локальных переменных в программе, написанной на языке С. Переменные глобального контекста доступны для всех потоков, созданных в рамках одного процесса. Переменные локального контекста доступны только для кодов определенного потока так же, как и локальные переменные функции. В NetWare можно создавать несколько групп потоков внутри одного процесса, и эти группы будут иметь свой групповой контекст. Переменные, принадлежащие групповому контексту, доступны всем потокам, входящим в группу, но недоступны остальным потокам.

Очевидно, что такая иерархическая организация контекстов ускоряет переключение потоков, так как при переключении с потока на поток в пределах одной группы нет необходимости заменять контексты групп или глобальные контексты – достаточно лишь заменить контексты потоков, имеющие меньший объем. Таким же образом при переключении с потока

одной группы на поток другой группы в пределах одного процесса изменяется не глобальный контекст, а лишь контекст группы. Переключение глобальных контекстов происходит только при переходе с потока одного процесса на поток другого.

### 4.3.4. Состояния потока

ОС выполняет планирование потоков, принимая во внимание их состояние. В мультипрограммной системе поток может находиться в одном из трех основных состояний:

– выполнение – активное состояние потока, во время которого поток обладает всеми необходимыми ресурсами и непосредственно выполняется процессором;

– ожидание – пассивное состояние потока, находясь в котором поток заблокирован по своим внутренним причинам (ждет осуществления некоторого события, например завершения операции ввода-вывода, получения сообщения от другого потока или освобождения какого-либо необходимого ему ресурса);

– готовность – также пассивное состояние потока, но в этом случае поток заблокирован в связи с внешним по отношению к нему обстоятельством (имеет все требуемые для него ресурсы, готов выполняться, однако процессор занят выполнением другого потока).

В течение времени своего выполнения каждый поток переходит из одного состояния в другое в соответствии с алгоритмом планирования потоков, принятым в данной операционной системе (рис. 4.3).

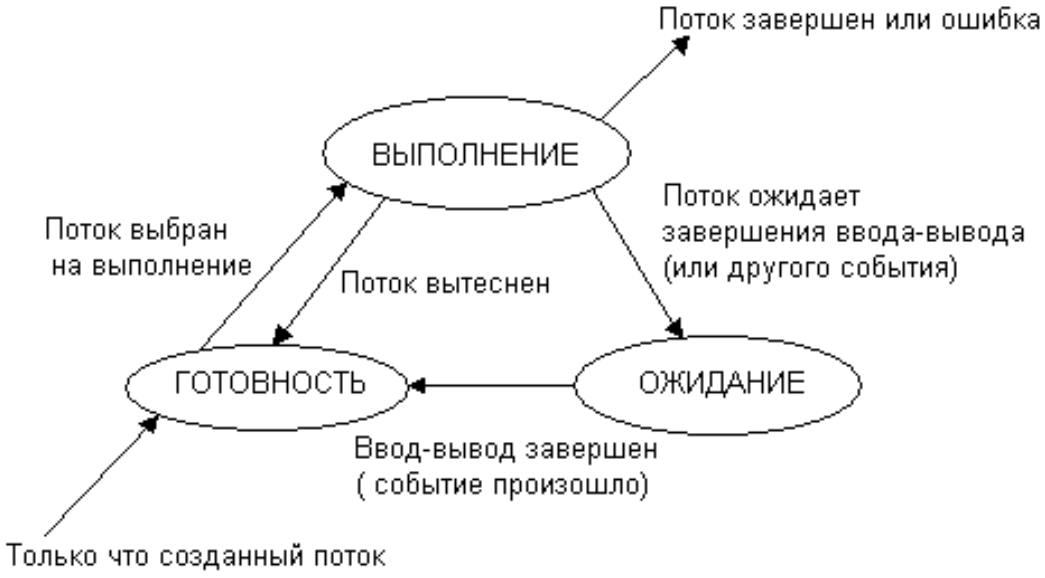



Рис. 4.3. Граф состояний потока в многозадачной среде

### 4.3.5. Вытесняющие и невытесняющие алгоритмы планирования

С общих позиций все множество алгоритмов планирования можно разделить на два класса: вытесняющие и невытесняющие алгоритмы планирования.

– невытесняющие (non-preemptive) алгоритмы основаны на том, что активному потоку позволяется выполняться, пока он по собственной инициативе не отдаст управление операционной системе, чтобы та выбрала из очереди другой готовый к выполнению поток;

– вытесняющие (preemptive) алгоритмы – это такие способы планирования потоков, в которых решение о переключении процессора с выполнения одного потока на выполнение другого потока принимается операционной системой, а не активной задачей.

Основным различием между вытесняющими и невытесняющими алгоритмами является степень централизации механизма планирования потоков. При вытесняющем мультипрограммировании функции планирования потоков целиком сосредоточены в операционной системе и программист пишет свое приложение, не заботясь о том, что оно будет выполняться одновременно с другими задачами. При этом операционная система выполняет следующие функции: определяет момент снятия с выполнения активного потока, запоминает его контекст, выбирает из очереди готовых потоков следующий, запускает новый поток на выполнение, загружая его контекст.

При невытесняющем мультипрограммировании механизм планирования распределен между операционной системой и прикладными программами. Прикладная программа, получив управление от операционной системы, сама определяет момент завершения очередного цикла своего выполнения и только затем передает управление ОС с помощью системного вызова. ОС формирует очереди потоков и выбирает в соответствии с некоторым правилом (например, с учетом приоритетов) следующий поток на выполнение.

Такой механизм создает проблемы как для пользователей, так и для разработчиков приложений. Для пользователей это означает, что управление системой теряется на произвольный период времени, который определяется приложением, а не пользователем. Если приложение тратит слишком много времени на выполнение какой-либо работы, например, на форматирование диска, пользователь не может переключиться с этой задачи на другую задачу, например, на текстовый редактор, в то время как форматирование продолжалось бы в фоновом режиме.

Поэтому разработчики приложений для операционной среды с невытесняющей многозадачностью вынуждены, возлагая на себя часть функций

планировщика, создавать приложения так, чтобы они выполняли свои задачи небольшими частями. Например, программа форматирования может отформатировать одну дорожку дискеты и вернуть управление системе. После выполнения других задач система возвратит управление программе форматирования, чтобы та отформатировала следующую дорожку. Подобный метод разделения времени между задачами работает, но он существенно затрудняет разработку программ и предъявляет повышенные требования к квалификации программиста. Программист должен обеспечить «дружественное» отношение своей программы к другим выполняемым одновременно с ней программам. Для этого в программе должны быть предусмотрены частые передачи управления операционной системе. Крайним проявлением «не дружественности» приложения является его зависание, которое приводит к общему краху системы. В системах с вытесняющей многозадачностью такие ситуации, как правило, исключены, так как центральный планирующий механизм имеет возможность снять зависшую задачу с выполнения.

Однако распределение функций планирования потоков между системой и приложениями не всегда является недостатком, а при определенных условиях может быть и преимуществом, потому что предоставляет разработчику приложений возможность самому проектировать алгоритм планирования, наиболее подходящий для данного фиксированного набора задач. Так как разработчик сам определяет в программе момент возвращения управления, при этом исключаются нерациональные прерывания программ в «неудобные» для них моменты времени. Кроме того, легко разрешаются проблемы совместного использования данных: задача во время каждого цикла выполнения использует их монопольно и уверена, что на протяжении этого периода никто другой не изменит данные. Существенным преимуществом невытесняющего планирования является более высокая скорость переключения с потока на поток.

Почти во всех современных операционных системах, ориентированных на высокопроизводительное выполнение приложений (UNIX, Windows NT / 2000 / XP / Vista, OS / 2, VAX / VMS), реализованы вытесняющие алгоритмы планирования потоков (процессов).

### 4.3.6. Алгоритмы планирования, основанные на квантовании

В основе многих вытесняющих алгоритмов планирования лежит концепция *квантования.* В соответствии с этой концепцией, каждому потоку для выполнения поочередно предоставляется ограниченный непрерывный период процессорного времени – *квант.* Смена активного потока происходит, если:

– поток завершился и покинул систему;

– произошла ошибка;
– поток перешел в состояние ожидания;
– исчерпан квант процессорного времени, отведенный данному потоку.

Поток, исчерпавший свой квант, переводится в состояние готовности и ожидает, когда ему будет предоставлен новый квант процессорного времени, а на выполнение в соответствии с определенным правилом выбирается новый поток из очереди готовых (рис. 4.4).

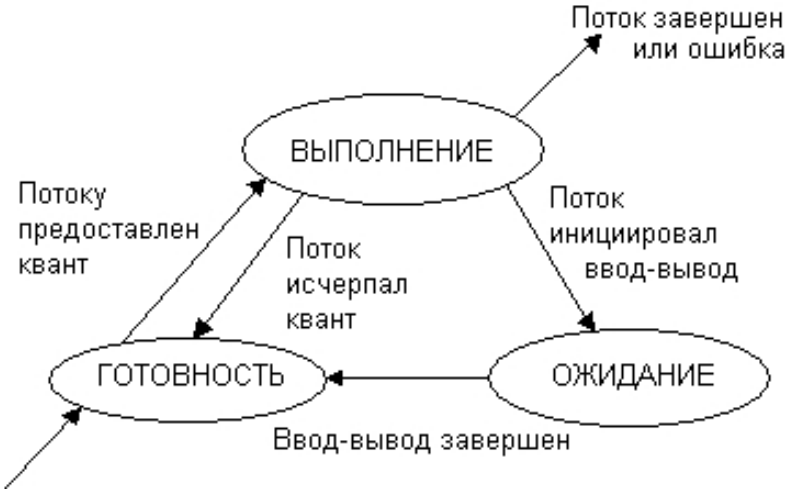



Рис. 4.4. Граф состояний потока в системе с квантованием

Кванты, выделяемые потокам, могут быть одинаковыми для всех потоков или различными.

Чем больше квант, тем выше вероятность того, что потоки завершатся в результате первого же цикла выполнения, и тем менее явной становится зависимость времени ожидания потоков от времени выполнения. При достаточно большом кванте алгоритм квантования вырождается в алгоритм последовательной обработки, присущий однопрограммным системам, при котором время ожидания задачи в очереди вообще никак не зависит от ее длительности.

Кванты, выделяемые одному потоку, могут иметь фиксированную величину, а могут изменяться в разные периоды жизни потока. Пусть, например, первоначально каждому потоку назначается достаточно большой квант, а величина каждого следующего кванта уменьшается до некоторой заранее заданной величины. В таком случае преимущество получают короткие задачи, которые успевают выполняться в течение первого кванта, а длительные вычисления будут проводиться в фоновом режиме. Можно представить себе алгоритм планирования, в котором каждый следующий квант, выделяемый определенному потоку, больше предыдущего. Такой подход

позволяет уменьшить накладные расходы на переключение задач в том случае, когда сразу несколько задач выполняют длительные вычисления.

Потоки получают для выполнения квант времени, но некоторые из них используют его не полностью, например, из-за необходимости выполнить ввод или вывод данных. В результате возникает ситуация, когда потоки с интенсивными обращениями к вводу-выводу используют только небольшую часть выделенного им процессорного времени. Алгоритм планирования может исправить эту ситуацию. В качестве компенсации за неиспользованные полностью кванты потоки получают привилегии при последующем обслуживании. Для этого планировщик создает две очереди готовых потоков. Очередь 1 образована потоками, которые пришли в состояние готовности в результате исчерпания кванта времени, а очередь 2 – потоками, у которых завершилась операция ввода-вывода. При выборе потока для выполнения прежде всего просматривается вторая очередь, и только если она пуста, квант выделяется потоку из первой очереди.

Многозадачные ОС теряют некоторое количество процессорного времени для выполнения вспомогательных работ во время переключения контекстов задач. При этом запоминаются и восстанавливаются регистры, флаги и указатели стека, а также проверяется статус задач для передачи управления. Затраты на эти вспомогательные действия не зависят от величины кванта времени, поэтому чем больше квант, тем меньше суммарные накладные расходы, связанные с переключением потоков.

### 4.3.7. Алгоритмы планирования, основанные на приоритетах

Другой важной концепцией, лежащей в основе многих вытесняющих алгоритмов планирования, является *приоритетное обслуживание.* Приоритетное обслуживание предполагает наличие у потоков некоторой изначально известной характеристики – приоритета, – на основании которой определяется порядок их выполнения. *Приоритет* – это число, характеризующее степень привилегированности потока при использовании ресурсов вычислительной машины, в частности, процессорного времени: чем выше приоритет, тем выше привилегии, тем меньше времени будет проводить поток в очередях.

Приоритет может выражаться целым или дробным, положительным или отрицательным значением. В некоторых ОС принято, что приоритет потока тем выше, чем больше (в арифметическом смысле) число, обозначающее приоритет. В других системах, наоборот, чем меньше число, тем выше приоритет.

В большинстве операционных систем, поддерживающих потоки, приоритет потока непосредственно связан с приоритетом процесса, в рам-

ках которого выполняется данный поток. Приоритет процесса назначается операционной системой при его создании. Значение приоритета включается в описатель процесса и используется при назначении приоритета потокам этого процесса. При назначении приоритета вновь созданному процессу ОС учитывает, является ли этот процесс системным или прикладным, каков статус пользователя, запустившего процесс, было ли явное указание пользователя на присвоение процессу определенного уровня приоритета. Поток может быть инициирован не только по команде пользователя, но и в результате выполнения системного вызова другим потоком. В этом случае при назначении приоритета новому потоку ОС должна принимать во внимание значение параметров системного вызова.

Во многих ОС предусматривается возможность изменения приоритетов в течение жизни потока. Изменение приоритета могут происходить по инициативе самого потока, когда он обращается с соответствующим вызовом к операционной системе, или по инициативе пользователя, когда он выполняет соответствующую команду. Кроме того, ОС сама может изменять приоритеты потоков в зависимости от ситуации, складывающейся в системе. В последнем случае приоритеты называются *динамическими* в отличие от неизменяемых, *фиксированных* приоритетов,

От того, какие приоритеты назначены потокам, существенно зависит эффективность работы всей вычислительной системы. В современных ОС во избежание разбалансировки системы, которая может возникнуть при неправильном назначении приоритетов, возможности пользователей влиять на приоритеты процессов и потоков стараются ограничивать. При этом обычные пользователи, как правило, не имеют права повышать приоритеты своим потокам, это разрешено делать (да и то в определенных пределах) только администраторам. В большинстве же случаев ОС присваивает приоритеты потокам по умолчанию.

В качестве примера рассмотрим схему назначения приоритетов потокам, принятую в операционной системе Windows NT (рис. 4.5). В системе определено 32 уровня приоритетов и два класса потоков – потоки реального времени и потоки с переменными приоритетами. Диапазон от 1 до 15 включительно отведен для потоков с переменными приоритетами, а от 16 до 31 – для более критичных ко времени потоков реального времени (приоритет 0 зарезервирован для системных целей).

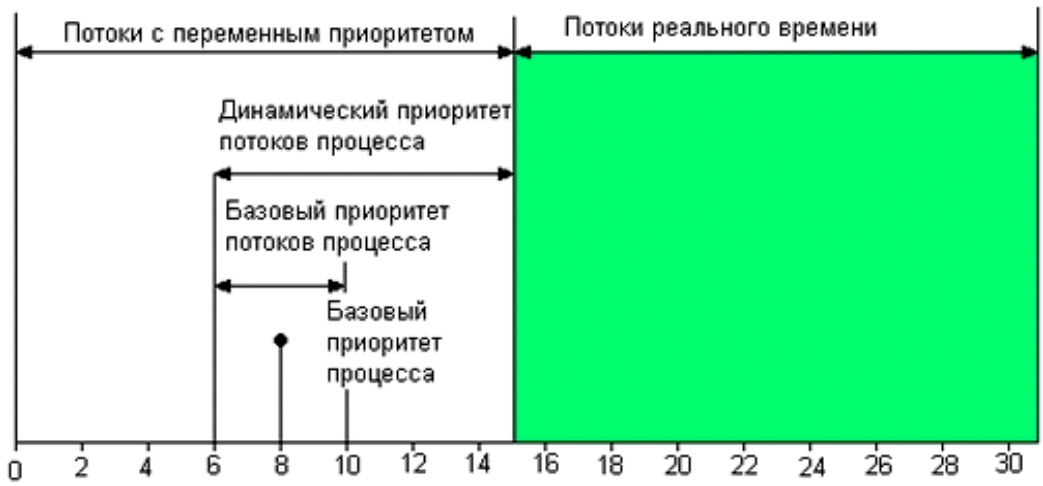



Рис. 4.5. Схема назначения приоритетов в Windows NT

При создании процесс в зависимости от класса получает по умолчанию базовый приоритет в верхней или нижней части диапазона. Базовый приоритет процесса в дальнейшем может быть повышен или понижен операционной системой. Первоначально поток получает значение базового приоритета из диапазона базового приоритета процесса, в котором он был создан. Пусть, например, значение базового приоритета некоторого процесса равно К. Тогда все потоки данного процесса получат базовые приоритеты из диапазона [К–2, К+2]. Отсюда видно, что, изменяя базовый приоритет процесса, ОС может влиять на базовые приоритеты его потоков.

В Windows NT с течением времени приоритет потока, относящегося к классу потоков с переменными приоритетами, может отклоняться от базового приоритета потока, причем эти изменения могут быть не связаны с изменениями базового приоритета процесса. В тех случаях, когда поток не полностью использовал отведенный ему квант, ОС может повышать приоритет потока (который в этом случае называется динамическим), или понижать приоритет, если квант был использован полностью. В зависимости от того, какого типа событие помешало потоку полностью использовать квант, ОС дифференцированно наращивает приоритет. В частности, ОС в большей степени повышает приоритет потокам, которые ожидают ввода с клавиатуры (интерактивным приложениям) и в меньшей степени – потокам, выполняющим дисковые операции. Именно на основе динамических приоритетов осуществляется планирование потоков. Начальной точкой отсчета для динамического приоритета является значение базового приоритета потока. Значение динамического приоритета потока ограничено снизу его базовым приоритетом, верхней же границей является нижняя граница диапазона приоритетов реального времени.

Существуют две разновидности приоритетного планирования: обслуживание с относительными приоритетами и обслуживание с абсолютными приоритетами.

В обоих случаях выбор потока на выполнение из очереди готовых осуществляется одинаково: выбирается поток, имеющий наивысший приоритет. Однако проблема определения момента смены активного потока решается по-разному. В системах с относительными приоритетами активный поток выполняется до тех пор, пока он сам не покинет процессор, перейдя в состояние ожидания (или же произойдет ошибка, или поток завершится). На рис. 4.6, *а* показан граф состояний потока в системе с относительными приоритетами.

В системах с абсолютными приоритетами выполнение активного потока прерывается кроме указанных выше причин, еще при одном условии: если в очереди готовых потоков появился поток, приоритет которого выше приоритета активного потока. В этом случае прерванный поток переходит в состояние готовности (рис. 4.6, *б*).


Рис. 4.6. Графы состояний потоков
*а* – в системах с относительными приоритетами;
*б* – в системах с абсолютными приоритетами

В системах, в которых планирование осуществляется на основе относительных приоритетов, минимизируются затраты на переключения процессора с одной работы на другую. С другой стороны, здесь могут возникать ситуации, когда одна задача занимает процессор слишком долго. Для систем разделения времени и реального времени такая дисциплина обслуживания не подходит: интерактивное приложение может ждать своей очереди часами, пока вычислительной задаче не потребуется ввод-вывод, зато в системах пакетной обработки, в т. ч. известной ОС OS / 360, относительные приоритеты используются широко.

В системах с абсолютными приоритетами время ожидания потока в очередях может быть сведено к минимуму, если ему назначить самый высокий приоритет. Такой поток будет вытеснять из процессора все остальные потоки, кроме потоков, имеющих такой же приоритет. Это делает планирование на основе абсолютных приоритетов подходящим для систем управления объектами, в которых важна быстрая реакция на событие.

### 4.3.8. Смешанные алгоритмы планирования

Во многих операционных системах алгоритмы планирования построены с использованием как концепции квантования, так и приоритетов. Например, в основе планирования может лежать квантование, но величина кванта и / или порядок выбора потока из очереди готовых определяется приоритетами потоков. Именно так реализовано планирование в системе Windows NT, где квантование сочетается с динамическими абсолютными приоритетами. Для выполнения выбирается готовый поток с наивысшим приоритетом. Ему выделяется квант времени. Если во время выполнения в очереди готовых появляется поток с более высоким приоритетом, то он вытесняет выполняемый поток. Вытесненный поток возвращается в очередь готовых, причем он становится впереди всех остальных потоков, имеющих такой же приоритет.

Рассмотрим более подробно алгоритм планирования в операционной системе UNIX System V Release 4. В этой ОС понятие «поток» отсутствует, и планирование осуществляется на уровне процессов. В системе UNIX System V Release 4 реализована вытесняющая многозадачность, основанная на использовании приоритетов и квантования.

Каждый процесс в зависимости от решаемой задачи относится к одному из трех определенных в системе приоритетных классов: классу реального времени, классу системных процессов или классу процессов разделения времени. Назначение и обработка приоритетов выполняется для разных классов по-разному. Процессы системного класса, зарезервирован-

ные для ядра, используют стратегию фиксированных приоритетов. Уровень приоритета процессу назначается ядром и никогда не изменяется.

Процессы реального времени также используют стратегию фиксированных приоритетов, но пользователь может их изменять. Так как при наличии готовых к выполнению процессов реального времени другие процессы не рассматриваются, то процессы реального времени надо тщательно проектировать, чтобы они не захватывали процессор на слишком долгое время. Характеристики планирования процессов реального времени включают две величины: уровень глобального приоритета и квант времени. Для каждого уровня приоритета по умолчанию имеется своя величина кванта времени. Процессу разрешается захватывать процессор на указанный квант времени, а по его истечении планировщик снимает процесс с выполнения.

Процессы разделения времени до появления UNIX System V Release 4 были единственным классом процессов, и по умолчанию UNIX System V Release 4 назначает новому процессу именно этот класс. Состав класса процессов разделения времени наиболее неопределенный и часто меняющийся в отличие от системных процессов и процессов реального времени. Для справедливого распределения времени процессора между процессами в этом классе используется стратегия динамических приоритетов. Величина приоритета, назначаемого процессам разделения времени, вычисляется пропорционально значениям двух составляющих: пользовательской части и системной части. Пользовательская часть приоритета может быть изменена администратором и владельцем процесса, но в последнем случае только в сторону его снижения.

Системная составляющая позволяет планировщику управлять процессами в зависимости от того, как долго они занимают процессор, не уходя в состояние ожидания. У тех процессов, которые потребляют большие периоды процессорного времени без ухода в состояние ожидания, приоритет снижается, а у тех процессов, которые часто уходят в состояние ожидания после короткого периода использования процессора, приоритет повышается. Таким образом, процессам, использующим больше ресурсов, дается низкий приоритет. Это означает, что они реже выбираются для выполнения. Это ущемление в правах компенсируется тем, что процессам с низким приоритетом даются бóльшие кванты времени, чем процессам с высокими приоритетами. Таким образом, хотя низкоприоритетный процесс и не работает так часто, как высокоприоритетный, но зато, когда он наконец выбирается для выполнения, ему отводится больше времени.

Другой пример относится к операционной системе OS / 2. Планирование здесь основано на использовании квантования и абсолютных дина-

мических приоритетов. На множестве потоков определены приоритетные классы: критический (time critical), серверный (server), стандартный (regular) и остаточный (idle), – в каждом из которых имеется 32 приоритетных уровня. Потоки критического класса имеют наивысший приоритет. К этому классу могут быть отнесены, например, системные потоки, выполняющие задачи управления сетью. Следующий по приоритетности класс предназначен, как это следует из его названия, для потоков, обслуживающих серверные приложения. К стандартному классу могут быть отнесены потоки обычных приложений. Потоки, входящие в остаточный класс, имеют самый низкий приоритет. К этому классу относится, например, поток, выводящий на экран заставку, когда в системе не выполняется никакой работы.

Поток из менее приоритетного класса не может быть выбран для выполнения, пока в очереди более приоритетного класса имеется хотя бы один поток. Внутри каждого класса потоки выбираются также по приоритетам. Потоки, имеющие одинаковое значение приоритета, обслуживаются в циклическом порядке.

Приоритеты могут изменяться планировщиком в следующих случаях:

– если поток находится в ожидании процессорного времени дольше, чем это задано системной переменной MAXWAIT, то его уровень приоритета будет автоматически увеличен операционной системой. При этом результирующее значение приоритета не должно превышать нижней границы диапазона приоритетов критического класса;

– если поток ушел на выполнение операции ввода-вывода, то после ее завершения он получит наивысшее значение приоритета своего класса;

– приоритет потока автоматически повышается, когда он поступает на выполнение.

ОС динамически устанавливает величину кванта, отводимого потоку для выполнения. Величина кванта зависит от загрузки системы и интенсивности подкачки. Параметры настройки системы позволяют явно задать границы изменения кванта. В любом случае он не может быть меньше 32 мс и больше 65536 мс. Если поток был прерван до истечения кванта, то следующий выделенный ему интервал выполнения будет увеличен на время, равное одному периоду таймера (около 32 мс), и так до тех пор, пока квант не достигнет заранее заданного при настройке ОС предела.

Благодаря такому алгоритму планирования в OS / 2, ни один поток не будет «забыт» системой и получит достаточно процессорного времени.

### 4.3.9. Планирование в системах реального времени

В системах реального времени, в которых главным критерием эффективности является обеспечение временных характеристик вычислительного процесса, планирование имеет особое значение. Любая система реального времени должна реагировать на сигналы управляемого объекта в течение заданных временных ограничений. Необходимость тщательного планирования работ облегчается тем, что в системах реального времени весь набор выполняемых задач известен заранее. Кроме того, часто в системе имеется информация о временах выполнения задач, моментах активизации, предельных допустимых сроках ожидания ответа и т. д. Эти данные могут быть использованы планировщиком для создания статического расписания или для построения адекватного алгоритма динамического планирования.

При разработке алгоритмов планирования для систем реального времени необходимо учитывать, какие последствия в этих системах возникают при несоблюдении временных ограничений. Если эти последствия катастрофичны, как, например, для системы управления полетами или атомной электростанцией, то операционная система реального времени, на основе которой строится управление объектом, называется *жесткой* (*hard*). Если же последствия нарушения временных ограничений не столь серьезны, т. е. сравнимы с той пользой, которую приносит система управления объектом, то система является *мягкой* (*soft*) системой реального времени. Примером мягкой системы реального времени является система резервирования билетов. Если из-за временных нарушений оператору не удается зарезервировать билет, это не очень страшно – можно просто послать запрос на резервирование заново.

В жестких системах реального времени время завершения выполнения каждой из критических задач должно быть гарантировано для всех возможных сценариев работы системы. Такие гарантии могут быть даны либо в результате исчерпывающего тестирования всех возможных сценариев поведения управляемого объекта и управляющих программ, либо в результате построения статического расписания, либо в результате выбора математически обоснованного динамического алгоритма планирования. При построении расписания надо иметь в виду, что для некоторых наборов задач в принципе невозможно найти расписание, при котором бы удовлетворялись заданные временные характеристики. С целью определения возможности существования расписания могут быть использованы различные критерии. Например, простейшим критерием может служить условие, что разность между предельным сроком выполнения задачи (после

появления запроса на ее выполнение) и временем ее вычисления (при условии непрерывного выполнения) всегда должна быть положительной. Очевидно, что такой критерий является необходимым, но недостаточным. Точные критерии, гарантирующие наличие расписания, являются очень сложными в вычислительном отношении.

В мягких системах реального времени предполагается, что заданные временные ограничения могут иногда нарушаться, поэтому здесь обычно применяются менее затратные способы планирования.

В зависимости от характера возникновения запросов на выполнение задач полезно разделять их на два типа: *периодические* и *спорадические*. Начиная с момента первоначального запроса все будущие моменты запроса периодической задачи можно определить заранее путем прибавления к моменту начального запроса величины, кратной известному периоду. Времена запросов на выполнение спорадических задач заранее не известны.

При выборе алгоритма планирования следует учитывать данные о возможной зависимости задач. Эта зависимость может выступать, например, в виде ограничений на последовательность выполнения задач или их синхронизации, вызванной взаимными исключениями (запрет на выполнение некоторых задач в течение определенных периодов времени).

С практической точки зрения алгоритмы планирования зависимых задач более важны, чем алгоритмы планирования независимых задач. При наличии дешевых микроконтроллеров нет смысла организовывать мультипрограммное выполнение большого количества независимых задач на одном компьютере, так как при этом значительно возрастает сложность программного обеспечения. Обычно одновременно выполняющиеся задачи должны обмениваться информацией и получать доступ к общим данным для достижения общей цели системы, т. е. они являются зависимыми задачами. Поэтому существование некоторого предпочтения последовательности выполнения задач или взаимного исключения скорее норма для систем управления реального времени, чем исключение.

Проблема планирования зависимых задач очень сложна, нахождение ее оптимального решения требует больших вычислительных ресурсов, сравнимых с теми, которые требуются для собственно выполнения задач управления. Решение этой проблемы возможно за счет следующих мер:

– разделение проблемы планирования на две части, чтобы одна часть выполнялась заранее, перед запуском системы, а вторая, более простая часть, – во время работы системы;

– предварительный анализ набора задач с взаимными исключениями может состоять, например, в выявлении так называемых запрещенных об-

ластей времени, в течение которых нельзя назначать выполнение задач, содержащих критические секции;

– введение ограничивающих предположений о поведении набора задач.

При таком подходе планирование приближается к статическому.

Возвращаясь к планированию независимых задач, рассмотрим классический алгоритм для жестких систем реального времени с одним процессором, разработанный в 1973 году Лью (Liu) и Лейландом (Layland). Алгоритм является динамическим, т. е. использует вытесняющую многозадачность и основан на относительных статических (неизменяемых в течение жизни задачи) приоритетах.

Суть алгоритма состоит в том, что всем задачам назначаются статические приоритеты в соответствии с величиной периодов выполнения. Задача с самым коротким периодом получает наивысший приоритет, а задача с наибольшим периодом выполнения – наименьший. При соблюдении всех ограничений этот алгоритм гарантирует выполнение временных ограничений для всех задач во всех ситуациях.

Если периоды повторения задач кратны периоду выполнения самой короткой задачи, то требование к максимальному коэффициенту загрузки процессора смягчается и может достигать 1.

Существуют также алгоритмы с динамическим изменением приоритетов, которые назначаются в соответствии с такими текущими параметрами задачи как, например, *конечный срок выполнения (deadline)*. При необходимости назначения некоторой задачи на выполнение выбирается та, у которой текущее значение разницы между конечным сроком выполнения и временем, требуемым для ее непрерывного выполнения, является наименьшим.

## 4.4. Моменты перепланировки

Для реализации алгоритма планирования ОС должна получать управление всякий раз, когда в системе происходит событие, требующее перераспределения процессорного времени. К таким событиям могут быть отнесены следующие:

– прерывание от таймера, сигнализирующее, что время, отведенное активной задаче на выполнение, закончилось. Планировщик переводит задачу в состояние готовности и выполняет перепланирование;

– активная задача выполнила системный вызов, связанный с запросом на ввод-вывод или на доступ к ресурсу, который в настоящий момент занят (например, файл данных). Планировщик переводит задачу в состояние ожидания и выполняет перепланирование;

– активная задача выполнила системный вызов, связанный с освобождением ресурса. Планировщик проверяет, не ожидается ли этот ресурс какой-либо задачей. Если да, то эта задача переводится из состояния ожидания в состояние готовности. При этом возможно, что задача, получившая ресурс, имеет более высокий приоритет, чем текущая активная задача. После перепланирования задача с бóльшим приоритетом получает доступ к процессору, вытесняя текущую задачу;

– внешнее (аппаратное) прерывание 1, которое сигнализирует о завершении периферийным устройством операции ввода-вывода, переводит соответствующую задачу в очередь готовых, и выполняется планирование;

– внутреннее прерывание сигнализирует об ошибке, которая произошла в результате выполнения активной задачи. Планировщик снимает задачу и выполняет перепланирование.

При возникновении каждого из этих событий планировщик выполняет просмотр очередей и решает вопрос о том, какая задача будет выполняться следующей. Помимо указанных, существует и ряд других событий, часто связанных с системными вызовами и требующих перепланировки. Например, запросы приложений и пользователей на создание новой задачи или повышение приоритета уже существующей задачи создают новую ситуацию, которая требует пересмотра очередей и, возможно, переключения процессора.

На рис. 4.7 показан фрагмент временной диаграммы работы планировщика в системе, где одновременно выполняются четыре потока. В данном случае неважно, по какому правилу выбираются потоки на выполнение и каким образом изменяются их приоритеты. Существенное значение имеют лишь события, вызывающие активизацию планировщика.

Первые четыре цикла работы планировщика, приведенные на рисунке, были инициированы прерываниями от таймера по истечении квантов времени.

Следующая передача управления планировщику была осуществлена в результате выполнения потоком 3 системного запроса на ввод-вывод (событие I / O). Планировщик перевел этот поток в состояние ожидания, а затем переключил процессор на поток 2. Поток 2 полностью использовал свой квант, произошло прерывание от таймера, и планировщик активизировал поток 1.

При выполнении потока 1 произошло событие R – системный вызов, в результате которого освободился некоторый ресурс (например, был закрыт файл). Это событие вызвало перепланировку потоков. Планировщик просмотрел очередь ожидающих потоков и обнаружил, что поток 4 ждет освобождения данного ресурса. Этот поток был переведен в состояние го-

товности, но, поскольку приоритет выполняющегося в данный момент потока 1 выше приоритета потока 4, планировщик вернул процессор потоку 1.

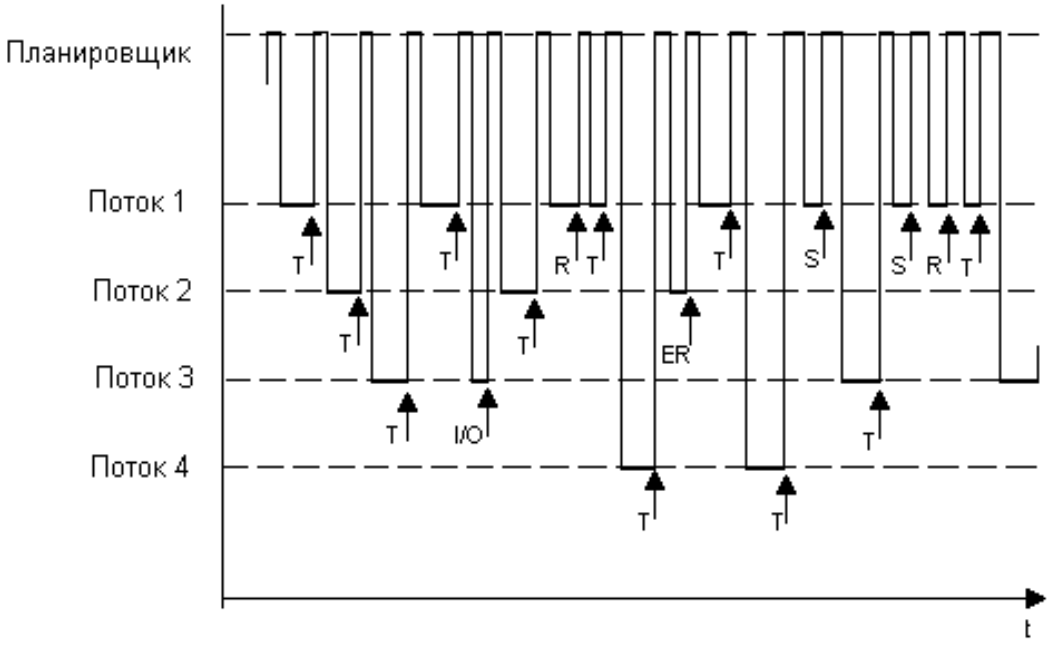



Рис. 4.7. Моменты перепланировки потоков

В следующем цикле работы планировщик активизировал поток 4, а затем, после истечения кванта и сигнала от таймера, управление получил поток 2. Этот поток не успел использовать свой квант, так как был снят с выполнения в результате возникшей ошибки (событие ER).

Далее планировщик предоставлял процессорное время потокам 1, 4 и снова 1. Во время выполнения потока 1 произошло прерывание S от внешнего устройства, сигнализирующее о том, что операция передачи данных завершена. Это событие активизировало работу планировщика, в результате которой поток 3, ожидавший завершения ввода-вывода, вытеснил поток 1, так как имел в этот момент более высокий приоритет.

Последний показанный на диаграмме период выполнения потока 1 прерывался несколько раз. Вначале это было прерывание от внешнего устройства S, затем программное прерывание R, вызвавшее освобождение ресурса, и, наконец, прерывание от таймера T. Каждое из этих трех прерываний вызвало перепланировку потоков. В двух первых случаях планировщик оставил выполняться поток 1, так как в очереди не оказалось более приоритетных потоков, а квант времени, выделенный потоку 1, еще не был исчерпан. Переключение потоков было выполнено только по прерыванию от таймера.

В системах реального времени для отработки статического расписания планировщик активизируется по прерываниям от таймера. Эти преры-

вания пронизывают всю временную ось, возникая через короткие постоянные интервалы времени. После каждого прерывания планировщик просматривает расписание и проверяет, не пора ли переключить задачи. Кроме прерываний от таймера, в системах реального времени перепланирование задач может происходить по прерываниям от внешних устройств – различного вида датчиков и исполнительных механизмов.

## 4.5. Вопросы и задания для самопроверки

1. Каковы наиболее характерные критерии эффективности вычислительных систем?

2. Назовите критерии эффективности систем пакетной обработки, систем разделения времени и систем реального времени.

3. Что такое мультипроцессорная обработка? Каковы основные архитектуры мультипроцессорных систем?

4. В чем состоит отличие процесса от потока?

5. Какие задачи включает в себя планирование потоков?

6. Опишите суть алгоритмов планирования потоков, основанных на квантовании. Приведите примеры операционных систем, где используются данные виды планирования.

7. Опишите суть алгоритмов планирования потоков, основанных на приоритетах. Приведите примеры операционных систем, где используются данные виды планирования.

8. Опишите вкратце алгоритмы планирования потоков в системах реального времени.

9. В чем состоит суть моментов перепланировки?

# ТЕМА 5. УПРАВЛЕНИЕ ПАМЯТЬЮ

Цель изучения темы – приобретение студентами знаний о функциях операционных систем по управлению памятью, а также принципах функционирования механизмов оперативной памяти современных операционных систем.

В результате изучения темы студенты должны:

– иметь понятие о функциях операционной системы по управлению памятью в мультипрограммной системе;

– знать типы адресов памяти и принципы их формирования;

– иметь представление об алгоритмах распределения памяти;

– знать принципы функционирования кэш-памяти.

## Содержание темы

1. Функции ОС по управлению памятью.
2. Типы адресов.
3. Алгоритмы распределения памяти.
3.1. Распределение памяти фиксированными разделами.
3.2. Распределение памяти динамическими разделами.
3.3. Перемещаемые разделы.
4. Свопинг и виртуальная память.
4.1. Страничное распределение.
4.2. Сегментное распределение.
4.3. Сегментно-страничное распределение.
5. Разделяемые сегменты памяти.
6. Кэширование данных.
6.1. Иерархия запоминающих устройств.
6.2. Кэш-память.
6.3. Принцип действия кэш-памяти.
6.2. Проблема согласования данных.
6.4. Способы отображения основной памяти на кэш.
6.5. Схемы выполнения запросов в системах с кэш-памятью.
7. Вопросы и задания для самопроверки.

## 5.1. Функции ОС по управлению памятью

Память является важнейшим ресурсом, требующим тщательного управления со стороны мультипрограммной операционной системы. Особая роль памяти объясняется тем, что процессор может выполнять инструкции программы только в том случае, если они находятся в памяти. Память распределяется как между модулями прикладных программ, так и между модулями самой операционной системы.

В ранних ОС управление памятью сводилось просто к загрузке программы и ее данных из некоторого внешнего накопителя (перфоленты, магнитной ленты или магнитного диска) в память. С появлением мультипрограммирования перед ОС были поставлены новые задачи, связанные с распределением имеющейся памяти между несколькими одновременно выполняющимися программами.

Функциями ОС по управлению памятью в мультипрограммной системе являются:

– отслеживание свободной и занятой памяти;

– выделение памяти процессам и освобождение памяти по завершении процессов;

– полное или частичное вытеснение кодов и данных 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;

– настройка адресов программы на конкретную область физической памяти.

Помимо первоначального выделения памяти процессам при их создании, ОС должна также заниматься динамическим распределением памяти, то есть выполнять запросы приложений на выделение им дополнительной памяти во время выполнения. После того как приложение перестает нуждаться в дополнительной памяти, оно может возвратить ее системе. Выделение памяти случайной длины в случайные моменты времени из общего пула памяти приводит к фрагментации памяти и вследствие этого к неэффективному ее использованию. Дефрагментация памяти является одной из функций операционной системы.

Во время работы ОС ей часто приходится создавать новые служебные информационные структуры, такие как описатели процессов и потоков, различные таблицы распределения ресурсов, буферы, используемые процессами для обмена данными, синхронизирующие объекты и т. п. Все эти системные объекты требуют памяти. В некоторых ОС заранее, уже во время

установки, резервируется некоторый фиксированный объем памяти для системных нужд. В других же ОС используется более гибкий подход, при котором память для системных целей выделяется динамически. В таком случае разные подсистемы ОС при создании своих таблиц, объектов, структур и т. п. обращаются к подсистеме управления памятью с запросами.

Защита памяти – еще одна важная задача ОС, которая состоит в том, чтобы не позволить выполняемому процессу записывать или читать данные из памяти, назначенной другому процессу. Эта функция, как правило, реализуется программными модулями ОС в тесном взаимодействии с аппаратными средствами.

## 5.2. Типы адресов

Для идентификации переменных и команд на разных этапах жизненного цикла программы используются символьные имена (метки), виртуальные и физические адреса.

*Символьные имена* присваивает пользователь при написании программы на алгоритмическом языке или ассемблере.

*Виртуальные адреса*, называемые иногда математическими или логическими адресами, вырабатывает транслятор, переводящий программу на машинный язык. Поскольку во время трансляции в общем случае неизвестно, в какое место оперативной памяти будет загружена программа, то транслятор присваивает переменным и командам виртуальные (условные) адреса, обычно считая по умолчанию, что начальным адресом программы будет нулевой адрес.

*Физические адреса* соответствуют номерам ячеек оперативной памяти, где в действительности расположены или будут расположены переменные и команды.

Совокупность виртуальных адресов процесса называется *виртуальным адресным пространством*. Диапазон возможных адресов виртуального пространства у всех процессов является одним и тем же. Например, при использовании 32-разрядных виртуальных адресов этот диапазон задается границами 0000000016 и FFFFFFFF16. Тем не менее, каждый процесс имеет собственное виртуальное адресное пространство: транслятор присваивает виртуальные адреса переменным и кодам каждой программе независимо.

Совпадение виртуальных адресов переменных и команд различных процессов не приводит к конфликтам, так как в случае, когда эти переменные одновременно присутствуют в памяти, операционная система отображает их на разные физические адреса.

В разных операционных системах используются разные способы структуризации виртуального адресного пространства. В одних ОС виртуальное адресное пространство процесса подобно физической памяти представлено в виде непрерывной *линейной* последовательности виртуальных адресов. Такую структуру адресного пространства называют также *плоской* (*flat*). При этом виртуальным адресом является единственное число, представляющее собой смещение относительно начала (обычно это значение 000...000) виртуального адресного пространства. Адрес такого типа называют линейным виртуальным адресом.

В других ОС виртуальное адресное пространство делится на части, называемые сегментами (или секциями, или областями). В этом случае, помимо линейного адреса, может быть использован виртуальный адрес, представляющий собой *пару* чисел (*n*, *m*), где *n* определяет сегмент, а *m* – смещение внутри сегмента.

Существуют и более сложные способы структуризации виртуального адресного пространства, когда виртуальный адрес образуется тремя или более числами.

Задачей операционной системы является отображение индивидуальных виртуальных адресных пространств всех одновременно выполняющихся процессов на общую физическую память. При этом ОС отображает либо все виртуальное адресное пространство, либо только определенную его часть. Процедура преобразования виртуальных адресов в физические должна быть максимально прозрачна для пользователя и программиста.

Существуют два принципиально отличающихся подхода к преобразованию виртуальных адресов в физические.

В первом случае замена виртуальных адресов на физические выполняется один раз для каждого процесса во время начальной загрузки программы в память. Специальная системная программа – *перемещающий загрузчик* – на основании имеющихся у нее исходных данных о начальном адресе физической памяти, в которую предстоит загружать программу, а также предоставленной транслятором информации об адресно-зависимых элементах программы выполняет загрузку программы, совмещая ее с заменой виртуальных адресов физическими.

Второй способ заключается в том, что программа загружается в память в виртуальных адресах в неизмененном виде, т. е. операнды инструкций и адреса переходов имеют те значения, которые выработал транслятор. В наиболее простом случае, когда виртуальная и физическая память процесса представляют собой единые непрерывные области адресов, операционная система выполняет преобразование виртуальных адресов в физиче-

ские по следующей схеме. При загрузке операционная система фиксирует смещение действительного расположения программного кода относительно виртуального адресного пространства. Во время выполнения программы при каждом обращении к оперативной памяти выполняется преобразование виртуального адреса в физический. Пусть, например, операционная система использует линейно-структурированное виртуальное адресное пространство и пусть некоторая программа, работающая под управлением этой ОС, загружена в физическую память начиная с физического адреса S. ОС запоминает значение начального смещения S и во время выполнения программы помещает его в специальный регистр процессора. При обращении к памяти виртуальные адреса данной программы преобразуются в физические путем прибавления к ним смещения S.

Последний способ является более гибким: в то время как перемещающий загрузчик жестко привязывает программу к первоначально выделенному ей участку памяти, динамическое преобразование виртуальных адресов позволяет перемещать программный код процесса в течение всего периода его выполнения. Однако использование перемещающего загрузчика более экономично, так как в этом случае преобразование каждого виртуального адреса происходит только один раз во время загрузки, а при динамическом преобразовании – при каждом обращении по данному адресу.

В некоторых случаях (обычно в специализированных системах), когда заранее точно известно, в какой области оперативной памяти будет выполняться программа, транслятор выдает исполняемый код сразу в физических адресах.

Необходимо различать *максимально возможное* виртуальное адресное пространство процесса и *назначенное* (*выделенное*) процессу виртуальное адресное пространство. В первом случае речь идет о максимальном размере виртуального адресного пространства, определяемом архитектурой компьютера, на котором работает ОС, и, в частности, разрядностью его схем адресации (32-битная, 64-битная и т. п.). Например, при работе на компьютерах с 32-разрядными процессорами Intel Pentium операционная система может предоставить каждому процессу виртуальное адресное пространство до 4 Гбайт ($2^{32}$). Однако это значение представляет собой только потенциально возможный размер виртуального адресного пространства, который редко на практике бывает необходим процессу. Процесс использует только часть доступного ему виртуального адресного пространства.

Назначенное виртуальное адресное пространство представляет собой набор виртуальных адресов, действительно нужных процессу для работы. Эти адреса первоначально назначает программе транслятор на основании

текста программы, когда создает кодовый (текстовый) сегмент, а также сегмент или сегменты данных, с которыми программа работает. Затем при создании процесса ОС фиксирует назначенное виртуальное адресное пространство в своих системных таблицах. В ходе своего выполнения процесс может увеличить размер первоначального назначенного ему виртуального адресного пространства, запросив у ОС создание дополнительных сегментов или увеличение размера существующих. В любом случае операционная система обычно следит за корректностью использования процессом виртуальных адресов: процессу не разрешается оперировать с виртуальным адресом, выходящим за пределы назначенных ему сегментов.

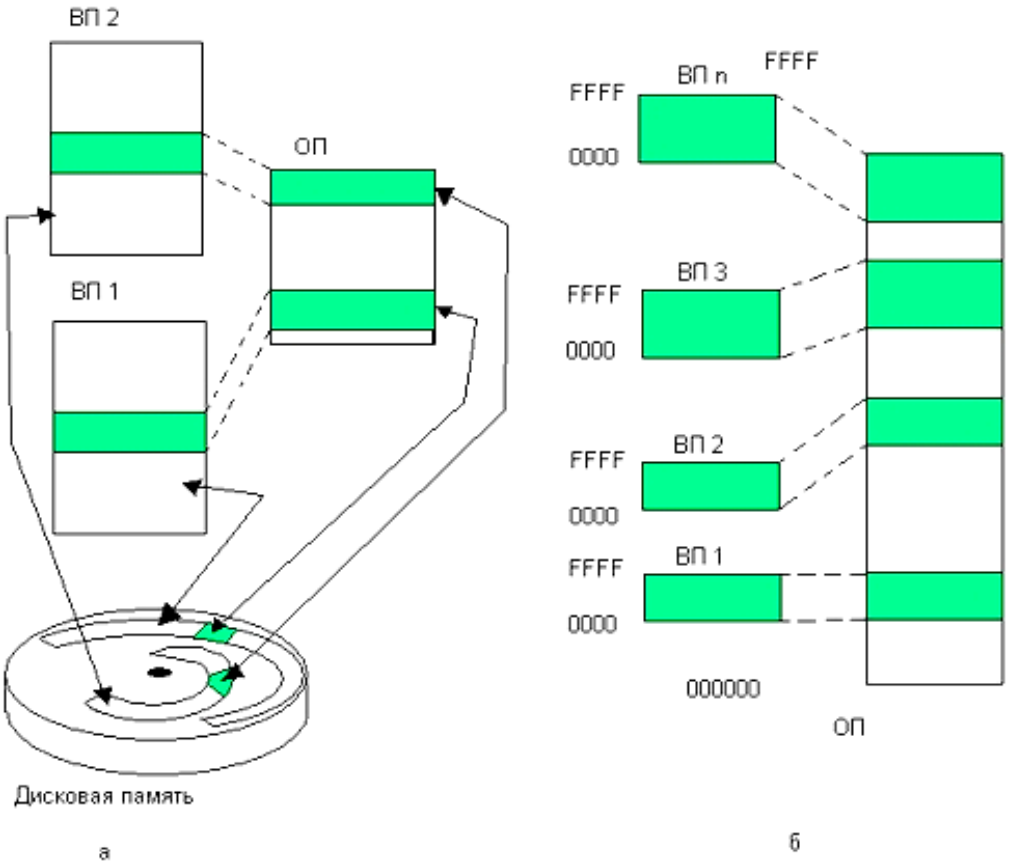



Рис. 5.1. Соотношение объемов виртуального адресного пространства и физической памяти:
*а* – виртуальное адресное пространство превосходит объем физической памяти,
*б* – виртуальное адресное пространство меньше объема физической памяти

Максимальный размер виртуального адресного пространства ограничивается только разрядностью адреса, присущей данной архитектуре компьютера, и, как правило, не совпадает с объемом физической памяти, имеющимся в компьютере.

Сегодня для машин универсального назначения типична ситуация, когда объем виртуального адресного пространства превышает доступный объ-

ем оперативной памяти. В таком случае операционная система для хранения данных виртуального адресного пространства процесса, не помещающихся в оперативную память, использует внешнюю память, которая в современных компьютерах представлена жесткими дисками (рис. 5.1, *а*). Именно на этом принципе основана *виртуальная память* – наиболее совершенный механизм, используемый в операционных системах для управления памятью.

Однако соотношение объемов виртуальной и физической памяти может быть и обратным. Так, в мини-компьютерах 80-х годов разрядности поля адреса нередко не хватало для того, чтобы охватить всю имеющуюся оперативную память. Несколько процессов могло быть загружено в память одновременно и целиком (рис. 5.1, *б*).

Необходимо подчеркнуть, что виртуальное адресное пространство и виртуальная память – различные механизмы и они не обязательно реализуются в операционной системе одновременно. Можно представить себе ОС, в которой поддерживаются виртуальные адресные пространства для процессов, но отсутствует механизм виртуальной памяти. Это возможно только в том случае, если размер виртуального адресного пространства каждого процесса меньше объема физической памяти.

Содержимое назначенного процессу виртуального адресного пространства, т. е. коды команд, исходные и промежуточные данные, а также результаты вычислений, представляет собой *образ процесса.*

Во время работы процесса постоянно выполняются переходы от прикладных кодов к кодам ОС, которые либо явно вызываются из прикладных процессов как системные функции, либо вызываются как реакция на внешние события или на исключительные ситуации, возникающие при некорректном поведении прикладных кодов. Для того чтобы упростить передачу управления от прикладного кода к коду ОС, а также для легкого доступа модулей ОС к прикладным данным (например, для вывода их на внешнее устройство), в большинстве ОС ее сегменты разделяют виртуальное адресное пространство с прикладными сегментами активного процесса. То есть сегменты ОС и сегменты активного процесса образуют единое виртуальное адресное пространство.

Обычно виртуальное адресное пространство процесса делится на две непрерывные части: системную и пользовательскую. В некоторых ОС (например, Windows NT, OS / 2) эти части имеют одинаковый размер – по 2 Гбайт, хотя в принципе деление может быть и другим, например, 1 Гбайт для ОС, и 2 Гбайт для прикладных программ. Часть виртуального адресного пространства каждого процесса, отводимая под сегменты ОС, является идентичной для всех процессов. Поэтому при смене активного процесса

заменяется только вторая часть виртуального адресного пространства, содержащая его индивидуальные сегменты, как правило, коды и данные прикладной программы (рис. 5.2). Архитектура современных процессоров отражает эту особенность структуры виртуального адресного пространства. Например, в процессорах Intel Pentium существует два типа системных таблиц: одна для описания сегментов, общих для всех процессов, а другая для описания индивидуальных сегментов данного процесса. При смене процесса первая таблица остается неизменной, а вторая заменяется новой.

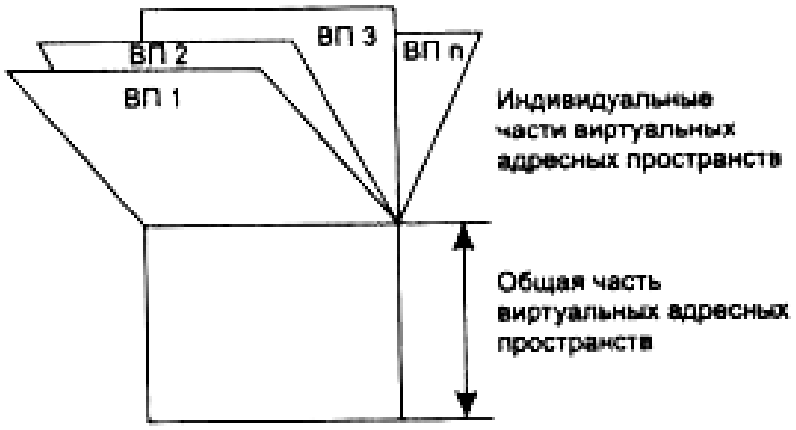



Рис. 5.2. Общая и индивидуальные части виртуальных адресных пространств

Описанное выше назначение двух частей виртуального адресного пространства – для сегментов ОС и для сегментов прикладной программы – является типичным, но не абсолютным. Имеются и исключения из общего правила. В некоторых ОС существуют системные процессы, порожденные для решения внутренних задач ОС. В этих процессах отсутствуют сегменты прикладной программы, но они могут расположить некоторые свои сегменты в общей части виртуального адресного пространства, а некоторые – в индивидуальной части, обычно предназначенной для прикладных сегментов. И наоборот, в общей системной части виртуального адресного пространства размещаются сегменты прикладного кода, предназначенные для совместного использования несколькими прикладными процессами.

Механизм страничной памяти в большинстве универсальных операционных систем применяется ко всем сегментам пользовательской части виртуального адресного пространства процесса. Исключения могут составлять специализированные ОС, например ОС реального времени, в которых некоторые сегменты жестко фиксируются в оперативной памяти и соответственно никогда не выгружаются на диск – это обеспечивает быструю реакцию определенных приложений на внешние события.

Системная часть виртуальной памяти в ОС любого типа включает область, подвергаемую страничному вытеснению (*paged*), и область, на которую страничное вытеснение не распространяется (*non-paged*). В невытесняемой области размещаются модули ОС, требующие быстрой реакции и / или постоянного присутствия в памяти, например, диспетчер потоков или код, управляющий заменой страниц памяти. Остальные модули ОС подвергаются страничному вытеснению, как и пользовательские сегменты.

Обычно аппаратура накладывает свои ограничения на порядок использования виртуального адресного пространства. Некоторые процессоры (например, MIPS) предусматривают для определенной области системной части адресного пространства особые правила отображения на физическую память. При этом виртуальный адрес прямо отображается на физический адрес (последний либо полностью соответствует виртуальному адресу, либо равен его части). Такая особая область памяти не подвергается страничному вытеснению, и поскольку достаточно трудоемкая процедура преобразования адресов исключается, то доступ к располагаемым здесь кодам и данным осуществляется очень быстро.

## 5.3. Алгоритмы распределения памяти

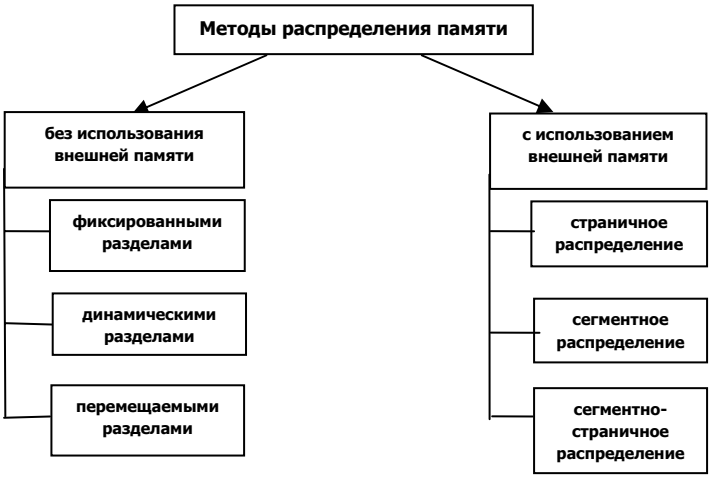



Рис. 5.3. Классификация методов распределения памяти

Следует ли назначать каждому процессу одну непрерывную область физической памяти или можно выделять память «кусками»? Должны ли сегменты программы, загруженные в память, находиться на одном месте в течение всего периода выполнения процесса или можно ее время от времени сдвигать? Что делать, если сегменты программы не помещаются в имеющуюся память? Разные ОС по-разному отвечают на эти и другие базовые вопросы управления памятью. Далее будут рассмотрены наиболее

общие подходы к распределению памяти, характерные для разных периодов развития операционных систем. Некоторые из них сохранили актуальность и широко используются в современных ОС, другие же представляют в основном только познавательный интерес, хотя их и сегодня можно встретить в специализированных системах.

На рис. 5.3 все алгоритмы распределения памяти разделены на два класса: алгоритмы, в которых используется перемещение сегментов процессов между оперативной памятью и диском, и алгоритмы, в которых внешняя память не привлекается.

### 5.3.1. Распределение памяти фиксированными разделами

Простейший способ управления оперативной памятью заключается в том, что память разбивается на несколько областей фиксированной величины, называемых *разделами*. Такое разбиение может быть выполнено вручную оператором во время старта системы или во время ее установки. После этого границы разделов не изменяются.

Очередной процесс, поступивший на выполнение, помещается либо в общую очередь (рис. 5.4, *а*), либо в очередь к некоторому разделу (рис. 5.4, *б*).

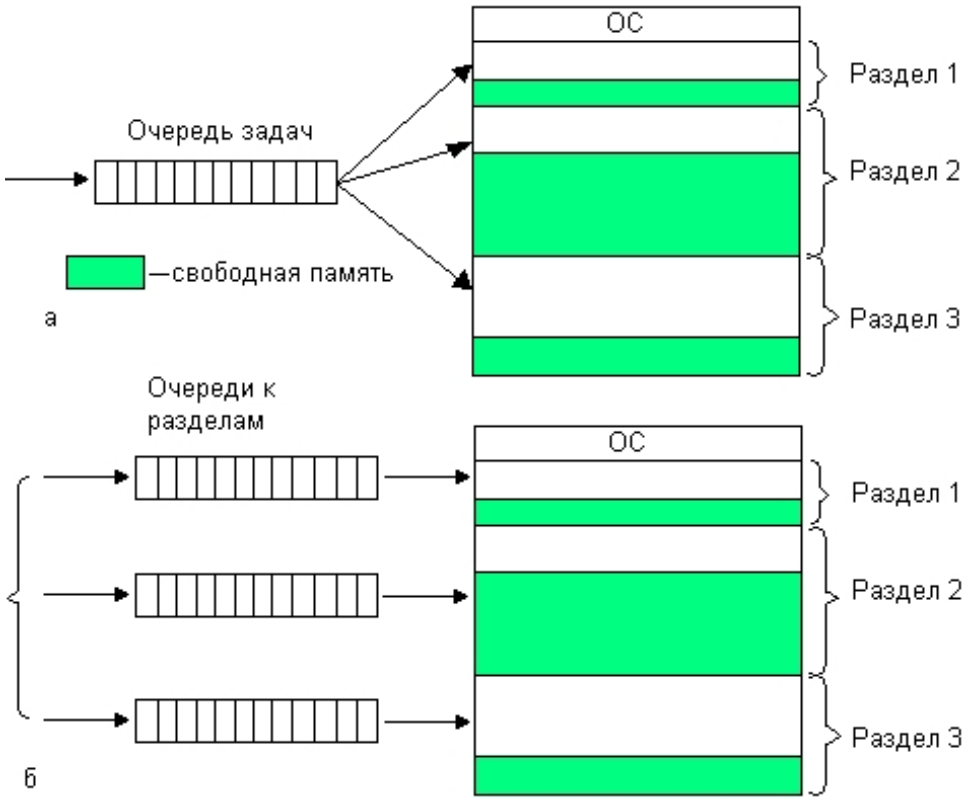



Рис. 5.4. Распределение памяти фиксированными разделами:
*а* – с общей очередью, *б* – с отдельными очередями

Подсистема управления памятью в этом случае выполняет следующие задачи:

– сравнивает объем памяти, требуемый для вновь поступившего процесса, с размерами свободных разделов и выбирает подходящий раздел;

– осуществляет загрузку программы в один из разделов и настройку адресов. Уже на этапе трансляции разработчик программы может задать раздел, в котором ее следует выполнять. Это позволяет сразу, без использования перемещающего загрузчика, получить машинный код, настроенный на конкретную область памяти.

При очевидном преимуществе – простоте реализации – данный метод имеет и существенный недостаток – жесткость. Так как в каждом разделе может выполняться только один процесс, то уровень мультипрограммирования заранее ограничен числом разделов. Независимо от размера программы она будет занимать весь раздел. Так, например, в системе с тремя разделами невозможно выполнять одновременно более трех процессов, даже если им требуется совсем мало памяти. С другой стороны, разбиение памяти на разделы не позволяет выполнять процессы, программы которых не помещаются ни в один из разделов, но для которых было бы достаточно памяти нескольких разделов.

Такой способ управления памятью применялся в ранних мультипрограммных ОС. Однако и сейчас метод распределения памяти фиксированными разделами находит применение в системах реального времени, в основном благодаря небольшим затратам на реализацию. Детерминированность вычислительного процесса систем реального времени (заранее известен набор выполняемых задач, их требования к памяти, а иногда и моменты запуска) компенсирует недостаточную гибкость данного способа управления памятью.

### 5.3.2. Распределение памяти динамическими разделами

В этом случае память машины заранее не делится на разделы. Сначала вся память, отводимая для приложений, свободна. Каждому вновь поступающему на выполнение приложению на этапе создания процесса выделяется вся необходимая ему память; если достаточный объем памяти отсутствует, то приложение не принимается на выполнение и процесс для него не создается. После завершения процесса память освобождается, и на это место может быть загружен другой процесс. Таким образом, в произвольный момент времени оперативная память представляет собой случайную последовательность занятых и свободных участков (разделов) произ-

вольного размера. На рис. 5.5 показано состояние памяти в различные моменты времени при использовании динамического распределения.

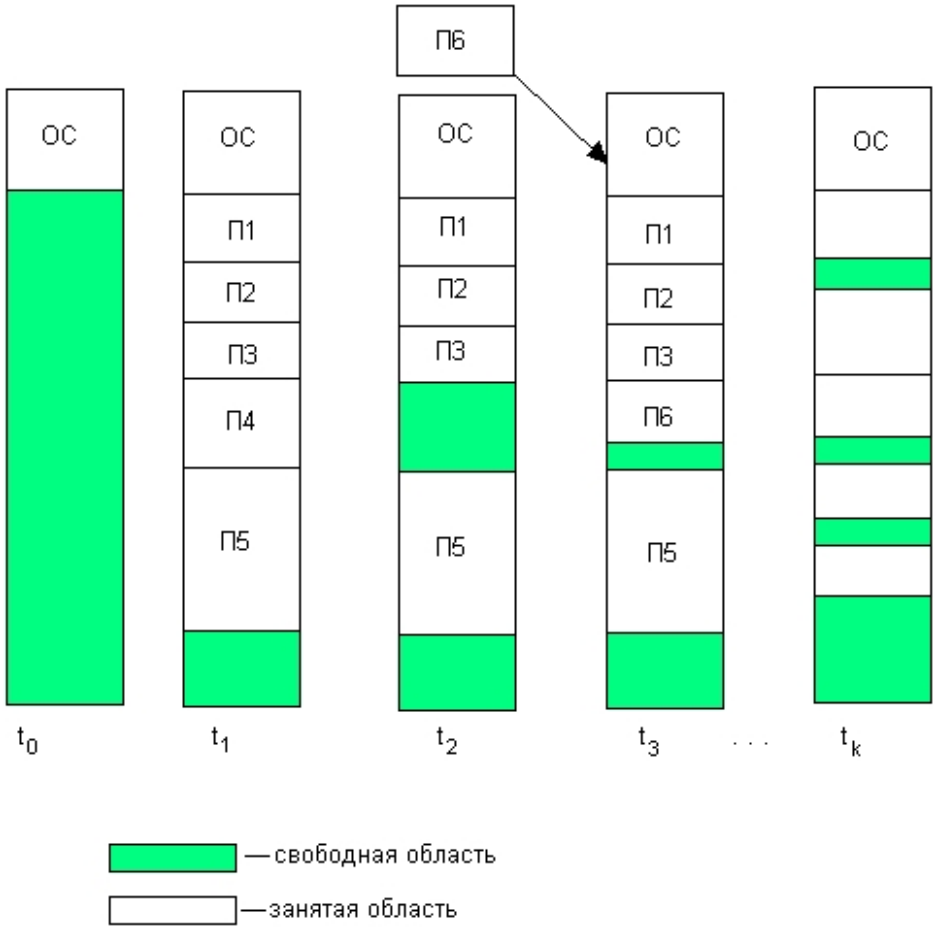



Рис. 5.5. Распределение памяти динамическими разделами

В момент $t_0$ в памяти находится только ОС, а к моменту $t_1$ память разделена между пятью процессами, причем процесс П4, завершаясь, покидает память. На освободившееся от процесса П4 место загружается процесс П6, поступивший в момент $t_3$.

Перечислим функции операционной системы, предназначенные для реализации данного метода управления памятью:

– ведение таблиц свободных и занятых областей, в которых указываются начальные адреса и размеры участков памяти;

– при создании нового процесса – анализ требований к памяти, просмотр таблицы свободных областей и выбор раздела, размер которого достаточен для размещения кодов и данных нового процесса. Выбор раздела может осуществляться по разным правилам, например, «первый попавшийся раздел достаточного размера», «раздел, имеющий наименьший достаточный размер» или «раздел, имеющий наибольший достаточный размер»;

– загрузка программы в выделенный ей раздел и корректировка таблиц свободных и занятых областей. Данный способ предполагает, что программный код не перемещается во время выполнения, а значит, настройка адресов может быть проведена единовременно во время загрузки;

– после завершения процесса корректировка таблиц свободных и занятых областей.

По сравнению с методом распределения памяти фиксированными разделами данный метод обладает гораздо большей гибкостью, но ему присущ очень серьезный недостаток – фрагментация памяти. *Фрагментация* – это наличие большого числа несмежных участков свободной памяти очень маленького размера (фрагментов). Настолько маленького, что ни одна из вновь поступающих программ не может поместиться ни в одном из участков, хотя суммарный объем фрагментов может составить значительную величину, намного превышающую требуемый объем памяти.

Распределение памяти динамическими разделами лежит в основе подсистем управления памятью многих мультипрограммных операционных системах 60-х – 70-х гг. XX в., в частности, такой популярной операционной системы, как OS / 360.

### 5.3.3. Перемещаемые разделы

Одним из методов борьбы с фрагментацией является перемещение всех занятых участков в сторону старших или младших адресов так, чтобы вся свободная память образовала единую свободную область (рис. 5.10).

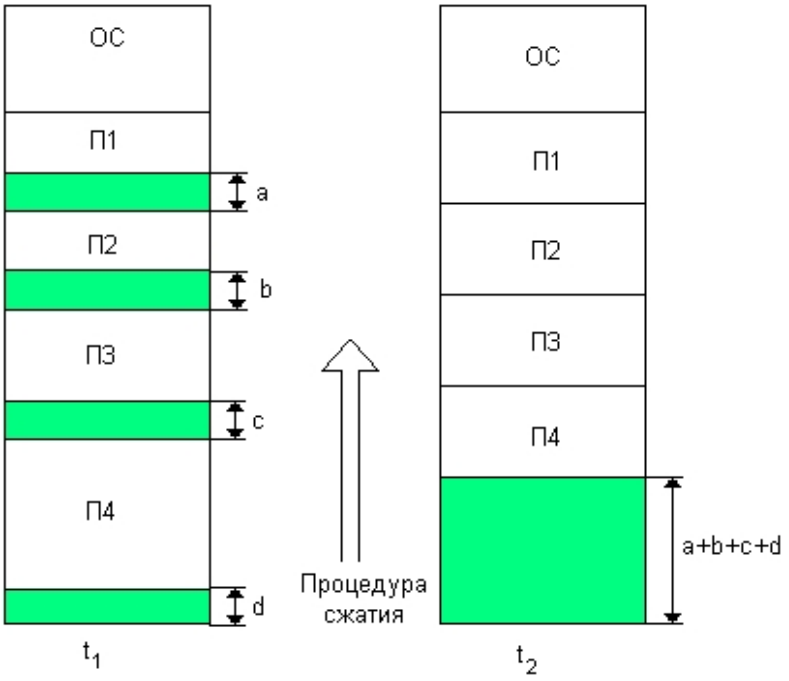



Рис. 5.6. Распределение памяти перемещаемыми разделами

В дополнение к функциям, выполняемым ОС при распределении памяти динамическими разделами, в данном случае она должна еще время от времени копировать содержимое разделов из одного места памяти в другое, корректируя таблицы свободных и занятых областей. Эта процедура называется *сжатием*. Сжатие может выполняться либо при каждом завершении процесса, либо только тогда, когда для вновь создаваемого процесса нет свободного раздела достаточного размера. В первом случае требуется меньше вычислительной работы при корректировке таблиц свободных и занятых областей, а во втором реже выполняется процедура сжатия.

Так как программы перемещаются по оперативной памяти в ходе своего выполнения, то в данном случае невозможно выполнить настройку адресов с помощью перемещающего загрузчика. Здесь более подходящим оказывается динамическое преобразование адресов.

Хотя процедура сжатия и приводит к более эффективному использованию памяти, она может потребовать значительного времени, что часто перевешивает преимущества данного метода.

Концепция сжатия применяется и при использовании других методов распределения памяти, когда отдельному процессу выделяется не одна сплошная область памяти, а несколько несмежных участков памяти произвольного размера (сегментов). Такой подход был использован в ранних версиях OS / 2, в которых память распределялась сегментами, а возникавшая при этом фрагментация устранялась путем периодического перемещения сегментов.

## 5.4. Свопинг и виртуальная память

Необходимым условием для того, чтобы программа могла выполняться, является ее нахождение в оперативной памяти. Только в этом случае процессор может извлекать команды из памяти и интерпретировать их, выполняя заданные действия. Объем оперативной памяти, имеющийся в компьютере, существенно сказывается на характере протекания вычислительного процесса. Он ограничивает число одновременно выполняющихся программ и размеры их виртуальных адресных пространств. В случаях, когда все задачи мультипрограммной смеси являются вычислительными, т. е. выполняют относительно мало операций ввода-вывода, разгружающих центральный процессор, для хорошей загрузки процессора может оказаться достаточным всего 3 – 5 задач. Однако если вычислительная система загружена выполнением интерактивных задач, то для эффективного использования процессора может потребоваться уже несколько десятков, а то и

сотен задач. Это иллюстрирует рис. 5.7, на котором показан график зависимости коэффициента загрузки процессора от числа одновременно выполняемых процессов и доли времени, проводимого этими процессами в состоянии ожидания ввода-вывода.

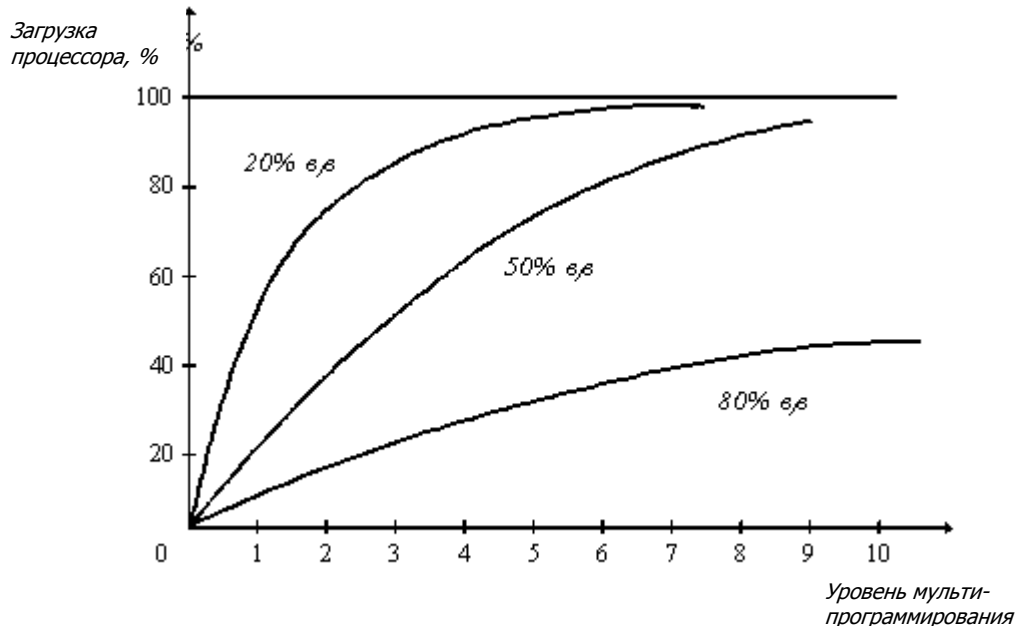



Рис. 5.7. Зависимость загрузки процессора от числа задач и интенсивности ввода-вывода

Большое количество задач, необходимое для высокой загрузки процессора, требует большого объема оперативной памяти. В условиях, когда для обеспечения приемлемого уровня мультипрограммирования имеющейся оперативной памяти недостаточно, был предложен метод организации вычислительного процесса, при котором образы некоторых процессов целиком или частично временно выгружаются на диск.

В мультипрограммном режиме, помимо активного процесса, т. е. процесса, коды которого в настоящий момент интерпретируются процессором, имеются приостановленные процессы, находящиеся в ожидании завершения ввода-вывода или освобождения ресурсов, а также процессы в состоянии готовности, стоящие в очереди к процессору. Образы таких неактивных процессов могут быть временно, до следующего цикла активности, выгружены на диск. Несмотря на то, что коды и данные процесса отсутствуют в оперативной памяти, ОС «знает» о его существовании и в полной мере учитывает это при распределении процессорного времени и других системных ресурсов. К моменту, когда подходит очередь выполнения выгруженного процесса, его образ возвращается с диска в оперативную память. Если при этом обнаруживается, что свободного места в оперативной памяти не хватает, то на диск выгружается другой процесс.

Такая подмена (*виртуализация*) оперативной памяти дисковой памятью позволяет повысить уровень мультипрограммирования. Объем оперативной памяти компьютера теперь не столь жестко ограничивает количество одновременно выполняемых процессов, поскольку суммарный объем памяти, занимаемой образами этих процессов, может существенно превосходить имеющийся объем оперативной памяти. *Виртуальным* называется ресурс, который пользователю или пользовательской программе представляется обладающим свойствами, которыми он в действительности не обладает. В данном случае в распоряжение прикладного программиста предоставляется виртуальная оперативная память, размер которой намного превосходит всю имеющуюся в системе реальную оперативную память. Пользователь пишет программу, а транслятор, используя виртуальные адреса, переводит ее в машинные коды так, как будто в распоряжении программы имеется однородная оперативная память большого объема. В действительности же все коды и данные, используемые программой, хранятся на дисках и только при необходимости загружаются в реальную оперативную память. Понятно, однако, что работа такой «оперативной памяти» происходит значительно медленнее.

Виртуализация оперативной памяти осуществляется совокупностью программных модулей ОС и аппаратных схем процессора и включает решение следующих задач:

– размещение данных в запоминающих устройствах разного типа, например, часть кодов программы – в оперативной памяти, а часть – на диске;

– выбор образов процессов или их частей для перемещения из оперативной памяти на диск и обратно;

– перемещение по мере необходимости данных между памятью и диском;

– преобразование виртуальных адресов в физические.

Очень важно то, что все действия по организации совместного использования диска и оперативной памяти – выделение места для перемещаемых фрагментов, настройка адресов, выбор кандидатов на загрузку и выгрузку – осуществляются операционной системой и аппаратурой процессора *автоматически*, без участия программиста, и никак не сказываются на логике работы приложений.

Виртуализация памяти может быть осуществлена на основе двух различных подходов:

– *свопинг (swapping)* – образы процессов выгружаются на диск и возвращаются в оперативную память *целиком*;

– *виртуальная память (virtual memory)* – между оперативной памятью и диском перемещаются *части* (сегменты, страницы и т. п.) образов процессов.

*Свопинг* представляет собой частный случай виртуальной памяти и, следовательно, более простой в реализации способ совместного использования оперативной памяти и диска. Однако подкачке свойственна избыточность: когда ОС решает активизировать процесс, для его выполнения, как правило, не требуется загружать в оперативную память все его сегменты полностью – достаточно загрузить небольшую часть кодового сегмента с подлежащей выполнению инструкцией и частью сегментов данных, с которыми работает эта инструкция, а также отвести место под сегмент стека. Аналогично при освобождении памяти для загрузки нового процесса часто вовсе не требуется выгружать другой процесс на диск целиком, достаточно вытеснить на диск только часть его образа. Перемещение избыточной информации замедляет работу системы, а также приводит к неэффективному использованию памяти. Кроме того, системы, поддерживающие свопинг, имеют еще один очень существенный недостаток: они не способны загрузить для выполнения процесс, виртуальное адресное пространство которого превышает имеющуюся в наличии свободную память.

Именно из-за указанных недостатков свопинг как основной механизм управления памятью почти не используется в современных ОС. На смену ему пришел более совершенный механизм *виртуальной памяти*, который, как уже было сказано, заключается в том, что при нехватке места в оперативной памяти на диск выгружаются только части образов процессов.

Ключевой проблемой виртуальной памяти, возникающей в результате многократного изменения местоположения в оперативной памяти образов процессов или их частей, является преобразование виртуальных адресов в физические. Решение этой проблемы, в свою очередь, зависит от того, какой способ структуризации виртуального адресного пространства принят в данной системе управления памятью. В настоящее время все множество реализаций виртуальной памяти может быть представлено тремя классами:

– *страничная виртуальная память* организует перемещение данных между памятью и диском страницами – частями виртуального адресного пространства, фиксированного и сравнительно небольшого размера;

– *сегментная виртуальная память* предусматривает перемещение данных сегментами – частями виртуального адресного пространства произвольного размера, полученными с учетом смыслового значения данных;

– *сегментно-страничная виртуальная память* использует двухуровневое деление: виртуальное адресное пространство делится на сегменты, а затем сегменты делятся на страницы. Единицей перемещения данных является страница. Этот способ управления памятью объединяет в себе элементы обоих предыдущих подходов.

Для временного хранения сегментов и страниц на диске отводится либо специальная область, либо специальный файл, которые во многих ОС по традиции продолжают называть областью, или файлом свопинга, хотя перемещение информации между оперативной памятью и диском осуществляется уже не в форме полного замещения одного процесса другим, а частями. Другое популярное название этой области – страничный файл (page file или paging file). Текущий размер страничного файла является важным параметром, оказывающим влияние на возможности операционной системы: чем больше страничный файл, тем больше приложений может одновременно выполнять ОС при фиксированном размере оперативной памяти. Однако необходимо учитывать, что увеличение числа одновременно работающих приложений за счет увеличения размера страничного файла замедляет их работу, так как значительная часть времени при этом тратится на перекачку кодов и данных из оперативной памяти на диск и обратно. Размер страничного файла в современных ОС является настраиваемым параметром, который выбирается администратором системы для достижения компромисса между уровнем мультипрограммирования и быстродействием системы.

### 5.4.1. Страничное распределение

На рис. 5.8 показана схема страничного распределения памяти. Виртуальное адресное пространство каждого процесса делится на части одинакового, фиксированного для данной системы размера, называемые *виртуальными страницами* (*virtual pages*). В общем случае размер виртуального адресного пространства процесса не кратен размеру страницы, поэтому последняя страница каждого процесса дополняется фиктивной областью.

Вся оперативная память машины также делится на части такого же размера, называемые *физическими страницами* (или блоками, или кадрами).

Размер страницы выбирается равным степени двойки: 512, 1024, 4096 байт и т. д. Это позволяет упростить механизм преобразования адресов.

При создании процесса ОС загружает в оперативную память несколько его виртуальных страниц (начальные страницы кодового сегмента и сегмента данных). Копия всего виртуального адресного пространства процесса находится на диске. Смежные виртуальные страницы не обязательно располагаются в смежных физических страницах. Для каждого процесса операционная система создает *таблицу страниц* – информационную структуру, содержащую записи обо всех виртуальных страницах процесса.

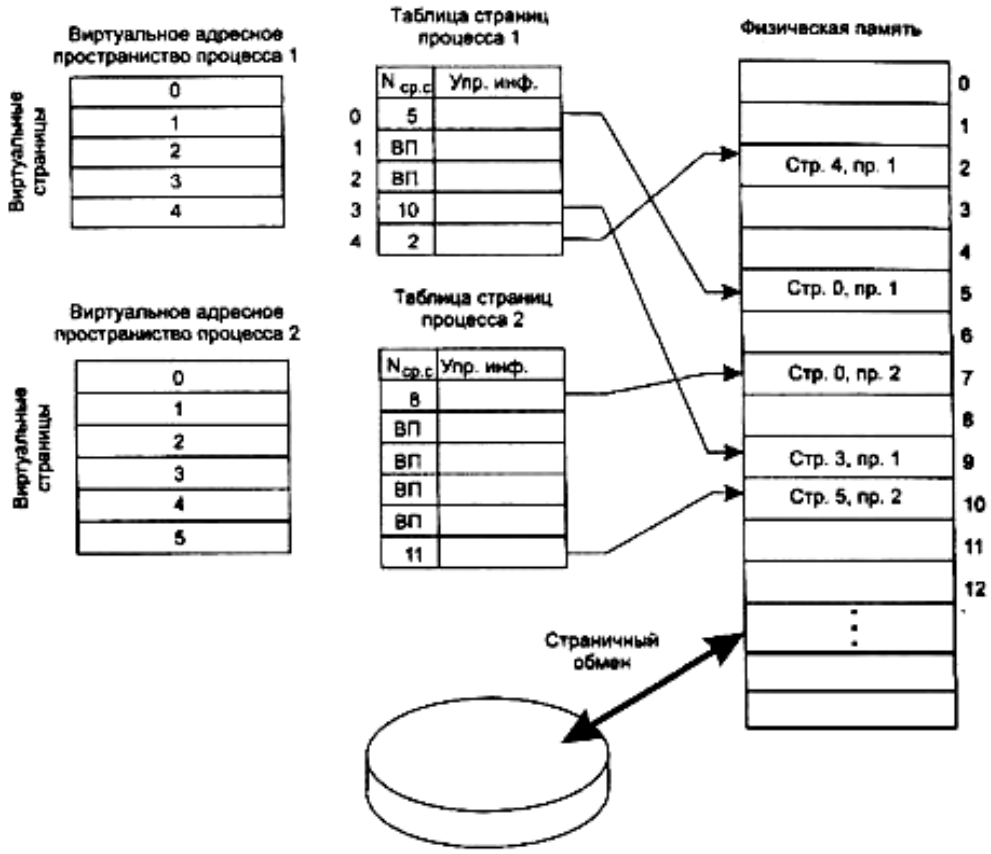



Рис. 5.8. Страничное распределение памяти

Запись таблицы, называемая *дескриптором страницы*, включает следующую информацию:

– *номер физической страницы*, в которую загружена данная виртуальная страница;

– *признак присутствия*, устанавливаемый в единицу, если виртуальная страница находится в оперативной памяти;

– *признак модификации* страницы, который устанавливается в единицу всякий раз, когда производится запись по адресу, относящемуся к данной странице;

– *признак обращения* к странице, называемый также *битом доступа,* который устанавливается в единицу при каждом обращении по адресу, относящемуся к данной странице.

Признаки присутствия, модификации и обращения в большинстве моделей современных процессоров устанавливаются аппаратно, схемами процессора при выполнении операции с памятью. Информация из таблиц страниц используется для решения вопроса о необходимости перемещения той или иной страницы между памятью и диском, а также для преобразования виртуального адреса в физический. Сами таблицы страниц, так же

как и описываемые ими страницы, размещаются в оперативной памяти. Адрес таблицы страниц включается в контекст соответствующего процесса. При активизации очередного процесса операционная система загружает адрес его таблицы страниц в специальный регистр процессора.

При каждом обращении к памяти выполняется поиск номера виртуальной страницы, содержащей требуемый адрес, затем по этому номеру определяется нужный элемент таблицы страниц, и из него извлекается описывающая страницу информация. Далее анализируется признак присутствия, и если данная виртуальная страница находится в оперативной памяти, то выполняется преобразование виртуального адреса в физический, т. е. виртуальный адрес заменяется указанным в записи таблицы физическим адресом. Если же нужная виртуальная страница в данный момент выгружена на диск, то происходит так называемое *страничное прерывание*. Выполняющийся процесс переводится в состояние ожидания, и активизируется другой процесс из очереди процессов, находящихся в состоянии готовности. Параллельно программа обработки страничного прерывания находит на диске требуемую виртуальную страницу (для этого операционная система должна помнить положение вытесненной страницы в страничном файле диска) и пытается загрузить ее в оперативную память. Если в памяти имеется свободная физическая страница, то загрузка выполняется немедленно, если же свободных страниц нет, то на основании принятой в данной системе стратегии замещения страниц решается вопрос о том, какую страницу следует выгрузить из оперативной памяти.

После того как выбрана страница, которая должна покинуть оперативную память, обнуляется ее бит присутствия и анализируется ее признак модификации. Если выталкиваемая страница за время последнего пребывания в оперативной памяти была модифицирована, то ее новая версия должна быть переписана на диск, если нет, то принимается во внимание, что на диске уже имеется предыдущая копия этой виртуальной страницы, и запись на диск не производится. Физическая страница объявляется свободной. Из соображений безопасности в некоторых системах освобождаемая страница обнуляется, чтобы невозможно было использовать содержимое выгруженной страницы.

Для хранения информации о положении вытесненной страницы в страничном файле ОС может использовать поля таблицы страниц или же другую системную структуру данных (например, дескриптор сегмента при сегментно-страничной организации виртуальной памяти).

Виртуальный адрес при страничном распределении может быть представлен в виде пары ($p$, $s_v$), где $p$ – порядковый номер виртуальной страницы процесса (нумерация страниц начинается с 0), а $s_v$ – смещение в пределах виртуальной страницы. Физический адрес также может быть представлен в виде пары ($n$, $S_f$), где $n$ – номер физической страницы, а $S_f$ – смещение в пределах физической страницы. Задача подсистемы виртуальной памяти состоит в отображении ($p$, $s_v$) в ($n$, $S_f$).

Прежде чем приступить к рассмотрению схемы преобразования виртуального адреса в физический, остановимся на двух базисных свойствах страничной организации.

Первое из них состоит в том, что объем страницы выбирается равным степени двойки – $2^{\kappa}$. Из этого следует, что смещение $s$ может быть получено простым отделением $k$ младших разрядов в двоичной записи адреса, а оставшиеся старшие разряды адреса представляют собой двоичную запись номера страницы; при этом неважно, является страница виртуальной или физической (рис. 5.9).

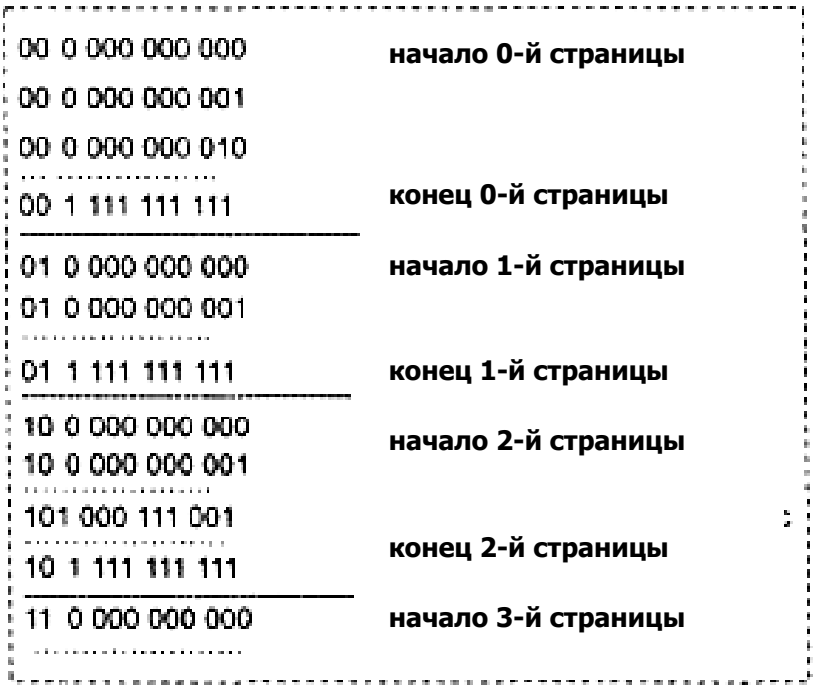



Рис. 5.9. Двоичное представление адресов

Из рисунка видно, что номер страницы и ее начальный адрес легко могут быть получены один из другого дополнением или отбрасыванием $k$ нулей, соответствующих смещению. Именно по этой причине часто говорят, что таблица страниц содержит начальный физический адрес страницы

в памяти, а не номер физической страницы, хотя на самом деле в таблице указаны только старшие разряды адреса. Начальный адрес страницы называется *базовым адресом.*

Второе свойство заключается в том, что в пределах страницы непрерывная последовательность виртуальных адресов однозначно отображается в непрерывную последовательность физических адресов, а значит, смещения в виртуальном и физическом адресах $s_v$ и $S_f$ равны между собой (рис. 5.10).

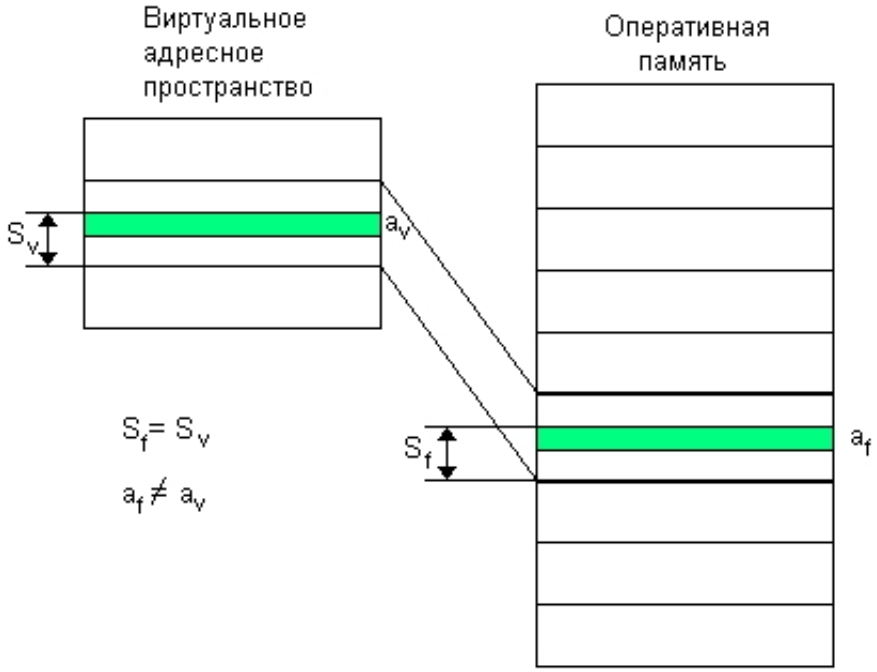



Рис. 5.10. При отображении виртуального адреса в физический смещение не изменяется

Отсюда следует простая схема преобразования виртуального адреса в физический (рис. 5.11). Младшие разряды физического адреса, соответствующие смещению, получаются переносом такого же количества младших разрядов из виртуального адреса. Старшие разряды физического адреса, соответствующие номеру физической страницы, определяются из таблицы страниц, в которой указывается соответствие виртуальных и физических страниц.

Итак, пусть произошло обращение к памяти по некоторому виртуальному адресу. Аппаратными схемами процессора выполняются следующие действия:

– из специального регистра процессора извлекается адрес АТ таблицы страниц активного процесса. На основании начального адреса таблицы страниц, номера виртуальной страницы $p$ (старшие разряды виртуального адреса) и длины отдельной записи в таблице страниц $L$ (систем-

ная константа) определяется адрес нужного дескриптора в таблице страниц: $a$ = AT + ($pxL$);

– из этого дескриптора извлекается номер соответствующей физической страницы – $n$;

– к номеру физической страницы присоединяется смещение $s$ (младшие разряды виртуального адреса).

Типичная машинная инструкция требует 3 – 4 обращений к памяти (выборка команды, извлечение операндов, запись результата). И при каждом обращении происходит либо преобразование виртуального адреса в физический, либо обработка страничного прерывания. Время выполнения этих операций в значительной степени влияет на общую производительность вычислительной системы, поэтому столь большое внимание разработчиков уделяется оптимизации виртуальной памяти.


Рис. 5.11. Схема преобразования виртуального адреса в физический при страничной организации памяти

Именно для уменьшения времени преобразования адресов во всех процессорах предусмотрен аппаратный механизм получения физического адреса по виртуальному. С той же целью размер страницы выбирается равным степени двойки, благодаря чему двоичная запись адреса легко разделяется на номер страницы и смещение, и в результате более длительная операция сложения в процедуре преобразования адресов заменяется операцией присоединения (конкатенации). Используются и другие способы ускорения преобразования, как, например, кэширование таблицы страниц – хра-

нение наиболее активно используемых записей в быстродействующих запоминающих устройствах, в частности в регистрах процессора.

Другим важным фактором, влияющим на производительность системы, является частота страничных прерываний, на которую, в свою очередь, влияют размер страницы и принятые в данной системе правила выбора страниц для выгрузки и загрузки. При неправильно выбранной стратегии замещения страниц могут возникать ситуации, когда система тратит бóльшую часть времени на подкачку страниц из оперативной памяти на диск и обратно.

При выборе страницы на выгрузку могут быть использованы различные критерии, смысл которых сводится к одному: на диск выталкивается страница, к которой в будущем, начиная с данного момента, дольше всего не будет обращений. Поскольку нельзя точно предсказать ход вычислительного процесса, то невозможно и точно определить страницу, подлежащую выгрузке. В таких условиях решение принимается на основе неких эмпирических критериев, часто основанных на предположении об инерционности вычислительного процесса. Так, например, из того, что страница не использовалась долгое время, делается вывод о том, что она, скорее всего, не будет использоваться и в ближайшее время. Однако привлечение критериев такого рода не исключает ситуаций, когда сразу после выгрузки страницы к ней происходит обращение и она снова должна быть загружена в память. Вероятность таких «напрасных» перемещений настолько велика, что в некоторых реализациях виртуальной памяти вообще отказываются от количественных критериев и предпочитают случайный выбор, при котором на диск выгружается первая попавшаяся страница. Возникающее при этом некоторое увеличение интенсивности страничного обмена компенсируется снижением вычислительных затрат на поддержание и анализ критерия выборки страниц на выгрузку.

Наиболее популярным критерием выбора страницы на выгрузку является число обращений к ней за последний период времени. Вычисление этого критерия происходит следующим образом. Операционная система ведет для каждой страницы программный счетчик. Значения счетчиков определяются значениями признаков доступа. Всякий раз, когда происходит обращение к какой-либо странице, процессор устанавливает в единицу признак доступа в относящейся к данной странице записи таблицы страниц. Операционная система периодически просматривает признаки доступа всех страниц во всех существующих в данный момент записях таблицы страниц. Если какой-либо признак оказывается равным 1 (т. е было обращение к странице), то система сбрасывает его в 0, увеличивая

при этом на единицу значение связанного с этой страницей счетчика обращений. Когда возникает необходимость удалить какую-либо страницу из памяти, ОС находит страницу, счетчик обращений которой имеет наименьшее значение. Для того чтобы критерий учитывал интенсивность обращений за последний период, ОС с соответствующей периодичностью обнуляет все счетчики.

Интенсивность страничного обмена может быть также снижена в результате так называемой *упреждающей загрузки*, в соответствии с которой при возникновении страничного прерывания в память загружается не одна страница, содержащая адрес обращения, а сразу несколько прилегающих к ней страниц. Здесь используется эмпирическое правило: если обращение произошло по некоторому адресу, то велика вероятность того, что следующие обращения произойдут по соседним адресам.

Другим важным резервом повышения производительности системы является правильный выбор размера страницы. С одной стороны, чтобы уменьшить частоту страничных прерываний, следовало бы увеличивать размер страницы. С другой стороны, если страница велика, то велика и фиктивная область в последней виртуальной странице каждого процесса. Если учесть, что в среднем фиктивная область в каждом процессе составляет половину страницы, то при большом объеме страницы суммарные потери могут составить существенную величину. Из этого следует, что выбор размера страницы является сложной оптимизационной задачей, требующей учета многих факторов. На практике же разработчики ОС и процессоров ограничиваются неким рациональным решением, пригодным для широкого класса вычислительных систем. Типичный размер страницы составляет несколько килобайт; например, наиболее распространенные процессоры x86 и Pentium компании Intel, а также операционные системы, устанавливаемые на этих процессорах, поддерживают страницы размером 4096 байт (4 Кбайт).

Размер страницы влияет также на количество записей в таблицах страниц. Чем меньше страница, тем более объемными являются таблицы страниц процессов и тем больше места они занимают в памяти. Учитывая, что в современных процессорах максимальный объем виртуального адресного пространства процесса, как правило, не меньше 4 Гбайт ($2^{32}$), то при размере страницы 4 Кбайт ($2^{12}$) и длине записи 4 байта для хранения таблицы страниц может потребоваться 4 Мбайт памяти. Выходом в такой ситуации является хранение в памяти только той части таблицы страниц, которая активно используется в данный период времени. Так как сама табли-

ца страниц хранится в таких же страницах физической памяти, что и описываемые ею страницы, то принципиально возможно временно вытеснять часть таблицы страниц из оперативной памяти.

Именно такого результата возможно достигнуть путем более сложной структуризации виртуального адресного пространства, при котором все множество виртуальных адресов процесса делится на разделы, а разделы в свою очередь делятся на страницы (рис. 5.12). Все страницы имеют одинаковый размер, а разделы содержат одинаковое количество страниц. Если размер страницы и количество страниц в разделе выбрать равными степени двойки ($2^k$ и $2^n$ соответственно), то принадлежность виртуального адреса к разделу и странице, а также смещение внутри страницы можно определить очень просто: младшие $k$ двоичных разрядов дают смещение, следующие $n$ разрядов представляют собой номер виртуальной страницы, а оставшиеся старшие разряды (обозначим их количество $t$) содержат номер раздела.

Для каждого раздела строится собственная таблица страниц. Количество дескрипторов в таблице и их размер подбираются такими, чтобы объем таблицы оказался равным объему страницы. Например, в процессоре Pentium при размере страницы 4 Кбайт длина дескриптора страницы составляет 4 байта и количество записей в таблице страниц, помещающейся на страницу, равняется соответственно 1024. Каждая таблица страниц описывается дескриптором, структура которого полностью совпадает со структурой дескриптора обычной страницы. Эти дескрипторы сведены в таблицу разделов, называемую также каталогом страниц. Физический адрес таблицы разделов активного процесса содержится в специальном регистре процессора и поэтому всегда известен операционной системе. Страница, содержащая таблицу разделов, никогда не выгружается из памяти, в противном случае работа виртуальной памяти была бы невозможна.

Выгрузка страниц с таблицами страниц позволяет сэкономить память, но при этом приводит к дополнительным временным затратам при получении физического адреса. Действительно, может случиться так, что та таблица страниц, которая содержит нужный дескриптор, в данный момент выгружена на диск. Тогда процесс преобразования адреса приостанавливается до тех пор, пока требуемая страница не будет снова загружена в память. Для уменьшения вероятности отсутствия страницы в памяти используются различные приемы, основным из которых является кэширование.

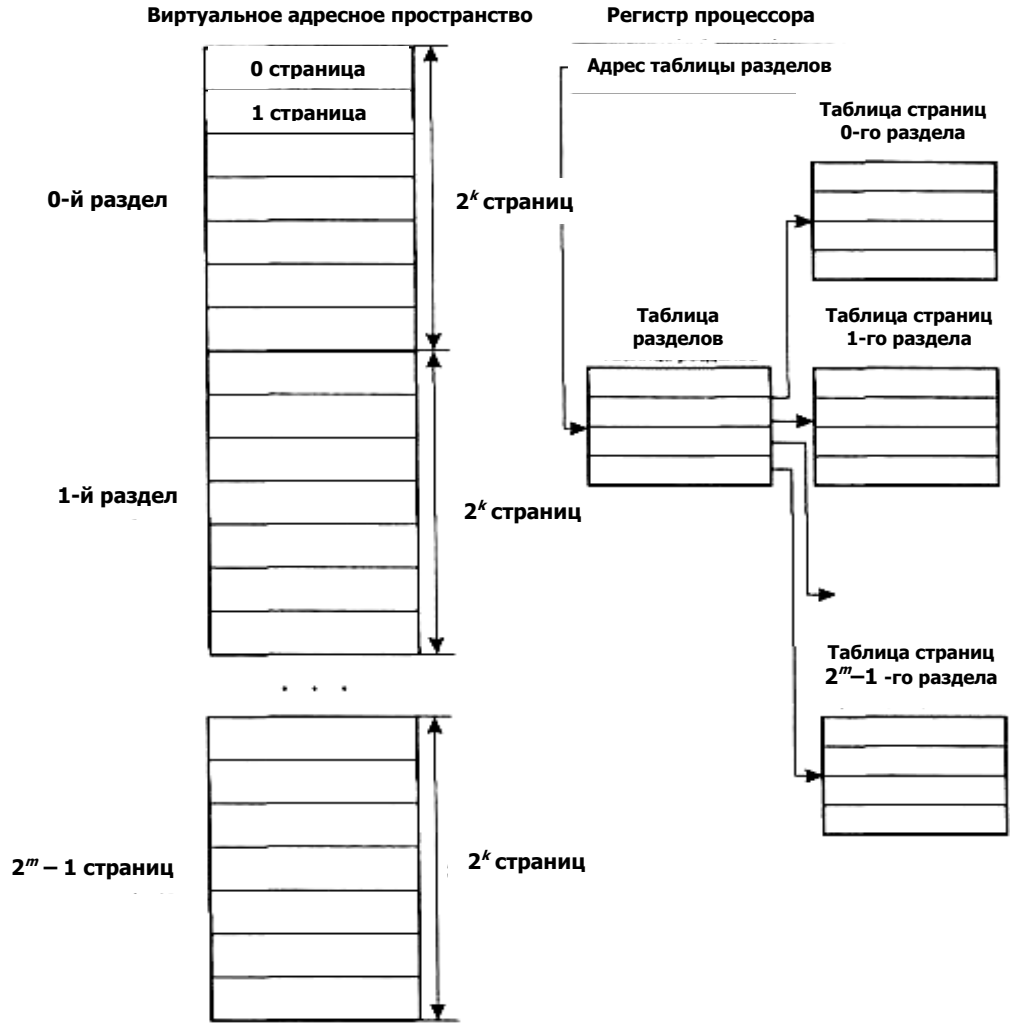



Рис. 5.12. Структура виртуального адресного пространства с разделами

Проследим более подробно схему преобразования адресов для случая двухуровневой структуризации виртуального адресного пространства (рис. 5.13):

– путем отбрасывания $k + n$ младших разрядов в виртуальном адресе определяется номер раздела, к которому принадлежит данный виртуальный адрес;

– по этому номеру из таблицы разделов извлекается дескриптор соответствующей таблицы страниц. Проверяется, находится ли данная таблица страниц в памяти. Если нет, происходит страничное прерывание и система загружает нужную страницу с диска;

– далее из этой таблицы страниц извлекается дескриптор виртуальной страницы, номер которой содержится в средних $n$ разрядах преобразуемого виртуального адреса. Снова выполняется проверка наличия данной страницы в памяти и при необходимости ее загрузка;

– из дескриптора определяется номер (базовый адрес) физической страницы, в которую загружена данная виртуальная страница. К номеру физической страницы пристыковывается смещение, взятое из *k* младших разрядов виртуального адреса. В результате получается искомый физический адрес.

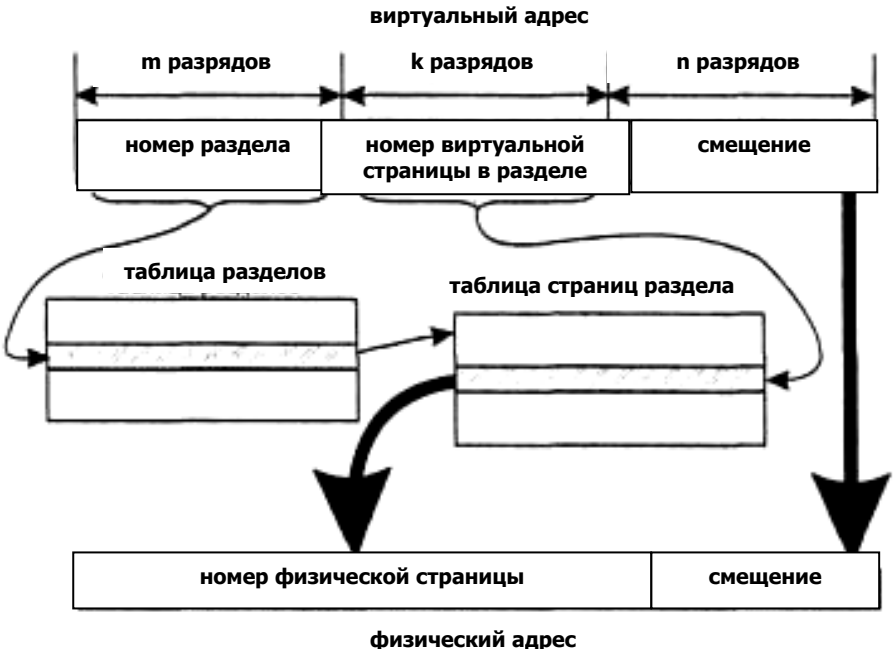



Рис. 5.13. Схема преобразования виртуального адреса
для двухуровневой структуризации адресного пространства

Страничное распределение памяти может быть реализовано в упрощенном варианте, без выгрузки страниц на диск. В этом случае все виртуальные страницы всех процессов постоянно находятся в оперативной памяти. Такой вариант страничной организации хотя и не предоставляет пользователю преимуществ работы с виртуальной памятью большого объема, но сохраняет другое достоинство страничной организации – позволяет успешно бороться с фрагментацией физической памяти. Во-первых, программу можно разбить на части и загрузить в разрозненные участки свободной памяти, во-вторых, при загрузке виртуальных страниц никогда не образуются неиспользуемые остатки, так как размеры виртуальных и физических страниц совпадают. Такой режим работы системы управления памятью используется в некоторых специализированных ОС, когда требуется высокая реактивность системы и способность выполнять переменный набор приложений (примером могут служить ОС семейства Novell NetWare 3.x и 4.x).

### 5.4.2. Сегментное распределение

При страничной организации виртуальное адресное пространство процесса делится на равные части *механически*, без учета смыслового значения данных. В одной странице могут оказаться и коды команд, и инициализируемые переменные, и массив исходных данных программы. Такой подход не позволяет обеспечить дифференцированный доступ к разным частям программы, а это свойство могло бы быть очень полезным во многих случаях. Например, можно было бы запретить обращаться с операциями записи в сегмент программы, содержащий коды команд, разрешив эту операцию для сегментов данных.

Кроме того, разбиение виртуального адресного пространства на части делает принципиально возможным совместное использование фрагментов программ разными процессами. Пусть, например, двум процессам требуется одна и та же подпрограмма, которая к тому же обладает свойством реентерабельности. Тогда коды этой подпрограммы могут быть оформлены в виде отдельного сегмента и включены в виртуальные адресные пространства обоих процессов. При отображении в физическую память сегменты, содержащие коды подпрограммы из обоих виртуальных пространств, проецируются на одну и ту же область физической памяти. Таким образом, оба процесса получат доступ к одной и той же копии подпрограммы (рис. 5.14).

Итак, виртуальное адресное пространство процесса делится на части – *сегменты*, размер которых определяется с учетом смыслового значения содержащейся в них информации. Отдельный сегмент может представлять собой подпрограмму, массив данных и т. п. Деление виртуального адресного пространства па сегменты осуществляется компилятором на основе указаний программиста или по умолчанию в соответствии с принятыми в системе соглашениями. Максимальный размер сегмента определяется разрядностью виртуального адреса, например, при 32-разрядной организации процессора он равен 4 Гбайт. При этом максимально возможное виртуальное адресное пространство процесса представляет собой набор из N виртуальных сегментов, каждый размером 4 Гбайт. В каждом сегменте виртуальные адреса находятся в диапазоне от 00000000 до FFFFFFFF. Сегменты не упорядочиваются друг относительно друга, так что общего для сегментов линейного виртуального адреса не существует, виртуальный адрес задается парой чисел: номером сегмента и линейным виртуальным адресом внутри сегмента.

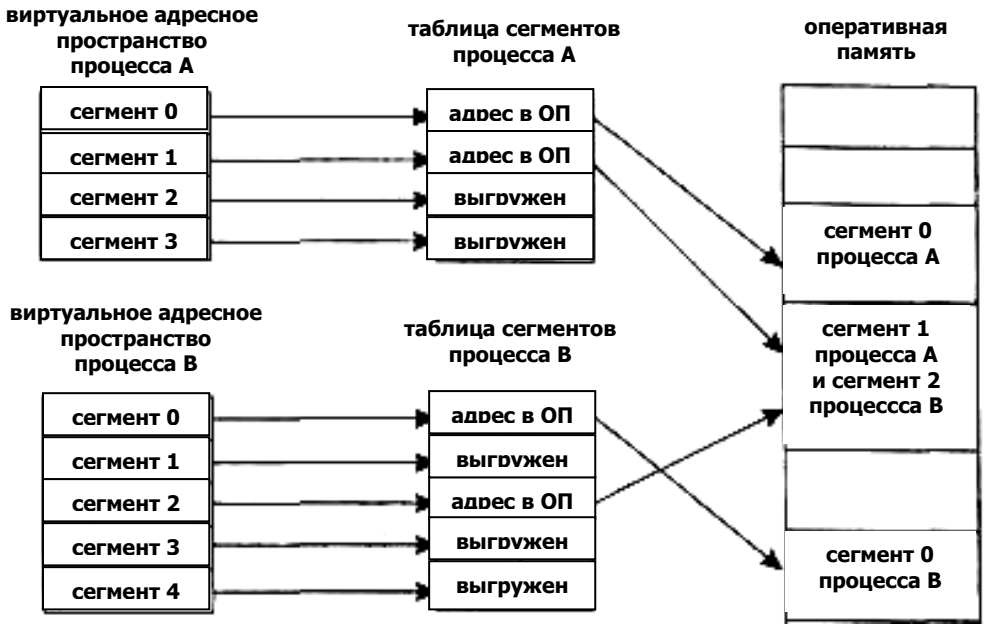



Рис. 5.14. Распределение памяти сегментами

При загрузке процесса в оперативную память помещается только часть его сегментов, полная копия виртуального адресного пространства находится в дисковой памяти. Для каждого загружаемого сегмента операционная система подыскивает непрерывный участок свободной памяти достаточного размера. Смежные в виртуальной памяти сегменты одного процесса могут занимать в оперативной памяти несмежные участки. Если во время выполнения процесса происходит обращение по виртуальному адресу, относящемуся к сегменту, который в данный момент отсутствует в памяти, то происходит прерывание. Операционная система приостанавливает активный процесс, запускает на выполнение следующий процесс из очереди, а параллельно организует загрузку нужного сегмента с диска. При отсутствии в памяти места, необходимого для загрузки сегмента, ОС выбирает сегмент на выгрузку, используя при этом критерии, аналогичные рассмотренным выше критериям выбора страниц при страничном способе управления памятью.

На этапе создания процесса во время загрузки его образа в оперативную память система создает *таблицу сегментов* процесса (аналогичную таблице страниц), в которой для каждого сегмента указывается:

– базовый физический адрес сегмента в оперативной памяти;

– размер сегмента;

– правила доступа к сегменту;

– признаки модификации, присутствия и обращения к данному сегменту, а также служебная информация.

Если виртуальные адресные пространства нескольких процессов включают один и тот же сегмент, то в таблицах сегментов этих процессов делаются ссылки на один и тот же участок оперативной памяти, в который данный сегмент загружается в единственном экземпляре.

Таким образом, сегментное распределение памяти имеет очень много общего со страничным распределением.

Механизмы преобразования адресов этих двух способов управления памятью тоже весьма схожи, однако в них имеются и существенные отличия, которые являются следствием того, что сегменты в отличие от страниц имеют произвольный размер. Виртуальный адрес при сегментной организации памяти может быть представлен парой ($g$, $s$), где $g$ – номер сегмента, $as$ – смещение в сегменте. Физический адрес получается путем *сложения* базового адреса сегмента, который определяется по номеру сегмента $g$ из таблицы сегментов и смещения $s$ (рис. 5.15).

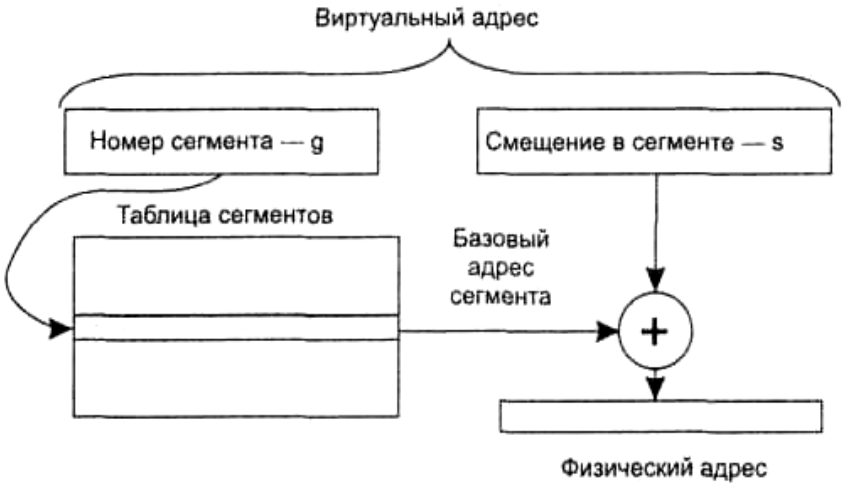



Рис. 5.15. Преобразование виртуального адреса при сегментной организации памяти

В данном случае нельзя обойтись операцией конкатенации, как это делается при страничной организации памяти. Поскольку размер страницы равен степени двойки, в двоичном виде он выражается числом с несколькими нулями в младших разрядах. Страницы имеют одинаковый размер, а значит, их начальные адреса кратны размеру страниц и выражаются также числами с нулями в младших разрядах. Поэтому ОС вносит в таблицы страниц не полные адреса, а номера физических страниц, которые совпадают со старшими разрядами базовых адресов. Сегмент же может в общем случае располагаться в физической памяти начиная с любого адреса, следовательно, для определения местоположения в памяти необходимо задавать его полный начальный физический адрес. Использование операции

сложения вместо конкатенации замедляет процедуру преобразования виртуального адреса в физический по сравнению со страничной организацией.

Другим недостатком сегментного распределения является избыточность. При сегментной организации единицей перемещения между памятью и диском является сегмент, имеющий в общем случае объем больший, чем страница. Однако во многих случаях для работы программы вовсе не требуется загружать весь сегмент целиком, достаточно было бы одной или двух страниц. Аналогично при отсутствии свободного места в памяти не стоит выгружать целый сегмент, когда можно обойтись выгрузкой нескольких страниц.

Но главный недостаток сегментного распределения – это фрагментация, возникающая из-за непредсказуемости размеров сегментов. В процессе работы системы в памяти образуются небольшие участки свободной памяти, в которые не может быть загружен ни один сегмент. Суммарный объем, занимаемый фрагментами, может составить существенную часть общей памяти системы, приводя к ее неэффективному использованию.

Система с сегментной организацией функционирует аналогично системе со страничной организацией: при каждом обращении к оперативной памяти выполняется преобразование виртуального адреса в физический, время от времени происходят прерывания, связанные с отсутствием нужных сегментов в памяти, при необходимости освобождения памяти некоторые сегменты выгружаются.

Одним из существенных отличий сегментной организации памяти от страничной является возможность задания дифференцированных прав доступа процесса к его сегментам. Например, один сегмент данных, содержащий исходную информацию для приложения, может иметь права доступа «только чтение», а сегмент данных, представляющий результаты, – «чтение и запись». Это свойство дает принципиальное преимущество сегментной модели памяти над страничной.

### 5.4.3. Сегментно-страничное распределение

Данный метод представляет собой комбинацию страничного и сегментного механизмов управления памятью и использует достоинства обоих подходов.

Так же, как и при сегментной организации памяти, виртуальное адресное пространство процесса разделено на сегменты. Это позволяет определять разные права доступа к разным частям кодов и данных программы.

Перемещение данных между памятью и диском осуществляется не сегментами, а страницами. Для этого каждый виртуальный сегмент и фи-

зическая память делятся на страницы равного размера, что позволяет более эффективно использовать память, сократив до минимума фрагментацию.

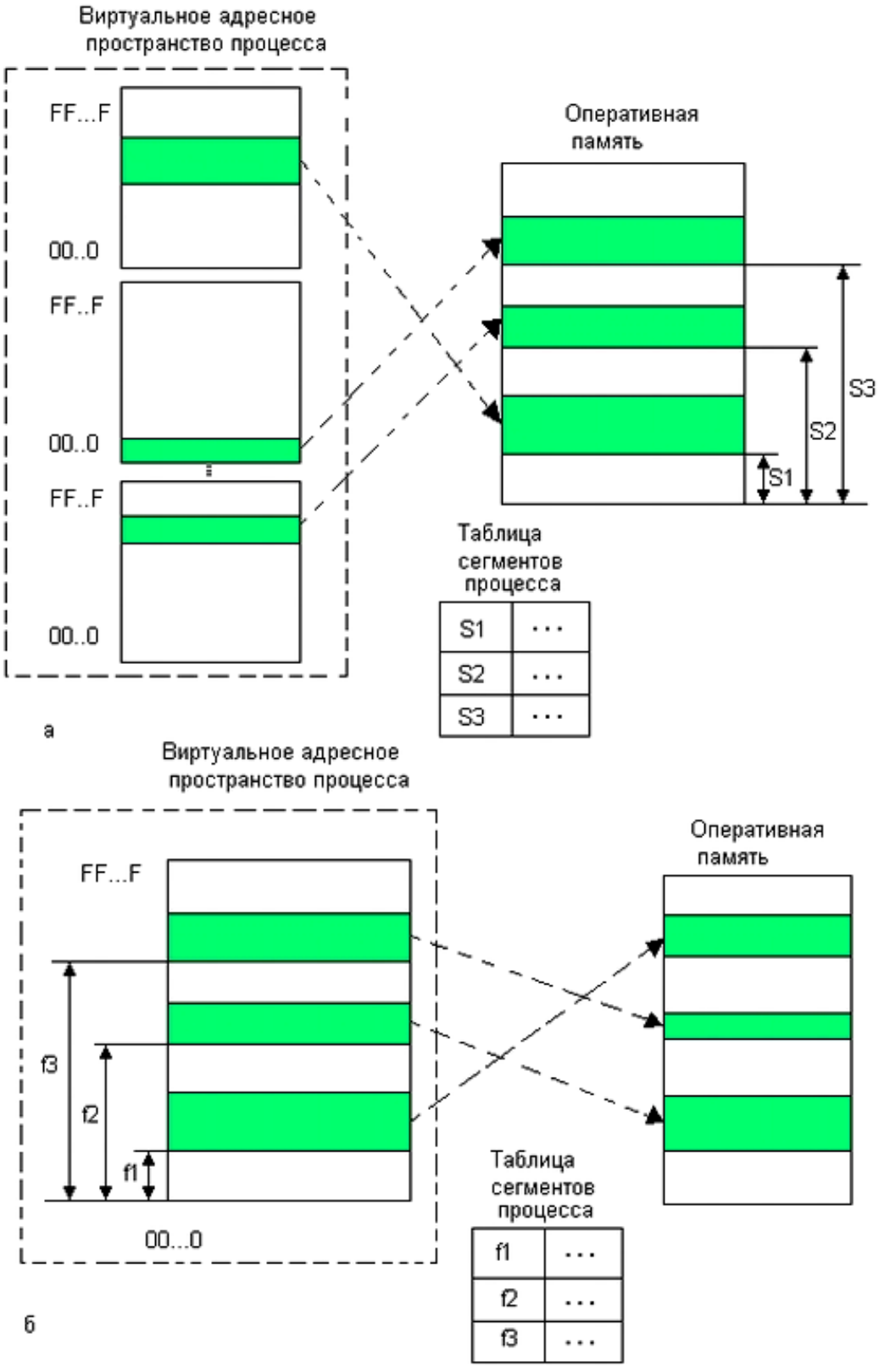



Рис. 5.16. Способы сегментации

В большинстве современных реализаций сегментно-страничной организации памяти в отличие от набора виртуальных диапазонов адресов при сегментной организации памяти (рис. 5.16, *а*) все виртуальные сегменты образуют одно непрерывное линейное виртуальное адресное пространство (рис. 5.16, *б*).

Координаты байта в виртуальном адресном пространстве при сегментно-страничной организации можно задать двумя способами. Во-первых, линейным виртуальным адресом, который равен сдвигу данного байта относительно границы общего линейного виртуального пространства, во-вторых, парой чисел, одно из которых является номером сегмента, а другое – смещением относительно начала сегмента. При этом, в отличие от сегментной модели, для однозначного задания виртуального адреса вторым способом необходимо указать также начальный виртуальный адрес сегмента с данным номером. Системы виртуальной памяти ОС с сегментно-страничной организацией используют второй способ, так как он позволяет непосредственно определить принадлежность адреса некоторому сегменту и проверить права доступа процесса к нему.

Для каждого процесса операционная система создает отдельную таблицу сегментов, в которой содержатся описатели (дескрипторы) всех сегментов процесса. Описание сегмента включает назначенные ему права доступа и другие характеристики, подобные тем, которые содержатся в дескрипторах сегментов при сегментной организации памяти. Однако имеется и принципиальное отличие. В поле базового адреса указывается не начальный физический адрес сегмента, отведенный ему в результате загрузки в оперативную память, а начальный линейный виртуальный адрес сегмента в пространстве виртуальных адресов (на рис. 5.16 базовые физические адреса обозначены S1, S2, S3, а базовые виртуальные адреса – fl, f2, f3).

Наличие базового виртуального адреса сегмента в дескрипторе позволяет однозначно преобразовать адрес, заданный в виде пары (номер сегмента, смещение в сегменте), в линейный виртуальный адрес байта, который затем преобразуется в физический адрес страничным механизмом.

Деление общего линейного виртуального адресного пространства процесса и физической памяти на страницы осуществляется так же, как это делается при страничной организации памяти. Размер страниц выбирается равным степени двойки, что упрощает механизм преобразования виртуальных адресов в физические. Виртуальные страницы нумеруются в пределах виртуального адресного пространства каждого процесса, а физические страницы – в пределах оперативной памяти. При создании процесса

в память загружается только часть страниц, остальные загружаются по мере необходимости. Время от времени система выгружает уже ненужные страницы, освобождая память для новых страниц. Операционная система ведет для каждого процесса таблицу страниц, в которой указывается соответствие виртуальных страниц физическим.

Базовые адреса таблицы сегментов и таблицы страниц процесса являются частью его контекста. При активизации процесса эти адреса загружаются в специальные регистры процессора и используются механизмом преобразования адресов.

Преобразование виртуального адреса в физический происходит в два этапа (рис. 5.17):

1) На первом этапе работает механизм сегментации. Исходный виртуальный адрес, заданный в виде пары (номер сегмента, смещение), преобразуется в линейный виртуальный адрес. Для этого на основании базового адреса таблицы сегментов и номера сегмента вычисляется адрес дескриптора сегмента. Анализируются поля дескриптора, выполняется проверка возможности выполнения заданной операции. Если доступ к сегменту разрешен, то вычисляется линейный виртуальный адрес путем сложения базового адреса сегмента, извлеченного из дескриптора, и смещения, заданного в исходном виртуальном адресе.

2) На втором этапе работает страничный механизм. Полученный линейный виртуальный адрес преобразуется в искомый физический адрес. В результате преобразования линейный виртуальный адрес представляется в том виде, в котором он используется при страничной организации памяти, – в виде пары (номер страницы, смещение в странице). Благодаря тому, что размер страницы выбран равным степени двойки, эта задача решается простым отделением некоторого количества младших двоичных разрядов. При этом в старших разрядах содержится номер виртуальной страницы, а в младших – смещение искомого элемента относительно начала страницы. Так, если размер страницы равен $2^k$, то смещением является содержимое младших $k$ разрядов, а остальные, старшие разряды содержат номер виртуальной страницы, которой принадлежит искомый адрес. Далее преобразование адреса происходит так же, как при страничной организации: старшие разряды линейного виртуального адреса, содержащие номер виртуальной страницы, заменяются номером физической страницы, взятым из таблицы страниц, а младшие разряды виртуального адреса, содержащие смещение, остаются без изменения.

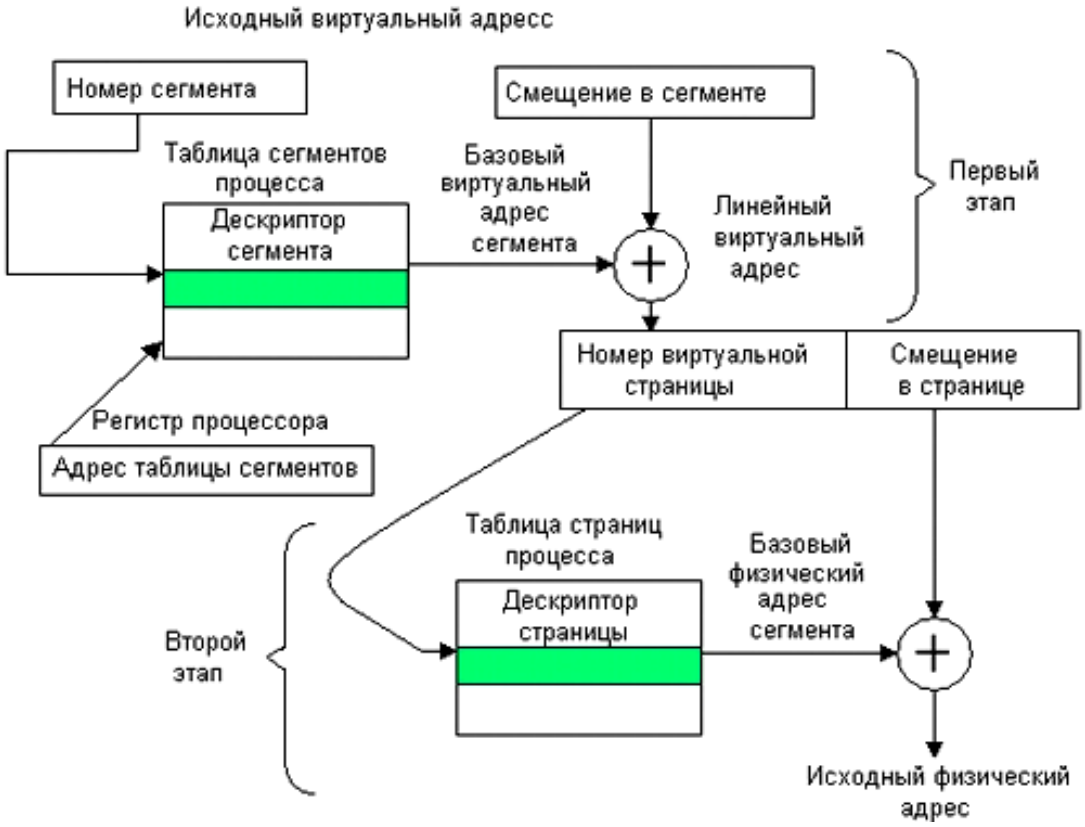



Рис. 5.17. Преобразование виртуального адреса в физический при сегментно-страничной организации памяти

Механизм сегментации и страничный механизм действуют достаточно независимо друг от друга. Поэтому нетрудно представить себе реализацию сегментно-страничного управления памятью, в которой механизм сегментации работает по вышеописанной схеме, а страничный механизм изменен. Он реализует двухуровневую схему, в которой виртуальное адресное пространство делится сначала на разделы, а затем на страницы. В таком случае преобразование виртуального адреса в физический происходит в несколько этапов. Сначала механизм сегментации обычным образом, используя таблицу сегментов, вычисляет линейный виртуальный адрес. Затем из данного виртуального адреса вычленяются номер раздела, номер страницы и смещение. Далее по номеру раздела из таблицы разделов определяется адрес таблицы страниц, а затем по номеру виртуальной страницы из таблицы страниц определяется номер физической страницы, к которому пристыковывается смещение. Именно такой подход реализован компанией Intel в процессорах i386, i486 и Pentium.

Рассмотрим еще одну возможную схему управления памятью, основанную на комбинировании сегментного и страничного механизма. Так же, как и в предыдущих случаях, виртуальное пространство процесса делится

на сегменты, а каждый сегмент, в свою очередь, делится на виртуальные страницы. Основное отличие состоит в том, что виртуальные страницы нумеруются не в пределах всего адресного пространства процесса, а в пределах сегмента. Виртуальный адрес в этом случае выражается тройкой (номер сегмента, номер страницы, смещение в странице).

Загрузка процесса выполняется операционной системой постранично, при этом часть страниц размещается в оперативной памяти, а часть – на диске. Для каждого процесса создается собственная таблица сегментов, а для каждого сегмента – своя таблица страниц. Адрес таблицы сегментов загружается в специальный регистр процессора, когда активизируется соответствующий процесс.

Таблица страниц содержит дескрипторы страниц, содержимое которых полностью аналогично содержимому ранее описанных дескрипторов страниц. А вот таблица сегментов состоит из дескрипторов сегментов, которые вместо информации о расположении сегментов в виртуальном адресном пространстве содержат описание расположения таблиц страниц в физической памяти. Это является вторым существенным отличием данного подхода от ранее рассмотренной схемы сегментно-страничной организации.

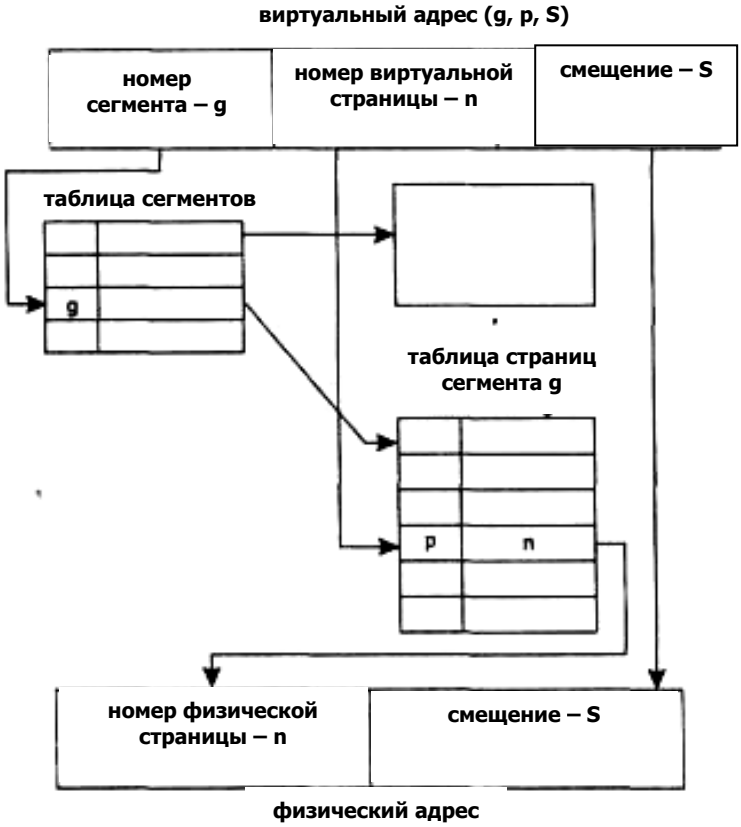



Рис. 5.18. Схема преобразования виртуального адреса в физический для сегментно-страничной организации памяти

На рис. 5.18 показана схема преобразования виртуального адреса в физический для данного метода.

1. По номеру сегмента, заданному в виртуальном адресе, из таблицы сегментов извлекается физический адрес соответствующей таблицы страниц.

2. По номеру виртуальной страницы, заданному в виртуальном адресе, из таблицы страниц извлекается дескриптор, в котором указан номер физической страницы.

3. К номеру физической страницы пристыковывается младшая часть виртуального адреса – смещение.

## 5.5. Разделяемые сегменты памяти

Подсистема виртуальной памяти представляет собой удобный механизм для решения задачи совместного доступа нескольких процессов к одному и тому же сегменту памяти, который в этом случае называется *разделяемой памятью* (*shared memory*).

Хотя основной задачей операционной системы при управлении памятью является защита областей оперативной памяти, принадлежащей одному из процессов, от доступа к ней остальных процессов, в некоторых случаях оказывается полезным организовать контролируемый совместный доступ нескольких процессов к определенной области памяти. Например, в том случае, когда несколько пользователей одновременно работают с некоторым текстовым редактором, нецелесообразно многократно загружать его код в оперативную память. Гораздо экономичней загрузить одну копию кода, которая обслуживала бы всех пользователей, работающих в данное время с этим редактором – для этого код редактора должен быть реентерабельным. Очевидно, что сегмент данных редактора не может присутствовать в памяти в единственном разделяемом экземпляре: для каждого пользователя должна быть создана копия этого сегмента, в которой помещается редактируемый текст и значения других переменных редактора, например, его конфигурация, индивидуальная для каждого пользователя.

Другим примером применения разделяемой области памяти может быть использование ее в качестве буфера при межпроцессном обмене данными. В этом случае один процесс пишет в разделяемую область, а другой – читает.

Для организации разделяемого сегмента при наличии подсистемы виртуальной памяти достаточно поместить его в виртуальное адресное пространство каждого процесса, которому нужен доступ к данному сегменту, а затем настроить параметры отображения этих виртуальных сегментов так, чтобы они соответствовали одной и той же области оперативной памяти. Детали такой настройки зависят от типа используемой в ОС модели виртуаль-

ной памяти: сегментной или сегментно-страничной. Чисто страничная организация не поддерживает понятие «сегмент», что делает невозможным решение рассматриваемой задачи. Например, при сегментной организации необходимо в дескрипторах виртуального сегмента каждого процесса указать один и тот же базовый физический адрес. При сегментно-страничной организации отображение на одну и ту же область памяти достигается за счет соответствующей настройки таблицы страниц каждого процесса.

В приведенном выше описании подразумевалось, что разделяемый сегмент помещается в индивидуальную часть виртуального адресного пространства каждого процесса (рис. 5.19, *а*) и описывается в каждом процессе индивидуальным дескриптором сегмента и индивидуальными дескрипторами страниц, если используется сегментно-страничный механизм. «Попадание» же этих виртуальных сегментов на общую часть оперативной памяти достигается за счет согласованной настройки операционной системой многочисленных дескрипторов для множества процессов.

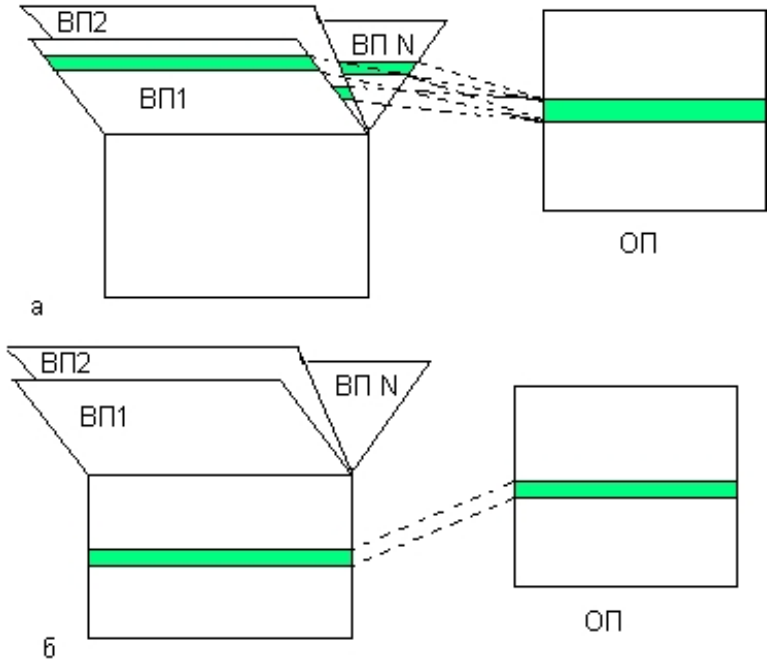



Рис. 5.19. Способы создания разделяемого сегмента памяти

Возможно и более экономичное для ОС решение этой задачи – помещение единственного разделяемого виртуального сегмента в общую часть виртуального адресного пространства процессов, т. е. в ту часть, которая обычно используется для модулей ОС (рис. 5.19, *б*). В этом случае настройка дескриптора сегмента и дескрипторов страниц выполняется только один раз, а все процессы пользуются этой настройкой и совместно используют часть оперативной памяти.

При работе с разделяемыми сегментами памяти ОС должна выполнять некоторые функции, общие для любых разделяемых между процессами ресурсов – файлов, семафоров и т. п. Эти функции состоят в поддержке схемы именования ресурсов, проверке прав доступа определенного процесса к ресурсу, а также в отслеживании количества процессов, пользующихся данным ресурсом, чтобы удалить его в случае ненадобности. Для того чтобы отличать разделяемые сегменты памяти от индивидуальных, дескриптор сегмента должен содержать поле, имеющее два значения: shared (разделяемый) или private (индивидуальный).

Операционная система может создавать разделяемые сегменты как по явному запросу, так и по умолчанию. В первом случае прикладной процесс должен выполнить соответствующий системный вызов, по которому ОС создает новый сегмент в соответствии с указанными в вызове параметрами: размером сегмента, разрешенными над ним операциями (чтение / запись) и идентификатором. Все процессы, выполнившие подобные вызовы с одним и тем же идентификатором, получают доступ к этому сегменту и используют его по своему усмотрению, например, в качестве буфера для обмена данными.

Во втором случае ОС сама в определенных ситуациях принимает решение о том, что нужно создать разделяемый сегмент. Типичным примером является поступление нескольких запросов на выполнение одного и того же приложения. Если кодовый сегмент приложения помечен в исполняемом файле как реентерабельный и разделяемый, то ОС не создает при поступлении нового запроса новую копию кодового сегмента этого приложения, а отображает уже существующий разделяемый сегмент в виртуальное адресное пространство процесса. При закрытии приложения каким-либо процессом ОС проверяет, существуют ли другие процессы, пользующиеся данным приложением, и если их нет, то удаляет данный разделяемый сегмент.

Разделяемые сегменты выгружаются на диск системой виртуальной памяти по тем же алгоритмам и с помощью тех же механизмов, что и индивидуальные.

## 5.6. Кэширование данных

### 5.6.1. Иерархия запоминающих устройств

Память вычислительной машины представляет собой иерархию запоминающих устройств (ЗУ), отличающихся средним временем доступа к данным, объемом и стоимостью хранения одного бита (рис. 5.20). Фунда-

ментом этой пирамиды запоминающих устройств служит внешняя память, как правило, представляемая жестким диском. Она имеет большой объем (десятки и сотни гигабайт), но скорость доступа к данным невысока. Время доступа к диску измеряется миллисекундами.

На следующем уровне располагается более быстродействующая (время доступа равно примерно 10 – 20 наносекундам) и менее объемная (от десятков мегабайт до нескольких гигабайт) оперативная память, реализуемая на относительно медленной динамической памяти DRAM.

Для хранения данных, к которым необходимо обеспечить быстрый доступ, используются компактные быстродействующие запоминающие устройства на основе статической памяти SRAM, объем которых составляет от нескольких десятков до нескольких сотен килобайт, а время доступа к данным обычно не превышает 8 нс.

И наконец, верхний уровень составляют внутренние регистры процессора, которые также могут быть использованы для промежуточного хранения данных. Общий объем регистров составляет несколько десятков байт, а время доступа определяется быстродействием процессора и равно в настоящее время примерно 2 – 3 нс.

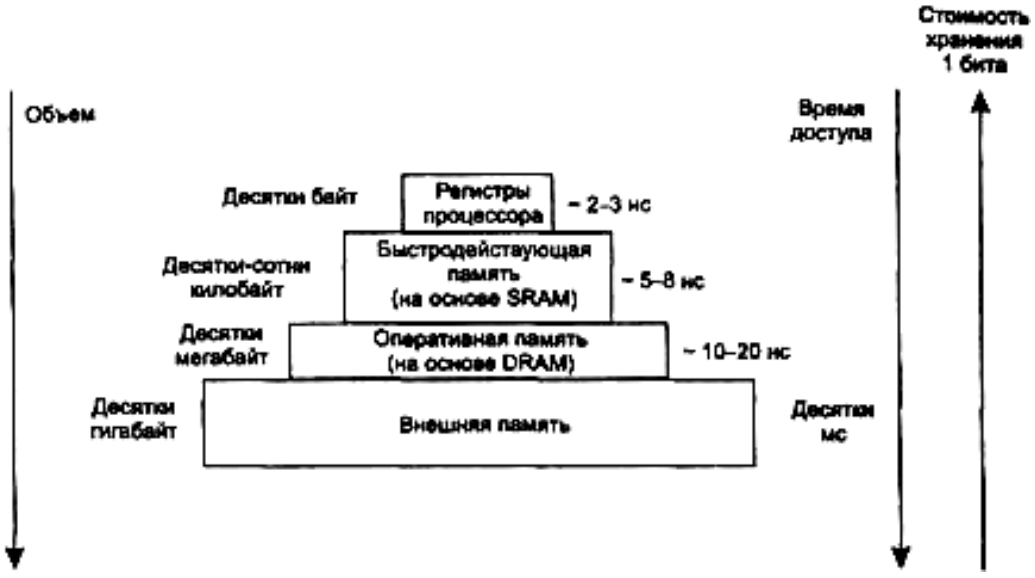



Рис. 5.20. Иерархия запоминающих устройств

Таким образом, можно констатировать закономерность: чем больше объем устройства, тем менее быстродействующим оно является. Более того, стоимость хранения данных в расчете на один бит также увеличивается с ростом быстродействия устройств. Кэш-память представляет некоторое компромиссное решение этой проблемы.

### 5.6.2. Кэш-память

*Кэш-память*, или *кэш (cache*), – это способ совместного функционирования двух типов запоминающих устройств, отличающихся временем доступа и стоимостью хранения данных, который за счет динамического копирования в «быстрое» ЗУ наиболее часто используемой информации из «медленного» ЗУ позволяет, с одной стороны, уменьшить среднее время доступа к данным, а с другой – экономить более дорогую быстродействующую память.

Неотъемлемым свойством кэш-памяти является ее прозрачность для программ и пользователей. Система не требует никакой внешней информации об интенсивности использования данных; ни пользователи, ни программы не принимают никакого участия в перемещении данных из ЗУ одного типа в ЗУ другого типа – это делается автоматически системными средствами.

Кэш-памятью или кэшем часто называют не только способ организации работы двух типов запоминающих устройств, но и одно из устройств – «быстрое» ЗУ. Оно стоит дороже и, как правило, имеет сравнительно небольшой объем. «Медленное» ЗУ далее будем называть основной памятью, противопоставляя ее вспомогательной кэш-памяти.

Кэширование – это универсальный метод, пригодный для ускорения доступа к оперативной памяти, к диску и к другим видам запоминающих устройств. Если кэширование применяется для уменьшения среднего времени доступа к оперативной памяти, то в качестве кэша используют быстродействующую статическую память. Если кэширование используется системой ввода-вывода для ускорения доступа к данным, хранящимся на диске, то роль кэш-памяти выполняют буферы в оперативной памяти, в которых оседают наиболее активно используемые данные. Виртуальную память также можно считать одним из вариантов реализации принципа кэширования данных, при котором оперативная память выступает в роли кэша по отношению к внешней памяти – жесткому диску. Правда, в этом случае кэширование используется не для того, чтобы уменьшить время доступа к данным, а для того, чтобы заставить диск частично подменить оперативную память за счет перемещения временно неиспользуемого кода и данных на диск с целью освобождения места для активных процессов. В результате наиболее интенсивно используемые данные «оседают» в оперативной памяти, остальная же информация хранится в более объемной и менее дорогостоящей внешней памяти.

### 5.6.3. Принцип действия кэш-памяти

Рассмотрим одну из возможных схем кэширования (рис. 5.21). Содержимое кэш-памяти представляет собой совокупность записей обо всех

загруженных в нее элементах данных из основной памяти. Каждая запись об элементе данных включает в себя:

– значение элемента данных;

– адрес, который этот элемент данных имеет в основной памяти;

– дополнительную информацию, которая используется для реализации алгоритма замещения данных в кэше и обычно включает признак модификации и признак действительности данных.

При каждом обращении к основной памяти по физическому адресу просматривается содержимое кэш-памяти с целью определения, не находятся ли там нужные данные. Кэш-память не является адресуемой, поэтому поиск нужных данных осуществляется по содержимому – по взятому из запроса значению поля адреса в оперативной памяти. Далее возможен один из двух вариантов развития событий:

– если данные обнаруживаются в кэш-памяти, т. е. произошло *кэш-попадание* (*cache-hit*), они считываются оттуда и результат передается источнику запроса;

– если нужные данные отсутствуют в кэш-памяти, т. е. произошел *кэш-промах* (*cache-miss*), они считываются из основной памяти, передаются источнику запроса и одновременно с этим копируются в кэш-память.


Структура кэш-памяти

| Адрес данных в основной памяти | Данные | Управляющая информация |
|---|---|---|
| | | |
| | | |
| | | |

Рис. 5.21. Схема функционирования кэш-памяти

Эффективность кэширования зависит от вероятности попадания в кэш. Докажем это путем нахождения зависимости среднего времени доступа к основной памяти от вероятности кэш-попаданий. Пусть имеется основное запоминающее устройство со средним временем доступа к данным

$t_l$ и кэш-память, имеющая время доступа $t_2$, очевидно, что $t_2 < t_l$. Пусть t – среднее время доступа к данным в системе с кэш-памятью, $a_p$ – вероятность кэш-попадания. По формуле полной вероятности имеем:

$$t – t,U – P) + t_2p – (t_2 –t,)p + tj$$

Среднее время доступа к данным в системе с кэш-памятью линейно зависит от вероятности попадания в кэш и изменяется от среднего времени доступа в основное запоминающее устройство $t_l$ при p = 0 до среднего времени доступа непосредственно в кэш-память $t_2$ при p = 1. Отсюда видно, что использование кэш-памяти имеет смысл только при высокой вероятности кэш-попадания.

Вероятность обнаружения данных в кэше зависит от разных факторов, таких, например, как объем кэша, объем кэшируемой памяти, алгоритм замещения данных в кэше, особенности выполняемой программы, время ее работы, уровень мультипрограммирования и других особенностей вычислительного процесса. Тем не менее, в большинстве реализаций кэш-памяти процент кэш-попаданий оказывается весьма высоким – свыше 90 %. Такое высокое значение вероятности нахождения данных в кэш-памяти объясняется наличием у данных объективных свойств: пространственной и временной локальности.

– *Временная локальность*. Если произошло обращение по некоторому адресу, то следующее обращение по тому же адресу с большой вероятностью произойдет в ближайшее время.

– *Пространственная локальность*. Если произошло обращение по некоторому адресу, то с высокой степенью вероятности в ближайшее время произойдет обращение к соседним адресам.

Основываясь на свойстве временной локальности, данные, только что считанные из основной памяти, размещают в запоминающем устройстве быстрого доступа, предполагая, что скоро они опять понадобятся. В начале работы системы, когда кэш-память еще пуста, почти каждый запрос к основной памяти выполняется «по полной программе»: просмотр кэша, констатация промаха, чтение данных из основной памяти, передача результата источнику запроса и копирование данных в кэш. Затем, по мере заполнения кэша, в полном соответствии со свойством временной локальности возрастает вероятность обращения к данным, которые уже были использованы на предыдущем этапе работы системы, т. е. к данным, которые содержатся в кэше и могут быть считаны значительно быстрее, чем из основной памяти.

Свойство пространственной локальности также используется для увеличения вероятности кэш-попадания: как правило, в кэш-память счи-

тывается не один информационный элемент, к которому произошло обращение, а целый блок данных, расположенных в основной памяти в непосредственной близости с данным элементом. Поскольку при выполнении программы очень высока вероятность, что команды выбираются из памяти последовательно одна за другой из соседних ячеек, то имеет смысл загружать в кэш-память целый фрагмент программы. Аналогично если программа ведет обработку некоторого массива данных, то ее работу можно ускорить, загрузив в кэш часть или даже весь массив. При этом учитывается высокая вероятность того, что значительное число обращений к памяти будет выполняться к адресам массива данных.

### 5.6.2. Проблема согласования данных

В процессе работы содержимое кэш-памяти постоянно обновляется, а значит, время от времени данные из нее должны вытесняться. Вытеснение означает либо объявление свободной соответствующей области кэш-памяти (сброс бита действительности), если вытесняемые данные за время нахождения в кэше не были изменены, либо в дополнение к этому копирование данных в основную память, если они были модифицированы. Алгоритм замены данных в кэш-памяти существенно влияет на ее эффективность. В идеале такой алгоритм должен, во-первых, быть максимально быстрым, чтобы не замедлять работу кэш-памяти, а во-вторых, обеспечивать максимально возможную вероятность кэш-попаданий. Поскольку из-за непредсказуемости вычислительного процесса ни один алгоритм замещения данных в кэш-памяти не может гарантировать оптимальный результат, разработчики ограничиваются рациональными решениями, которые, по крайней мере, не очень замедляют работу кэша – запоминающего устройства, изначально призванного быть быстрым.

Наличие в компьютере двух копий данных – в основной памяти и в кэше – порождает проблему согласования данных. Если происходит запись в основную память по некоторому адресу, а содержимое этой ячейки находится в кэше, то в результате соответствующая запись в кэше становится недостоверной. Рассмотрим два подхода к решению этой проблемы.

1) *Сквозная запись* (*write through*). При каждом запросе к основной памяти, в том числе и при записи, просматривается кэш. Если данные по запрашиваемому адресу отсутствуют, то запись выполняется только в основную память. Если же данные, к которым выполняется обращение, находятся в кэше, то запись выполняется одновременно в кэш и основную память.

2) *Обратная запись* (*write back*). При возникновении запроса к памяти выполняется просмотр кэша, и если запрашиваемых данных там нет, то за-

пись выполняется только в основную память. В противном же случае запись производится *только в кэш-память*, при этом в описателе данных делается специальная отметка (признак модификации), которая указывает на то, что при вытеснении этих данных из кэша необходимо переписать их в основную память, чтобы актуализировать устаревшее содержимое основной памяти.

В некоторых алгоритмах замещения предусматривается первоочередная выгрузка модифицированных, или, как еще говорят, «грязных» данных. Модифицированные данные могут выгружаться не только при освобождении места в кэш-памяти для новых данных, но и в «фоновом режиме», когда система не очень загружена.

### 5.6.4. Способы отображения основной памяти на кэш

Алгоритмы поиска и замещения данных в кэше зависят от того, каким образом основная память отображается на кэш-память. Принцип прозрачности требует, чтобы отображение основной памяти на кэш-память не зависело от работы программ и пользователей. При кэшировании данных из оперативной памяти широко используются две основные схемы: случайное отображение и детерминированное отображение.

При *случайном* отображении элемент оперативной памяти в общем случае может быть размещен в произвольном месте кэш-памяти. Для того чтобы в дальнейшем можно было найти нужные данные в кэше, они помещаются туда вместе со своим адресом, т. е. тем адресом, который данные имеют в оперативной памяти. При каждом запросе к оперативной памяти выполняется поиск в кэше, причем критерием поиска выступает адрес оперативной памяти из запроса. Очевидная схема простого перебора для поиска нужных данных в случае кэша оказывается непригодной из-за недопустимо больших временных затрат.

Для кэшей со случайным отображением используется так называемый *ассоциативный поиск*, при котором сравнение выполняется не последовательно с каждой записью кэша, а параллельно со всеми его записями (рис. 5.22). Признак, по которому выполняется сравнение, называется *тегом* (*tag*). В данном случае тегом является адрес данных в оперативной памяти. Электронная реализация такой схемы приводит к удорожанию памяти, причем стоимость существенно возрастает с увеличением объема запоминающего устройства. Поэтому ассоциативная кэш-память используется в тех случаях, когда для обеспечения высокого процента попадания достаточно небольшого объема памяти.

В кэшах, построенных на основе случайного отображения, вытеснение старых данных происходит только в том случае, когда вся кэш-память

заполнена и нет свободного места. Выбор данных на выгрузку осуществляется среди всех записей кэша. Обычно этот выбор основывается на тех же приемах, что и в алгоритмах замещения страниц, например выгрузка данных, к которым дольше всего не было обращений, или данных, к которым было меньше всего обращений.

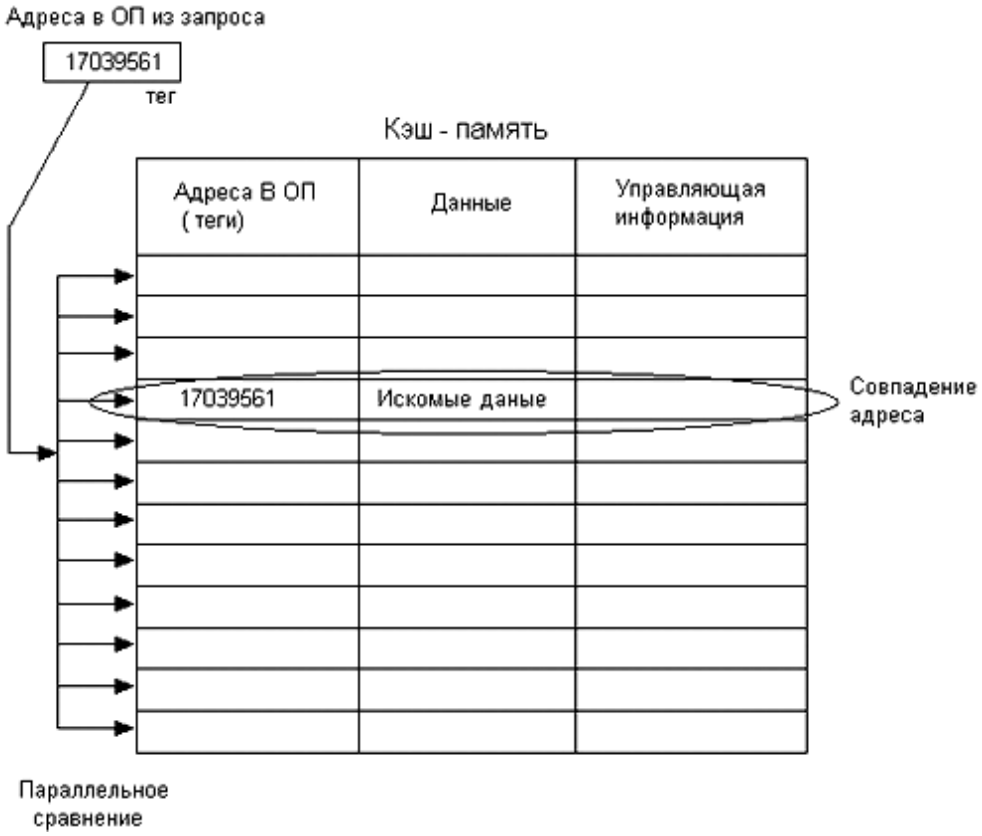



Рис. 5.22. Ассоциативный поиск в кэше со случайным отображением

Второй, *детерминированный* способ отображения предполагает, что любой элемент основной памяти всегда отображается в одно и то же место кэш-памяти. В этом случае кэш-память разделена на строки, каждая из которых предназначена для хранения одной записи об одном элементе данных и имеет свой номер. Между номерами строк кэш-памяти и адресами оперативной памяти устанавливается соответствие «один ко многим»: одному номеру строки соответствует несколько (обычно достаточно много) адресов оперативной памяти.

В качестве отображающей функции может использоваться простое выделение нескольких разрядов из адреса оперативной памяти, которые интерпретируются как номер строки кэш-памяти, такое отображение называется *прямым*. Например, пусть в кэш-памяти может храниться 1024 записи, т. е. кэш имеет 1024 строки, пронумерованные от 0 до 1023. Тогда

любой адрес оперативной памяти может быть отображен на адрес кэш-памяти простым отделением 10 двоичных разрядов (рис. 5.23).

При поиске данных в кэше используется быстрый прямой доступ к записи по номеру строки, полученному путем обработки адреса оперативной памяти из запроса. Однако, поскольку в найденной строке могут находиться данные из любой ячейки оперативной памяти, младшие разряды адреса которой совпадают с номером строки, необходимо выполнить дополнительную проверку. Для этих целей каждая строка кэш-памяти дополняется тегом, содержащим старшую часть адреса данных в оперативной памяти. При совпадении тега с соответствующей частью адреса из запроса констатируется кэш-попадание.

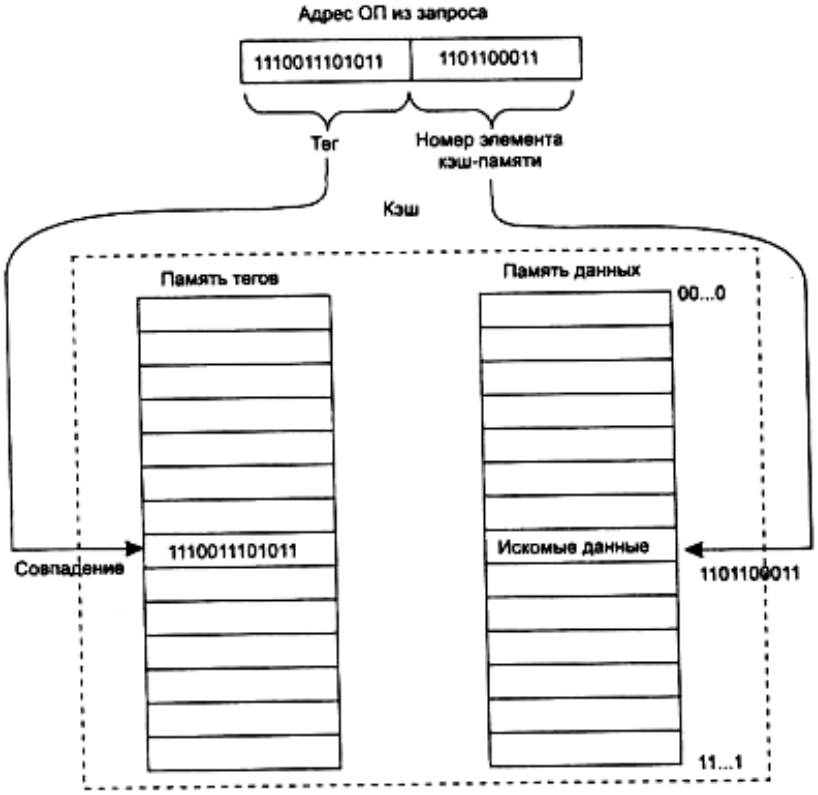



Рис. 5.23. Прямое отображение

Если же произошел кэш-промах, то данные считываются из оперативной памяти и копируются в кэш. Если строка кэш-памяти, в которую должен быть скопирован элемент данных из оперативной памяти, содержит другие данные, то последние вытесняются из кэша. Заметим, что процесс замещения данных в кэш-памяти на основе прямого отображения существенно отличается от процесса замещения данных в кэш-памяти со случайным отображением. Во-первых, вытеснение данных происходит не только в случае отсутствия свободного места в кэше, во-вторых, никакого

выбора данных на замещение не существует. Во многих современных процессорах кэш-память строится на основе сочетания этих двух подходов, что позволяет найти компромисс между сравнительно низкой стоимостью кэша с прямым отображением и интеллектуальностью алгоритмов замещения в кэше со случайным отображением. При смешанном подходе произвольный адрес оперативной памяти отображается не на один адрес кэш-памяти (как это характерно для прямого отображения) и не на любой адрес кэш-памяти (как это делается при случайном отображении), а на некоторую группу адресов. Все группы пронумерованы. Поиск в кэше осуществляется вначале по номеру группы, полученному из адреса оперативной памяти из запроса, а затем в пределах группы путем ассоциативного просмотра всех записей в группе на предмет совпадения старших частей адресов оперативной памяти (рис. 5.24).

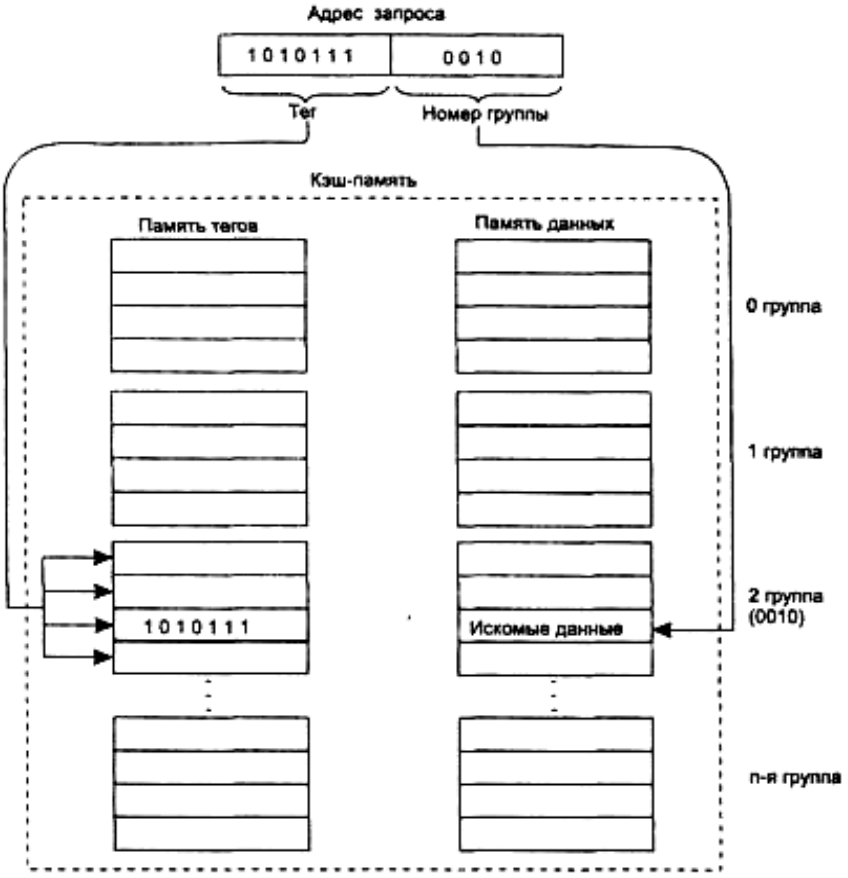



Рис. 5.24. Комбинирование прямого и случайного отображения

При промахе данные копируются по любому свободному адресу из однозначно заданной группы. Если свободных адресов в группе нет, то выполняется вытеснение данных. Поскольку кандидатов на выгрузку несколько – все записи из данной группы, – алгоритм замещения может учесть ин-

тенсивность обращений к данным и тем самым повысить вероятность попаданий в будущем. Таким образом, в данном способе комбинируется прямое отображение на группу и случайное отображение в пределах группы.

### 5.6.5. Схемы выполнения запросов в системах с кэш-памятью

На рис. 5.25 приведена обобщенная схема работы кэш-памяти. Большая часть ветвей этой схемы уже была подробно рассмотрена выше, поэтому остановимся здесь только на некоторых особых случаях.

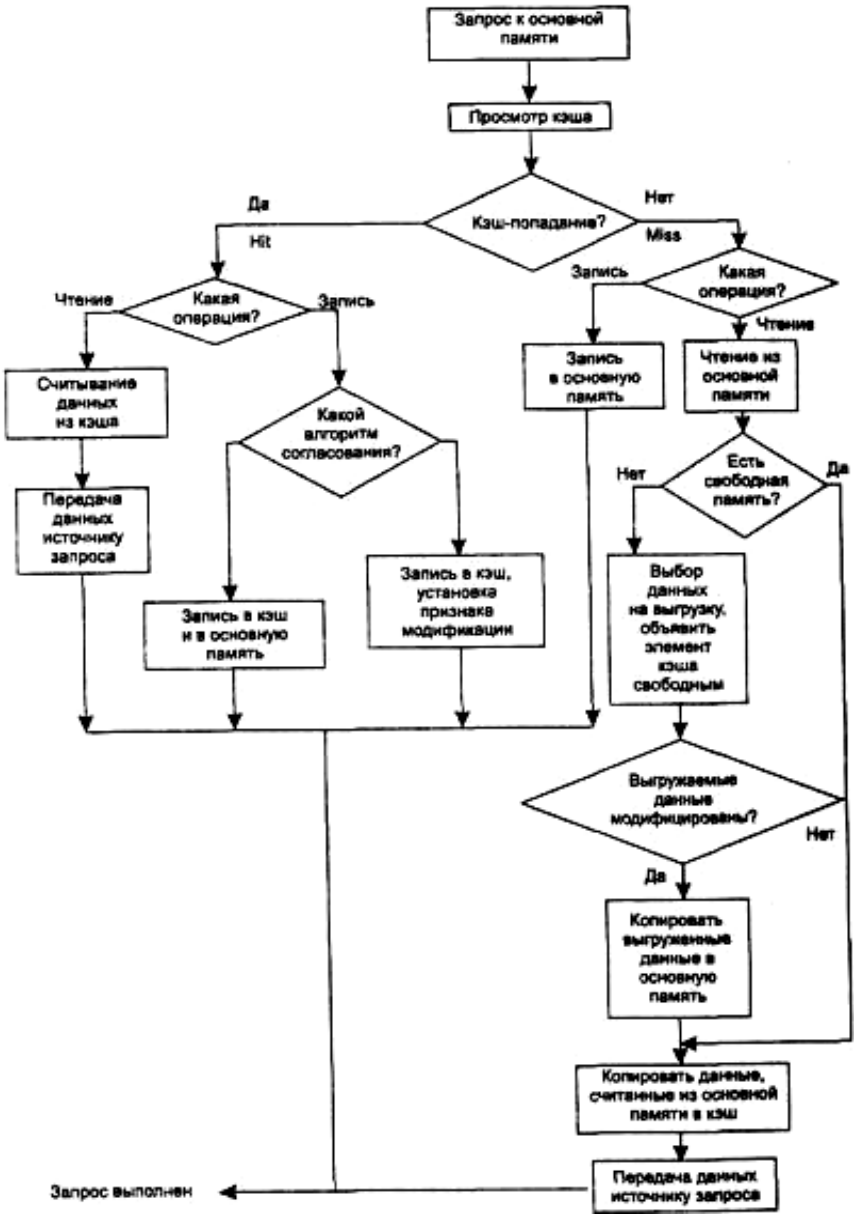



Рис. 5.25. Схема выполнения запроса к памяти в системе, использующей кэширование

Из схемы видно, что, когда выполняется запись, кэш просматривается только с целью согласования содержимого кэша и основной памяти. Ес-

ли происходит промах, то запросы на запись не вызывают никаких изменений содержимого кэша. В некоторых же реализациях кэш-памяти при отсутствии данных в кэше они копируются туда из основной памяти независимо от того, выполняется запрос на чтение или на запись.

В соответствии с описанной логикой работы кэш-памяти следует, что при возникновении запроса сначала просматривается кэш, а затем, если произошел промах, выполняется обращение к основной памяти. Однако часто реализуется и другая схема работы кэша: поиск в кэше и в основной памяти начинается одновременно, а затем в зависимости от результата просмотра кэша операция в основной памяти либо продолжается, либо прерывается.

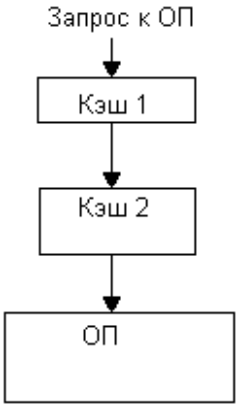



Рис. 5.26. Двухуровневое кэширование

При выполнении запросов к оперативной памяти во многих вычислительных системах используется двухуровневое кэширование. Кэш первого уровня имеет меньший объем и большее быстродействие, чем кэш второго уровня. Кэш второго уровня играет роль основной памяти по отношению к кэшу первого уровня (рис. 5.26).

На рис. 5.27 показана схема выполнения запроса на чтение в системе с двухуровневым кэшем. Сначала делается попытка обнаружить данные в кэше первого уровня. Если произошел промах, поиск продолжается в кэше второго уровня. Если нужные данные отсутствуют и здесь, происходит считывание данных из основной памяти. Время доступа к данным оказывается минимальным, когда кэш-попадание происходит уже на первом уровне, несколько бóльшим при обнаружении данных на втором уровне и обычным временем доступа к оперативной памяти, если нужных данных нет ни в том, ни в другом кэше. При считывании данных из оперативной памяти происходит их копирование в кэш второго уровня, а если данные считываются из кэша второго уровня, то копируются в кэш первого уровня.

При работе такой иерархической организованной памяти необходимо обеспечить непротиворечивость данных на всех уровнях. Кэши разных уровней могут согласовывать данные разными способами. Пусть, например, кэш первого уровня использует сквозную запись, а кэш второго уровня – обратную запись. Именно такая комбинация алгоритмов согласования применена в процессоре Pentium при одном из возможных вариантов его работы.

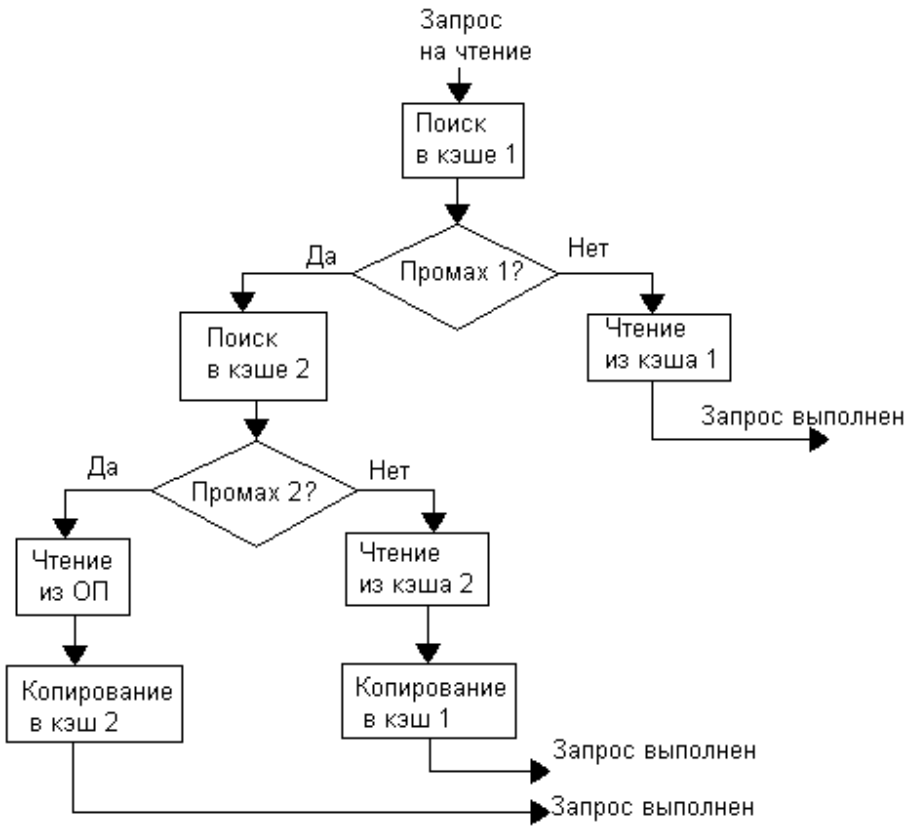



Рис. 5.27. Схема выполнения запроса на чтение в системе с двухуровневым кэшем

На рис. 5.28 приведена схема выполнения запроса на запись в такой системе. При модификации данных необходимо убедиться, что они отсутствуют в кэшах. В этом случае выполняется запись только в оперативную память. Если данные обнаружены в кэше первого уровня, то вступает в силу алгоритм сквозной записи: выполняется запись в кэш первого уровня и передается запрос на запись в кэш второго уровня, играющий в данном случае роль основной памяти. Запись в кэш второго уровня, в соответствии с алгоритмом обратной записи, принятом на данном уровне, сопровождается установкой признака модификации, и при этом никакой записи в оперативную память не производится. Если данные найдены в кэше второго уровня, то, как и в предыдущем случае, выполняется запись в этот кэш и устанавливается признак модификации.

Рассмотренные в данном разделе проблемы кэширования охватывают только такой класс систем организации памяти, в котором на каждом уровне имеется одно каптирующее устройство. Существует и другой класс систем памяти, главной отличительной особенностью которого является наличие нескольких кэшей одного уровня. Этот вариант характерен для распределенных систем обработки информации – мультипроцессорных компьютеров и компьютерных сетей.

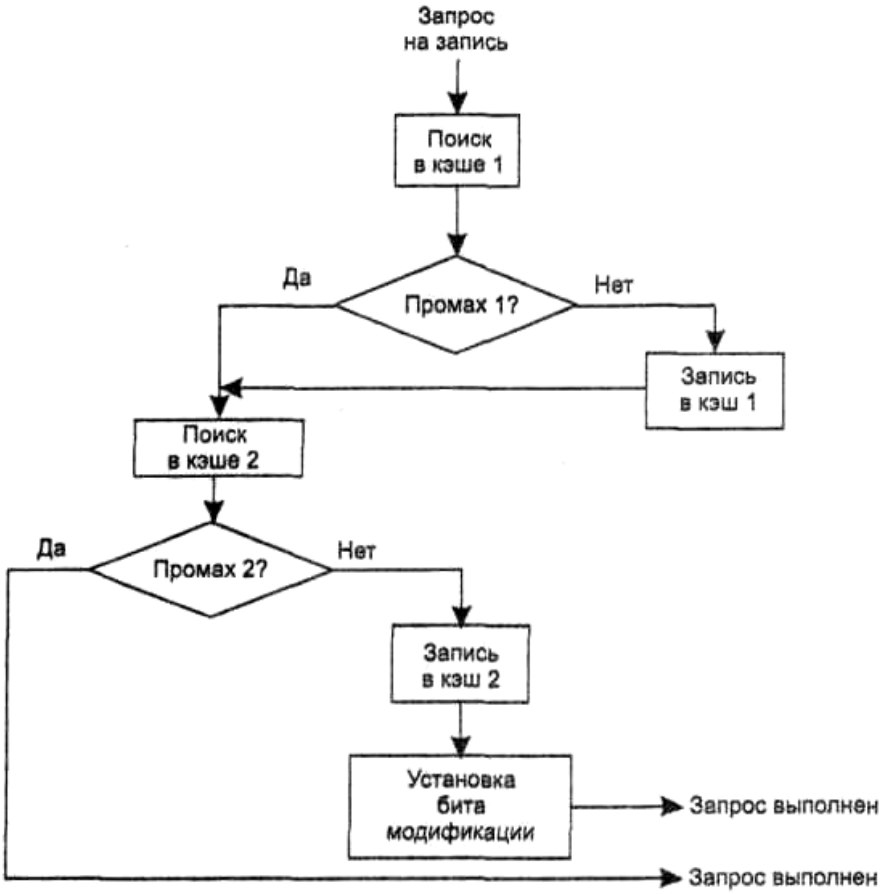



Рис. 5.28. Схема выполнения запроса на запись в системе с двухуровневым кэшем

## 5.7. Вопросы и задания для самопроверки

1. Чем ограничивается максимальный размер физической памяти, которую можно установить в компьютере определенной модели?

2. Чем ограничивается максимальный размер виртуального адресного пространства, доступного приложению?

3. Какой из следующих методов распределения памяти может рассматриваться как частный случай виртуальной памяти?

а) распределение фиксированными разделами;

б) распределение динамическими разделами;

в) страничное распределение;

г) сегментное распределение;

д) сегментно-страничное распределение.

4. Распределение памяти перемещаемыми разделами основано на применении процедуры сжатия. Имеет ли смысл использовать данную процедуру при страничном распределении? При сегментном?

5. Поясните разные значения термина «свопинг».

6. Как величина файла подкачки влияет на производительность системы?

7. Где хранятся таблицы страниц и таблицы сегментов?

8. Какие характеристики содержит таблица сегментов и таблица страниц при сегментно-страничной организации памяти?

9. Пусть ОС реализует выгрузку страниц на основе критерия «выгружается страница, которая не использовалась дольше остальных». Предложите алгоритм вычисления данного критерия, использующий аппаратно-устанавливаемые биты доступа.

10. Почему загрузка и выгрузка данных из кэш-памяти производится блоками?

11. Как обеспечивается согласование данных в кэше с помощью методов обратной и сквозной записи?

12. Известно, что с помощью программных конвейеров данными могут обмениваться только процессы-родственники. Все процессы в UNIX являются родственниками, так как все они потомки специального процесса, инициализирующего систему. Почему механизм программных конвейеров не работает для двух произвольных процессов?

# ТЕМА 6. ФАЙЛОВАЯ СИСТЕМА

Цель изучения темы – приобретение студентами знаний о принципах функционирования механизмов различных файловых систем, их сходствах и различиях.

В результате изучения темы студенты должны:

– иметь понятие о структуре файловой системы, именах и типах файлов, их логической и физической организации;

– иметь представление об организации файловой системы NTFS как одной из наиболее распространенных и имеющих наибольшее количество общих черт с остальными файловыми системами;

– иметь представление о файловых операциях и механизмах контроля доступа в ОС Windows NT и UNIX.

## Содержание темы

1. Логическая организация файловой системы.
1.1. Цели и задачи файловой системы.
1.2. Типы файлов.
1.3. Иерархическая структура файловой системы.
1.4. Имена файлов.
1.5. Монтирование.
1.6. Атрибуты файлов.
1.7. Логическая организация файла.
2. Физическая организация NTFS.
2.1. Структура тома NTFS.
2.2. Структура файлов NTFS.
2.3. Каталоги NTFS.
3. Файловые операции.
3.1. Способы организации файловых операций.
3.2. Открытие файла.
3.3. Обмен данными с файлом.
3.4. Блокировки файлов.
3.5. Стандартные файлы ввода и вывода, перенаправление вывода.
4. Контроль доступа к файлам.
4.1. Доступ к файлам как частный случай доступа к разделяемым ресурсам.
4.2. Механизм контроля доступа.
4.3. Организация контроля доступа в ОС UNIX.
4.4. Организация контроля доступа в ОС Windows NT.
5. Вопросы и задания для самопроверки.

## 6. 1. Логическая организация файловой системы

Одной из основных задач операционной системы является предоставление пользователю удобств при работе с данными, хранящимися на дисках. Для этого ОС подменяет физическую структуру хранящихся данных удобной для пользователя логической моделью. Логическая модель файловой системы материализуется в виде дерева каталогов, 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 логической и физической структурой.

### 6.1.1. Цели и задачи файловой системы

*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 здесь имеются исключения. Одним из таких исключений является так называемый электронный диск, когда в оперативной памяти создается структура, имитирующая файловую систему.

Назовем основные цели использования файла.

– *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 другим, при этом создатель файла или администратор могут определить права доступа к нему других пользователей. Эти цели реализуются в ОС файловой системой.

*Файловая система* (ФС) – 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.

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 ФС берет на себя. Она распределяет дисковую память, поддерживает именование файлов, отображает имена файлов в соответствующие адреса во внешней памяти, обеспечивает доступ к данным, поддерживает разделение, защиту и восстановление файлов.

Таким образом, ФС играет роль промежуточного слоя, экранирующего все сложности физической организации долговременного хранилища данных и создающего для программ более простую логическую модель этого хранилища, а также предоставляющего им набор удобных в использовании команд для манипулирования файлами.

Задачи, решаемые ФС, зависят от способа организации вычислительного процесса в целом. Самый простой тип – ФС в однопользовательских и однопрограммных ОС, к числу которых относится, например, MS-DOS. Основными функциями такой ФС является:

– именование файлов;

– программный интерфейс для приложений;

– отображения логической модели ФС на физическую организацию хранилища данных;

– устойчивость ФС к сбоям питания, ошибкам аппаратных и программных средств.

Задачи ФС усложняются в операционных однопользовательских мультипрограммных ОС, которые хотя и предназначены для работы одного пользователя, но дают ему возможность запускать одновременно несколько процессов. Одной из первых ОС этого типа стала OS / 2. К перечисленным выше добавляется новая задача совместного доступа к файлу из нескольких процессов. Файл в этом случае является разделяемым ресурсом, а значит, ФС должна решать весь комплекс проблем, связанных с такими ресурсами. В частности, в ФС должны быть предусмотрены средства блокировки файла и его частей, предотвращения гонок, исключение тупиков, согласование копий и т. п.

В многопользовательских системах появляется еще одна задача: защита файлов одного пользователя от несанкционированного доступа другого.

### 6.1.2. 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 др.

*Обычные файлы* содержат информацию произвольного характера, которую записывает в них пользователь или которая образуется в результате работы системных и пользовательских программ. 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, содержащих 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 и используемые для унификации механизма 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.

### 6.1.3. 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.

Граф, описывающий иерархию каталогов, может быть деревом или сетью. Каталоги образуют дерево, если файлу разрешено входить только в один каталог, и сеть – если файл может входить сразу в несколько каталогов. 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.

### 6.1.4. Имена файлов

Все типы файлов имеют символьные имена. В иерархически организованных файловых системах обычно используется три типа имен файлов: простые, составные и относительные.

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.

В иерархических файловых системах разным файлам разрешено иметь одинаковые простые символьные имена при условии, что они принадлежат разным каталогам. 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.

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.

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.

### 6.1.5. 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. Каким же 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, т. е. 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.

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.

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, причем один из дисков является системным.

Файловая система, расположенная на системном диске, назначается корневой. Для связи иерархий файлов в корневой файловой системе выбирается некоторый существующий каталог. После выполнения монтирования выбранный каталог становится корневым каталогом второй файловой системы. Через этот каталог монтируемая файловая система присоединяется как поддерево к общему дереву.

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.

### 6.1.6. Атрибуты файлов

Понятие «файл» включает не только хранимые им данные и имя, но и атрибуты. Атрибуты – это информация, описывающая свойства файла. Приведем 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;
- время 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;
- длина ключа.

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 списке, связанных со структуриза-

цией файла. В однопользовательской ОС в наборе атрибутов будут отсутствовать характеристики, имеющие отношение к пользователям и защите (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.

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. 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.

В обоих 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.

### 6.1.7. 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. 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 ФС 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.

Модель файла, в соответствии с которой содержимое файла представляется неструктурированной последовательностью (потоком) байт, стала популярной вместе с ОС UNIX, а теперь широко используется в большинстве современных ОС, в том числе в MS-DOS, Windows NT / 2000, NetWare. Неструктурированная модель файла позволяет 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. Здесь 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).

Операционная система не может поддерживать все возможные способы структурирования данных в файле, поэтому в тех ОС, в которых вообще существует поддержка логической структуризации файлов, она предназначена 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. 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 ло-

гической организации размер записи фиксирован в пределах файла, а записи в различных файлах, принадлежащих одной и той же файловой системе, могут иметь различный размер.

Другой способ структуризации состоит в представлении данных в виде последовательности записей, размер которых изменяется в пределах одного файла. Для поиска нужной записи система должна последовательно прочитать все предшествующие записи. Вычислить адрес нужной записи по ее номеру при такой логической организации файла невозможно, а следовательно, невозможно применить 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, допускающие более быстрый прямой доступ к отдельной логической записи. В индексированном файле 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 адрес 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. Записи в индексированных файлах могут иметь произвольную длину.

Вышесказанное в большей степени относится к обычным файлам, которые могут быть как структурированными, так и неструктурированными. Другие типы файлов обладают определенной структурой, известной файловой системе. Например, файловая система должна понимать структуру данных, хранящихся в файле-каталоге или файле типа «символьная связь».

## 6.2. Физическая организация NTFS

Файловая система NTFS была разработана в качестве основной файловой системы для ОС Windows NT в начале 90-х годов XX в. с учетом опыта разработки файловых систем FAT и HPFS (основная файловая сис-

тема для OS / 2), а также других существовавших в то время файловых систем. Основными отличительными свойствами NTFS являются:

– поддержка больших файлов и больших дисков объемом до 264 байт;

– восстанавливаемость после сбоев и отказов программ и аппаратуры управления дисками;

– высокая скорость операций, в том числе для больших дисков;

– низкий уровень фрагментации, в том числе для больших дисков;

– гибкая структура, допускающая развитие за счет добавления новых типов записей и атрибутов файлов с сохранением совместимости с предыдущими версиями ФС;

– устойчивость к отказам дисковых накопителей;

– поддержка длинных символьных имен;

– контроль доступа к каталогам и отдельным файлам.

### 6.2.1. Структура тома NTFS

В отличие от разделов FAT и s5 / ufs все пространство тома NTFS представляет собой либо файл, либо часть файла. Основой структуры тома NTFS является главная таблица файлов (Master File Table, MFT), которая содержит по крайней мере одну запись для каждого файла тома, включая одну запись для самой себя. Каждая запись MFT имеет фиксированную длину, зависящую от объема диска, – 1, 2 или 4 Кбайт. Для большинства дисков, используемых сегодня, размер записи MFT равен 2 Кбайт, который мы далее будет считать размером записи по умолчанию.

Все файлы на томе NTFS идентифицируются номером файла, который определяется позицией файла в MFT. Этот способ идентификации файла близок к способу, используемому в файловых системах s5 и ufs, где файл однозначно идентифицируется номером его записи в области индексных дескрипторов.

Весь том NTFS состоит из последовательности кластеров, что отличает эту файловую систему от рассмотренных ранее, где на кластеры делилась только область данных. Порядковый номер кластера в томе NTFS называется логическим номером кластера (Logical Cluster Number, LCN). Файл NTFS также состоит из последовательности кластеров, при этом порядковый номер кластера внутри файла называется виртуальным номером кластера (Virtual Cluster Number, VCN).

Базовая единица распределения дискового пространства для файловой системы NTFS – непрерывная область кластеров, называемая отрезком. В качестве адреса отрезка NTFS использует логический номер его первого кластера, а также количество кластеров в отрезке *k*, т. е. пара (LCN, *k*). Та-

ким образом, часть файла, помещенная в отрезок и начинающаяся с виртуального кластера VCN, характеризуется адресом, состоящим из трех чисел: (VCN, LCN, *k*).

Для хранения номера кластера в NTFS используются 64-разрядные указатели, что дает возможность поддерживать тома и файлы размером до 264 кластеров. При размере кластера в 4 Кбайт это позволяет использовать тома и файлы, состоящие из 64 миллиардов килобайт.

Структура тома NTFS. Загрузочный блок тома NTFS располагается в начале тома, а его копия – в середине тома. Загрузочный блок содержит стандартный блок параметров BIOS, количество блоков в томе, а также начальный логический номер кластера основной копии MFT и зеркальную копию MFT.

Далее располагается первый отрезок MFT, содержащий 16 стандартных, создаваемых при форматировании записей о системных файлах NTFS. Назначение этих файлов описано в таблице MFT (табл. 6.1).

В NTFS файл целиком размещается в записи таблицы MFT, если это позволяет сделать его размер. В том же случае, когда размер файла больше размера записи MFT, в запись помещаются только некоторые атрибуты файла, а остальная часть файла размещается в отдельном отрезке тома (или нескольких отрезках). Часть файла, размещаемая в записи MFT, называется резидентной частью, а остальные части — нерезидентными. Адресная информация об отрезках, содержащих нерезидентные части файла, размещается в атрибутах резидентной части.

Таблица 6.1

Главная таблица файлов NTFS

| **Номер** | **Системный файл** | **Имя файла** | **Назначение файла записи** |
|---|---|---|---|
| 0 | Главная таблица файлов | $Mft | Содержит полный список файлов тома NTFS |
| 1 | Копия главной таблицы файлов | $MftMirr | Зеркальная копия первых трех записей MFT |
| 2 | Файл журнала | $LogFile | Список транзакций, который используется для восстановления файловой системы после сбоев |
| 3 | Том | $Volume | Имя тома, версия NTFS и другая информация о томе |

*Окончание табл. 6.1*

| 1 | 2 | 3 | 4 |
|---|---|---|---|
| 4 | Таблица определения атрибутов | $AttrDef | Таблица имен, номеров и описаний атрибутов |
| 5 | Индекс корневого каталога | $. | Корневой каталог |
| 6 | Битовая карта кластеров | $Bitmap | Разметка использованных кластеров тома |
| 7 | Загрузочный сектор раздела | $Boot | Адрес загрузочного сектора раздела |
| 8 | Файл плохих кластеров | $BadClus | Файл, содержащий список всех обнаруженных на томе плохих кластеров |
| 9 | Таблица квот | $Quota | Квоты используемого пространства на диске для каждого пользователя |
| 10 | Таблица преобразования регистра символов | $Upcase | Используется для преобразования регистра символов для Unicode |
| 11 – 15 | Зарезервированы для будущего использования | | |

Некоторые системные файлы являются полностью резидентными, а некоторые имеют и нерезидентные части, которые располагаются после первого отрезка MFT.

Нулевая запись MFT содержит описание самой MFT, в том числе и такой ее важный атрибут, как адреса всех ее отрезков. После форматирования MFT состоит из одного отрезка, но после создания первого же несистемного файла для хранения его атрибутов требуется еще один отрезок, так как изначально непрерывная последовательность кластеров MFT уже завершена системными файлами.

Сама таблица MFT рассматривается как файл, к которому применим метод размещения в томе в виде набора нескольких произвольно расположенных отрезков.

### 6.2.2. Структура файлов NTFS

Каждый файл и каталог на томе NTFS состоит из набора атрибутов. Отметим, что имя файла и его данные также рассматриваются как атрибуты файла, то есть в трактовке NTFS, кроме атрибутов, у файла нет никаких других компонентов.

Каждый атрибут файла NTFS состоит из полей: тип атрибута, длина атрибута, значение атрибута и, возможно, имя атрибута. Тип атрибута, длина и имя образуют заголовок атрибута.

Имеется системный набор атрибутов, определяемых структурой тома NTFS. Системные атрибуты имеют фиксированные имена и коды их типа, а также определенный формат. Могут применяться также атрибуты, определяемые пользователями. Их имена, типы и форматы задаются исключительно пользователем. Атрибуты файлов упорядочены по убыванию кода атрибута, причем атрибут одного и того же типа может повторяться несколько раз. Существуют два способа хранения атрибутов файла: резидентное хранение в записях таблицы MFT и нерезидентное хранение вне ее, во внешних отрезках. Таким образом, резидентная часть файла состоит из резидентных атрибутов, а нерезидентная – из нерезидентных атрибутов. Сортировка может осуществляться только по резидентным атрибутам.

Системный набор включает следующие атрибуты:

– *Attribute List* (список атрибутов) – список атрибутов, из которых состоит файл; содержит ссылки на номер записи MFT, где расположен каждый атрибут; этот редко используемый атрибут нужен только в том случае, если атрибуты файла не умещаются в основной записи и занимают дополнительные записи MFT;

– *File Name* (имя файла) содержит длинное имя файла в формате Unicode, а также номер входа в таблице MFT для родительского каталога; если этот файл содержится в нескольких каталогах, то у него будет несколько атрибутов типа File Name; этот атрибут всегда должен быть резидентным;

– *MS-DOS Name* (имя MS-DOS) содержит имя файла в формате 8.3;

– *Version* (версия) содержит номер последней версии файла;

– *Security Descriptor* (дескриптор безопасности) содержит информацию о защите файла: список прав доступа ACL и поле аудита, которое определяет, какого рода операции над этим файлом нужно регистрировать;

– *Volume Version* (версия тома) используется только в системных файлах тома;

– *Volume Name* (имя тома) – символьное имя тома;

– *Data* (данные) – содержит обычные данные файла;

– *MFT bitmap* (битовая карта MFT) содержит карту использования блоков на томе;

– *Index Root* (корень индекса) – корень B-дерева, используемого для поиска файлов в каталоге;

– *Index Allocation* (размещение индекса) – нерезидентные части индексного списка B-дерева;

– *Standard Information* (стандартная информация) хранит всю остальную стандартную информацию о файле, которую трудно связать с каким-либо из других атрибутов файла, например, время создания файла, время обновления и другие.

Файлы NTFS в зависимости от способа размещения делятся на небольшие, большие, очень большие и сверхбольшие.

*Небольшие файлы* (small). Если файл имеет небольшой размер, то он может целиком располагаться внутри одной записи MFT, имеющей, например, размер 2 Кбайт. Небольшие файлы NTFS состоят из следующих атрибутов:

- стандартная информация (SI – standard information);
- имя файла (FN – file name);
- данные (Data);
- дескриптор безопасности (SD – security descriptor).

Из-за того что файл может иметь переменное количество атрибутов, а также из-за переменного размера атрибутов, нельзя наверняка утверждать, что файл уместится внутри записи. Однако обычно файлы размером менее 1500 байт помещаются внутри записи MFT (размером 2 Кбайт).

*Большие файлы* (large). Если данные файла не помещаются в одну запись MFT, это отражается в заголовке атрибута Data, который содержит признак того, что этот атрибут является нерезидентным, т. е. находится в отрезках вне таблицы MFT. В этом случае атрибут Data содержит адресную информацию (LCN, VCN, *k*) каждого отрезка данных.

*Сверхбольшие файлы* (extremely huge). Для сверхбольших файлов в атрибуте Attribute List можно указать несколько атрибутов, расположенных в дополнительных записях MFT. Кроме того, можно использовать двойную косвенную адресацию, когда нерезидентный атрибут будет ссылаться на другие нерезидентные атрибуты, поэтому в NTFS не может быть атрибутов слишком большой для системы длины.

*Очень большие файлы* (huge). Если файл настолько велик, что его атрибут данных, хранящий адреса нерезидентных отрезков данных, не помещается в одной записи, то этот атрибут помещается в другую запись MFT, а ссылка на такой атрибут помещается в основную запись файла. Ссылка содержится в атрибуте Attribute List. Сам атрибут данных попрежнему содержит адреса нерезидентных отрезков данных.

### 6.2.3. Каталоги NTFS

Каждый каталог NTFS представляет собой один вход в таблицу MFT и содержит атрибут Index Root. Индекс содержит список файлов, входящих в каталог. Индексы позволяют сортировать файлы для ускорения поиска, основанного на значении определенного атрибута. Обычно в файловых системах файлы сортируются по имени. NTFS позволяет использовать для сортировки любой атрибут, если он хранится в резидентной форме.

Имеются две формы хранения списка файлов.

*Небольшие каталоги* (small indexes). Если количество файлов в каталоге невелико, то список файлов может быть резидентным в записи в MFT, являющейся каталогом. Для резидентного хранения списка используется единственный атрибут – Index Root. Список файлов содержит значения атрибутов файла. По умолчанию это имя файла, а также номер записи MTF, содержащей начальную запись файла.

*Большие каталоги* (large indexes). По мере роста каталога список файлов может потребовать нерезидентной формы хранения. Однако начальная часть списка всегда остается резидентной в корневой записи каталога в таблице MFT. Имена файлов резидентной части списка файлов являются узлами так называемого B-дерева (двоичного дерева). Остальные части списка файлов размещаются вне MFT. Для их поиска используется специальный атрибут Index Allocation, представляющий собой адреса отрезков, хранящих остальные части списка файлов каталога. Одни части списков являются листьями дерева, а другие являются промежуточными узлами, т. е. содержат наряду с именами файлов атрибут Index Allocation, указывающий на списки файлов более низких уровней.

Узлы двоичного дерева делят весь список файлов на несколько групп. Имя каждого файла-узла является именем последнего файла в соответствующей группе. Считается, что имена файлов сравниваются лексикографически, т. е. сначала принимаются во внимание коды первых символов двух сравниваемых имен, при этом имя считается меньшим, если код его первого символа имеет меньшее арифметическое значение; при равенстве кодов первых символов сравниваются коды вторых символов имен и т. д.

Поиск в каталоге уникального имени файла, которым в NTFS является номер основной записи о файле в MFT, по его символьному имени происходит следующим образом. Сначала искомое символьное имя сравнивается с именем первого узла в резидентной части индекса. Если искомое имя меньше, это означает, что его нужно искать в первой нерезидентной группе, для чего из атрибута Index Allocation извлекается адрес отрезка (VCN, $LCN_j$, $K_j$), хранящего имена файлов первой группы. Среди имен этой группы поиск осуществляется прямым перебором имен и сравнением до полного совпадения всех символов искомого имени с хранящимся в каталоге именем. При совпадении из каталога извлекается номер основной записи о файле в MFT, и остальные характеристики файла берутся уже оттуда. Если же искомое имя больше имени первого узла резидентной части индекса, то его сравнивают с именем второго узла, и если искомое имя меньше, то описанная процедура применяется ко второй нерезидентной группе имен, и т. д.

В результате вместо перебора большого количества имен (в худшем случае – всех имен каталога) выполняется сравнение с гораздо меньшим количеством имен узлов и имен в одной из групп каталога.

Если одна из групп каталога становится слишком большой, ее также делят на группы; последние имена каждой новой группы оставляют в исходном нерезидентном атрибуте Index Root, а все остальные имена новых групп переносят в новые нерезидентные атрибуты типа Index Root. К исходному нерезидентному атрибуту Index Root добавляется атрибут размещения индекса, указывающий на отрезки индекса новых групп. Если теперь при поиске искомого имени в нерезидентной части индекса первого уровня какое-либо сравнение показывает, что искомое имя оказывается меньше, чем одно из хранящихся там имен, это говорит о том, что в данном атрибуте точного сравнения имени уже быть не может и нужно перейти к подгруппе имен следующего уровня дерева.

## 6.3. Файловые операции

### 6.3.1. Способы организации файловых операций

Файловая система ОС должна предоставлять пользователям набор операций работы с файлами, оформленный в виде системных вызовов. Этот набор обычно состоит из таких системных вызовов, как creat (создать файл), read (читать из файла), write (записать в файл) и некоторых других.

Чаще всего с одним и тем же файлом пользователь выполняет не одну операцию, а последовательность операций. Например, при работе текстового редактора с файлом, в котором содержится некоторый документ, пользователь обычно считывает несколько страниц текста, редактирует эти данные и записывает их на место считанных, а затем считывает страницы из другой области файла, и т. п. После большого количества операций чтения и записи пользователь завершает работу с данным файлом и переходит к другому.

Какие бы операции ни выполнялись над файлом, ОС необходимо выполнить ряд универсальных для всех операций действий:

1) по символьному имени файла найти его характеристики, которые хранятся в файловой системе на диске;

2) скопировать характеристики файла в оперативную память, так как только таким образом программный код может их использовать;

3) на основании характеристик файла проверить права пользователя на выполнение запрошенной операции (чтение, запись, удаление, просмотр атрибутов файла);

4) очистить область памяти, отведенную под временное хранение характеристик файла.

Кроме того, каждая операция включает ряд уникальных для нее действий, например, чтение определенного набора кластеров диска, удаление файла и т. п.

Операционная система может выполнять последовательность действий над файлом двумя способами:

– для каждой операции выполняются как универсальные, так и уникальные действия; такая схема иногда называется схемой без запоминания состояния операций (stateless);

– все универсальные действия выполняются в начале и конце последовательности операций, а для каждой промежуточной операции выполняются только уникальные действия.

Большинство файловых систем поддерживает второй способ организации файловых операций как более экономичный и быстрый. Первый способ обладает преимуществом: он более устойчив к сбоям в работе системы, так как каждая операция является самодостаточной и не зависит от результата предыдущей. Поэтому первый способ иногда применяется в распределенных сетевых файловых системах (например, в Network File System, NFS компании Sun), когда сбои из-за потерь пакетов или отказов одного из сетевых узлов более вероятны, чем при локальном доступе к файлам.

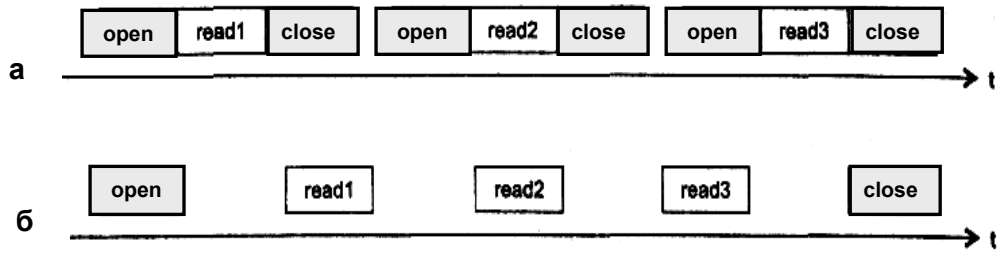



Рис. 6.1. Два способа выполнения файловых операций

При втором способе в файловой системе вводятся два специальных системных вызова: open – открытие файла, и close – закрытие файла.

Системный вызов открытия файла open выполняется перед началом любой последовательности операций с файлом, а вызов закрытия файла close – после окончания работы с файлом. Основной задачей вызова open является преобразование символьного имени файла в его уникальное числовое имя, копирование характеристик файла из дисковой области в буфер оперативной памяти и проверка прав пользователя на выполнение запрошенной операции. Вызов close освобождает буфер с характеристиками файла и делает невозможным продолжение операций с файлом без его повторного открытия.

Операции открытия и закрытия файла в той или иной форме утвердились в операционных системах давно. Даже в операционной системе OS / 360, существовала макрокоманда OPEN, по которой в специальном буфере, называемом DCB (Data Control Block), собирались из различных источников все нужные характеристики набора данных (понятие, близкое к современному понятию файла), используемые затем при выполнении операций чтения и записи.

Далее основные системные вызовы файловых операций рассматриваются более детально на примере их реализации в ОС UNIX, где они приобрели тот вид, который сегодня поддерживается практически всеми операционными системами.

### 6.3.2. Открытие файла

Системный вызов open в ОС UNIX работает с двумя аргументами: символьным именем открываемого файла и режимом открытия файла. Режим открытия сообщает системе, какие операции будут выполняться над файлом в последовательности операций до закрытия файла по системному вызову close, например, только чтение, только запись или чтение и запись.

При открытии файла ОС сначала выполняет преобразование первого аргумента системного вызова, то есть символьного имени файла, в его уникальное числовое имя, которым в традиционных файловых системах UNIX является номер индексного дескриптора. Эта процедура была рассмотрена выше при описании файловой системы s5.

По номеру индексного дескриптора inode файловая система находит нужную запись на диске и копирует из нее характеристики файла в оперативную память.

Для хранения копии индексного дескриптора используются буферные области системного виртуального пространства. Характеристики индексного дескриптора, перенесенные в оперативную память, помещаются в структуру так называемого виртуального дескриптора vnode (virtual node). Структура vnode включает поля индексного дескриптора файла inode, а также несколько дополнительных полей, полезных при выполнении операций с файлом:

1) *состояние индексного дескриптора в памяти*, отражающее:

– заблокирован ли файл;

– ждет ли снятия блокировки с файла какой-либо процесс;

– отличается ли представление характеристик файла в памяти от своей дисковой копии в результате изменения содержимого индексного дескриптора;

– отличается ли представление файла в памяти от своей дисковой копии в результате изменения содержимого файла;

– является ли файл точкой монтирования;

2) *логический номер устройства файловой системы, содержащей файл*;

3) *номер индексного дескриптора*. В дисковом индексном дескрипторе это поле отсутствует, так как номер определяется положением дескриптора относительно начала области индексных дескрипторов;

4) *счетчик ссылок на данную структуру vnode*.

С одним и тем же файлом в какой-то период времени могут работать различные процессы, но операционная система не создает для каждого процесса отдельную копию структуры vnode, а для каждого файла, с которым в данный момент работает хотя бы один процесс, хранит ровно одну копию виртуального дескриптора. При очередном открытии файла ОС по номеру логического устройства и номеру индексного дескриптора, определяемым при преобразовании символьного имени, проверяет, имеется ли в системной памяти структура vnode открываемого файла, и если имеется, то счетчик ссылок на нее увеличивается на единицу. При очередном закрытии этого файла счетчик ссылок уменьшается на единицу, и если он становится равным 0, то буфер, хранящий данный vnode, считается свободным.

Использование единственной копии характеристик файла и некоторых характеристик файловых операций, например, признака блокировки, общих для всех работающих с файлом процессов, экономит системную память. Тем не менее, существуют характеристики, индивидуальные для каждого процесса, выполняющего некоторую последовательность операций с определенным файлом. Для их хранения в UNIX используется структура типа file, которая так же, как и vnode, хранится в системной области памяти.

При каждом открытии процессом файла ОС проверяет права пользовательского процесса на выполнение запрошенной операции с файлом и, если проверка прошла успешно, создает в системной области памяти новую структуру file, которая описывает как открытый файл, так и операции, которые процесс собирается производить с файлом.

Структура file содержит следующие поля:

– признак режима открытия (только для чтения, для чтения и записи и т. п.);

– указатель на структуру vnode;

– текущее смещение в файле (переменная offset) при операциях чтения / записи;

– счетчик ссылок на данную структуру;

– указатель на структуру, содержащую права процесса, открывшего файл; данная структура находится в дескрипторе процесса;

– указатели на предыдущую и последующую структуры file, связывающие все такие структуры в двойной список.

Переменная offset, хранящаяся в структуре file, позволяет ОС запоминать текущее положение условного указателя в последовательности байт файла. При открытии файла эта переменная указывает на начальный или конечный байт файла в зависимости от заданного режима открытия. После выполнения операций чтения или записи указатель сдвигается на то количество байт, которое было прочитано или записано в результате операции. Следующая операция застает указатель в том состоянии, в котором его оставила предыдущая операция. Прикладной программист может явно управлять положением указателя с помощью системного вызова 1 seek.

При каждом новом открытии какого-либо файла ОС создает новую структуру file и помещает ее в дважды связанный список (рис. 6.2). Обычно под хранение структур file в системной области отводится ограниченная область, поэтому общее количество открытых файлов всеми процессами в любой момент ограничено.

После создания структуры file операционная система помещает указатель на нее в таблицу открытых файлов процесса, которая находится в контексте процесса. Если процесс несколько раз открывает один и тот же файл, то структура file создается для каждой операции открытия. Так как контекст процесса-родителя в UNIX наследуется процессом-потомком, то потомок наследует и указатели на все открытые родителем файлы, получая возможность выполнять над ними операции.

Системный вызов open возвращает в пользовательский процесс дескриптор файла, который представляет собой номер записи в таблице открытых файлов процесса. Дескриптор файла имеет локальное значение только для того процесса, который открыл файл, для разных процессов одно и то же значение дескриптора указывает на разные операции, в общем случае над разными файлами.

После открытия файла его дескриптор используется во всех дальнейших операциях с файлом вплоть до явного закрытия файла. Таким образом, дескриптор файла является временным уникальным именем, но не файла, а определенной последовательности операций с этим файлом.

Для открытия файла /bin/prog 1.exe в режиме «только для чтения» прикладной программист может использовать следующее выражение на языке С:

*fd* = open("/bin/progl exe". 0_RDONLY).

Здесь *fd* – это целочисленная переменная, сохраняющая значение дескриптора открытого файла. Ее значение должно использоваться в операциях обмена данными с файлом /bin/progI.exe. При неудачной попытке от-

крытия файл, (нет прав для выполнения затребованной операции, неверное имя файла) переменной *fd* присваивается значение –1, которое является индикатором ошибки для всех системных вызовов UNIX.

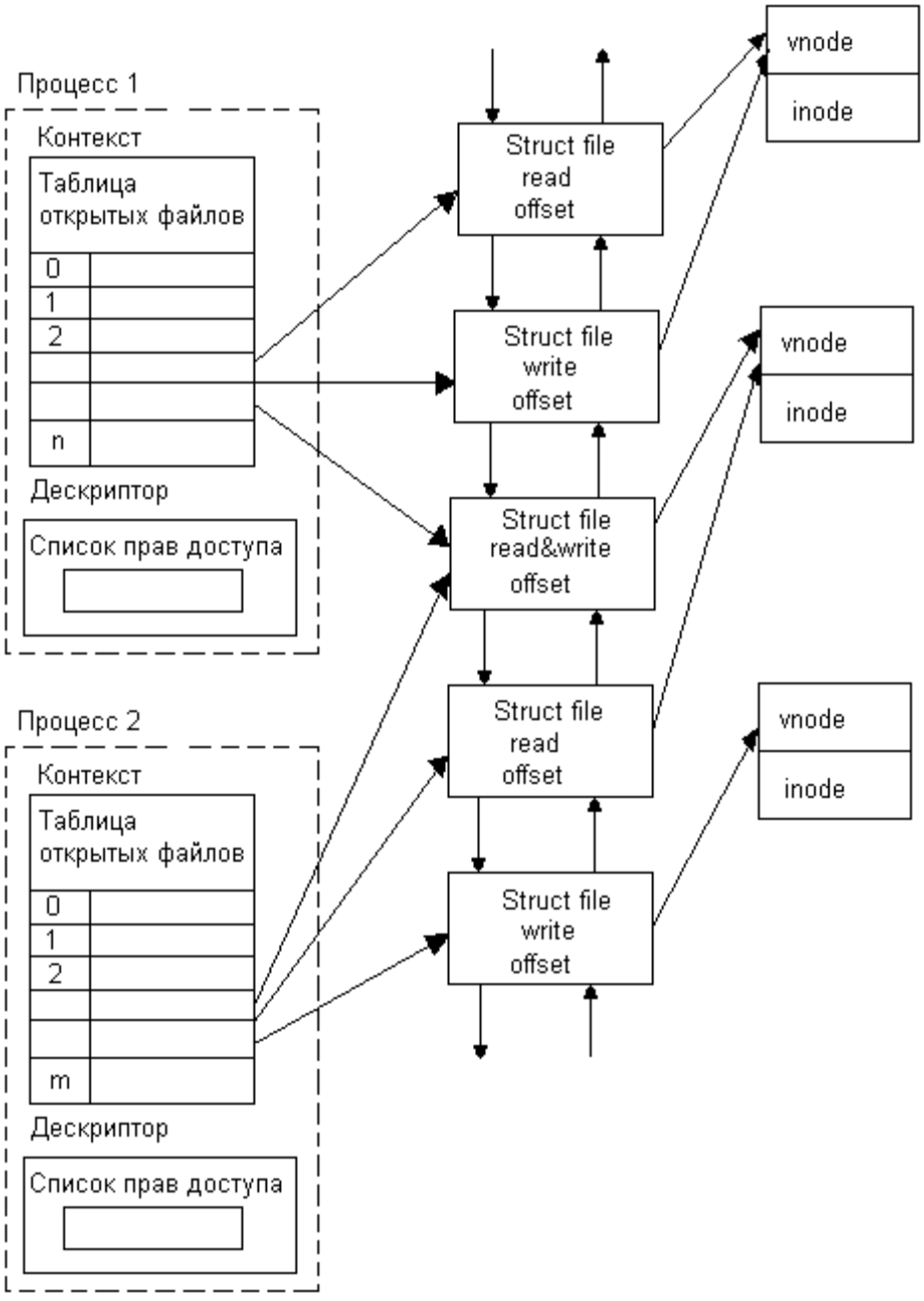



Рис. 6.2. Связь процесса с открытыми файлами

### 6.3.3. Обмен данными с файлом

Для обмена данными с предварительно открытым файлом в ОС UNIX существуют системные вызовы read и write. В том случае, когда необходимо явным образом указать, с какого байта файла необходимо читать или записывать данные, используется также системный вызов seek.

Системный вызов чтения данных из файла read имеет три аргумента

*read(fd buffer nbytes)*.

Первый аргумент *fd* является целочисленной переменной, имеющей значение дескриптора открытого файла. Второй аргумент *buffer* является указателем на область пользовательской памяти, в которую система должна поместить считанные данные. Количество байт этой области памяти задается третьим целочисленным аргументом *nbytes*. Функция read возвращает действительное количество считанных байт (оно может отличаться от заданного, если, например, была задана область чтения, выходящая за пределы файла) или код ошибки –1. Начало дисковой области, которую нужно прочитать с помощью вызова read, явно в этом системном вызове не указывается. Чтение начинается с того байта, на который указывает смещение offset в структуре file. На это смещение указывает запись с номером *fd* в таблице открытых файлов процесса. После выполнения вызова read смещение offset наращивается на количество прочитанных байт.

Вид системного вызова записи данных write аналогичен вызову read:

*write(fd buffer.nbytes)*.

Функция write записывает *nbytes* из буфера оперативной памяти *buffer* в файл, описываемый дескриптором *fd*. Функция write, так же как и read, возвращает вызвавшей ее программе значение реально переданных ею байт или код ошибки.

Рассмотрим пример, в котором прикладная программа работает с файлом, состоящем из записей фиксированной длины в 50 байт:

```
fd = open("/doc/qwery/base12. txt", 0_RDWR);
read(fd, bufferl,50);
read(fd,buffer2, 2500);
...
Iseek(fd, 150, 0);
 write (fd, output, 300);
```

В приведенном фрагменте программы после открытия файла /doc/query/base12.txt для чтения и записи выполняется чтение первой записи файла, а затем читается область файла, включающая еще 50 записей, начиная со 2 по 51. После обработки считанных записей производится перемещение указателя смещения в файле на начало четвертой записи и запись результатов в шесть последовательных записей, начиная с четвертой. Фрагмент завершается закрытием файла с помощью системного вызова close.

Все описанные системные вызовы являются синхронными, т. е. пользовательский процесс переводится в состояние ожидания до тех пор, пока операция ввода-вывода не завершится.

Описанный набор системных вызовов, появившийся в ОС UNIX еще в 70-х гг. XX в., стал стандартом де-факто для современных операционных систем. Эти традиционные системные вызовы часто в конкретных ОС дополняются оригинальными системными вызовами ввода-вывода, например, операциями асинхронного типа. На основе системных вызовов ввода-вывода обычно строятся более мощные библиотечные функции ввода-вывода, составляющие прикладной интерфейс ОС.

### 6.3.4. Блокировки файлов

Блокировки файлов и отдельных записей в файлах являются средством синхронизации между работающими в кооперации процессами, пытающимися использовать один и тот же файл одновременно.

Процессы могут иметь соответствующие права доступа к файлу, но одновременное использование этих прав, в особенности права записи, может привести к некорректным результатам. Примером тому служит одновременное редактирование одного и того же документа несколькими пользователями. Если доступ к файлу не управляется блокировками, то каждый пользователь, имеющий право записи в файл, работает со своей копией данных файла. Результат такого редактирования непредсказуем: он зависит от того, в какой последовательности записывали изменения в файл применяемые пользователями приложения-редакторы.

Многопользовательские операционные системы обычно поддерживают специальный системный вызов, позволяющий программисту установить и проверить блокировки на файл и его отдельные области. В UNIX такой системный вызов называется fcntl. В его аргументах указывается дескриптор файла, для которого нужно установить или проверить блокировки, тип операции (блокирование или проверка, блокирование доступа для чтения или для записи), а также область блокирования – смещение от начала файла и размер в байтах. При проверке наличия блокировок, установленных другими процессами, вызов fcntl немедленно возвращает управление с сообщением результата. При установке блокировки можно задать два режима работы системного вызова: с переходом процесса в состояние ожидания, если блокировку установить невозможно (синхронный системный вызов), и с немедленным возвратом в такой ситуации с сообщением отрицательного результата (асинхронный вызов). Запрошенная блокировка записи не может быть установлена в том случае, если другой процесс уже

установил свою блокировку записи на тот же файл. То есть блокировка записи является исключительной. Блокировки чтения не являются исключительными и могут устанавливаться на файл в том случае, если их области действия не перекрываются. Если на определенную область файла установлена блокировка чтения, то на эту область нельзя установить блокировку записи. В UNIX существуют два режима действия блокировок – консультативный (advisory) и обязательный (mandatory). Основным рекомендуемым для использования режимом является консультативный. При нем операционная система не занимается блокированием операций с файлом, а только устанавливает признаки блокирования областей в структурах file, поддерживающих операции с файлами. Кооперирующиеся процессы обязательно должны проверять наличие блокировок на файл, чтобы синхронизировать свою работу. Если же блокировки установлены, но процесс не проверяет их, то операционная система не запрещает доступ процесса к файлу, когда процесс делает системные вызовы read или write.

В обязательном режиме запрет на выполнение операции с заблокированным файлом поддерживает операционная система, поэтому процесс в любом случае не получит доступа к такому файлу. Однако при работе в этом режиме операционная система тратит много усилий и времени на его поддержание, поэтому его использование обычно не рекомендуется.

### 6.3.5. Стандартные файлы ввода и вывода, перенаправление вывода

В ОС UNIX были введены в свое время такие понятия, как «стандартный файл ввода», «стандартный файл вывода» и «стандартный файл ошибок». Эти три уже открытых файла существуют у любого пользовательского процесса с момента его возникновения. Процесс в любое время может организовать ввод данных из стандартного файла ввода, выполнив следующий системный вызов:

*read(stdio. buffer. nbytes).*

Здесь *stdio* – предопределенное имя константы, обозначающей дескриптор стандартного файла ввода.

Аналогично, так как stdout – предопределенное имя дескриптора стандартного файла вывода, процесс может вывести данные в стандартный файл вывода, применив следующий системный вызов:

*write(stout, buffer, nbytes):*

За стандартным файлом ошибок закреплено имя stderr.

Фактически при создании нового процесса ОС помещает в его таблицу открытых файлов три записи: с номером 0 – для стандартного файла ввода

(следовательно, stdin всегда имеет значение 0), с номером 1 – для стандартного файла вывода (stdout = l), и с номером 2 – для стандартного файла ошибок (stderr = 2). Соответственно создаются и три структуры типа file, на которые указывают первые три записи таблицы открытых файлов процесса.

В начальный период существования процесса эти три структуры file связываются операционной системой с одним файлом. В качестве этого файла выступает специальный файл – терминал, с которого вошел в систему пользователь. Такое назначение стандартных файлов достаточно естественно. Прикладные программы, запускаемые пользователем в ходе сеанса работы, чаще всего выводят результаты и сообщения об ошибках на экран терминала, за которым работает пользователь, и с клавиатуры этого же терминала считывают команды и другие исходные данные.

Модель стандартных файлов ввода-вывода рассчитана в основном на алфавитно-цифровые терминалы, управление которыми хорошо описывается потоком выводимых байт, который отображается в виде строк символов на экране, а также потоком вводимых байт, порождаемым последовательными нажатиями клавиш.

Наиболее известной программой, широко использующей стандартные файлы ввода-вывода, является интерпретатор команд, называемый также оболочкой (shell) операционной системы. Интерпретатор читает вводимые пользователем с клавиатуры команды (из стандартного файла ввода) и либо выполняет их самостоятельно, с помощью своих внутренних функций (такие команды называются внутренними), либо интерпретирует команду как имя исполняемого файла на диске, который необходимо запустить на выполнение в качестве отдельного процесса (внешние команды). Сообщения выводятся на экран терминала – стандартный файл вывода.

Стандартные файлы ввода и ввода широко используются не только интерпретатором команд, но и самими командами. Многие внутренние и внешние команды устроены так, что они либо читают свои исходные данные из стандартного файла ввода, либо выводят результаты в стандартный файл вывода. Если команда делает то и другое, она называется фильтром.

Рассмотрим несколько примеров команд UNIX, работающих со стандартными файлами ввода и вывода:

– *ls dir2* – читает записи каталога dir2 и выводит их в определенном символьном формате в стандартный файл вывода;

– *wc* – фильтр, который читает последовательность байт из стандартного файла ввода, подсчитывает число слов, строк или символов в считанных данных и выводит результат в стандартный файл вывода;

– *who* – выводит в стандартный файл информацию о пользователях, работающих в системе.

Интерпретатор команд выполняет также такую важную функцию, как перенаправление стандартного ввода и вывода. Под этим понимается замена файла-терминала, используемого по умолчанию в качестве стандартных файлов ввода и вывода, на произвольный файл. Механизм перенаправления основан на том, что приложение не знает, какой именно файл является стандартным, а просто использует определенный дескриптор в качестве указателя на этот файл. Поэтому для перенаправления ввода-вывода достаточно создать процесс выполнения команды с нестандартной связью стандартной записи в таблице открытых файлов.

Перенаправление осуществляется с помощью специальных конструкций командного языка. Для указания интерпретатору о необходимости перенаправить стандартный ввод на файл file используется следующая конструкция:

*< file*

Для перенаправления стандартного вывода требуется следующая конструкция:

*> file*

Например, показанная ниже командная строка запишет данные о содержимом каталога dir2 в файл a.txt:

*Is dir2 > a.txt*

Механизм перенаправления ввода-вывода, введенный ОС UNIX, получил широкое распространение в интерпретаторах команд многих операционных систем, например MS-DOS, Windows, OS / 2.

## 6.4. Контроль доступа к файлам

### 6.4.1. Доступ к файлам как частный случай доступа к разделяемым ресурсам

Файлы – это частный и самый популярный вид разделяемых ресурсов, доступ к которым операционная система должна контролировать. Существуют и другие виды ресурсов, с которыми пользователи работают в режиме совместного использования. Прежде всего, это различные внешние устройства: принтеры, модемы, графопостроители и т. п. Область памяти, используемая для обмена данными между процессами, также является примером разделяемого ресурса, и сами процессы в некоторых случаях высту-

пают в этой роли, например, когда пользователи ОС посылают процессам сигналы, на которые те должны реагировать.

Во всех этих случаях действует общая схема: пользователи пытаются выполнить с разделяемым ресурсом определенные операции, а ОС должна решать, имеют ли пользователи на это право. Пользователи являются субъектами доступа, а разделяемые ресурсы – объектами. Пользователь осуществляет доступ к объектам операционной системы не непосредственно, а с помощью прикладных процессов, которые запускаются от его имени. Для каждого типа объектов существует набор операций, которые с ними можно выполнять. Например, для файлов это операции чтения, записи, удаления, выполнения; для принтера – перезапуск, очистка очереди документов, приостановка печати документа и т. д. Система контроля доступа ОС должна предоставлять средства для задания прав пользователей по отношению к объектам дифференцированно по операциям, например, пользователю может быть разрешена операция чтения и выполнения файла и запрещена операция удаления.

Во многих операционных системах реализованы механизмы, позволяющие управлять доступом к объектам различного типа с единых позиций. Так, представление устройств ввода-вывода в виде специальных файлов в операционных системах UNIX является примером такого подхода: в этом случае при доступе к устройствам используются те же атрибуты безопасности и алгоритмы, что и при доступе к обычным файлам и каталогам. Еще дальше продвинулась в этом направлении операционная система Windows NT. В ней используется унифицированная структура – объект безопасности, – которая создается не только для файлов и внешних устройств, но и для любых разделяемых ресурсов: секций памяти, синхронизирующих примитивов типа семафоров и мьютексов и т. п. Это позволяет использовать в Windows NT для контроля доступа к ресурсам любого вида общий модуль ядра – менеджер безопасности.

В качестве субъектов доступа могут выступать как отдельные пользователи, так и группы пользователей. Определение индивидуальных прав доступа для каждого пользователя позволяет максимально гибко задать политику расходования разделяемых ресурсов в вычислительной системе. Однако этот способ приводит в больших системах к чрезмерной загрузке администратора рутинной работой по повторению одних и тех же операций для пользователей с одинаковыми правами. Объединение таких пользователей в группу и задание прав доступа в целом для группы является одним из основных приемов администрирования в больших системах.

У каждого объекта доступа существует владелец. Владельцем может быть как отдельный пользователь, так и группа пользователей. Владелец объекта имеет право выполнять с ним любые допустимые для данного объекта операции. Во многих операционных системах существует особый пользователь (superuser, root, administrator), который имеет все права по отношению к любым объектам системы, не обязательно являясь их владельцем. Под таким именем работает администратор системы, которому необходим полный доступ ко всем файлам и устройствам для управления политикой доступа.

Различают два основных подхода к определению прав доступа.

– *Избирательный доступ* имеет место, когда для каждого объекта сам владелец может определить допустимые операции с объектами. Этот подход называется также произвольным (от discretionary – предоставленный на собственное усмотрение) доступом, так как позволяет администратору и владельцам объектов определить права доступа произвольным образом, по их желанию. Между пользователями и группами пользователей в системах с избирательным доступом нет жестких иерархических взаимоотношений, т. е. взаимоотношений, которые определены по умолчанию и которые нельзя изменить. Исключение делается только для администратора, по умолчанию наделяемого всеми правами.

– *Мандатный доступ* (от mandatory – обязательный, принудительный) – это такой подход к определению прав доступа, при котором система наделяет пользователя определенными правами по отношению к каждому разделяемому ресурсу (в данном случае файлу) в зависимости от того, к какой группе пользователь отнесен. От имени системы выступает администратор, а владельцы объектов лишены возможности управлять доступом к ним по своему усмотрению. Все группы пользователей в такой системе образуют строгую иерархию, причем каждая группа пользуется всеми правами группы более низкого уровня иерархии, к которым добавляются права данного уровня. Членам какой-либо группы не разрешается предоставлять свои права членам групп более низких уровней иерархии. Мандатный способ доступа близок к схемам, применяемым для доступа к секретным документам: пользователь может входить в одну из групп, отличающихся правом на доступ к документам с соответствующим грифом секретности, например «для служебного пользования», «секретно», «совершенно секретно» и «государственная тайна». При этом пользователи группы «совершенно секретно» имеют право работать с документами «секретно» и «для служебного пользования», так как эти виды доступа разрешены для более низких в иерархии групп. Однако сами пользователи не распоряжаются правами доступа – этой возможностью наделен только особый чиновник учреждения.

Мандатные системы доступа считаются более надежными, но менее гибкими; обычно они применяются в специализированных вычислительных системах с повышенными требованиями к защите информации. В универсальных системах используются, как правило, избирательные методы доступа.

Далее будут рассмотрены механизмы контроля доступа к таким объектам, как файлы и каталоги, но необходимо понимать, что эти же механизмы могут использоваться в современных операционных системах для контроля доступа к объектам любого типа и отличия заключаются лишь в наборе операций, характерных для того или иного класса объектов.

### 6.4.2. Механизм контроля доступа

Каждый пользователь и каждая группа пользователей обычно имеют символьное имя, а также уникальный числовой идентификатор. При выполнении процедуры логического входа в систему пользователь сообщает свое символьное имя и пароль, а операционная система определяет соответствующие числовые идентификаторы пользователя и групп, в которые он входит. Все идентификационные данные, в том числе имена и идентификаторы пользователей и групп, пароли пользователей, а также сведения о вхождении пользователя в группы, хранятся в специальном файле (файл /etc/passwd в UNIX) или специальной базе данных (в Windows NT).

Вход пользователя в систему порождает процесс-оболочку, который поддерживает диалог с пользователем и запускает для него другие процессы. Процесс-оболочка получает от пользователя символьное имя и пароль и находит по ним числовые идентификаторы пользователя и его групп. Эти идентификаторы связываются с каждым процессом, запущенным оболочкой для данного пользователя. Говорят, что процесс выступает от имени данного пользователя и данных групп пользователей. В наиболее типичном случае любой порождаемый процесс наследует идентификаторы пользователя и групп от процесса-родителя.

Определить права доступа к ресурсу – значит определить для каждого пользователя набор операций, которые ему разрешено применять к данному ресурсу. В разных операционных системах для одних и тех же типов ресурсов может быть определен свой список дифференцируемых операций доступа. Для файловых объектов этот список может включать:

- создание файла;
- уничтожение файла;
- открытие файла;

– закрытие файла;
– чтение файла;
– запись в файл;
– дополнение файла;
– поиск в файле;
– получение атрибутов файла;
– установка новых значений атрибутов;
– переименование;
– выполнение файла;
– чтение каталога;
– смена владельца;
– изменение прав доступа.

Набор файловых операций ОС может состоять из большого количества элементарных операций, а может включать всего несколько укрупненных операций. Приведенный выше список является примером первого подхода, который позволяет весьма тонко управлять правами доступа пользователей, но создает значительную нагрузку на администратора. Пример укрупненного подхода демонстрируют операционные системы семейства UNIX, в которых существуют всего три операции с файлами и каталогами: читать (read, r), писать (write, w) и выполнить (execute, x). Хотя в UNIX для операций используется всего три названия, в действительности им соответствует гораздо больше операций. Например, содержание операции «выполнить» зависит от того, к какому объекту она применяется. Если операция «выполнить файл» интуитивно понятна, то операция «выполнить каталог» интерпретируется как поиск в каталоге определенной записи. Поэтому администратор UNIX, по сути, располагает бóльшим списком операций, чем это кажется на первый взгляд.

В ОС Windows NT разработчики применили гибкий подход: они реализовали возможность работы с операциями над файлами на двух уровнях: по умолчанию администратор работает на укрупненном уровне (уровень стандартных операций), а при желании может перейти на элементарный уровень (уровень индивидуальных операций).

В общем случае права доступа могут быть описаны матрицей прав доступа, в которой столбцы соответствуют всем файлам системы, строки – всем пользователям, а на пересечении строк и столбцов указываются разрешенные операции (рис. 6.3).

| Имена пользователей | Имена файлов: modern.txt | win.exe | class.dbf | unix.ppt |
|---|---|---|---|---|
| kira | читать | выполнять | — | выполнять |
| genua | читать | выполнять | — | выполнять читать |
| nataly | читать | — | — | выполнять читать |
| victor | читать писать | — | создать | — |
| | | | | |

Рис. 6.3. Матрица прав доступа

Практически во всех операционных системах матрица прав доступа хранится «по частям», т. е. для каждого файла или каталога создается так называемый список управления доступом (Access Control List, ACL), в котором описываются права пользователей и групп пользователей на выполнение операций по отношению к этому файлу или каталогу. Список управления доступа является частью характеристик файла или каталога и хранится на диске в соответствующей области, например, в индексном дескрипторе inode файловой системы ufs. Не все файловые системы поддерживают списки управления доступом, например, его не поддерживает файловая система FAT, так как она разрабатывалась для однопользовательской однопрограммной операционной системы MS-DOS, для которой задача защиты от несанкционированного доступа неактуальна.

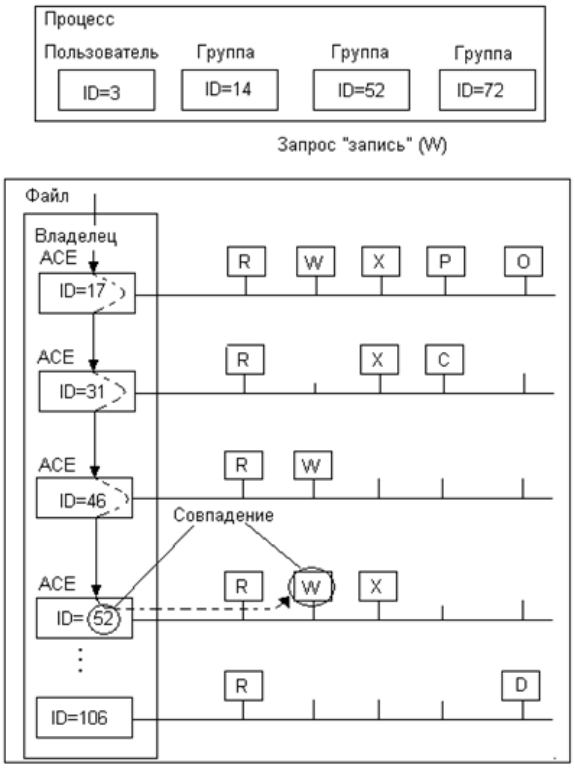



Рис. 6.4. Проверка прав доступа

Обобщенно формат списка управления доступом можно представить в виде набора идентификаторов пользователей и групп пользователей, в котором для каждого идентификатора указывается набор разрешенных операций над объектом (рис. 6.4). Говорят, что список ACL состоит из элементов управления доступом (Access Control Element, ACE), при этом каждый элемент соответствует одному идентификатору. Список ACL с добавленным к нему идентификатором владельца называют характеристиками безопасности.

В приведенном на рисунке примере процесс, который выступает от имени пользователя с идентификатором 3 и групп с идентификаторами 14, 52 и 72, пытается выполнить операцию записи (W) в файл. Файлом владеет пользователь с идентификатором 17. Операционная система, получив запрос на запись, находит характеристики безопасности файла на диске или в буферной системной области и последовательно сравнивает все идентификаторы процесса с идентификатором владельца файла и идентификаторами пользователей и групп в элементах ACE. В данном примере один из идентификаторов группы, от имени которой выступает процесс, а именно 52, совпадает с идентификатором одного из элементов ACE. Так как пользователю с идентификатором 52 разрешена операция чтения (признак W имеется в наборе операций этого элемента), то ОС разрешает процессу выполнение операции.

Описанная обобщенная схема хранения информации о правах доступа и процедуры проверки имеет в каждой операционной системе свои особенности, которые рассмотрим далее на примере операционных систем UNIX и Windows NT.

### 6.4.3. Организация контроля доступа в ОС UNIX

В ОС UNIX права доступа к файлу или каталогу определяются для трех субъектов:

- владельца файла (идентификатор User ID, UID);
- членов группы, к которой принадлежит владелец (Group ID, GID);
- всех остальных пользователей системы.

Учитывая, что в UNIX определены всего три операции над файлами и каталогами (чтение, запись, выполнение), характеристики безопасности файла включают девять признаков, задающих возможность выполнения каждой из трех операций для каждого из трех субъектов доступа. Например, если владелец файла разрешил себе выполнение всех трех операций, для членов группы – чтение и выполнение, а для всех остальных пользователей – только выполнение, то девять характеристик безопасности файла выглядят следующим образом:

*rwx r-x r—*

Здесь *r*, *w* и *x* обозначают операции чтения, записи и выполнения соответственно. Именно в таком виде выводит информацию о правах доступа к файлам команда просмотра содержимого каталога 1s. Суперпользователю UNIX все виды доступа позволены всегда, поэтому его идентификатор (он имеет значение 0) не фигурирует в списках управления доступом.

С каждым процессом UNIX связаны два идентификатора: пользователя, от имени которого был создан этот процесс, и группы, к которой принадлежит данный пользователь. Эти идентификаторы носят название реальных идентификаторов пользователя: Real User ID, RUID и реальных идентификаторов группы: Real Group ID, RGID. Однако при проверке прав доступа к файлу используются не эти идентификаторы, а так называемые эффективные идентификаторы пользователя: Effective User ID, EUID и эффективные идентификаторы группы: Effective Group ID, EGID (рис. 6.5).

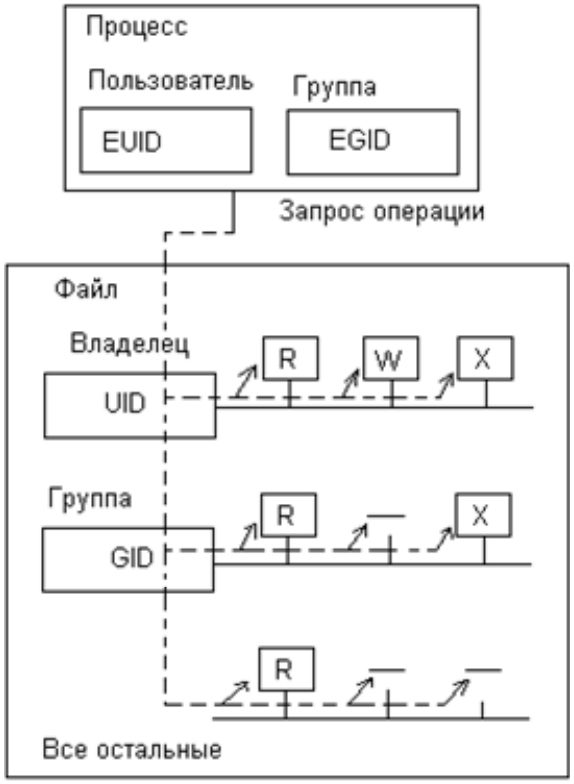



Рис. 6.5. Проверка прав доступа в UNIX

Введение эффективных идентификаторов позволяет процессу выступать в некоторых случаях от имени пользователя и группы, отличных от тех, которые ему достались при рождении. В исходном состоянии эффективные идентификаторы совпадают с реальными.

Случаи, когда процесс выполняет системный вызов exec запуска приложения, хранящегося в некотором файле, в UNIX связаны со сменой процессом исполняемого кода. В рамках данного процесса начинает выполняться новый код, и если в характеристиках безопасности этого файла указаны признаки разрешения смены идентификаторов пользователя и группы, то происходит смена эффективных идентификаторов процесса. Файл имеет два признака разрешения смены идентификатора – Set User ID on execution

(SUID) и Set Group ID on execution (SGID), которые разрешают смену идентификаторов пользователя и группы при выполнении данного файла.

Механизм эффективных идентификаторов позволяет пользователю получать некоторые виды доступа, которые ему явно не разрешены, но только с помощью вполне ограниченного набора приложений, хранящихся в файлах с установленными признаками смены идентификаторов. Пример такой ситуации приведен на рис. 6.6.

Первоначально процесс А имел эффективные идентификаторы пользователя и группы (12 и 23 соответственно), совпадающие с реальными. На каком-то этапе работы процесс запросил выполнение приложения из файла b.exe. Процесс может выполнить файл b.exe, хотя его эффективные идентификаторы не совпадают с идентификатором владельца и группы файла, так как выполнение разрешено всем пользователям.

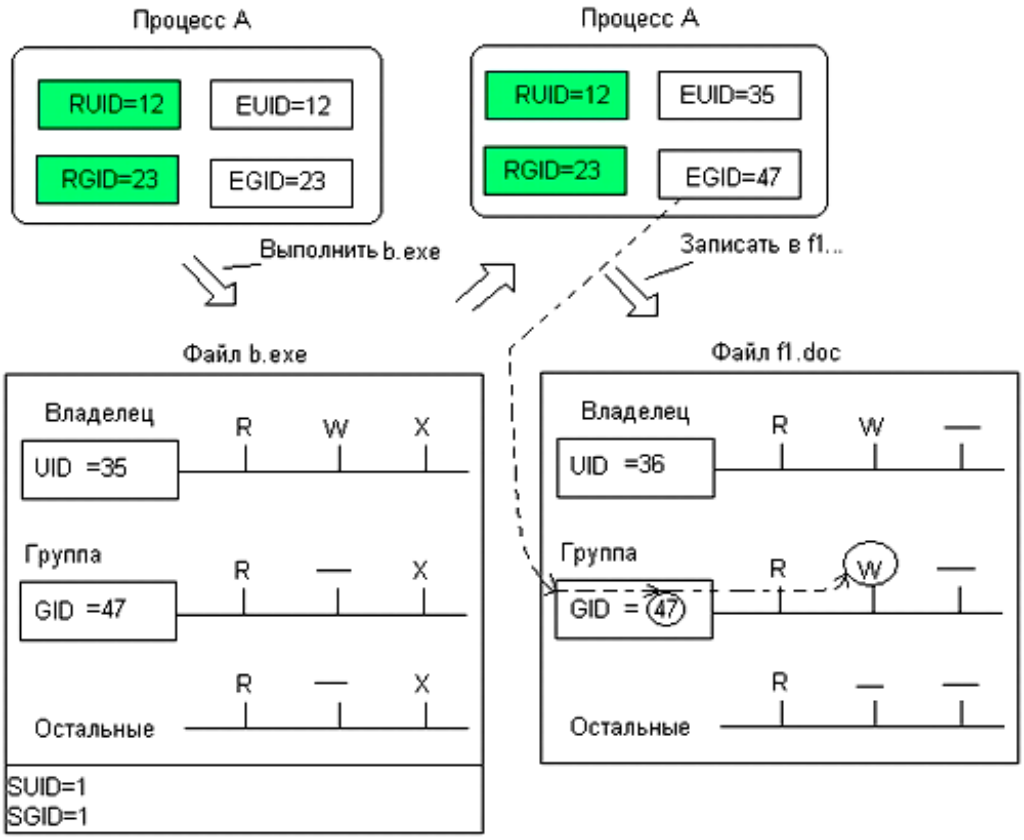



Рис. 6.6. Смена эффективных идентификаторов процесса

Файл b.exe имеет установленные признаки смены идентификаторов SUID и SGID, поэтому одновременно со сменой кода процесс меняет и значения эффективных идентификаторов (35 и 47). Вследствие этого при последующей попытке записать данные в файл f 1 .doc процессу А это удается, так как его новый эффективный идентификатор группы совпадает с

идентификатором группы файла f1.doc. Без смены идентификаторов эта операция для процесса А была бы запрещена.

Описанный механизм преследует те же цели, что и рассмотренный выше механизм подчиненных сегментов процессора Pentium.

Использование модели файла как универсальной модели разделяемого ресурса позволяет в UNIX применять одни и те же механизмы для контроля доступа к файлам, каталогам, принтерам, терминалам и разделяемым сегментам памяти.

Система управления доступом ОС UNIX была разработана в 70-е гг. XX в. и с тех пор мало изменилась. Эта достаточно простая система позволяет во многих случаях решить поставленные перед администратором задачи по предотвращению несанкционированного доступа, однако такое решение иногда требует слишком больших ухищрений или же вовсе не может быть реализовано.

### 6.4.4. Организация контроля доступа в ОС Windows NT

Система управления доступом в ОС Windows NT отличается высокой степенью гибкости, которая достигается за счет большого разнообразия субъектов и объектов доступа, а также детализации операций доступа.

Для разделяемых ресурсов в Windows NT применяется общая модель объекта, который содержит такие характеристики безопасности, как набор допустимых операций, идентификатор владельца, список управления доступом. Объекты в Windows NT создаются для любых ресурсов – файлов, каталогов, устройств, секций памяти, процессов – в том случае, когда они являются или становятся разделяемыми. Характеристики объектов в Windows NT делятся на две части – общую часть, состав которой не зависит от типа объекта, и индивидуальную, определяемую типом объекта.

Все объекты хранятся в древовидных иерархических структурах, элементами которых являются объекты-ветви (каталоги) и объекты-листья (файлы). Для объектов файловой системы такая схема отношений является прямым отражением иерархии каталогов и файлов. Для объектов других типов иерархическая схема отношений имеет свое содержание, например, для процессов она отражает связи «родитель-потомок», а для устройств отражает принадлежность к определенному типу устройств и связи устройства с другими устройствами, например SCSI-контроллера с дисками.

Проверка прав доступа для объектов любого типа выполняется централизованно с помощью монитора безопасности (Security Reference Monitor), работающего в привилегированном режиме. Централизация функций контроля доступа повышает надежность средств защиты инфор-

мации операционной системы по сравнению с распределенной реализацией, когда в различных модулях ОС имеются свои процедуры проверки прав доступа и вероятность ошибки программиста от этого возрастает.

Для системы безопасности Windows NT характерно наличие большого количества различных предопределенных (встроенных) субъектов доступа – как отдельных пользователей, так и групп. Так, в системе всегда имеются такие пользователи, как Administrator, System и Guest, а также группы Users, Administrators, Account Operators, Server Operators, Everyone и другие. Эти встроенные пользователи и группы наделены некоторыми правами, облегчая администратору работу по созданию эффективной системы разграничения доступа. При добавлении нового пользователя администратору остается только решить, к какой группе или группам отнести этого пользователя. Конечно, администратор может создавать новые группы, а также добавлять права к встроенным группам для реализации собственной политики безопасности, но во многих случаях встроенных групп оказывается вполне достаточно.

Windows NT поддерживает три класса операций доступа, которые отличаются типом субъектов и объектов, участвующих в этих операциях:

– *разрешения* (permissions) – это множество операций, которые могут быть определены для субъектов всех типов по отношению к объектам любого типа: файлам, каталогам, принтерам, секциям памяти и т. д. Разрешения по своему назначению соответствуют правам доступа к файлам и каталогам в ОС UNIX;

– *права* (user rights) определяются для субъектов типа группа на выполнение некоторых системных операций: установку системного времени, архивирование файлов, выключение компьютера и т. п. В этих операциях участвует особый объект доступа – операционная система в целом. В основном именно права, а не разрешения отличают одну встроенную группу пользователей от другой. Некоторые права у встроенной группы являются также встроенными – их у данной группы нельзя удалить. Остальные права встроенной группы можно удалять (или добавлять из общего списка прав);

– *возможности пользовател*ей (user abilities) определяются для отдельных пользователей на выполнение действий, связанных с формированием их операционной среды, например, изменение состава главного меню программ, возможность пользоваться пунктом меню Run (выполнить) и т. п. За счет уменьшения набора возможностей, которые по умолчанию доступны пользователю, администратор может «заставить» пользователя работать с той операционной средой, которую администратор считает наиболее подходящей и ограждающей пользователя от возможных ошибок.

Права и разрешения, данные группе, автоматически предоставляются ее членам, позволяя администратору рассматривать большое количество пользователей как единицу учетной информации и минимизировать свои действия.

Проверка разрешений доступа процесса к объекту производится в Windows NT в основном в соответствии с общей схемой доступа.

При входе пользователя в систему для него создается так называемый токен доступа (access token), включающий идентификатор пользователя и идентификаторы всех групп, в которые входит пользователь. В токене также имеется список управления доступом (ACL) по умолчанию, который состоит из разрешений и применяется к создаваемым процессом объектам, и список прав пользователя на выполнение системных действий.

Все объекты, включая файлы, потоки, события, токены доступа, при создании снабжаются дескриптором безопасности. Дескриптор безопасности содержит список управления доступом – ACL. Владелец объекта, обычно пользователь, создавший его, обладает правом избирательного управления доступом к объекту и может изменять ACL объекта, чтобы позволить или не позволить другим осуществлять доступ к объекту. Встроенный администратор Windows NT, в отличие от суперпользователя UNIX, может не иметь некоторых разрешений на доступ к объекту. Для реализации этой возможности идентификаторы администратора и группы администраторов могут входить в ACL, как и идентификаторы рядовых пользователей. Однако администратор все же имеет возможность выполнить любые операции с любыми объектами, так как он всегда может стать владельцем объекта, а затем уже как владелец получить полный набор разрешений. Однако вернуть владение предыдущему владельцу объекта администратор не может, поэтому пользователь всегда может узнать о том, что с его файлом или принтером работал администратор.

При запросе процессом некоторой операции доступа к объекту в Windows NT управление всегда передается монитору безопасности, который сравнивает идентификаторы пользователя и групп пользователей из токена доступа с идентификаторами, хранящимися в элементах ACL объекта. В отличие от UNIX в элементах ACL Windows NT могут существовать списки как разрешенных, так и запрещенных для пользователя операций.

Система безопасности могла бы осуществлять проверку разрешений каждый раз, когда процесс использует объект. Но список ACL состоит из многих элементов, процесс в течение своего существования может иметь доступ ко многим объектам, и количество активных процессов в каждый момент времени велико. Поэтому проверка выполняется только при каждом открытии, а не при каждом использовании объекта.

Для смены в некоторых ситуациях процессом своих идентификаторов в Windows NT используется механизм олицетворения (impersonation). В Windows NT существуют простые субъекты и субъекты-серверы. Простой субъект – это процесс, которому не разрешается смена токена доступа и соответственно смена идентификаторов. Субъект-сервер – это процесс, работающий в качестве сервера и обслуживающий процессы своих клиентов (например, процесс файлового сервера). Такому процессу разрешается получить токен доступа у процесса-клиента, запросившего у сервера выполнение некоторого действия, и использовать его при доступе к объектам.

В Windows NT однозначно определены правила, по которым вновь создаваемому объекту назначается список ACL. Если вызывающий код во время создания объекта явно задает все права доступа к вновь создаваемому объекту, то система безопасности приписывает этот ACL объекту.

Если же вызывающий код не снабжает объект списком ACL, а объект имеет имя, то применяется принцип наследования разрешений. Система безопасности просматривает ACL того каталога объектов, в котором хранится имя нового объекта. Некоторые из входов ACL каталога объектов могут быть помечены как наследуемые. Это означает, что они могут быть приписаны новым объектам, создаваемым в этом каталоге.

В том случае, когда процесс не задал явно список ACL для создаваемого объекта и объект-каталог не имеет наследуемых элементов ACL, используется список ACL по умолчанию из токена доступа процесса.

Наследование разрешений употребляется наиболее часто при создании нового объекта. Особенно эффективно оно при создании файлов, так как эта операция выполняется в системе наиболее часто.

## 6.5. Вопросы и задания для самопроверки

1. Дайте определения понятиям «файл» и «файловая система».
2. Опишите структуру тома NTFS.
3. Перечислите основные атрибуты файла.
4. Назовите два способа организации файловых операций. Чем они отличаются?
5. Какую информацию содержит индексный дескриптор файла?
6. Для чего используются блокировки файла?
7. Приведите примеры стандартных файлов ввода-вывода.
8. Назовите два основных метода доступа к файлам и каталогам.
9. Какие операции в ОС UNIX может выполнять с файлом, имеющим параметры доступа «rwxr-xr--» владелец этого файла?
10. Какие три класса операций доступа поддерживает Windows NT?

## ЛИТЕРАТУРА

1. Гранже, М. OS / 2 : Принципы построения и установка / М. Гранже, Ф. Менсье. – М. : Мир, 1991.

2. Гук, М. Процессоры Pentium И, Pentium Pro и просто Pentium / М. Гук. – СПб : «Питер», 1999.

3. Дэй, М. Программирование NLM в NetWare 4.0 / Дэй, М., Кунц, М., Маршалл, Д. – М. : «ЛОРИ», 1994.

4. Зубанов, Ф. В. Перспектива : Windows NT 5.0 / Зубанов, Ф. В. – М. : Издательский отдел «Русская редакция» ТОО «Channel Trading Ltd.», 1998.

5. Каплан, А. Windows 2000 изнутри / А. Каплан, М. Ш. Нильсен. – ДМК, 2000.

6. Кастер, Х. Основы Windows NT и NTFS / Х. Кастер. – М. : Издательский отдел «Русская редакция» ТОО «Channel Trading Ltd.», 1996.

7. Максвелл, С. Ядро Linux в комментариях / С. Максвелл; пер. с англ. – К. : «ДиаСофт», 2000.

8. Минаси, М. OS / 2 : Warp изнутри. Т. 1, 2 / М. Минаси, Б. Камарда. – СПб : «Питер», 1996.

9. Олифер, В. Г.Компьютерные сети. Принципы, технологии, протоколы / В. Г. Олифер, Н. А. Олифер. – СПб : «Питер», 1999.

10. Рихтер, Д. Windows для профессионалов : Программирование для Windows NT 4.0 и Windows 95 на базе Win32 API / Д. Рихтер; пер. с англ. – М. : Издательский отдел «Русская редакция» ТОО «Channel Trading Ltd.», 1997.

11. Робачевский, А. Операционная система UNIX / А. Робачевский. – BHV, 1999.

12. Фролов, А. В. Защищенный режим процессоров Intel 80286, 80386, 80486: Практическое руководство по использованию защищенного режима / Фролов, А. В., Фролов, Г. В. – М. : «Диалог-МИФИ», 1993.

13. Хьюз, Дж. Ф. Сети NetWare 5. Руководство от Novell / Дж. Ф. Хьюз, В. Т. Блейер. – Вильямс, 2000.

14. Bach, Maurice J. The Design of the UNIX Operating System / Maurice J. Bach. – Prentice-Hall, 1986.

15. Black, Uyless. Internet Security protocols : protecting IP Traffic / Uyless Black. – Prentice-Hall, 2000.

16. Distributed Systems, 2 / e. Edited by S. Mullender. – Addison-Wesley, 1998.

17. Ford, W. Computer Communications Security / W. Ford. – Prentice-Hall, 1994.

18. Goodheart, B. The Magic Garden Explained : The Internals of UNIX System V Release 4, An Open Systems Design / B. Goodheart, J. Cox. – Prentice Hall, 1994.

19. Stalling, W. Operating Systems / W. Stalling – Prentice Hall, 1995.

20. Tannenbaum, A. S. Modern Operating Systems / A. S. Tannenbaum. – Prentice-Hall, 1992.

21. Tannenbaum, A. S. Distributed Operating Systems / A. S. Tannenbaum. – Prentice-Hall, 1995.

*Учебное издание*

ТРАВКИН Олег Николаевич

# СИСТЕМНОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ

Учебно-методический комплекс для студентов специальностей
1-40 01 01 «Программное обеспечение информационных технологий»,
1-40 02 01 «Вычислительные машины, системы и сети»

Редактор *Н. М. Важенина*

Дизайн обложки *В. А. Виноградовой*

Подписано в печать 27.02.09. Формат 60x84 1/16. Гарнитура Таймс. Бумага офсетная.
Ризография. Усл.-печ. л. 12,99. Уч.-изд. л. 12,14. Тираж 75 экз. Заказ 361.

Издатель и полиграфическое исполнение:
учреждение образования «Полоцкий государственный университет»

ЛИ № 02330/0133020 от 30.04.2004 ЛП № 02330/0133128 от 27.05.2004

211440 г. Новополоцк, ул. Блохина, 29