Федеральное агентство по образованию
АМУРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ГОУВПО "АмГУ"
Факультет математики и информатики

УТВЕРЖДАЮ
Зав. кафедрой МАиМ

_____ Т.В. Труфанова

«___»_____ 2007 г.

ОПЕРАЦИОННЫЕ СИСТЕМЫ И СЕТЕВЫЕ ТЕХНОЛОГИИ

УЧЕБНО-МЕТОДИЧЕСКИЙ КОМПЛЕКС ПО ДИСЦИПЛИНЕ

для специальности 010501 – "Прикладная математика и информатика"

Составитель: А.В. Рыженко

Благовещенск
2007 г.

ББК

Печатается по решению редакционно-издательского совета факультета математики и информатики Амурского государственного университета

Рыженко А.В.

Операционные системы и сетевые технологии: Учебно-методический комплекс по дисциплине для студентов АмГУ очной формы обучения специальности 010501 "Прикладная математика и информатика". – Благовещенск: Амурский гос. ун-т, 2007. – 371 с.

Учебно-методический комплекс по дисциплине "Операционные системы и сетевые технологии" предназначен для студентов специальности 010501 – "Прикладная математика и информатика" очной формы обучения, призван помочь ведущим преподавателям и студентам в организации процесса изучения дисциплины. Комплекс содержит рабочую программу дисциплины, план-конспект лекций, материалы для проведения практических семинаров, контролирующие материалы для осуществления промежуточного и итогового контроля, справочный материал и библиографический список.

I. РАБОЧАЯ ПРОГРАММА ДИСЦИПЛИНЫ

Рабочая программа по дисциплине "Операционные системы и сетевые технологии" для специальности 010501 – "Прикладная математика и информатика".

Курс 3. Семестр 5, 6. Лекции 72 (36+36) час. Экзамен 6 семестр. Практические (семинарские) занятия 72 (36+36) час. Зачет 5 семестр. Лабораторные занятия (нет). Самостоятельная работа 76 час. Всего часов 220 час.

Составитель А.Н. Гетман, доцент. Факультет математики и информатики. Кафедра математического анализа и моделирования. Благовещенск, 2006 г.

1. ЦЕЛИ И ЗАДАЧИ ДИСЦИПЛИНЫ, ЕЕ МЕСТО В УЧЕБНОМ ПРОЦЕССЕ

1.1. Цель преподавания дисциплины.

Цель курса – изучение принципов построения, назначения, теоретических основ функционирования и практического использования операционных систем как эффективного средства управления процессами обработки данных в современных ЭВМ.

Обучение студентов принципам организации, построения современных локальных и глобальных компьютерных сетей, методологии передачи данных, построению различных структур обмена данными между ЭВМ, методике выбора оборудования и расчета основных параметров систем и устройств сетевой обработки данных, разработке алгоритмов обмена данными в сетях.

Изучение принципов организации и работы в сети Internet, основных ресурсов Internet (WWW, FTP, e-mail, Telnet и др.), ознакомление со средствами навигации и поисковыми системами. Приобретение навыков разработки WEB-документов.

1.2. Задачи изучения дисциплины.

По окончанию данного курса студент должен знать и уметь использовать:

-современное состояние теории операционных систем;

-принципы и методы разработки, построения современных операционных систем;

-владеть такими понятиями как вычислительный процесс и файловая система, их реализация с помощью операционной системы;

-уметь создавать программы, расширяющие возможности операционных систем;

-иметь устойчивые практические навыки работы с MS DOS, Windows;

-ознакомиться с операционными системами класса Unix и NetWare;

-современные подходы к реализации сетей ЭВМ;

-способы управления сетями и алгоритмы передачи данных;

-технологию работы с Microsoft Internet Explorer, реализовывать наиболее

важные сервисы Internet, искать требуемую информацию в сети;

-самостоятельно создавать Web-страницы.

1.3. Перечень дисциплин с указанием разделов (тем), усвоение которых студентами необходимо при изучении данной дисциплины.

Дисциплина «Операционные системы и сетевые технологии» тесно связана с изучением курсов государственного образовательного стандарта «Информатика» и «Системное и прикладное программное обеспечение» и является основой для изучения дальнейших дисциплин, использующих ЭВМ и практику программирования.

2. СОДЕРЖАНИЕ ДИСЦИПЛИНЫ

2.1. Наименование тем, их содержание, объем в лекционных часах.

5 семестр

Тема 1. Принципы построения современных операционных систем (6 часов).

Назначение и функции операционных систем, их классификация. История развития операционных систем. Автономные и сетевые операционные системы. Архитектура и способы построения. Основные требования к современным операционным системам.

Тема 2. Загрузка программ (2 часа).

Абсолютная и относительная загрузка. Разделы памяти. Базовая адресация. Загрузка операционной системы.

Тема 3. Управление оперативной памятью (6 часов).

Функции ОС по управлению памятью. Алгоритмы распределения памяти. Система с базовой виртуальной адресацией. Сегментная и страничная виртуальная память. Свопинг и виртуальная память. Кэш-память.

Тема 4. Вычислительный процесс (6 часов).

Мультипрограммирование. Планирование процессов и потоков. Компьютер и внешние события. Синхронизация процессов и потоков. Примитивы синхронизации. Аппаратная поддержка мультипрограммирования.

Тема 5. Файловая система (4 часа).

Файлы с точки зрения пользователя. Простые и сложные файловые системы. Логическая и физическая организация файловой системы. Дескриптор файла. Файловые операции. Контроль доступа к файлам. Дополнительные возможности файловых систем.

Тема 6. Задачи операционной системы по управлению внешними устройствами (2 часа).

Доступ к внешним устройствам. Порты передачи данных. Драйверы внешних устройств. Функции и архитектура драйверов, запросы к драйверу. Сервисы ядра, доступные драйверам.

Тема 7. Обеспечение безопасности в операционных системах (2 часа).

Основные понятия безопасности. Базовые технологии безопасности – аутентификация, авторизация, ресурсные квоты, аудит.

Тема 8. Сетевые операционные системы (4 часа).

Сетевые и распределенные ОС. Функциональные компоненты сетевой ОС. Сетевые службы и сервисы. Концепции распределенной обработки в сетевых ОС. Сетевая безопасность.

Тема 9. Обзор архитектур современных операционных систем (4 часа).

MVS, OS/390, z/OS. Семейство Unix. Семейство CP/M. Интерфейсы и основные стандарты в области системного программного обеспечения.

6 семестр

Тема 10. Общие принципы построения вычислительных сетей (4 часа).

Основные понятия компьютерных систем. Эволюция вычислительных систем. Предпосылки создания компьютерных сетей. Основные программные и аппаратные компоненты сети. Проблемы построения сетей. Топология физических связей. Организация совместного использования линий связи. Адресация компьютеров. Физическая и логическая структуризация сети. Сетевые службы.

Тема 11. Архитектура открытых систем (4 часа).

Понятие «открытая система». Многоуровневый подход. Интерфейс. Уровни модели OSI. Источники стандартов. Протокол. Стандартные стеки коммуникационных протоколов.

Тема 12. Основы передачи данных. Каналы связи (6 часов).

Линии связи, их характеристика. Пропускная способность, помехоустойчивость и достоверность линии связи. Аппаратура линий связи. Стандарты кабелей. Методы передачи дискретных данных – аналоговая модуляция, цифровое и логическое кодирование, дискретная модуляция аналоговых сигналов. Асинхронная и синхронная передачи.

Методы передачи данных канального уровня. Коммутация каналов. Коммутация пакетов и сообщений.

Тема 13. Локальные сети (4 часа).

Цели создания и преимущества использования локальных сетей. Особенности организации локальных сетей. Топология локальных сетей. Базовые технологии, протоколы и стандарты локальных сетей. Программное обеспечение локальных сетей. Построение локальных сетей по стандартам физического и канального уровней. Роль и функции администратора локальных сетей.

Тема 14.Сетевой уровень как средство построения больших сетей (4 часа).

Принципы объединения сетей на основе протоколов сетевого уровня. Реализация межсетевого взаимодействия средствами TCP/IP. Адресация в IP-сетях. Протокол IP. Протоколы маршрутизации в IP-сетях. Основные характеристики маршрутизаторов и концентраторов. Корпоративные сети.

Тема 15. Глобальные сети (6 часов).

Основные понятия и определения глобальных сетей. Типы глобальных сетей. Глобальные сети на основе выделенных линий. Глобальные сети на основе сетей с коммутаций каналов. Компьютерные глобальные сети с коммутацией пакетов. Глобальные сети с удаленным доступом.

Тема 16. Глобальная сеть Internet (4 часа).

Краткая история Internet. Структура и основные принципы работы в Internet. Способы доступа к Internet. Адресация в Internet. Возможности, предоставляемые сетью Internet. Защита информации в сети. Internet в России.

Тема 17. Средства анализа и управления сетями (4 часа).

Функции и архитектура систем управления сетями. Архитектура систем управления сетями. Стандарты систем управления. Стандарты систем управления SNMP, OSI. Мониторинг и анализ сетей.

2.2. Практические и семинарские занятия, их содержание и объем в

часах.

5 семестр

Тема 1. Операционная система MS-DOS (2 часа).

Тема 2. Настройка конфигурации в MS-DOS: файлы CONFIG.SYS, AUTOEXEC.BAT. Настройка на национальные стандарты, использование оперативной памяти, кэширование дисков, диалоговые файлы конфигурации, оптимизация файлов конфигурации (4 часа).

Тема 3. Командные файлы MS-DOS: выполнение командного файла, пошаговое выполнение командных файлов, проверка условий и переходы в командном файле, создание диалоговых командных файлов (4 часа).

Тема 4. Операционные системы семейства Windows. Основные особенности, понятия и приемы работы (4 часа).

Тема 5. Операционные системы WINDOWS 98, WINDOWS 2000. Распределение ресурсов. Настройка (4 часа).

Тема 6. Командные файлы (2 часа).

Тема 7. Иерархическая организация памяти ЭВМ (2 часа).

Тема 8. Понятие процесса. Служебные процессы ОС (2 часа).

Тема 9. Файловые системы. Структура файлов и способы их организации (2 часа).

Тема 10. Организация работы с внешними устройствами (2 часа).

Тема 11. Реестры Windows (4 часа).

Тема 12. Современные операционные системы (UNIX, Linux, NetWare, Windows NT и др.) (4 часа).

6 семестр

Тема 13. Организация и функционирование компьютерных сетей. Топология сетей (2 часа).

Тема 14. Модель OSI (2 часа).

Тема 15. Методы доступа и протоколы передачи данных (2 часа).

Тема 16. Методы передачи данных (2 часа).

Тема 17. Локальные сети (2 часа).

Тема 18. Глобальные сети (4 часа).

Тема 19. Глобальная сеть Internet. Способы доступа к Internet. Адресация в Internet (2 часа).

Тема 20. Возможности, предоставляемые сетью Internet (4 часа).

Тема 21. WWW. Средства навигации в WWW (2 часа).

Тема 22. Стратегия поиска информации в Internet (2 часа).

Тема 23. Режим работы off-line. Хранение информации о WEB-страницах (2 часа).

Тема 24. Создание WEB-документов (4 часов).

Тема 25. Защита информации в Internet (2 часа).

Тема 26. Средства анализа и управления сетями. Стандарты систем управления SNMP, OSI (4 часа).

2.3. Перечень промежуточных форм контроля знаний студентов.

По данному курсу предполагается систематическое проведение контрольных работ. Типовой расчет включает в себя задания по всему семестру.

Варианты тем типовых расчетов (6 семестр):

1.BIOS.

2.Конфигурация системы MS DOS.

3.Обзор реестров Windows.

4.Использование реестров Windows для решения конкретных задач пользователя.

5.Варианты сохранения и восстановления реестра.

6.Операционные системы класса UNIX. История развития и общая характеристика.

7.Операционная система Open BSD.

8.Операционная система FreeBSD.

9.Операционные системы Linux. Сравнение различных версий.

10. Установка и настройка ОС Linux Mandrake.

11. Установка и настройка ОС Alt Linux.

12. Установка и настройка ОС ASP Linux.

13. Сетевые операционные системы. Сравнительный анализ.

14. Операционные системы мобильных компьютеров.

15. Операционные системы реального времени.

16. Сравнительный анализ браузеров – просмотрщиков web-страниц (Internet Explorer, Mozilla Firefox, Netscape Naviganor, Opera и др.).

17. Сравнительный анализ почтовых клиентов (Outlook Express, The Bat!, Mozilla ThunderBird и др.).

18. Поисковые системы Интернета.

19. Язык HTML как средство разработки WEB-приложений.

20. Особенности организации локальной сети.

Выбор темы типового расчета оговариваются с каждым студентом отдельно.

2.4. Самостоятельная работа студентов.

1.Выполнение типового расчета (20 часов).

2.Установка на ПК и настройка виртуальной машины – Vmware (10 часов).

3.Инсталляция на виртуальную машину различных ОС (Dos, Windows9x, 2k, XP, Linux – Alt, ASP, Mandrake, Unix – FreeBSD). Настройка операционных систем, работа с ними, настройка сети (26 часов).

4.Работа с ресурсами Internet (WWW, FTP, e-mail, Telnet и др.) (10 часов).

5.Работа с дополнительной литературой при подготовке к практическим занятиям и написании рефератов (10 часов).

2.5. Вопросы к зачету (5 семестр).

1.Определение и назначение ОС. История создания ОС.

2.Основные функции ОС. Классификации ОС.

3.Понятие «процесс». Состояние процесса.

4.Планирование процессов. Понятие очереди.

5.Планирование процесса. Критерии планирования.

6.Стратегия планирования «Первый пришел – первый обслуживается»

(FCFS).

7.Стратегия планирования «Shortest Job Firs» (SJF).

8.Приоритетное планирование.

9.«Карусельная стратегия планирования».

10. Планирование с использованием многоуровневой очереди (Multilevel Queue Sheduling).

11. Планирование с использованием многоуровневой очереди с обратными связями (Multilevel Feedback Queue Sheduling).

12. Свопинг.

13. Понятие смежного и несмежного размещения процессов.

14. Однопрограммный режим.

15. Мультипрограммный режим с фиксированными границами.

16. Мультипрограммный режим с переменными разделами и уплотнением памяти.

17. Основные стратегии заполнения свободного раздела.

18. Страничная организация памяти. Базовый метод. Аппаратная поддержка страничной организации памяти.

19. Сегментная организация памяти. Базовый метод.

20. Разделение сегмента между несколькими процессами.

21. Управление виртуальной памятью. Разбиение на страницы по запросу.

22. Файл. Основные виды манипуляции с файлом.

23. Файловая система, ее основные функции.

24.Иерархия данных в ОС.

25. Объединение в блоки и буферизация.

26. Последовательная и индексно-последовательная организация файлов.

27. Прямая и библиотечная организация файлов.

28. Связанное и несвязанное распределение файлов.

29. Распределение памяти при помощи списка секторов.

30. Поблочное распределение памяти.

31. Дескриптор файла.

32. Управление доступом к файлам.
33. Доступ к внешним устройствам. Порты передачи данных.
34. Драйверы внешних устройств. Функции и архитектура драйверов, запросы к драйверу. Сервисы ядра, доступные драйверам.
35. Базовые технологии безопасности – аутентификация, авторизация, ресурсные квоты, аудит.
36. Классы вирусов и основные антивирусные средства.
37. Защита от сбоев, защита от несанкционированного доступа.
38. Сетевые и распределенные ОС. Функциональные компоненты сетевой ОС.
39. Концепции распределенной обработки в сетевых ОС. Сетевая безопасность.
40. Обзор архитектур современных операционных систем.
41. Файлы конфигурации и командные файлы ОС. Назначение и использование.

2.7. Экзаменационные вопросы (6 семестр).

1.Основные понятия компьютерных систем. Эволюция вычислительных систем. Предпосылки создания компьютерных сетей.

2.Основные программные и аппаратные компоненты сети. Проблемы построения сетей.

3.Топология физических связей.

4.Организация совместного использования линий связи. Адресация компьютеров.

5.Физическая и логическая структуризация сети. Сетевые службы.

6.Понятие «открытая система». Архитектура открытых систем.

7.Многоуровневый подход. Уровни модели OSI.

8.Протокол. Стандартные стеки коммуникационных протоколов.

9.Основы передачи данных. Линии связи, их характеристика.

10. Пропускная способность, помехоустойчивость и достоверность линии

связи.

11. Аппаратура линий связи. Стандарты кабелей.
12. Методы передачи дискретных данных. Асинхронная и синхронная передачи.
13. Методы передачи данных канального уровня.
14. Коммутация каналов.
15. Коммутация пакетов и сообщений.
16. Цели создания и преимущества использования локальных сетей. Особенности организации локальных сетей.
17. Топология локальных сетей.
18. Базовые технологии, протоколы и стандарты локальных сетей.
19. Построение локальных сетей по стандартам физического и канального уровней.
20. Принципы объединения сетей на основе протоколов сетевого уровня.
21. Реализация межсетевого взаимодействия средствами TCP/IP. Адресация в IP-сетях.
22. Протокол IP. Протоколы маршрутизации в IP-сетях.
23. Основные характеристики маршрутизаторов.
24. Основные характеристики концентраторов.
25. Корпоративные сети.
26. Основные понятия и определения глобальных сетей. Типы глобальных сетей.
27. Глобальные сети на основе выделенных линий.
28. Глобальные сети на основе сетей с коммутаций каналов.
29. Компьютерные глобальные сети с коммутацией пакетов.
30. Глобальные сети с удаленным доступом.
31. Глобальная сеть Internet. Краткая история Internet.
32. Структура и основные принципы работы в Internet.
33. Способы доступа к Internet.
34. Адресация в Internet.

35. Возможности, предоставляемые сетью Internet.
36. Основные концепции WWW. Программы просмотра. Домашняя страница.
37. Средства навигации в WWW. Поисковые системы.
38. E-mail. Принципы работы с электронной почтой.
39. Сервис FTP – протокол передачи файлов.
40. Система GOPHER.
41. Система USNET.
42. Система Telnet – взаимодействие с другими компьютерами.
43. Защита информации в сети.
44. Функции и архитектура систем управления сетями.
45. Стандарты систем управления (SNMP, OSI).
46. Мониторинг и анализ сетей.

3. УЧЕБНО-МЕТОДИЧЕСКИЕ МАТЕРИАЛЫ ПО ДИСЦИПЛИНЕ

3.1. Перечень обязательной (основной) литературы.

1.Золотов С. Протоколы Internet. – СПб.: BHV – Санкт-Петербург, 1998.

2.Иртегов Д. Введение в операционные системы. – СПб.: БХВ-Петербург, 2002.

3.Компьютерные сети. Учебный курс, 2-е изд. – Microsoft Press, Русская редакция, 1998.

4.Олифер В.Г., Олифер Н.А. Компьютерные сети. Принципы, технологии, протоколы. – СПб.: Питер, 2002.

5.Олифер В.Г., Олифер Н.А. Сетевые операционные системы. – СПб.: Питер, 2002.

6.Петров В.Н. Информационные системы. – СПб.: Питер, 2002.

3.2. Перечень дополнительной литературы.

1.Анин Б. Защита компьютерной информации. – СПб.: БХВ–Санкт-Петербург, 2000.

2.Баурн С. Операционная система UNIX. – М.: Мир, 1986.

3.Данкин Р. Профессиональная работа с MS DOS. М.: Мир, 1993.

4.Каплан Нильсен Windows 2000 изнутри. – М.: ДМК, 2000.

5.Краковяк С. Основы организации и функционирования ОС ЭВМ. – М: Мир, 1988.

6.Панкратов Е. Операционная система MS DOS 6.22. Справочное пособие.– М.: Познавательная книга плюс, 2001.

7.Робачевский А. Операционная система UNIX. – BHV, 1999.

3.3. Методическое обеспечение курса.

1. Резниченко Е.С., Решетнева Т.Г., Чугунова О.В. Интернет-технологии. Учебно-методическое пособие. Благовещенск: Амурский гос. ун-т, 2002.

3.4. Перечень наглядных и иных пособий.

1. Карточки с заданиями к практическим семинарам / *А.В. Рыженко*.

3.5. Средства обеспечения освоения дисциплины.

1. Операционная система Linux.
2. Операционная система Windows.
3. Операционная система MS-DOS.
4. Операционная система OS/2.
5. Операционная система NETWARE.

4. МАТЕРИАЛЬНО-ТЕХНИЧЕСКОЕ ОБЕСПЕЧЕНИЕ ДИСЦИПЛИНЫ

Компьютерный класс кафедры МАиМ.

5. КРИТЕРИИ ОЦЕНКИ ЗНАНИЙ

5.1. Требования к знаниям студентов, предъявляемые на зачете.

На зачете студенту предлагается ответить на один вопрос из предлагаемого списка и ответить на дополнительные вопросы по теме.

Знания студента оцениваются на «зачтено» при полном ответе на вопрос и удовлетворительном ответе на дополнительные вопросы преподавателя.

Оценка «не зачтено» ставится при незнании вопроса, предлагаемого студенту на зачетном занятии.

5.2. Требования к знаниям студентов, предъявляемые на экзамене.

Необходимым условием допуска на экзамен является выполнение всех лабораторных работ по дисциплине. В экзаменационный билет входят три вопроса из различных разделов курса.

Знания студента оцениваются на «отлично» при полном изложении теоретического материала экзаменационного билета, ответах на дополнительные вопросы со свободной ориентацией в материале и других литературных источниках.

Оценка «хорошо» ставится при твердых знаниях студентом всех разделов курса (в пределах конспекта лекций) и при преимущественно правильных ответах на дополнительные вопросы части (допускаются нетвердое знание одного – двух вопросов билета).

Оценку «удовлетворительно» студент получает, если дает неполные ответы на теоретические вопросы билета, показывая поверхностное знание учебного материала, владение основными понятиями и терминологией; при неверном ответе на билет или на дополнительные вопросы. Допускается полное незнание одного из вопросов билета.

Оценка «неудовлетворительно» выставляется за незнание студентом одного из разделов курса, если студент не дает ответы на теоретические вопросы билета, показывая лишь фрагментарное знание учебного материала, незнание основных понятий и терминологии, при полном незнании двух вопросов из трех предлагаемых в билете.

II. ГРАФИК САМОСТОЯТЕЛЬНОЙ УЧЕБНОЙ РАБОТЫ СТУДЕНТОВ ПО ДИСЦИПЛИНЕ НА КАЖДЫЙ СЕМЕСТР С УКАЗАНИЕМ ЕЕ СОДЕРЖАНИЯ, ОБЪЕМА В ЧАСАХ, СРОКОВ И ФОРМ КОНТРОЛЯ

График самостоятельной учебной работы студентов по дисциплине (с указанием ее содержания, объема в часах, сроков и форм контроля) приведен в рабочей программе дисциплины.

III. МЕТОДИЧЕСКИЕ РЕКОМЕНДАЦИИ ПО ПРОВЕДЕНИЮ ПРАКТИЧЕСКИХ СЕМИНАРОВ, ДЕЛОВЫХ ИГР, РАЗБОРУ СИТУАЦИЙ И Т. П. СПИСОК РЕКОМЕНДУЕМОЙ ЛИТЕРАТУРЫ (ОСНОВНОЙ И ДОПОЛНИТЕЛЬНОЙ)

Проведение деловых игр, разбор ситуаций и т. п. рабочей программой дисциплины не предусмотрены.

Форма проведения практического семинара: а) приветствие студентов, 1 мин.; б) определение личного состава студенческой группы, 4 мин.; в) объявление тематики и вопросов практического семинара, 1 мин.; г) выполнение заданий, 70 мин.; д) подведение итогов семинара, 13 мин.; е) прощание со студентами, 1 мин.

Список рекомендуемой литературы (основной и дополнительной) приведен в рабочей программе.

IV. КРАТКИЙ КОНСПЕКТ ЛЕКЦИЙ (ПО КАЖДОЙ ТЕМЕ) ИЛИ ПЛАН-КОНСПЕКТ

5 семестр

Операционные системы и среды

В англоязычной технической литературе термин *System Software (системное* программное обеспечение) означает программы и комплексы программ, являющиеся общими для всех, кто совместно использует технические средства компьютера, и применяемое как для автоматизации разработки (создания) новых программ, так и для организации выполнения программ, так и для организации выполнения программ существующих. *С* этих позиций системное программное обеспечение может быть разделено на следующие пять групп**:**

1.Операционные системы.

2.Системы управления файлами.

3.Интерфейсные оболочки для взаимодействия пользователя с ОС и программные среды.

4.Системы программирования.

5. Утилиты.

Рассмотрим группы системных программ.

1. Под ***операционной системой (ОС)*** обычно понимают комплекс управляющих и обрабатывающих программ, который, с одной стороны, выступает как интерфейс между аппаратурой компьютера и пользователем с его задачами, а с другой – предназначен для наиболее эффективного использования ресурсов вычислительной системы и организации надёжных вычислений. Любой из компонентов прикладного программного обеспечения обязательно работает под управлением ОС. На рис. 1 изображена обобщенная структура программного обеспечения вычислительной системы. Видно, что ни один из компонентов программного обеспечения, за исключением самой ОС, не имеет непосредственного доступа к аппаратуре компьютера. Даже пользователи взаимодействуют со своими программами через интерфейс ОС. Любые их команды, прежде чем попасть в прикладную программу, сначала проходят через ОС.

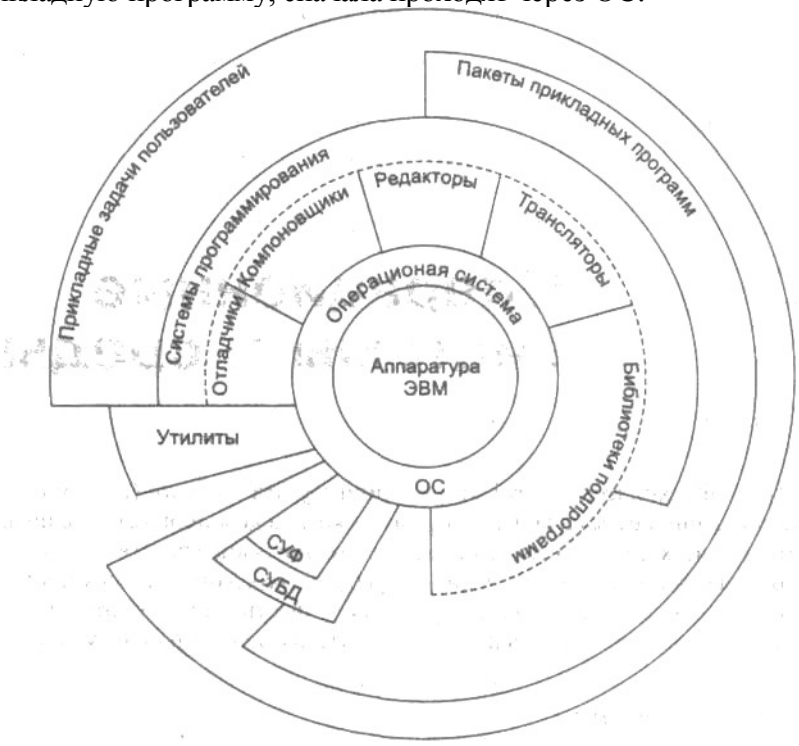

Рис. 1. Обобщенная структура программного обеспечения вычислительной системы

Основными **функциями**, которые выполняет **ОС**, являются следующие:

– прием от пользователя (или от оператора системы) заданий или команд, сформулированных на соответствующем языке – в виде директив (команд) оператора или

в виде указаний (своеобразных команд) с помощью соответствующего манипулятора (например, помощью мыши), - и их обработка;

– приём и использование программных запросов на запуск, приостановку, остановку других программ;

– загрузка в оперативную память подлежащих исполнению программ;

– инициализация программы (передача ей управления, в результате чего процессор исполняет программу);

– идентификация всех программ и данных;

– обеспечение работы систем управлений файлами (СУФ) и/или систем управления базами данных (СУБД), что позволяет резко увеличить эффективность всего программного обеспечения;

– обеспечений режима мультипрограммирования, то есть выполнение двух или более программ на одном процессоре, создающее видимость их одновременного исполнения;

– обеспечение функций по организации и управлению всеми операциями ввода/вывода;

– удовлетворение жёстким ограничениям на время ответа в режиме реального времени (характерно для соответствующих ОС);

– распределение памяти, а в большинстве современных систем и организация виртуальной памяти;

– планирование и диспетчеризация задач в соответствии с заданным стратегией и дисциплинами обслуживания;

– организация механизмов обмена сообщениями и данными между выполняющимися, программами;

– защита одной программы от влияния другой; обеспечение сохранности данных;

– предоставление услуг на случай частичного, сбоя системы;

– обеспечение работы систем программирования, с помощью которых пользователи готовят свои программы.

2. Назначение системы управления файлами – организация более удобного доступа к данным, организованным как файлы. Именно благодаря системе управления файлами вместо низкоуровневого доступа к данным с указанием конкретных физических адресов нужной нам записи используется логический доступ с указанием имени файла и записи в нем. Как правило, все современные ОС имеют соответствующие системы управления файлами. Однако выделение этого вида системного программного обеспечения в отдельную категорию представляется целесообразным, поскольку ряд ОС позволяет работать с несколькими файловыми системами (либо с одной из нескольких, либо сразу с несколькими одновременно). В этом случае, говорят о монтируемых, файловых системах (дополнительную систему управления файлами можно установить), и в этом смысле они самостоятельны. Более того, можно назвать примеры простейших ОС, которые могут работать, и без файловых систем, а значит, им необязательно иметь систему управления файлами, либо они могут работать с одной из выбранных файловых систем. Надо, однако, понимать, что любая система управления файлами не существует сама по себе – она разработана для работы в конкретной ОС и с конкретной файловой системой. Можно сказать, что всем известная файловая система FAT (fife allocation table)1 имеет множество, реализаций как система управления файлами. Например, FAT-16 для самой MS-DOS, super-FAT для OS/2, FAT для Windows NT и т.д. Другими словами, для работы с файлами, организованными в соответствии с некоторой файловой системой, для каждой ОС должна быть разработана соответствующая система управления файлами; и эта система управления файлами будет работать только в, той ОС, для которой она и создана.

Для удобства взаимодействия с ОС могут использоваться дополнительные интерфейсные оболочки. Их основное назначение – либо расширить возможности по управлению ОС, либо изменить встроенные в систему возможности. В качестве классических примеров интерфейсных оболочек, и соответствующих операционных сред,

выполнения программ можно, назвать различные варианты графического интерфейса X Window в системах семейства UNIX (например, K Desktop Environment в Linux), PM Shell или Object Desktop в OS/2 с графическим интерфейсом Presentation Manager; наконец, можно указать разнообразные варианты интерфейсов дли семейства ОС Windows компании Microsoft, которые заменяют Explorer и могут напоминать либо UNIX с его графическим интерфейсом, либо OS/2, либо MAC OS. Следует отметить, что о семействе ОС кампании Microsoft с общим интерфейсом, реализуемым программными модулями с названием Explorer (в файле system.ini, который находится в каталоге Windows, имеется строка SHELL=EXPLORER.EXE), все же можно сказать, что заменяемой в этих системах является только интерфейсная оболочка, в то время как сама операционная среда остается неизменной; она интегрирована в ОС. Другими словами, операционная среда определяется программными интерфейсами, т.е. API (application program interface). Интерфейс прикладного программирования (API') включает в себя управление процессами, памятью и вводом/выводом.

Ряд операционных систем могут организовывать выполнение программ, созданных для других ОС. Например, в OS/2 можно выполнять как программы, созданные для самой OS/2, так и программы, предназначенные для выполнения в среде MS-DOS и Windows 3.x. Соответствующая операционная среда организуется в ОС в рамках отдельной виртуальной машины. Аналогично, в системе Linux можно создать условия для выполнения некоторых программ, написанных для Windows *95/98.* Определенными возможностями исполнения программ, созданных для иной операционной среды, обладает и Windows NT. Эта система позволяет выполнять некоторые программы, созданные для MS-DOS, OS/2 1.x, Windows 3.x. Правда, в своем последнем семействе ОС Window XP разработчики решили отказаться от поддержки возможности выполнения DOS-программ.

Наконец, к этому классу системного программного обеспечения следует отнести и эмуляторы, позволяющие смоделировать в одной операционной, системе, какую-либо другую) машину или операционную систему. Так, известна система эмуляции,WMWARE, которая позволяет запустить в среде Linux любую, другую ОС, например, Window. Можно, наоборот, создать эмулятор, работающий в среде Window, который позволит смоделировать компьютер, работающий под управлением любой ОС, в том числе и под Linux.

Таким образом, термин операционная среда означает соответствующий интерфейс, необходимый программам для обращения к ОС с целью получить определенный сервис – выполнить операцию ввода/вывода, получить или освободить участок памяти и т.д.

3. *Система программирования* на рис. 1 представлена прежде всего такими компонентами, как транслятор с соответствующего языка, библиотеки подпрограмм, редакторы, компоновщики и отладчики. Не бывает самостоятельных (оторванных от ОС) систем программирования. Любая система программирования может работать только в соответствующей ОС, под которую она и создана, однако при этом она может позволять разрабатывать программное обеспечение и под другие ОС. Например, одна из популярных систем программирования на языке C/C++ от фирмы Watcom для OS/2 позволяет получать программы и для самой OS/2, и для DOS, и для Windows.

В том случае, когда создаваемые программы должны работать совсем на другой аппаратной базе, говорят о кросс-системах. Так, для ПК на базе микропроцессоров семейства i80x86 имеется большое количество кросс-систем, позволяющих создавать программное обеспечение для различных микропроцессоров и микроконтроллеров.

4. Наконец, под *утилитами* понимают специальные системные программы, с помощью которых можно как обслуживать саму ОС, так и подготавливать для работы носители данных, выполнять перекодирование данных осуществлять оптимизацию размещение данных на носителе и производить некоторые другие работы, связанные с обслуживанием вычислительной системы. К утилитам следует отнести и программу

разбиения на магнитных дисках на разделы, и программу форматирования, программу переноса основных системных файлов самой ОС. Также к утилитам относятся и небезызвестные комплексы программ от фирмы Symantec, носящие имя Питера Нортона (создателя этой фирмы и соавтора популярного набора утилит для первых IBM PC). Естественно, что утилиты могут работать только в соответствующей операционной среде.

1 Основные понятия операционной среды

Операционная система выполняет функции управления вычислительными процессами в вычислительной системе, распределяет ресурсы вычислительной системы между различными вычислительными процессами и образует программную среду, в которой выполняются прикладные программы пользователей. Такая среда называется операционной.

Любая программа имеет дело с некоторыми исходными данными, которые она обрабатывает, и порождает в конечном итоге некоторые выходные данные, результаты вычислений. Очевидно, что в абсолютном большинстве случаев исходные данные попадают в оперативную память (с которой непосредственно работает процессор, выполняя вычисления по программе) с внешних (периферийных) устройств. Аналогично и результаты вычислений, в конце концов, должны быть выведены на внешние устройства. Следует заметить, что программирование операций ввода/вывода относится, пожалуй, к наиболее сложным и трудоемким задачам. Дело в том, что при создании таких программ без использования современных систем программирования, как говорится, «по старинке», нужно знать не только архитектуру процессора (его состав, назначение основных регистров, систему команд процессора, форматы данных и т.п.), но и архитектуру подсистемы ввода/вывода (соответствующие интерфейсы, протоколы обмена данными, алгоритм работы контроллера устройства ввода/вывода и т.д.). Именно поэтому развитие системного программирования и самого системного программного обеспечения пошло по пути выделения наиболее часто встречающихся операций и создания для них соответствующих программных модулей, которые можно в дальнейшем использовать в большинстве вновь создаваемых программ.

Например, в далекие пятидесятые годы, на заре развития вычислительных систем, при разработке первых систем программирования, прежде всего создавали программные модули для подсистемы ввода/вывода, а уже затем – вычисления часто встречающихся математических операций и функций. Благодаря этому при создании прикладных программ программисты могли просто обращаться к соответствующим функциям ввода/вывода и иным функциям и процедурам, что избавляло их от необходимости, каждый раз создавать, все программные компоненты «с нуля» и от необходимости знать во всех подробностях особенности работы контроллеров ввода/вывода и соответствующих интерфейсов.

Следующий шаг в автоматизации создания готовых к выполнению машинных двоичных программ заключался в том, что транслятор с алгоритмического языка более высокого уровня, нежели 1-е ассемблеры, уже сам мог подставить вместо высокоуровневого оператора типа READ или WRITE все необходимые вызовы к готовым библиотечным программным модулям. Состав и количества *библиотек* систем программирования постепенно увеличивались. В конечном итоге возникла ситуация, когда при создании двоичных машинных программ программисты могут вообще; не знать многих деталей управления конкретными ресурсами вычислительной системы, а должны только обращаться к некоторой программной подсистеме с соответствующими вызовами и получать от неё необходимые функции и сервисы. Эта программная подсистема и есть ОС, а набор её функций, сервисов и правила обращения к ним как раз и образуют то базовое понятие, которое мы называем операционной средой. Т. об., можно сказать, что термин операционная среда означает, прежде всего, соответствующие интерфейсы,

необходимые программам и пользователям *для* обращения к ОС с целью получить определенные сервисы.

Можно спросить: а чем отличаются *системные программные модули,* реализующие основные системные функции, от тех программных модулей, что пишутся прикладными программистами? Ответ простой: тем, что эти модули, как правило, используются всеми прикладными программами. Поэтому нет особого смысла на этапе создания машинной двоичной программы (которую и исполняет процессор) присоединять соответствующие системные программные модули к телу программы. Выгоднее просто обращаться к этим программным модулям, указывая их адреса и передавая им необходимые параметры, поскольку они уже и так находятся в, основной памяти, ибо нужны всем. Другими словами, эти основные системные программные модули вводят в состав самой ОС.

Параллельное существование терминов «операционная система» и «операционная среда» вызвано тем, что ОС в общем случае может поддерживать несколько операционных сред. Например, операционная система OS/2 Warp может выполнять следующие программы:

– так называемые *«нативные»* программы, созданные с учетом соответствующего «родного» 32-битового программного интерфейса этой ОС;

– 16-битовые программы созданные для систем OS/2 первого поколения;

–16-битовые приложения, разработанные для выполнения в операционной среде MS-DOS или PC DOS;

–16-битовые приложения, созданные для операционной среды Windows 3.x;

– саму операционную оболочку Windows 3.x и уже в ней – созданные для нее программы.

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

Понятия вычислительного процесса и ресурса

Понятие ***«вычислительный процесс»*** (или просто – «процесс») является одним из основных при рассмотрений ОС. Как понятие, процесс является определённым видом абстракции, и мы будем придерживаться следующего неформального определения. Последовательный процесс (иногда называемый ***«задачей»)*** – это выполнение отдельной программы с её данными на последовательном процессоре. В концепции, которая получила наибольшее распространение в 70-е годы, ***задача (task)*** – это совокупность связанных между собой, и образующих единое целое программных модулей и данных, требующая ресурсов вычислительной системы для своей реализации. В последующие годы задачей стали называть единицу работы, для выполнения которой предоставляется центральный процессор. Вычислительный процесс может включать в себя несколько задач. Концептуально процессор рассматривается в двух аспектах: во-первых, он является

носителем данных и, во-вторых он (одновременно) выполняет операции; связанные с их обработкой.

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

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

Ресурсы могут быть разделяемыми, когда несколько процессов могут их использовать одновременно (в один и тот же момент времени) или параллельно (в течение некоторого интервала времени процессы используют ресурс попеременно), а могут быть и неделимыми (рис: 1.1).

Рис. 1.1. Классификации ресурсов

При разработке первых систем ресурсами считать процессорное время, память, каналы ввода/вывода и периферийные устройства. Однако очень скоро понятие ресурса стало гораздо более универсальным и общим. Различного рода программные и информационные ресурсы также могут быть определены для системы как объекты, которые могут разделяться и распределяться, и доступ к которым необходимо соответствующим образом контролировать. В настоящее время понятие ресурса превратилось в абстрактную структуру с целым рядом атрибутов, характеризующих способы доступа к этой структуре и ее физическое представление в системе. Более того, помимо системных ресурсов, о которых мы сейчас говорили, как ресурс стали толковать и такие объекты, как сообщения и синхросигналы, которыми обмениваются задачи.

В первых вычислительных системах любая программа могла выполняться только после полного завершения предыдущей. Поскольку эти первые вычислительные системы были построены в соответствии с принципами, изложенными в известной работе Яноша Джон фон Неймана, все подсистемы и устройства компьютера управлялись исключительно центральным процессором. Центральный процессор осуществлял и выполнение вычислений, и управления операциями ввода/вывода данных. Соответственно, пока осуществлялся обмен данными между оперативной памятью и внешними устройствами, процессор не мог выполнять вычисления. Введение в состав вычислительной машины специальных контроллеров позволило совместить во времени (распараллелить) операции вывода полученных данных и последующие вычисления на центральном процессоре. Однако всё равно процессор продолжал часто и долго

простаивать, дожидаясь завершения очередной операции ввода/вывода. Поэтому было предложено организовать так называемый мультипрограммный (мультизадачный) режим работы вычислительной системы. Суть его заключается в том, что пока одна программа (один вычислительный процесс шли задача, как мы теперь говорим) ожидает завершений очередной операции ввода/вывода, другая программа (а точнее, другая задача) может быть поставлена на решение (рис. 1.2).

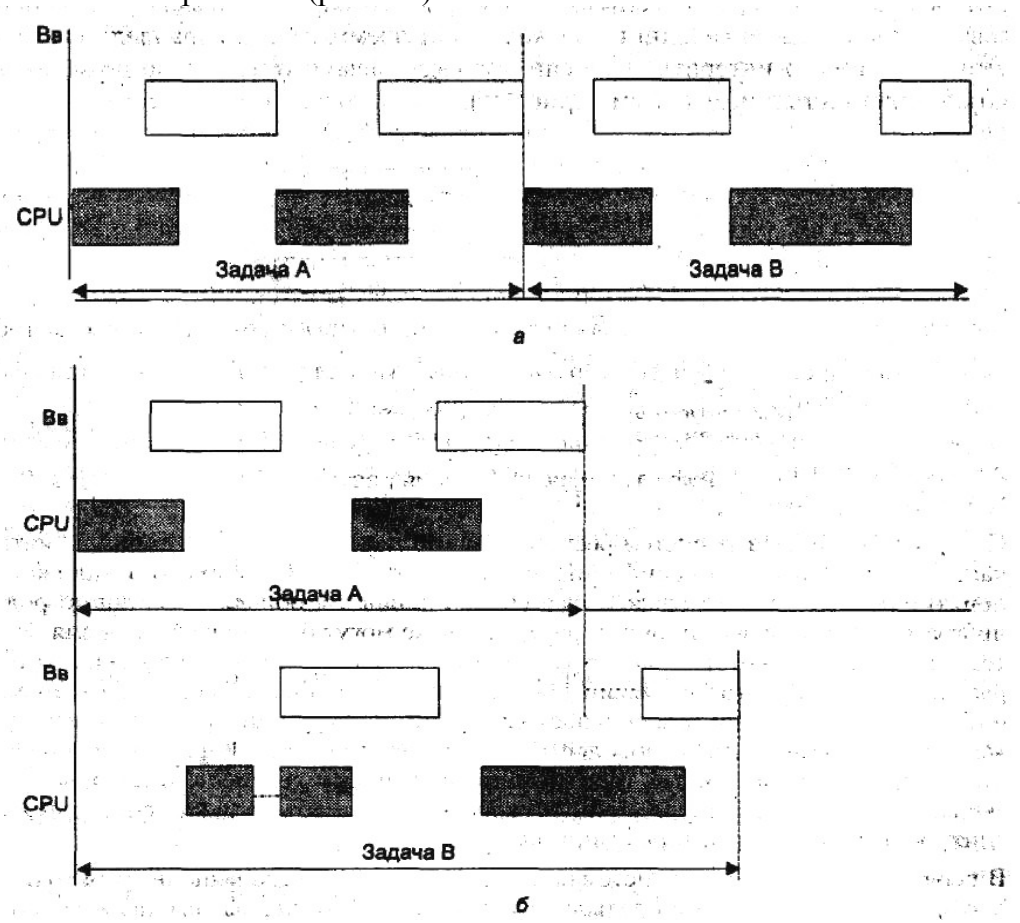

Рис. 1.2. Пример выполнения двух программ: а – однопрограммный режим; б – мультипрограммный режим

Из рис. 1.2, на котором качестве примера изображена такая гипотетическая ситуация, видно, что благодаря совмещению во времени выполнения двух программ общее время выполнения двух задач получается меньше, чем если бы мы выполняли их до очереди (запуск одной только после полного завершения другой). Из этого же рисунка видно, что во время выполнения каждой задачи в общем случае становится больше, чем, если бы мы выполняли каждую из них как единственную.

При мультипрограммировании повышается пропускная способность системы, но отдельный процесс никогда не может быть выполнен быстрее, «чем если бы он выполнялся в однопрограммном режиме (всякое разделение ресурсов замедляет работу одного из участников за счет дополнительных затрат времени на ожидание освобождения ресурса).

Как мы уже отмечали, операционная система поддерживает *мультипрограммирование* (*многопроцессность*) и старается эффективно использовать ресурсы путем организации к ним очередей запросов, составляемых тем или иным способом. Это требование достигается поддерживанием в памяти более одного процесса, ожидающего процессор, и более одного процесса, готового использовать другие ресурсы, как только последние станут доступными. Общая схема выделения ресурсов такова. При необходимости использовать какой–либо ресурс (оперативную память, устройство ввода/вывода, массив данных и т. п.) задача обращается к супервизору операционной системы – ее

центральному управляющему модулю, который может состоять из нескольких модулей, например: супервизор ввода/вывода, супервизор прерываний, супервизор программ, диспетчер задач и т. д.– посредством специальных вызовов (команд, директив) и сообщает о своём требовании. При этом указывается вид ресурса и, если надо, его объем (например, количество адресуемых ячеек оперативной памяти, количество дорожек или секторов на системном диске, устройство печати и объем выводимых данных и т. п.).

Директива обращения к операционной системе передает ей управление, переводя процессор в привилегированный режим работы, если такой существует. Не все вычислительные комплексы имеют два (и более) режима работы: привилегированный (режим супервизора), пользовательский, режим эмуляции какого-нибудь другого компьютера и т. д.

Ресурс может быть выделен задаче, обратившейся *к* супервизору с соответствующим запросом, если:

– он свободен и в системе нет запросов от задач более высокого приоритета к этому же ресурсу;

– текущий запрос и ранее выданные запросы допускают, совместное использование ресурсов;

– ресурс используется задачей низшего приоритета и может быть временно отобран (разделяемый ресурс).

Получив запрос, операционная система либо удовлетворяет его и возвращает управление задаче, выдавшей данный запрос, либо, если ресурс занят, ставит задачу в очередь к ресурсу, переводя ее в состояние ожидания (блокируя). Очередь, к ресурсу может быть организована несколькими способами, но чаще всего это осуществляется с помощью списковой структуры.

После окончания работы с ресурсом задача опять с помощью специального вызова супервизора (посредством соответствующей директивы) сообщает операционной системе об отказе от ресурса, или операционная система забирает ресурс сама, если управление возвращается супервизору после выполнения какой-либо системной функции. Супервизор операционной системы, получив управление по этому обращению, освобождает ресурс и проверяет, имеется ли очередь к освободившемуся ресурсу.

Если очередь есть в зависимости от принятой *дисциплины обслуживания* (правила обслуживания)[1] и приоритетов заявок он выводит из состояния ожидания задачу, ждущую ресурс, и переводит ее в состояние готовности к выполнению. После этого управление либо передается данной задаче, либо возвращается той, которая только что освободила ресурс.

При выдаче запроса на ресурс задача может указать, хочет ли она владеть ресурсом монопольно или допускает совместное использование с другими задачами. Например, с файлом можно работать монопольно, а можно и совместно с другими задачами.

Если в системе имеется некоторая совокупность ресурсов, то управлять их использованием можно на основе определенной стратегии. Стратегия подразумевает четкую формулировку целей, следуя которым можно добиться эффективного распределения ресурсов.

При организации управления ресурсами всегда требуется принять решение о том, что в данной ситуации выгоднее: быстро обслуживать отдельные наиболее важные запросы, предоставлять всем процессам равные возможности, либо обслуживать максимально возможное количество процессов и наиболее полно использовать ресурсы.

Диаграмма состояний процесса

Необходимо различать системные управляющие процессы, представляющие работу супервизора операционной системы и занимающиеся распределением и управлением ресурсов, от всех других процессов: системных обрабатывающих Процессов, которые не входят в ядро операционной системы, и процессов пользователя. Для системных

управляющих процессов в большинстве операционных систем ресурсы распределяются изначально и однозначно. Эти процессы управляют ресурсами системы, за использование которых существует конкуренция между всеми остальными процессами. Поэтому исполнение системных управляющих программ не принято называть процессами. Термин задача можно употреблять только по отношению к процессам пользователей и к системным обрабатывающим процессам. Однако это справедливо не для всех ОС. Например, в так называемых «микроядерных» ОС (в качестве примера можно привести ОС реального времени QNX фирмы Quantum Software systems) большинство управляющих программных модулей самой ОС и даже драйверы имеют статус высокоприоритетных процессов, для выполнения которых необходимо выделить соответствующие ресурсы. Аналогично и в UNIX-системах выполнение системных программных модулей тоже имеет статус системных процессов, которые получают ресурсы для своего исполнения.

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

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

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

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

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

В обычных ОС, как правило, процесс появляется при запуске какой-нибудь программы. ОС организует (порождает или выделяет) для нового процесса соответствующий дескриптор процесса, и процесс (задача) начинает развиваться (выполняться). Поэтому пассивного состояния не существует. В ОС реального времени (ОСРВ) ситуация иная. Обычно при проектировании системы реального времени уже заранее бывает известен состав программ (задач), которые должны будут выполняться. Известны и многие их параметры, которые необходимо учитывать при распределении ресурсов (например, объем памяти, приоритет, средняя длительность выполнения, открываемые файлы, используемые устройства и т. п.). Поэтому для них заранее заводят дескрипторы задач с тем, чтобы впоследствии не тратить драгоценное время на организацию дескриптора и поиск для него необходимых ресурсов. Таким образом, в ОСРВ многие процессы (задачи) могут находиться в состоянии бездействия, что мы и отобразили на рис. 1.3, отделив это состояние от остальных состояний пунктиром.

За время своего существования процесс может неоднократно совершать переходы из одного состояния в другое. Это обусловлено, обращениями *к* операционной системе с запросами ресурсов и выполнения системных функций, которые предоставляет операционная система, взаимодействием с другими процессами, появлением сигналов прерывания от таймера, каналов и устройств ввода/вывода, а также других устройств. Возможные переходы процесса из одного состояний в другое отображены в виде графа состояний на рис. 1.3. Рассмотрим эти переходы из одного состояния в другое более подробно.

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

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

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

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

– по прерыванию от внешнего инициативного (если по сигналу запроса на прерывание от него должна запускаться некоторая задача) устройства (сигнал о свершении некоторого события может запускать соответствующую задачу);

– при наступлении запланированного времени запуска программы.

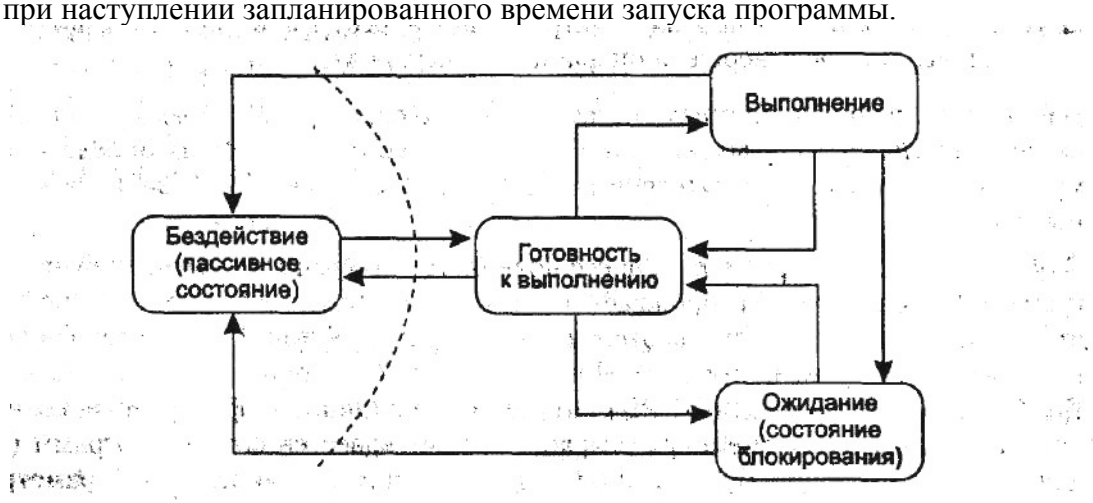

Рис. 1.3. Граф состояний процесса

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

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

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

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

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

– процесс блокируется (переводится в состояние ожидания) либо, вследствие запроса операции ввода/вывода (которая должна быть выполнена прежде, чем он сможет продолжить исполнение), либо в силу невозможности предоставить ему ресурс, запрошенный в настоящий момент (причиной перевода в состояние ожидания может быть

и отсутствие сегмента или страницы в случае организации механизмов виртуальной памяти), а также по команде оператора на приостановку задачи или по требованию через супервизор от другой задачи.

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

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

Реализация понятия последовательного процесса в ОС

Для того чтобы операционная система могла управлять процессами, она должна располагать всей необходимой для этого информацией. С этой целью на каждый процесс заводится специальная информационная структура, называемая дескриптором процесса (описателем задачи, блоком управления задачей). В общем случае дескриптор процесса содержит следующую информацию:

– идентификатор процесса (так называемый **PID – process identificator**);

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

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

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

–защищенную область памяти (или адрес такой зоны), в которой хранятся текущие значения регистров процессора, если процесс прерывается, не закончив работы. Эта информация называется ***контекстом задачи***.

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

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

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

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

Описатели задач, как правило, постоянно располагаются в оперативной памяти с целью ускорить работу супервизора, который организует их в списки (очереди) и отображает изменение, состояния процесса перемещением соответствующего описателя из одного списка в другой. Для каждого состояния (за исключением состояния выполнения для однопроцессорной системы) операционная система ведет соответствующий список задач, находящихся в этом состоянии. Однако для состояния ожидания может быть не один список, а столько, сколько различных видов ресурсов могут вызывать состояние ожидания. Например, состояний ожидания завершения операций ввода/вывода может быть столько, сколько устройств ввода/вывода имеется в системе.

В некоторых ОС количество описателей определяется жестко и заранее (на этапе генерации варианта операционной системы или в конфигурационном файле, который используется при загрузке ОС), в других - по мере необходимости система может выделять участки памяти под новые описатели. Например, в OS/2 максимально

возможное количество описателей задач определяется в конфигурационном файле CONFIG.SYS, а в Windows NT оно в явном виде не задаётся. Справедливости ради стоит заметить, что в упомянутом файле указывается количество не процессов, а именно задач, и под задачей в данном случае понимается как процесс, так и поток этого же процесса, называемый потоком или тредом.

Например, строка в файле CONFIG.SYS **THREADS=1024** указывает, что в системе может параллельно существовать и выполняться до 1024 задач, включая вычислительные процессы и их потоки.

В ОС реального времени чаще всего количество процессов фиксируется; и, следовательно, целесообразно заранее определять (на этапе генерации или конфигурирования ОС) количество дескрипторов. Для использования таких ОС в качестве систем общего назначения (что сейчас встречается редко, а в недалеком прошлом достаточно часто в качестве вычислительных систем общего назначения приобретали мини-ЭВМ и устанавливали на них ОС реального времени) обычно количество дескрипторов берётся с некоторым запасом, и появление новой задачи связывается с заполнением этой информационной структуры. Поскольку дескрипторы процессов постоянно располагаются в оперативной памяти (с целью ускорить работу диспетчера), то их количество не должно быть очень большим. При необходимости иметь большое количество задач один и тот же дескриптор может в разное время предоставляться для разных задач, но это сильно снижает скорость реагирования системы.

Для более эффективной обработки данных в системах реального времени целесообразно иметь постоянные задачи, полностью или частично всегда существующие в системе независимо от того, поступило на них требование или нет. Каждая постоянная задача обладает некоторой собственной областью оперативной памяти (ОЗУ-резидентные задачи) независимо от того, выполняется задача в данный момент или нет. Эта область, в частности, может использоваться для хранения данных; полученных задачей ранее. Данные могут храниться в ней и тогда, когда задача находится в состоянии ожидания или даже в состоянии бездействия.

Для аппаратной поддержки работы операционных систем с этими информационными структурами (дескрипторами задач) в процессорах могут быть реализованы соответствующие механизмы. Так, например, в микропроцессорах Intel i80x86, начиная с 80286, имеется специальный регистр TR (task register), указывающий местонахождение TSS (сегмента состояния задачи), в котором при переключении с задачи на задачу автоматически сохраняется содержимое регистров процессора. Как правило, в современных ОС для этих микропроцессоров дескриптор задачи включает в себя TSS. Другими словами, дескриптор задачи больше по размеру, чем TSS, и включает в себя такие традиционные поля, как идентификатор задачи, её имя, тип, приоритет и т. п.

Процессы и треды

Понятие процесса было введено для реализации идей мультипрограммирования. Напомним, в свое время различали термины «мультизадачность» и «мультипрограммирование». Таким образом, для реализации «мультизадачности» в ее исходном, толковании необходимо было тоже ввести соответствующую сущность. Такой сущностью и стали так называемые «легковесные» процессы, или, как их теперь преимущественно называют, – ***потоки*** или ***треды*** (нити). Рассмотрим эти понятия подробнее.

Когда говорят о ***процессах*** (process), то тем самым хотят отметить, что операционная система поддерживает их обособленность: у каждого процесса имеется свое виртуальное адресное пространство, каждому процессу назначаются свои ресурсы – файлы, окна, семафоры и т. д. Такая обособленность нужна для того, чтобы защитить один процесс от другого, поскольку они, совместно используя все ресурсы вычислительной системы, конкурируют друг с другом. В общем случае процессы просто никак не связаны между

собой и могут принадлежать даже разным пользователям, разделяющим одну вычислительную систему. Другими словами, в случае процессов ОС считает их совершенно не связными и независимыми. При этом именно ОС берет на себя роль арбитра в конкуренции между процессами по поводу ресурсов.

Однако желательно иметь еще и возможность задействовать внутренний параллелизм, который может быть в самих процессах. Такой внутренний параллелизм встречается достаточно часто и его использование позволяет ускорить их решение. Например, некоторые операции, выполняемые приложением, могут требовать для своего исполнения достаточно длительного использования центрального процессора. В этом случае при интерактивной работе с приложением пользователь вынужден долго ожидать завершения заказанной операции, и не может управлять приложением до тех пор, пока операция не выполнится до самого конца. Такие ситуации встречаются достаточно часто, например, при обработке больших изображений в графических редакторах. Если же программные модули, исполняющие такие длительные операции, оформлять в виде самостоятельных «подпроцессов» (легковесных или облегченных процессов - потоков, можно также воспользоваться термином *задача),* которые будут выполняться параллельно с другими «подпроцессами» (потоками, задачами), то у пользователя появляется возможность параллельно выполнять несколько операций в рамках одного приложения (процесса). Легковесными эти задачи называют потому, что операционная система не должна для них организовывать полноценную виртуальную машину. Эти задачи не имеют своих собственных ресурсов, они развиваются в том же виртуальном адресном пространстве, могут пользоваться теми же файлами, виртуальными устройствами и иными ресурсами, что и данный процесс. Единственное, что им необходимо иметь - это процессорный ресурс. В однопроцессорной системе треды (задачи) разделяют между собой процессорное время так же, как это делают обычные процессы, а в мультипроцессорной системе могут выполняться одновременно, если не встречают конкуренции из-за обращения к иным ресурсам.

Главное, что обеспечивает многопоточность – это возможность параллельно выполнять несколько видов операций в одной прикладной программе. Параллельные вычисления (а, следовательно, и более эффективное использование ресурсов центрального процессора, и меньшее суммарное время выполнения задач) теперь уже часто реализуется на уровне тредов, и программа, оформленная в виде нескольких тредов в рамках одного процесса, может быть выполнена быстрее за счет параллельного выполнения ее отдельных частей. Например, если электронная таблица или текстовый процессор были разработаны с учетом возможностей многопоточной обработки, то пользователь может запросить пересчет своего рабочею листа или слияние нескольких документов и одновременно продолжать заполнять таблицу или открывать для редактирования следующий документ.

Особенно эффективно можно использовать многопоточность для выполнения распределенных приложений; например, многопоточный сервер может параллельно выполнять запросы сразу нескольких клиентов. Как известно, операционная система OS/2 одной из первых среди ОС, используемых на ПК, ввела многопоточность. В середине девяностых годов для этой ОС было создано очень большое количество приложений, в которых использование механизмов многопоточной обработки реально приводило к существенно большей скорости выполнения вычислений.

Итак, сущность «поток» была введена для того, чтобы именно с помощью этих единиц распределять процессорное время между возможными работами. Сущность «процесс» предполагает, что при диспетчеризации нужно учитывать все ресурсы, закрепленные за ним. А при манипулировании тредами можно менять только контекст задачи, если мы переключаемся с одной задачи на другую в рамках одного процесса. Все остальные вычислительные ресурсы при этом не затрагиваются. Каждый процесс всегда состоит по

крайней мере из одного потока и только если имеется внутренний параллелизм, программист может «расщепить» один тред на несколько параллельных.

Потребность в потоках (threads) возникла еще на однопроцессорных вычислительных системах, поскольку они позволяют организовать вычисления более эффективно. Для использования достоинств многопроцессорных систем с общей памятью треды уже просто необходимы, так как позволяют не только реально ускорить выполнение тех задач, которые допускают их естественное распараллеливание, но и загрузить процессорные элементы работой, чтобы они не простаивали. Заметим, однако, что желательно, чтобы можно было уменьшить взаимодействие тредов между собой, ибо ускорение от одновременного выполнения параллельных потоков может быть сведено к минимуму из-за задержек синхронизации и обмена данными.

Каждый тред выполняется строго последовательно и имеет свой собственный ~~п~~програ~~п~~ммный счетчик и стек. Треды, как и процессы, могут порождать треды - потомки, поскольку любой процесс состоит по крайней мере из одного треда. Подобно традиционным процессам (то есть процессам, состоящим из одного треда), каждый тред может находится в одном из активных состояний. Пока один тред заблокирован (или просто находится в очереди готовых к исполнению задач), тред того же процесса может выполняться. Треды разделяют процессорное время так же, как это делают обычные процессы, в соответствии с различными вариантами диспетчеризации.

Как мы уже знаем, все трёды имеют одно и то же виртуальное адресное пространство своего процесса. Это означает, что они разделяют одни и те же глобальные переменные. Поскольку каждый тред может иметь доступ к каждому виртуальному адресу, один тред может использовать стек другого треда. Между потоками нет полной защиты, так как это, во-первых; невозможно, а во-вторых, не нужно. Все потоки одного процесса всегда решают общую задачу одного пользователя, и механизм стоков используется здесь для белее быстрого решения путем ее распараллеливания. При этом программисту очень важно получить в свое распоряжение удобные средства организации взаимодействия частей программы. Повторим, что кроме разделения адресного пространства, все треды разделяют так же набор открытых файлов, используют общие устройства, выделенные процессу, имеют одни и те же наборы сигналов, семафоры и т. п. А что у тредов будет их собственным? Собственными являются программный стек, рабочие регистры процессора, потоки-потомки, состояние.

Вследствие того, что треды, относящиеся к одному процессу, выполняются в одном и том же виртуальном адресном пространстве, между ними легко организовать тесное взаимодействие, в отличие от процессов, для которых нужны специальные механизмы обмена сообщениями и данными. Более того, программист, создающий многопоточное приложение; может заранее продумать работу множества тредов процесса таким образом, чтобы они могли взаимодействовать наиболее выгодным способом, а не участвовать в конкуренции за предоставление ресурсов тогда, когда этого можно избежать.

Для того чтобы можно было эффективно организовать параллельное выполнение рассмотренных сущностей (процессов и тредов), в архитектуру современных процессоров включена возможность работать со специальной информационной структурой, описывающей ту или иную сущность. Для этого уже на уровне архитектуры микропроцессора используется понятие «задача» (task). Оно как бы объединяет в себе обычный и «легковесный» процессы. Это понятие и поддерживаемая для него на уровне аппаратуры информационная структура позволяют в дальнейшем при разработке операционной системы построить соответствующие дескрипторы, как для процесса, так и для треда. Отличаться эти дескрипторы будут прежде всего тем, что дескриптор треда может хранить только контекст приостановленного вычислительного процесса, тогда как дескриптор процесса (process) должен уже содержать поля, описывающие тем или иным способом ресурсы, выделенные этому процессу. Другими словами, тот же task state segment (сегмент состояния задачи), используется, как основа для дескриптора процесса.

Каждый тред (в случае использования так называемой «плоской» модели памяти – может быть оформлен в виде самостоятельного сегмента, что приводит к тому, что простая (не многопоточная) программа будет иметь всего один сегмент кода в виртуальном адресном пространстве.

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

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

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

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

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

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

6. Всякий раз, когда, какой-либо из ваших потоков пытается воспользоваться общим ресурсом вычислительного процесса, которому он принадлежит, вы обязаны тем или иным образом легализовать и защитить свою деятельность. Хорошим средством для этого являются критические секции, семафоры и очереди сообщений. Если вы протестировали свое приложение и не обнаружили ошибок синхронизации, то это еще не значит, что их там нет. Пользователь может создать самые непредсказуемые ситуации. Это очень ответственный момент в разработке многопоточных приложений.

7. Не возлагайте на поток несколько функций. Сложные функциональные отношения затрудняют понимание общей структуры приложения, его алгоритм. Чем проще и менее многозначна каждая из рассматриваемых ситуаций, тем больше вероятность, что ошибок удаётся избежать.

Прерывания

Прерывания представляют собой механизм, позволяющий координировать параллельное функционирование отдельных устройств вычислительной системы и реагировать на особые состояния, возникающие, при работе процессора. Таким образом, прерывание - это принудительная передача управления от выполняемой программы к системе (а через нее - к соответствующей программе обработки прерывания), происходящая при возникновении определенного события.

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

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

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

1. Установление факта прерывания (прием сигнала на прерывание) и идентификация прерывания (в операционных системах иногда осуществляется повторно, на шаге 4).
2. Запоминание состояния прерванного процесса. Состояние процесса определяется прежде всего значением счетчика команд (адресом следующей команды, который, например, в i80x86 определяется регистрами CS и IP – указателем команды), содержимым регистров процессора и может включать также спецификацию режима (например, режим пользовательский или привилегированный) и другую информацию.
3. Управление аппаратно передается подпрограмме обработки прерывания. В простейшем случае в счетчик команд заносится начальный адрес подпрограммы обработка прерываний, а в соответствующие регистры – информация из слова состояния. В более развитых процессорах, например в том же i80x86 и последующих 32-битовых микропроцессорах, начиная с i80x86, осуществляется достаточно сложная процедура определения начального адреса соответствующей подпрограммы обработки прерывания и не менее сложная процедура инициализации рабочих регистров процессора.
4. Сохранение информации о прерванной программе, которую не удалось спасти на шаге 2 с помощью действий аппаратуры. В некоторых вычислительных системах предусматривается запоминание довольно большого объема информации о состоянии прерванного процесса.
5. Обработка прерывания. Эта работа может быть выполнена той же подпрограммой, которой было передано управление на шаге 3, но в ОС чаще всего она реализуется путем последующего вызова соответствующей подпрограммы.
6. Восстановление информации, относящейся к прерванному процессу (этап, обратный шагу 4).
7. Возврат в прерванную программу.

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

На рис. 1.4 показано, что при возникновении запроса на прерывание естественный ход вычислений нарушается и управление передается программе обработки возникшего прерывания. При этом средствами аппаратуры сохраняется (как правило, с помощью механизмов стековой памяти) адрес той команды, с которой следует продолжить выполнение прерванной программы. После выполнения обработки прерывания управление возвращается прерванной ранее программе посредством занесения в указатель команд сохраненного адреса команды. Однако такая схема используется только в самых простых программных средах. В мультипрограммных операционных системах обработка

прерываний происходит по более сложным схемам, о чем будет более подробно написано ниже.

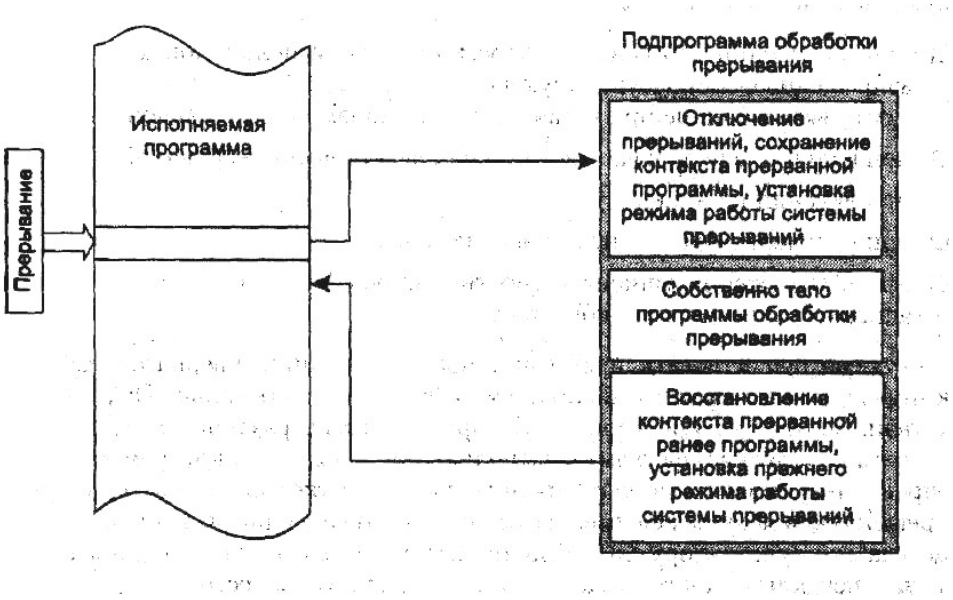

Рис. 1.4. Обработка прерывания

Итак, главные функции механизма прерываний:

– распознавание или классификация прерываний;

– передача управления соответственно обработчику прерываний;

– корректное возвращение к прерванной программе.

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

Прерывания, возникающие при работе вычислительной системы, можно разделить на два основных класса: *внешние* (их иногда называют асинхронными) и *внутренние* (синхронные).

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

– прерывания от таймера;

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

– прерывания по нарушению питания;

– прерывания с пульта оператора вычислительной системы;

– прерывания от другого процессора или другой вычислительной системы.

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

– при нарушении адресации (в адресной части выполняемой команды указан запрещенный или несуществующий адрес, обращение к отсутствующему сегменту или странице при организации механизмов виртуальной памяти);

– при наличии в поле кода операции незадействованной двоичной комбинации;

– при делении на нуль;

– при переполнении или исчезновении порядка;

– при обнаружении ошибок четности, ошибок в работе различных устройств аппаратуры средствами контроля.

Могут еще существовать прерывания при обращении к супервизору ОС - в некоторых компьютерах часть команд может использовать только ОС, а не пользователи.

Соответственно в аппаратуре предусмотрены различные режимы работы, и пользовательские программы выполняются в режиме, в котором эти привилегированные команды не исполняются. При попытке использовать команду, запрещенного данном режиме, происходит внутреннее прерывание и управление передается супервизору ОС. К привилегированным командам относятся и команды переключения режима работа центрального процессора.

Наконец, существуют собственна *программные прерывания.* Эти прерывания происходят по соответствующей команде прерывания, то есть по этой команде процессор осуществляет практически те же действия, что и при обычных внутренних прерываниях. Данный механизм был специально введен для того, чтобы переключение на системные программные модули происходило не просто как переход в подпрограмму, а точно таким же образом, как и обычное прерываний. Этим обеспечивается автоматическое переключение процессора в привилегированным режим с возможностью исполнения любых команд.

Сигналы, вызывающие прерывания, формируются вне процессора или самом процессоре; они могут возникать одновременно. Выбор одного из них для обработки осуществляется на основе приоритетов, приписанных каждому типу прерывания. Очевидно, что прерывания от схем контроля процессор должны обладать наивысшим приоритетом (если аппаратура работает неправильно, то не имеет смысла продолжать обработку информации). На рис. 1.5 изображен обычный порядок (приоритеты) обработки прерываний в зависимости от типа прерываний. Учет приоритета может быть встроен в технические средства, а также определяться ОС, т. е. кроме аппаратно реализованных приоритетов прерывания большинство вычислительных машин и комплексов допускают программно-аппаратное управление порядком обработки сигналов прерывания. Второй способ, дополняя первый, позволяет применять различные дисциплины обслуживания прерываний.

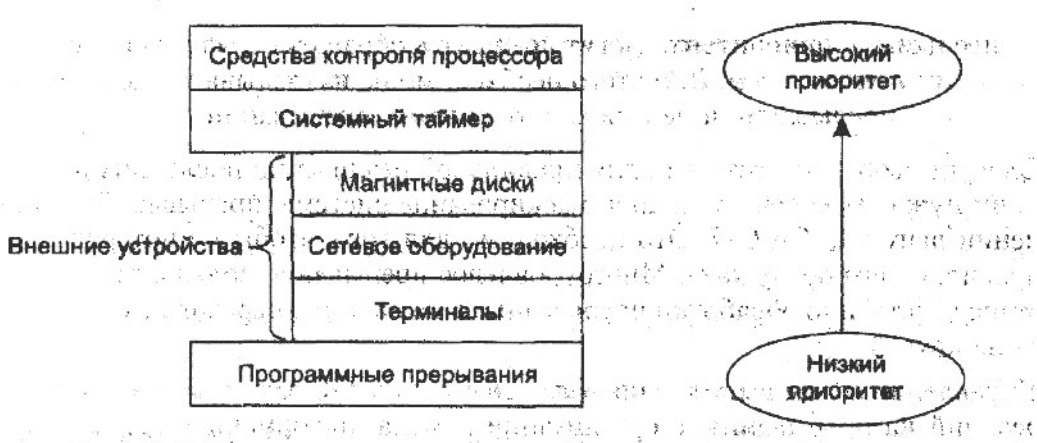

Рис. 1.5. Распределение прерываний по уровням приоритета

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

отмечалось) присваивается приоритеты. Сигнал с более высоким приоритетом обрабатывается в первую очередь, обработка остальных сигналов прерывания откладывается.

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

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

– *с абсолютными приоритетами,* то есть всегда обслуживается прерывание с наивысшим приоритетом. Для реализации этого режима необходимо на время обработки прерывания замаскировать все запросы с более, низким приоритетом. При этом возможно многоуровневое прерывание, то есть прерывание программ обработки прерываний. Число уровней прерывания в этом режиме изменяется и зависит от приоритета запроса.

– по *принципу стека,* ила, как иногда говорят, *по дисциплине LCFS* (last come first served – последним пришёл – первым обслужен), то есть запросы с более низким приоритетом могут прерывать обработку прерывания с более высоким приоритетом. Для этого необходимо не накладывать маски ни на один сигнал прерывания и не выключать систему прерываний.

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

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

Как мы уже знаем, при появлении запроса на прерывание система прерываний идентифицирует сигнал и, если прерывания разрешены, управление передается на соответствующую подпрограмму обработки. Из рис. 1.4 видно, что в подпрограмме обработки прерывания имеются две служебные секции. Это – первая секция, в которой осуществляется сохранение контекста прерванной задачи, который не смог быть сохранен на 2-м шаге, и последняя, заключительная секция, в которой, наоборот, осуществляется восстановления контекста. Для того чтобы система прерываний не среагировала повторно на сигнал запроса на прерывание, она обычно автоматически закрывает (отключает) прерывания, поэтому необходимо потом в подпрограмме обработки прерываний вновь включать систему прерываний. Установка рассмотренных режимов обработки прерываний (с относительными и абсолютными приоритетами, и по правилу LCFS) осуществляется в конце первой секции подпрограммы обработки. Таким образом, на время выполнения центральной секции (в случае работы в режимах с абсолютными приоритетами и по дисциплине LCFS) прерывания разрешены. На время работы заключительной секции подпрограммы обработки система прерываний должна быть

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

Супервизор прерываний, прежде всего, сохраняет в дескрипторе текущей задачи рабочие регистры процессора, определяющие контекст прерываемого вычислительного процесса. Далее он определяет ту подпрограмму, которая должна выполнить действия, связанные с обслуживанием настоящего (текущего) запроса на прерывание. Наконец, перед тем как передать управление этой подпрограмме, супервизор прерываний устанавливает необходимый режим обработки прерывания. После выполнения подпрограммы, обработки прерывания управление передается супервизору, на этот раз уже на тот модуль, который занимается диспетчеризацией задач. И уже диспетчер, задач, в свою очередь в соответствии с принятым режимом распределения процессорного времени (между выполняющимися процессами) восстановит контекст той задачи, которой будет решено выделить процессор. Рассмотренная нами схема проиллюстрирована на рис. 1.6.

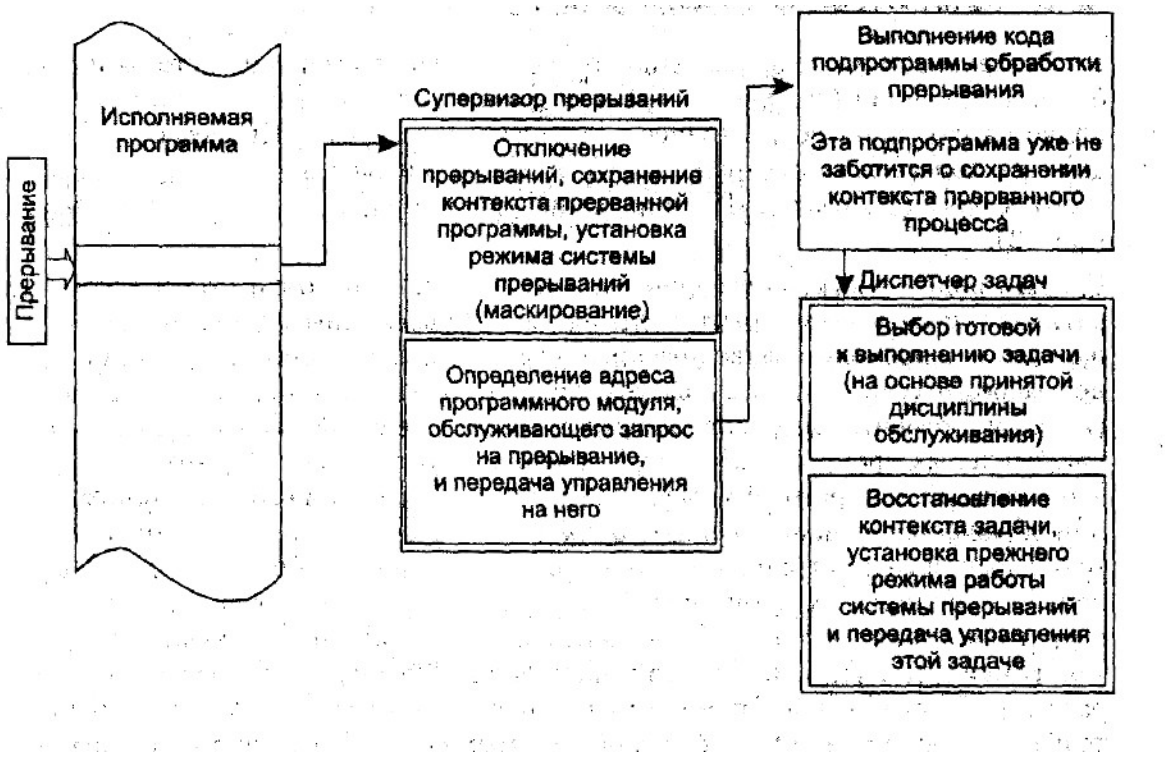

Рис. 1.6. Обработка прерывания при участии супервизоров ОС

Как мы видим из рис. 1.6, здесь нет непосредственного возврата в прерванную ранее программу прямо из самой подпрограммы обработки прерывания. Для прямого непосредственного возврата достаточно адрес возврата сохранить в стеке, что и делает аппаратура процессора. При этом стек легко обеспечивает возможность возврата в случае вложенных прерываний, поскольку он всегда реализует дисциплину LCFS (last come – first served).

Однако если бы контекст процессов сохранялся просто в стеке, как это обычно реализуется аппаратурой, а не в описанных выше дескрипторах задач, то у нас не было бы возможности гибко подходить к выбору той задачи, которой нужно передать, процессор после завершения работы подпрограммы обработки прерывания. Естественно, что это только общий принцип. В конкретных процессорах и в конкретных ОС могут существовать некоторые отступления от рассмотренной системы и/или дополнения к ней. Например, в современных процессорах часто имеются специальные аппаратные возможности для сохранения контекста прерываемого процесса непосредственно в его

дескрипторе, то есть дескриптор процесса: (по крайней мере, его часть) становится структурой данных, которую поддерживает аппаратура.

Для полного понимания принципов создания и механизмов реализации, рассматриваемых далее современных ОС необходимо знать архитектуру персональных компьютеров и, в частности, особенности системы прерывания.

Основные виды ресурсов

Рассмотрим кратко основные виды ресурсов вычислительной системы и способы их разделения (см. рис. 1.1). Прежде всего, одним из важнейших ресурсов является сам процессор, точнее – процессорное время. Процессорное время делится попеременно (параллельно). Имеется множество методов разделения этого ресурса.

Вторым видом ресурсов вычислительной системы можно считать память. Оперативная память может быть разделена и одновременным, способом (то есть в памяти одновременно может располагаться несколько процессов или, по крайней мере, текущие фрагменты, участвующее в вычислениях), и попеременно, (в разные моменты оперативная, память может предоставляться для разных вычислительных процессов). Память – очень интересный вид ресурса. Дело в том, что в каждый конкретный момент времени процессор при выполнении вычислений обращается к очень ограниченному числу ячеек оперативной памяти. С этой точки зрения желательно память разделять для возможно большего числа параллельно исполняемых процессов. С другой стороны, как правило, чем больше оперативной памяти может быть выделено для конкретного текущего процесса, тем лучше будут условия для его выполнения. Поэтому проблема эффективного разделения оперативной памяти между параллельно выполняемыми вычислительными процессами является одной из самых актуальных.

Когда говорят о внешней памяти (например, память на магнитных дисках), то собственно память и доступ к ней считаются разными видами ресурса. Собственно внешняя память может разделяться одновременно, а доступ к ней - попеременно.

Если говорить о внешних устройствах, то они, как правило, могут разделяться параллельно, если используются механизмы прямого доступа. Если же устройство работает с последовательным доступом, то оно не может считаться разделяемым ресурсом. Простыми и наглядными примерами внешних устройств, которые не могут быть разделяемыми, являются принтер и накопитель на машинной ленте. Действительно, если допустить, что принтер можно разделять между двумя процессами, которые смогут его использовать попеременно, то результаты печати, скорее всего, не смогут быть использованы – фрагменты выведенного текста могут перемешаться таким образом, что в них невозможно будет разобраться. Аналогично обстоит дело и с накопителем на магнитной ленте. Если один процесс начнет что-то читать или писать, а второй при этом запросит перемотку ленты на, ее начало, то оба вычислительных процесса не смогут выполнить свои вычисления.

Очень важным видом ресурсов являются программные модули. Прежде всего, мы будем рассматривать системные программные модули, поскольку именно они обычно и рассматриваются как программные ресурсы, и, в принципе, могут быть распределены между выполняющимися процессами.

Как известно, программные модули могут быть однократно и многократно (или повторно) используемыми. Однократно используемыми называют такие программные модули, которые могут быть правильно выполнены только один раз. Это означает, что в процессе своего выполнения они могут испортить себя: либо повреждается часть кода; либо – исходные данные, от которых зависит ход вычислений. Очевидно, что однократно используемые программные модули являются неделимым ресурсом. Более того, их обычно вообще не распределяют как ресурс системы. Системные однократно используемые программные модули, как правило, используются только на этапе загрузки ОС. При этом следует иметь в виду тот очевидный, факт, что собственно двоичные

файлы, которые обычно хранятся на системном диске, и в которых и записаны эти модули, не портятся, а потому могут быть повторно использованы при следующем запуске ОС.

Повторно используемые программные модули, в свою очередь, могут быть не-привилегированными, привилегированными и реентерабельными.

Привилегированные программные модули работают в так называемом привилегированном режиме, то есть при отключенной системе прерываний (часто говорят, что прерывания закрыты), так, что никакие внешние события не могут нарушить естественный порядок вычислений. В результате программный модуль выполняется до своего конца, после чего он может быть вновь вызван на исполнение из другой задачи (другого вычислительного процесса). С позиции стороннего наблюдателя по отношению к вычислительным процессам, которые попеременно (причём, возможно, неоднократно) в течении срока своей жизни вызывают некоторый привилегированный программный модуль, такой модуль будет выступать как попеременно разделяемый ресурс. Структура привилегированных программных модулей изображена на рис.1.7. Здесь в первой секций программного модуля выключается система прерываний. В последней секции, напротив, включена система прерываний.

Непривилегированные программное модули - это обычные программные модули, которые могут быть прерваны во время своей работы. Следовательно, в общем случае их нельзя считать разделяемыми, потому что если после прерывания выполнения такого модуля, исполняемого в рамках одного вычислительного процесса, запустить его ещё раз по требованию другого вычислительного процесса, то промежуточные результаты для прерванных вычислений могут быть потеряны. В противоположность этому, реентерабельные программные модули (reenterable – допускающий повторные обращения) допускают повторное многократное прерывание своего исполнения и повторный их запуск по обращению из других задач (вычислительных процессов). Для этого реентерабельные программные модули должны быть созданы таким образом, чтобы было обеспечено сохранение промежуточных вычислений для прерываемых вычислений и возврат к ним, когда вычислительный процесс возобновляется с прерванной ранее точки. Это может быть реализовано двумя способами: с помощью статических и динамических методов выделения памяти под сохраняемые значения. Основной, наиболее часто используемый динамический способ выделения памяти для сохранения всех промежуточных результатов вычисления, относящихся к реентерабельному программному модулю, может быть проиллюстрирован с помощью рис. 1.8.

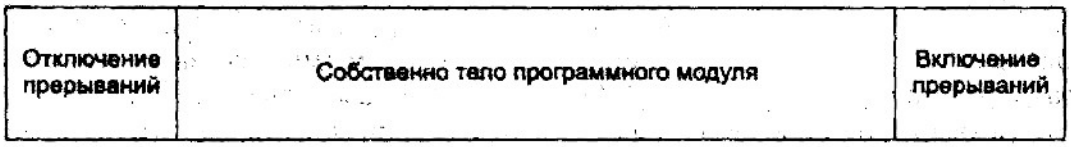

Рис. 1.7. Структура привилегированного программного модуля

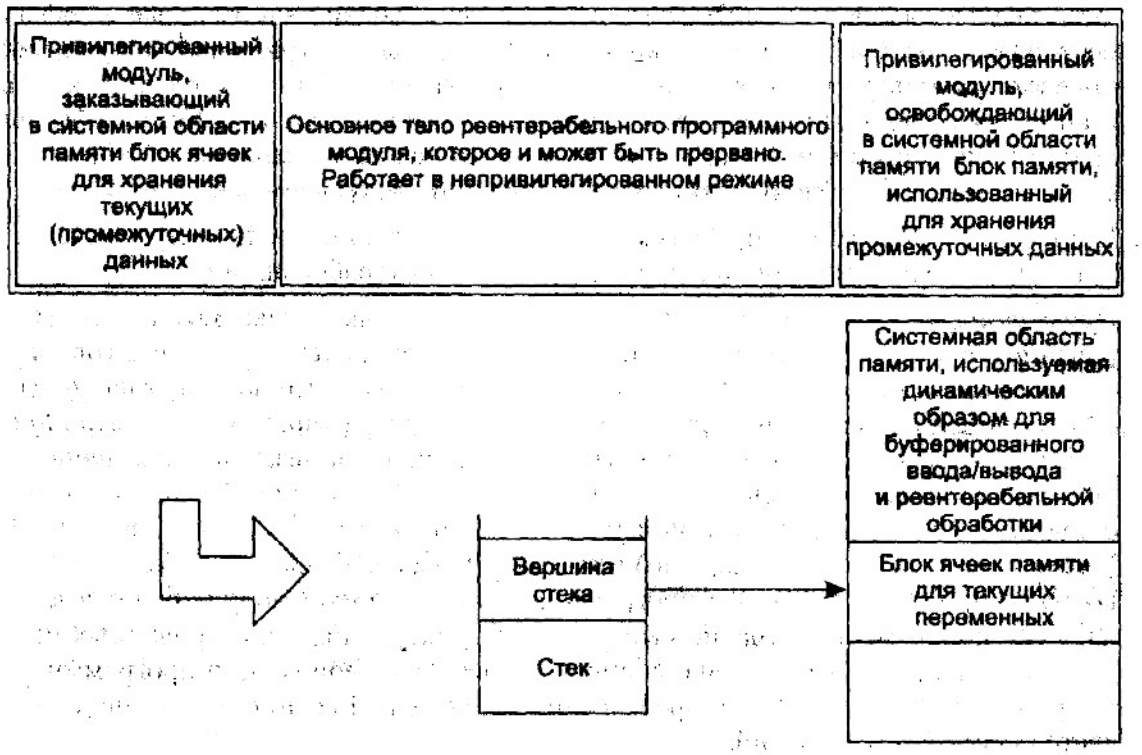

Рис. 1.8. Реентерабельный программный модуль

Основная идея построения и работы реентерабельного программного модуля, структура которого представлена на рис. 1.8, заключается в том, что в первой (головной) своей части с помощью обращения из системной привилегированной секции осуществляется запрос на получение в системной области памяти блока ячеек, необходимого для размещения всех текущих (промежуточных) данных. При этом на вершину стека помещается указатель на начало области данных и ее объем. Все текущие переменные реентерабельного программного модуля в этом случае располагаются в системной области памяти. Поскольку в конце привилегированной секции система прерываний включается, то во время работы центральной (основной) части реентерабельного модуля возможно ее прерывание. Если прерывание не возникает, то в третьей (заключительной) секции осуществляется запрос на освобождение использованного блока системной области памяти. Если же во время работы центральной секции возникает прерывание и другой вычислительный процесс обращается к тому же самому реентерабельному программному модулю, то для этого нового процесса вновь заказывается новый блок памяти в системной области памяти, и на вершину стека записывается новый указатель. Очевидно, что возможно многократное повторное вхождение в реентерабельный программный модуль до тех пор, пока в области системной памяти, выделяемой специально для реентерабельной обработки, есть свободные ячейки, число которых достаточно для выделения нового блока.

Что касается статического способа выделения памяти, то здесь речь может идти, например, о том, что заранее для фиксированного числа вычислительных процессов резервируются области памяти, в которых будут располагаться переменные реентерабельных программных модулей для каждого процесса – своя область памяти. Чаще всего в качестве таких процессов выступают процессы ввода-вывода и речь идет о

реентерабельных драйверах (реентерабельный драйвер может управлять параллельно несколькими однотипными устройствами).

Кроме реентерабельных программных модулей существуют еще повторно входимые (от re-entrance). Этим термином называют программные модули, которые ноже допускают свое многократное параллельное использование, но, в отличие от реентерабельных их нельзя прерывать. Повторно входимые программные модули состоят из привилегированных секций и повторное обращение к ним возможно только после завершения какой-нибудь из таких секций. После выполнения очередной привилегированной секции управление может быть передано супервизору, и если он предоставит возможность выполняться другому процессу, то возможно повторное вхождение в рассматриваемый программный модуль. Другими словами, в повторно входимых программных модулях четко предопределены все допустимые (возможные) точки входа. Следует отметить, что повторно входимые программные модули встречаются гораздо чаще реентерабельных (повторно прерываемых).

Наконец, имеются и информационные ресурсы, то есть в качестве ресурсов могут выступать данные. Информационные ресурсы могут существовать как в виде переменных, находящихся в оперативной памяти, так и в виде файлов. Если процессы используют данные только для чтения, то такие информационные ресурсы можно разделить. Если же процессы могут изменять информационные ресурсы, то необходимо специальным образом организовать работу с такими данными. Это одна из наиболее сложных проблем.

Классификация операционных систем

Широко известно высказывание, согласно которому любая наука начинается с классификации. Само собой, что вариантов классификации может быть очень много, все будет зависеть от выбранного признака, по которому один объект мы будем отличать от другого. Однако, что касается ОС, здесь уже давно сформулировалось относительно небольшое количество классификаций: по назначению, по режиму обработки задач, по способу взаимодействия с системой и, наконец, по способам построения (архитектурным особенностям систем).

Во введении мы уже, дали «определение» операционной системы (ОС). Поэтому просто повторим, что основным предназначением ОС является организация эффективных и надежных вычислений, создание различных интерфейсов для взаимодействия с этими вычислениями и с самой вычислительной системой.

Прежде всего, различают ОС общего и специального назначения. ОС специального назначения, в свою очередь, подразделяются на следующие: для переносимых микрокомпьютеров и различных встроенных систем, организации и ведения баз данных, решения задач реального времени и т. п. По режиму обработки задач различают ОС, обеспечивающие однопрограммный и мультипрограммный режимы. Под мультипрограммированием понимается способ организации вычислений, когда на однопроцессорной вычислительной системе создается видимость одновременного выполнения нескольких программ. Любая задержка в решении программы (например, для осуществления операций ввода/вывода данных) используется для выполнения других (таких же, либо менее важных) программ. Иногда при этом говорят о мультизадачном режиме. При этом, вообще говоря; мультипрограммный и мультизадачный режимы - это не синонимы, хотя и близкие понятия. Основное принципиальное отличие в этих терминах заключается в том, что мультипрограммный режим обеспечивает параллельное выполнение нескольких приложений и при этом программисты, создающие эти программы не должны заботиться о механизмах организации их параллельной работы. Эти функций берет на себя сама ОС, именно она распределяет между выполняющимися приложениями ресурсы вычислительной системы, осуществляет необходимую синхронизацию вычислений и взаимодействие. Мультизадачный режим, наоборот, предполагает, что забота о параллельном выполнении и взаимодействии приложений

ложится как раз на прикладных программистов. В современной технической и, тем более, научно-популярной литературе об этом различии часто забывают, тем самым внося некоторую путаницу. Можно, однако, заметить, что современные ОС для ПК реализуют и мультипрограммный, и мультизадачный режимы.

При организации работ с вычислительной системой в диалоговом режиме можно говорить об однопользовательских (однотерминальных) и мультитерминальных ОС. В мультитерминальных ОС с одной вычислительной системой одновременно могут работать несколько пользователей, каждый со своего терминала. При этом у пользователей возникает иллюзия, что у каждого из них имеется своя собственная вычислительная система. Очевидно, что для организации мультитерминального доступа к вычислительной системе необходимо мультипрограммный режим работы. В качестве одного из примеров мультитерминальных ОС для ПК можно назвать Linux.

Основной особенностью операционных систем реального времени (ОСРВ) является обеспечение обработки поступающих заданий в течение заданных интервалов времени, которые нельзя превышать. Поток заданий в общем случае не является планомерным и не может регулироваться оператором (характер следования событий можно предсказать лишь в редких случаях), то есть задания поступают непредсказуемые моменты времени и без всякой очередности. В ОС, не предназначенных для решения задач реального времени, имеются некоторые накладные расходы процессорного времени на этапе инициирования (при выполнении которого ОС распознает все пожелания пользователей относительно решения своей задачи, загружает в оперативную память нужную программу и выделяет другие необходимые для ее выполнения ресурсы). В ОСРВ подобные затраты могут отсутствовать, так как набор задач обычно фиксирован и вся информация о задачах известна еще до поступления запросов. Для продленной реализации режима реального времени необходима (хотя этого и недостаточно) организация мультипрограммирования. Мультипрограммирование является основным средством повышения производительности вычислительной системы, а для решения задач реального времени производительность становится важнейшим фактором. Лучшие характеристики по производительности для систем реального времени обеспечиваются однотерминальными ОСРВ. Средства организации мультитерминального режима всегда замедляют работу системы в целом, но расширяют функциональные возможности системы. Одной из наиболее известных ОСРВ для ПК является ОС QNX.

По основному архитектурному принципу ОС разделяются на микроядерные и монолитные. В некоторой степени это разделение тоже условно, однако можно в качестве яркого примера микроядерной ОС привести ОСРВ QNX, тогда как в качестве монолитной можно назвать Windows 95/98 или ОС Linux. Ядро ОС Windows мы не можем изменить, нам не доступны его исходные коды и у нас нет программы для сборки (компиляции) этого ядра. А вот в случае с Linux мы можем, сами собрать ядро, которое нам необходимо, включив в него те необходимые программные модули и драйверы, которые мы считаем целесообразным включить в ядро (а не обращаться к ним из ядра).

Глава 2. Управление задачами и памятью в операционных системах

Итак, время центрального процессора и оперативная память являются основными ресурсами в случае реализации мультипрограммных вычислений.

Оперативная память - это важнейший ресурс любой вычислительной системы, поскольку без нее (как, впрочем, и без центрального процессора) невозможно выполнение ни одной программы. Мы уже отмечали, что память является разделяемым ресурсом. От выбранных механизмов распределения памяти между выполняющимися процессорами очень сильно зависят и эффективности использования ресурсов системы, и ее производительность, и возможности, которыми могут пользоваться программисты при создании своих программ. Способы распределения времени центрального процессора

тоже сильно влияют и на скорость выполнения отдельных вычислений, и на общую эффективность вычислительной системы.

Понятий процесса (задачи) нам уже известно. Мы не будем стараться разделять понятия процесс (process) и поток (thread), вместо этого используя как бы обобщающий термин task (задача). В других разделах, если специально это не оговаривается, под задачей или процессом следует понимать практически одно и то же. Сейчас же мы будем говорить о разделении ресурса центрального процессора, поэтому термин задача может включать в себя понятие треда (потока).

Итак, ОС выполняет следующие основные функции, связанные с управлением задачами:

– создание и удаление задач;

– планирование процессов и диспетчеризация задач;

– синхронизация задач, обеспечение их средствами коммуникации.

Система управления задачами обеспечивает происхождение их через компьютер. В зависимости от состояния процесса ему должен быть предоставлен тот или иной ресурс. Например, новый процесс необходимо разместить в основной памяти – следовательно, ему необходимо выделить часть адресного пространства. Новый порожденный поток текущего процесса необходимо включить в общий список задач, конкурирующих между собой за ресурсы центрального процессора.

Создание и удаление задач осуществляется по соответствующим запросам от пользователей или от самих задач. Задача может породить новую задачу. При этом между процессами появляются «родственные» отношения, порождающая задача называется «предком», «родителем», а порождённая - «потомком» «сыном» или «дочерней задачей». «Предок» может приостановить или удалить свою дочернюю задачу, тогда как «потомок» не может управлять «предком».

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

Очевидно, что распределение ресурсов влияют конкретные потребности тех задач, которые должны выполняться параллельно. Другими словами, можно столкнуться с ситуациями, когда невозможно эффективно распределять ресурсы с тем, чтобы они не простаивали. Например, всем выполняющимся процессам требуется некоторое устройство с последовательным доступом. Но поскольку, как мы уже знаем, то не может распределяться между параллельно выполняющимися процессами, то процессы вынуждены будут очень долго ждать своей очереди. Таким образом, недоступность одного ресурса может привести к тому, что длительное время не будут использоваться и многие другие ресурсы.

Если же мы возьмем набор таких процессов, которые не будут конкурировать между собой за неразделяемые ресурсы при параллельном выполнении, то, скорее всего процессы смогут выполниться быстрее, (из-за отсутствия дополнительных ожиданий), да и имеющиеся в системе ресурсы будут использоваться более эффективно. Итак, возникает задача подбора такого множества процессов, что при выполнении они будут как можно реже конфликтовать из-за имеющихся в системе ресурсов. Такая задача называется планированием вычислительных процессов. Задача планирования процессов возникла очень давно - в первых пакетных ОС при планировании пакетов задач, которые должны были выполняться на компьютере и оптимально использовать его ресурсы. В настоящее время актуальность не так велика. На первый план уже очень давно вышли задачи динамического (или краткосрочного планирования), то есть текущего наиболее эффективного распределения, возникающего практически при каждом событии. Задачи динамического планирования стали называть *диспетчеризацией.*

Очевидно, что планирование, распределения ресурсов между уже выполняющимися процессами и потоками. Основное отличие между долгосрочным и краткосрочным

планировщиками заключается в частоте запуска: краткосрочный планировщик, например, может запускаться каждые 30 или 100 мс, долгосрочный – один раз за несколько минут (или чаще; тут многое зависит от общей длительности решения заданий пользователей).

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

Краткосрочный планировщик решает, какая из задач, находящихся в очереди готовых к выполнению, должна быть передана на исполнение. В большинстве современных операционных систем, с которыми мы сталкиваемся, долгосрочный планировщик отсутствует.

Планирование и диспетчеризация процессов и задач

Стратегии планирования

Прежде всего, следует отметить, что при рассмотрении стратегий планирования, как правило, идёт речь о краткосрочном планировании, то есть диспетчеризации. Долгосрочное планирование, как мы уже отметили, заключается в подборе таких вычислительных процессов, которые бы меньше всего конкурировали между собой за ресурсы вычислительной системы.

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

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

– отдавать предпочтение более коротким процессам;

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

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

Дисциплины диспетчеризации

Когда говорят о диспетчеризации, то всегда в явном или неявном виде имеют в виду понятие задачи (потока). Если ОС не поддерживает механизм тредов, то можно заменять понятие задачи на понятие процесса. Так как эти термины часто используются именно в таком смысле, мы вынуждены будем использовать термин «процесс» как синоним термина «задача».

Известно большое количество правил (дисциплин диспетчеризации), в соответствии, с которыми формируется список (очередь) готовых к выполнению задач. Различают два больших класса дисциплин обслуживания - бесприоритетные и приоритетные. При бесприоритетном обслуживании выбор задачи производится в некотором заранее установленном порядке без учета их относительной важности и времени обслуживания. При реализации приоритетных дисциплин обслуживания отдельным задачам предоставляется преимущественное право попасть в состояние исполнения. Перечень дисциплин обслуживания и их классификация приведены на рис. 2.1.

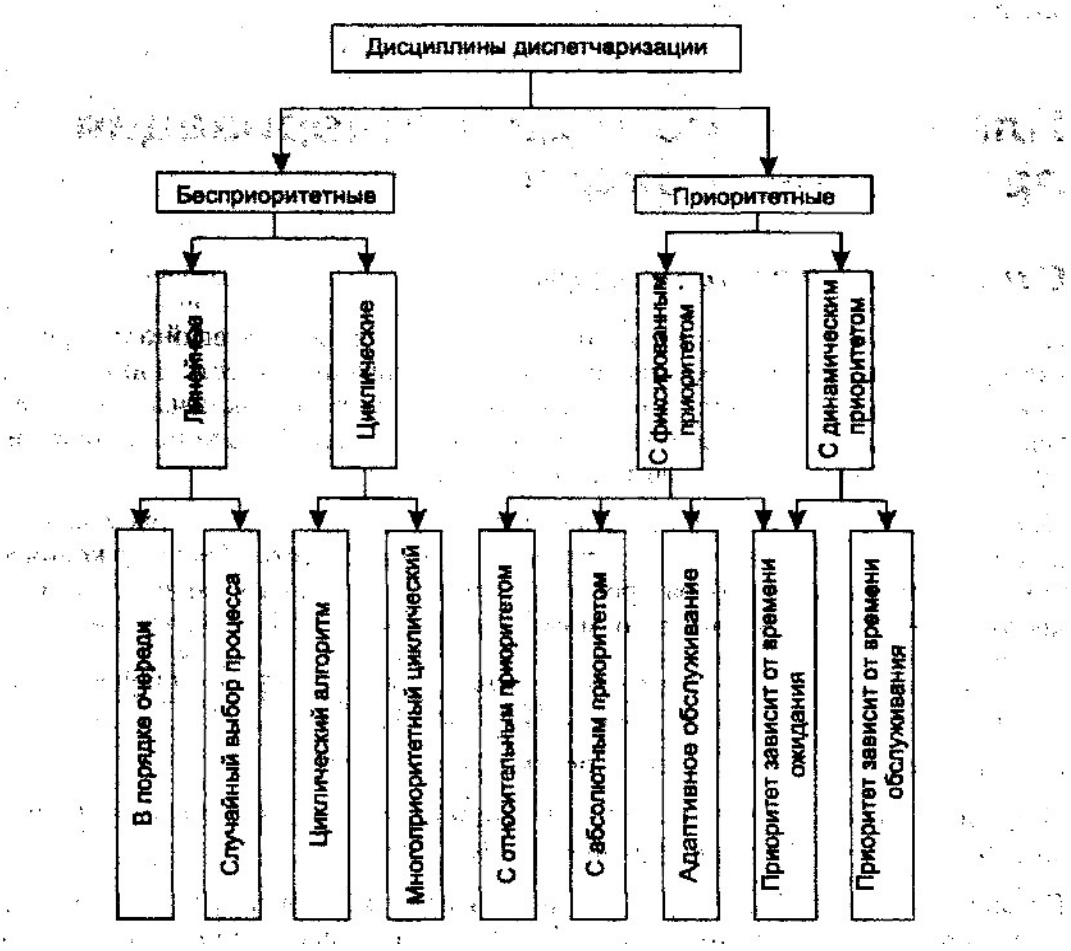

Рис. 2.1. Дисциплины диспетчеризации

Запомним о приоритетах следующее:

□ приоритет, присвоенный задаче, может являться величиной постоянной;

□ приоритет задачи может изменяться в процессе ее решения.

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

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

Самой простой в реализации является дисциплина FGFS (first come – first served), согласно которой задачи обслуживаются в порядке очереди, то есть в порядке их появления. Те задачи, которые были заблокированы в процессе работы (попали в какое-либо из состояний ожидания, например, из-за операций ввода/вывода), после перехода в состояние готовности ставятся в эту очередь готовности перед теми задачами, которые еще не выполнились. Другими словами, образуются две очереди (рис. 2.2) одна очередь образуется из новых задач, а вторая очередь – из ранее выполнявшихся, но попавших в состояние ожидание. Такой подход позволяет реализовать стратегию обслуживания «по возможности заканчивать вычисления в порядке их появления. Эта дисциплина обслуживания не требует внешнего вмешательства в ход вычислений, при ней не происходит перераспределение процессорного времени. Существующие дисциплины диспетчеризации процессов могут быть разбиты на два класса - вытесняющие, (pre-emptive) и не вытесняющие, (non-preemptive). В первых пакетных ОС часто реализовывали параллельное выполнение, заданий без принудительного перераспределения процессора между задачами. В большинстве современных ОС для мощных вычислительных систем, а также и в ОС для ПК, ориентированных на

высокопроизводительное выполнение приложений (Windows NT, OS/2, Linux), реализована вытесняющая многозадачность. Можно сказать, что рассмотренная дисциплина относится к не вытесняющим.

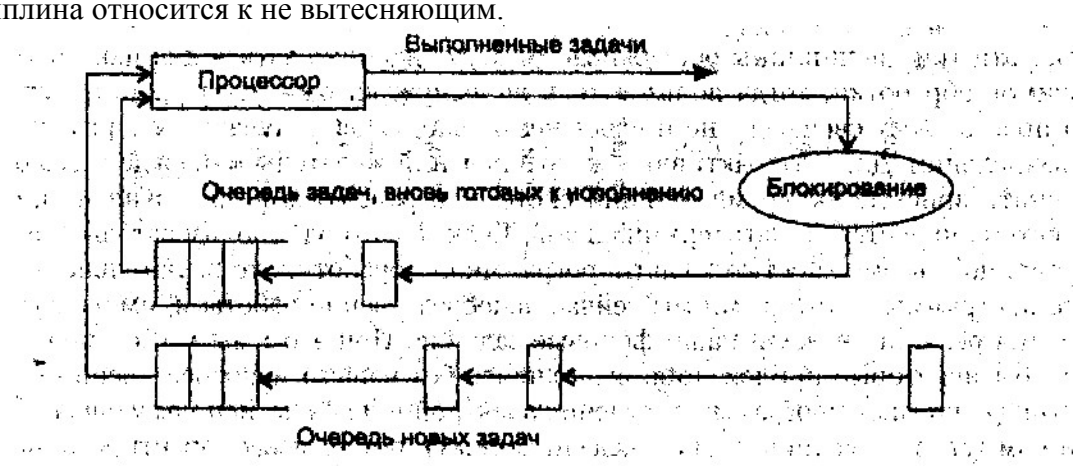

Рис. 2.2 Дисциплины диспетчеризации FCFS

К достоинствам этой дисциплины, прежде всего можно отнести простоту реализации и малые расходы системных ресурсов на формирование очереди задач.

Однако эта дисциплина приводит к тому, что при увеличении загрузки вычислительной системы растет и среднее время ожидания обслуживания, причем короткие задания (требующие небольших затрат машинного времени) вынуждены ожидать столько же, сколько и трудоемкие задания. Избежать этого недостатка позволяют дисциплины SJN и SRT.

Дисциплина обслуживания SJN (shortest job next, что означает: следующим будет выполняться кратчайшее задание) требует, чтобы для каждого задания была известна оценка в потребностях машинного времени. Необходимость сообщать ОС характеристики задач, в которых описывались бы потребности в ресурсах вычислительной системы, привела к тому, что были разработаны соответствующие языковые средства. В частности, язык JCL (job control language, язык управления заданиями) был одним из наиболее известных. Пользователи вынуждены были указывать предполагаемое время выполнения, и для того, чтобы они не злоупотребляли возможностью указать заведомо меньшее время выполнения (с целью получить результаты раньше других), ввели подсчет реальных потребностей. Диспетчер задач, сравнивал заказанное время и время выполнения, и в случае превышения указанной оценки в данном, ресурсе ставил данное задание не в начало, а в конец очереди. Еще в некоторых ОС в таких случаях использовалась система штрафов, при которой в случае превышения заказанного машинного времени оплата вычислительных ресурсов осуществлялась уже по другим расценкам.

Дисциплина обслуживания SJN предполагает, что имеется только одна очередь заданий, готовых к выполнению. И задания, которые в процессе своего исполнения были временно заблокированы (например, ожидали завершения операций ввода/вывода), вновь попадают в конец очереди готовых к выполнению наравне с вновь поступившими. Это приводят к тому, что задания, которым требуется очень много времени для своего завершения, вынуждены ожидать процессор наравне с длительными работами, что не всегда хорошо.

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

Все эти три дисциплины обслуживания могут использоваться для пакетных режимов обработки, когда пользователь не вынужден ожидать реакции системы, а просто сдает своё задание и через несколько часов получает свои результаты вычислений. Для интерактивных же вычислений желательно, прежде всего обеспечить приемлемое время реакций системы и равенство в обслуживании, если система является мультитерминальной. Если же это однопользовательская система, но с возможностью

мультипрограммной обработки, то желательно, чтобы те программы, с которыми мы сейчас непосредственно работаем, имели лучшее время реакции, нежели ниши фоновые, задания. При этом мы можем пожелать, чтобы некоторое приложения выполняясь без нашего непосредственного участия (например, программа получения электронной, почты, использующая модем и коммутируемые линии для передачи данных), тем не менее, гарантированно получали необходимую им долю процессорного времени. Для решения подобных проблем используется дисциплина обслуживания, называемая RR (round robin, круговая, карусельная), и приоритетные методы обслуживания.

Дисциплина обслуживания RR предполагает, что каждая задача получает процессное время порциями (говорят: квантами времени, q). После окончания кванта времени q задача снимается с процессора и он передаётся следующей задаче. Снятая задача ставится в конец очереди задач, готовых к выполнению. Эти дисциплина обслуживания иллюстрируется рис. 2.3. Для оптимальной работы необходимо правильно выбрать закон, по которому кванты времени выделяются задачам

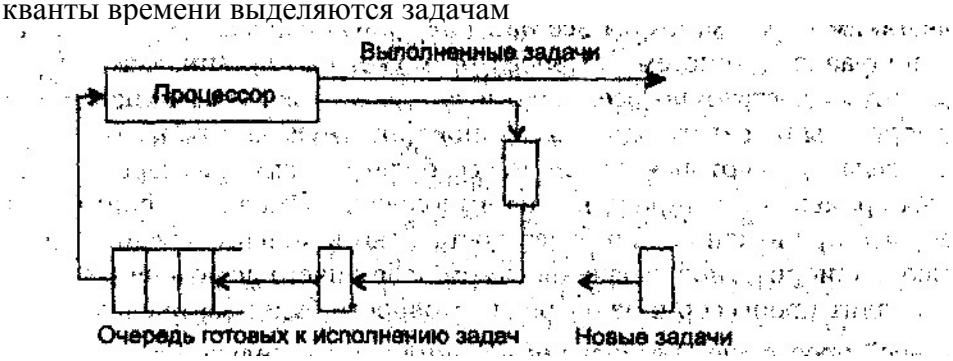

Рис. 2.3. Карусельная дисциплина диспетчеризации (round robin)

Величина кванта времени q выбирается как компромисс между приемлемым временем реакции системы на запросы пользователей (с тем, чтобы их простейшие запросы не вызывали длительного ожидания) и накладными расходами на смену контекста задач. Очевидно, что при прерываниях ОС вынуждена достаточно большой объем информации о текущем (прерываемому поставить дескриптор снятой задачи в очередь, загрузить контекст задачи, которая теперь будет выполняться (её дескриптор был первым в очереди готовых к исполнению). Если величина q велика, то при увеличении очереди к выполнению задач реакция системы станет плохой. Если же величина мала, то относительная доля накладных расходов на переключения между исполняющимися задачами, станет большой и это ухудшит производительность. В некоторых ОС есть возможность указывать в явном виде величину q либо диапазон ее возможных значений, поскольку система будет, стараться выбирать оптимальное значение сама. Например, в OS/2 в файле CONFIG.SYS есть возможность с помощью оператора TIMESLICE указать минимальное и максимальное значение для кванта q. Так, например, строка TIMESLICE=32.256 указывает, что минимальное значение q равно 32, а максимальное – 256. Если некоторая задача (тред) была прервана, поскольку выделенный ей квант времени q израсходован, то следующий выделенный ему интервал будет увеличен на время, равное одному периоду таймера (около 32 мс), и так до тех пор, пока квант времени не станет равным максимальному значению, указанному в операторе TIMESLICE. Этот метод позволяет OS/2 уменьшить накладные расходы на переключение задач только в том случае, если несколько задач параллельно выполняют длительные вычисления.

Дисциплина диспетчеризации RR– это одна из самых распространенных дисциплин. Однако бывают ситуация, когда ОС не поддерживает в явном виде дисциплину карусельной диспетчеризации. Например, в некоторые ОС реального времени используется диспетчер задач, работающий по принципам абсолютных приоритетов (процессор, предоставляется задаче с максимальным приоритетом, а при равенстве приоритетов он действует по принципу очерёдности). Другими словами, снять задачу с

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

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

Вытесняющие и не вытесняющие алгоритмы диспетчеризации

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

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

При не вытесняющей многозадачности механизм распределения между системой и прикладными программами. Прикладная программа, получив управление от операционной системы, сама определяет момент завершения своей очередной итераций и передает управление супервизору ОС с помощью соответствующего системного вызова. При этом естественно, что диспетчер задач, так же как и в случае вытесняющей мультизадачности, формирует очереди задач и выбирает в соответствии с некоторым алгоритмом (например, с учетом порядка поступления задач или их приоритетов) следующую задачу на выполнение. Такой механизм создает некоторые проблемы как пользователей, так и для разработчиков.

Для пользователей это означает, что управление системой может теряться на некоторый произвольный период времени, который определяется процессом выполнения приложения (а не системой, старающейся всегда обеспечить приемлемое время реакции на запросы пользователей). 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 или графический редактор, в то время как форматирование продолжалось бы в фоновом

режиме). Эта ситуация нежелательна, так как пользователи обычно не хотят долго ждать, когда машина завершит свою задачу.

Поэтому разработчики приложений для не вытесняющей операционной среды, на себя функции диспетчера задач, должны создавать приложения так, они выполняли свои задачи небольшими частями. Так, упомянутая выше программа форматирования может отформатировать одну дорожку дискеты и 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.

Например, в ныне уже забытой операционной среде Windows 3.x нативные приложения этой системы разделяли между собой процессорное время именно таким образом. И программисты сами должны были обеспечивать «дружественное» отношение своей программы к другим выполняемым одновременно с ней программам, достаточно часто отдавая управление ядру системы. Крайним проявлением «не дружественности» приложения является его зависание, которое приводит к общему краху системы. В системах с вытесняющей многозадачностью такие ситуации, как правило, исключены, т.к. центральный механизм диспетчеризации, во-первых, обеспечивает все задачи процессорным временем, а во вторых, даёт возможность иметь надёжные механизмы для мониторинга вычислений, и позволяет снять зависшую задачу с выполнения.

Однако распределённые функции диспетчеризации между системой и приложениями не всегда являются недостатком, а при определённых условиях может быть и преимуществом, потому что даёт возможность разработчику приложений самому проектировать алгоритм распределения процессорного времени, наиболее подходящий для данного фиксированного, набора задач. Так как разработчик сам определяет в программе момент времени отдачи управления, то при этом исключаются нерациональные прерывания программ в «неудобные» для них моменты времени. Кроме того, легко разрешаются проблемы совместного использование данных: задача во время каждой итерации, использует их монопольно и уверена, что на протяжении этого периода никто другой не изменит эти данные. Примером эффективного использования не вытесняющей многозадачности является операционная система Novell, NetWare, в которой в значительной степени благодаря этому достигнута высокая скорость выполнения файловых операций. Менее удачным оказалось, использование невытесняющей многозадачности в операционной среде Windows 3.x.

Качество диспетчеризации и гарантии обслуживания

Одна из проблем, которая возникает при выборе подходящей дисциплины обслуживания – это гарантия обслуживания. Дело в том, что при некоторых дисциплинах, например при использовании дисциплины абсолютных приоритетов, низкоприоритетные процессы оказываются обделенными многими ресурсами и, прежде всего, процессорным временем. Возникает реальная дискриминация низкоприоритетных задач, и ряд таких процессов, имеющих к тому же большие потребности в ресурсах, могут очень длительное время откладываться или, в конце концов, вообще могут быть не выполнены. Известны случаи, когда вследствие высокой загрузки вычислительной системы отдельные процессы так и не были выполнены, несмотря на то, что прошло несколько лет с момента их планирования. Поэтому вопрос гарантии обслуживания является очень актуальным.

Более жестким требованием к системе, чем просто гарантированное завершение процесса, является его гарантированное завершение, к указанному моменту времени или за указанный интервал времени. Существуют различные дисциплины диспетчеризации, учитывающие жесткие временные ограничения, но не существует дисциплин, которые могли бы предоставить больше процессорного времени, чем может быть в принципе выделено.

Планирование с учетом жестких временных ограничений легко реализовать, организуя очередь готовых к выполнению процессов в порядке возрастания их временных ограничений. Основным недостатком такого простого упорядочения, является то, что процесс (за счет других процессов) может быть обслужен быстрее, чем это ему реально необходимо. Для того чтобы избежать этого, проще всего процессорное время выделять все-таки квантами. Гарантировать, обслуживание можно следующими тремя способами:

– выделять минимальную долю процессорного времени некоторому классу процессов, если, по крайней мере, один из них готов к исполнению. Например, можно отводить 20% от каждых 10 мс процессам реального времени, 40% от каждых 2с – интерактивным процессам и 10% от каждых 5 мин – пакетным фоновым процессам;

– выделять столько процессорного времени некоторому процессу, чтобы он мог выполнить свои вычисления к сроку.

Для сравнения алгоритмов диспетчеризации обычно используются следующие критерии:

– Использование (загрузка) центрального процессора (CPU utilization). В большинстве персональных систем средняя загрузка процессора не превышает 2-3 %, доходя в моменты выполнений сложных вычислений и до 100 %. В реальных системах, где компьютеры выполняют очень много работы, например, в серверах, загрузка процессора колеблется в предела 15–40 % для легко загруженного процессора и до 90-100% – для сильно загруженного процессора.

– Пропускная способность (CPU throughput). Пропускная способность процессора может измеряться количеством процессов, которые выполняются в единицу времени.

– Время оборота (turnaround time). Для некоторых процессов важным критерием является полное время выполнения, то есть интервал от момента появления процесса во входной очереди до момента его завершения. Это время названо временем оборота и включает время ожидания во входной очереди, время ожидания в очереди готовых процессов, время ожидания в очередях к оборудованию, время выполнения в процессоре и время ввода/вывода.

– Время ожидания (waiting time). Под временем ожидания понимается суммарное время нахождения процесса в очереди готовых процессов.

– Время отклика (response time). Для интерактивных программ важным показателем является время отклика или время, прошедшее от момента попадния процесса во входную очередь до момента первого обращения к терминалу. Очевидно, что простейшая стратегия краткосрочного планировщика должна быть направлена на максимизацию средних значений загруженности и пропускной способности, времени ожидания и времени «отклика».

Правильное планирование процессов сильно влияет на производительность всей системы. Можно выделить следующие главные причины, приводящие к уменьшению производительности системы:

– Накладные расходы на переключение процессора. Они определяются не только переключениями контекстов задач, но и (при переключении на процессы другого приложения) перемещениями страниц виртуальной памяти, а также необходимостью, обновления данных в кэше (коды и данные, одной задачи, находящиеся в кэше, не нужны другой задаче и будут заменены, что приводит к дополнительным задержкам).

– Переключение на другой процесс в тот момент, когда текущий процесс выполняет критическую секцию, а другие процессы активно ожидают входа в свою критическую секцию. В этом случае потери будут особо велики (хотя вероятность прерывания выполнения коротких критических секций мала).

В случае использования мультипроцессорных систем применяются следующие методы повышения производительности системы:

– совместное планирование, при котором все потоки одного приложения (неблокированные) одновременно выбираются для выполнения процессорами и одновременно снимаются с них (для сокращения переключений контекста);

– планирование, при котором находящиеся в критической секции задачи не прерываются, а активно ожидающие входа в критическую секцию задачи не выбираются до тех пор, пока вход в секцию не освободится;

– планирование с учетом так называемых «советов» программы (во время ее выполнения). Например, в известной своими новациями ОС Mach имелись два класса таких советов (hints) – указания (разной степени категоричности) о снятии текущего процесса с процессора, а также указания о том процессе, который должен быть выбран взамен текущего.

Диспетчеризация задач с использованием динамических приоритетов

При выполнении программ, реализующих какие-либо задачи контроля и управления (что характерно, прежде всего, для систем реального времени), может случиться такая ситуация, когда одна или несколько задач не могут быть реализованы (решены) в течение длительного промежутка времени из-за возросшей нагрузки в вычислительной системе. Потери, связанные с невыполнением таких задач, могут оказаться больше, чем потери от невыполнения программ с более высоким приоритетом. При этом оказывается целесообразным временно изменить приоритет «аварийных»– задач (для которых истекает отпущенное для них время обработки). После выполнения этих задач их приоритет восстанавливается. Поэтому почти в любой ОС реального времени имеются средства для изменения приоритета программ. Есть такие средства и во многих ОС, которые не относятся к классу ОСРВ. Введение механизмов динамического изменения приоритетов позволяет реализовать более быструю реакцию системы на короткие запросы пользователей, что очень важно при интерактивной работе, но при этом гарантировать выполнение любых запросов.

Рассмотрим, например, как реализован механизм динамических приоритетов в ОС UNIX, которая, как известно, не относится к ОСРВ. Приоритет процесса вычисляется следующим образом. Во-первых, в вычислении участвуют значения двух полей дескриптора процесса – p_nice и p_cpu. Первое из них назначается пользователем явно или формируется по умолчанию с помощью системы программирования. Второе поле формируется диспетчером задач (планировщиком разделения времени) и называется системной составляющей или текущим приоритетом. Другими словами, каждый процесс имеет два атрибута приоритета, с учетом которого и распределяется между исполняющимися задачами процессорное время: *текущий приоритет*, на основании которого происходит планирование, и заказанный *относительный приоритет*, называемый nice number (или просто nice).

Схема нумерации (числовых значений) текущих приоритетов различна для различных версий UNIX. Например, более высокому значению текущего приоритета может, соответствовать более низкий фактический приоритет планирования. Разделение между приоритетами режима ядра и задачи также зависит от версии. Рассмотрим частный случай, когда текущий приоритет процесса варьируется в диапазоне от 0 (низкий приоритет) до 127 (наивысший приоритет). Процессы, выполняющиеся в режиме задачи, имеют более низкий приоритет, чем в режиме ядра. Для режима задачи приоритет меняется в диапазоне 0-65, для режима ядра – 66-95 (системный диапазон). Процессы, приоритеты которых лежат в диапазоне 96-127, являются процессами с фиксированным приоритетом, не изменяемым операционной системой, и предназначены для поддержки приложений реального времени.

Процессу, ожидающему недоступного в данный момент ресурса, система определяет значение приоритета сна, выбираемое ядром из диапазона системных приоритетов и

связанное с событием, вызвавшее это состояние. Когда процесс пробуждается, ядро устанавливает значение текущего приоритета процесса равным приоритету сна. Поскольку приоритет такого процесса находится в системном диапазоне и выше, чем приоритет режима задачи, вероятность предоставления процессу вычислительных ресурсов весьма велика. Такой подход позволяет в частности, быстро завершить системный вызов, выполнение которого, в свою очередь, может блокировать некоторые системные ресурсы.

После завершения системного вызова перед возвращением в режим задачи ядро восстанавливает приоритет режима задачи, сохраненный перед выполнением системного вызова. Это может привести к понижению приоритета, что, в свою очередь, вызовет переключение контекста.

Текущий приоритет процесса в режиме задачи p_priuser, как мы только что отметили, зависит от значения nice number и степени использования вычислительных ресурсов p_cpu:

p_priuser - a*p_nice - b*p_cpu

Задача планировщика разделения времени – справедливо распределить вычислительный ресурс между конкурирующими процессами. Для принятия решения о выборе следующего запускаемого процесса планировщику необходима информация об использовании процессора. Эта составляющая приоритета уменьшается обработчиком прерываний таймера по каждому «тику» таймера. Таким образом, пока процесс выполняется в режиме задачи, его текущий приоритет линейно уменьшается.

Каждую секунду ядро пересчитывает текущие приоритеты процессов, готовых к запуску (приоритеты которых меньше некоторого порогового значения, в нашем примере эта величина равна 65), последовательно увеличивая их. Это осуществляется за счет того, что ядро последовательно уменьшает отрицательную компоненту времени использования процессора. Как результат, эти действия приводят к перемещению процессов в более приоритетные очереди и повышают вероятность их последующего запуска.

Возможно использование следующей формулы:

p_cpu = p_cpu/2

Это правило проявляет недостаток нивелирования приоритетов при повышении загрузки системы. Происходит это потому что, в таком случае каждый процесс получает незначительней объем вычислительных ресурсов и, следовательно, имеет малую составляющую. p_cpi, которая еще более уменьшается благодаря формуле пересчета величины p_cpi. В результате степень использования, процессора перестает оказывать заметное влияние на приоритет, и низкоприоритетные процессы (то есть процессы с высоким значением nice number), практически «отлучаются» от вычислительных ресурсов систему.

В некоторых версиях ОС UNIX для пересчета значения p_cpu используется другая формула;

p_cpu = p_cpu*(2*load)/(2*load+l)

Здесь параметр load равен среднему числу процессов, находившихся в очереди на выполнение за последнюю секунду, и характеризует среднюю загрузку системы за этот период времени. Такой алгоритм позволяет частично избавиться от недостатка планирования по формуле p_cpu=p_cpu/2, поскольку при значительной загрузке системы уменьшение p_cpu при пересчете будет происходить медленнее.

Описанные алгоритмы планирования позволяют учесть интересы низкоприоритетных процессов, так как в результате длительного ожидания очереди на запуск приоритет таких процессов увеличивается, соответственно увеличивается и вероятность запуска. Эти алгоритмы также обеспечивают более вероятный выбор планировщиком интерактивных процессов по отношению к вычислительным (фоновым). Такие задачи, как командный интерпретатор или редактор, большую часть времени проводят в ожидании ввода, имея, таким образом, высокий приоритет (приоритет сна). При наступлении ожидаемого

события (например, пользователь осуществил ввод данных) им сразу же предоставляются вычислительные ресурсы. Фоновые процессы, потребляющие значительные ресурсы процессора, имеют высокую составляющую p_cpu и, как следствие, менее высокий приоритет.

Аналогичные механизмы имеют место и в таких ОС, как OS/2 или Windows NT. Правда, алгоритмы изменения приоритета задач в этих системах иные. Например, в Windows NT каждый поток (тред) имеет базовый уровень приоритета, который лежит в диапазоне от двух уровней ниже базового приоритета процесса, его породившего, до двух уровней выше этого приоритета, как показано на рис. 2.4. Базовый приоритет процесса определяет, сколь сильно метут различаться приоритеты потоков процесса и как они соотносится с приоритетами потоков других процессов. Поток наследует этот базовый приоритет и может изменять его так, чтобы он стал немного больше или немного меньше. В результате получается приоритет планирования, с которым поток и начинает исполняться. В процессе исполнения потока его приоритет может отклоняться от базового.

На рис. 2.4 показан динамический приоритет потока, нижней границей которого является базовый приоритет потока, а верхняя – зависит от вида работ, исполняемых потоком. Например, если поток обрабатывает пользовательский ввод, то диспетчер задач Windows NT поднимает его динамический приоритет, если же он выполняет вычисления, то диспетчер постепенно снижаем его приоритет до базового. Снижая приоритет одного процесса и поднимая приоритет другого, подсистемы могут управлять относительной приоритетностью потоков внутри процесса.

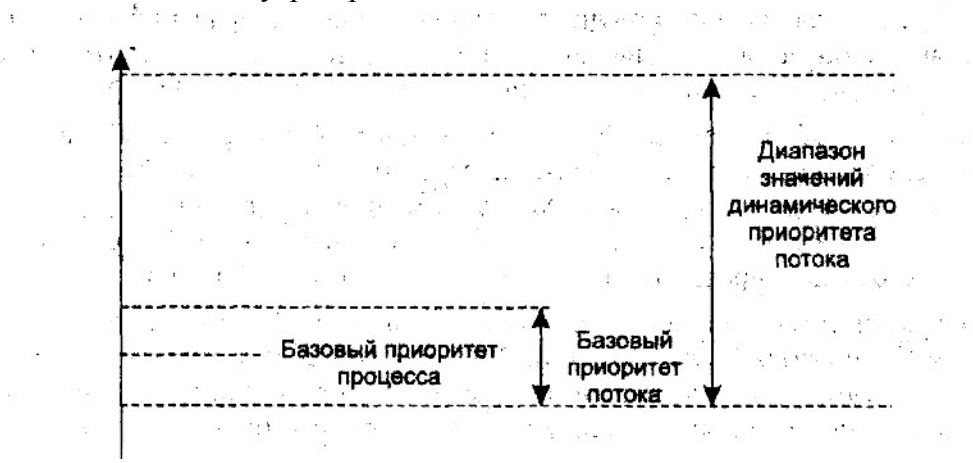

Рис. 2.4. Схема динамического изменения приоритетов в Windows NT

Для определения порядка выполнения потоков диспетчер использует систему приоритетов, направляя на выполнение потоки с высоким приоритетом раньше потоков с низкими приоритетами. Система прекращает исполнение или вытесняет (preempts) текущий поток, если становится готовой к выполнению другая задача (поток) с более высоким приоритетом.

Существует группа очередей – по одной для каждого приоритета. Windows NT поддерживает 32 уровня приоритетов; потоки делятся на два класса; реального времени и переменного приоритета. Потоки реального времени, имеющие приоритеты от 16 до 31 – это высокоприоритетные потоки, используемыми программами с критическим временем выполнения, то есть требующие немедленного внимания системы (по терминологии Microsoft).

Диспетчер задач просматривает очереди, начиная с самой приоритетной. При этом если очередь пустая, то есть, нет готовых к выполнению задач с таким приоритетом, осуществляется переход к следующей очереди. Следовательно, если есть задачи, требующие процессор немедленно, они будут обслужены в первую очередь. Для собственно системных модулей, функционирующих в статусе задач, зарезервирована очередь с номером 0.

Большинство потоков в системе относятся к классу переменного приоритета с уровнями приоритета (номером очереди) от 1 до 15.Эти очереди используются потоками с переменным приоритетом (variable-priority), так как диспетчер задач корректирует их приоритеты потере выполнениям задан для оптимизации отклика системы. Диспетчер приостанавливает, исполнение текущего потока после того, как тот израсходует свой квант времени. При этом если прерванный тред – это поток переменною приоритета, то, диспетчер задач понижает его приоритет на единицу и перемещает в другую очередь. Таким образом, приоритет, потока, выполняющего много вычислений постепенно понижается (до значения его базового приоритета). С другой стороны, диспетчер повышает приоритет потока после освобождения задачи (потока) из состояния ожидания. Обычно добавка к приоритету потока определяется кодом исполнительной системы находящимся, вне ядра ОС, однако величина этой добавки зависит от типа события, которого ожидал заблокированный тред. Так, например поток, ожидавший ввода очередного байта с клавиатуры, получает большую добавку к значению, своего приоритета, чем процесс ввода/вывода, работавший с дисковым накопителем, Однако в любом случае значение приоритета не может достигнуть 16.

В операционной системе OS/2 схема динамической приоритетной диспетчеризации несколько иная, хотя и похожа на рассмотренную. В OS/2 имеются четыре класса задач. И для каждого класса задач имеется своя группа приоритетов с интервалом значений от 0 до 31. Итого, 128 различных уровней и, соответственно, 128 возможных очередей готовых к выполнению задач (тредов, потоков).

Класс задач, имеющих самые высокие значения приоритета, называется критическим (time critical). Этот класс предназначается для задач, которые мы в обиходе называем задачами реального времени, то есть для них должен быть обязательно предоставлен определенный минимум процессорного времени. Наиболее часто встречающимися задачами, которые относят к этому классу, являются задачи коммуникаций (например, задача управления последовательным портом, принимающим биты с коммутируемой линии, к которой подключен модем, или задачи управления сетевым оборудованием). Если такие задачи не получат управление в нужный момент времени, то сеанс связи может прерваться.

Следующий класс задач имеет название *приоритетного*. Поскольку к этому классу относят задачи, которые выполняют по отношению к остальным задачам роль сервера, то его еще иногда называют *серверным*. Приоритет таких задач должен быть выше, это будет гарантировать, что запрос на некоторую функцию со стороны обычных задач выполнится сразу, а не будет дожидаться, пока до него дойдет очередь на фоне других пользовательских приложений.

Большинство задач относят к обычному классу, его еще называют регулярным или стандартным. По умолчанию система программирования порождает задачу, относящуюся именно к этому классу. Наконец, существует еще класс фоновых задач называемой в OS/2 остаточным. Программы этого класса получают процессорное время только тогда, когда нет задач т других классов, которым сейчас нужен процессор. В качестве примера такой задачи можно привести программу проверки электронной почты.

Внутри каждого из вышеописанных классов задачи, имеющие одинаковый уровень приоритета, выполняются в соответствий с дисциплиной round-robin. Переход от одного треда к другому происходит либо по окончании отпущенного ему кванта времени, либо по системному прерыванию, передающему управление задаче с более высоким приоритетом (таким, образом, система вытесняет задачи с более низким приоритетом для выполнения задач с более высоки приоритетом и может обеспечить быструю реакцию на важные события).

OS/2 самостоятельно изменяет приоритет выполняющийся программ независимо от уровня, установленного самим приложением. Этот механизм называется повышением приоритета. ОС изменяем приоритет задачи в следующих трех случаях:

□ Увеличение приоритета активной задачи (foreground boost). Приоритет задачи автоматически увеличивается, когда она становится активной. Это снижает время реакции активного приложения на действия пользователя по сравнению с фоновыми программами.

□ Увеличение приоритета ввода/вывода (input/output boot). По завершении операции ввода/вывода задача получает самый высокий уровень приоритета ее класса. Таким образом, обеспечивается завершение всех незаконченных операций ввода/вывода.

□ Увеличение приоритета «забытых» задач (starvation boost). Если задача не получает управление в течение достаточно долгого времени (этот промежуток времени задает оператор MAXWAIT в файле CQNFIG.sys) диспетчер задач OS/2 временно присваивает ей уровень приоритета, не превышающий критический. В результате переключение на такую «забытую» программу происходит быстрее. После выполнения приложения течение одного кванта времени, его приоритет вновь снижается до остаточного. В сильно загруженных системах этот механизм позволяет программам с остаточным приоритетом работать хотя бы в краткие интервалы времени. В противном случае они вообще никогда бы не получили управление.

Если нам нет, необходимости использовать метод динамического изменения приоритета, то с помощью оператора PRIGPITY - ABSOLUTE в файле CONFIG.SYS можно ввести дисциплину абсолютных приоритетов; по умолчанию оператор PRIORITY имеет значение DYNAMIC.

Память и отображения, виртуальное адресное пространство

Если не принимать во внимание программирование на машинном языке (эта технология практически не используется уже очень давно), то можно сказать, что программист обращается к памяти с помощью некоторого набора логических имен, которые чаще всего являются символьными, а не числовыми и для которого отсутствует отношение порядка. Другими словами, в общем случае множество переменных неупорядочено, хотя отдельные переменные и могут иметь частичную упорядоченность (например, элементы массива). Имена переменных и входных точек программных модулей составляют пространство имен.

С другой стороны, существует понятие физической оперативной памяти, собственно с которой и работает процессор, извлекая из нее команды и данные и помещая в нее результаты вычислений. *Физическая память* представляет собой упорядоченное множество ячеек, и все они пронумированны, то есть к каждой из них можно обратиться, указав ее порядковый номер (адрес). Количество ячеек физической памяти ограничено и фиксировано.

Системное программное обеспечение должно связать каждое указанное пользователем имя с физической ячейкой памяти, то есть осуществить, отображение пространства имен на физическую память компьютера. В общем случае это отображение осуществляется в два этапа (рис. 2.5), сначала системой программирования, а затем операционной системой (с помощью специальных программных модулей управления памятью и использования соответствующих аппаратных средств вычислительной системы). Между этими этапами обращения к памяти имеют форму виртуального или логического адреса. При этом можно сказать, что множество всех допустимых значений виртуального адреса для некоторой программы определяет ее виртуальное адресное пространство или виртуальную память. Виртуальное адресное пространство программы, прежде всего, зависит от архитектуры процессора и от системы программирования и практически не зависит от объема реальной физической памяти, установленной в компьютер. Можно еще сказать, что адреса команд и переменных в готовой машинной программе, подготовленной к выполнению системой программирования, как раз и являются виртуальными адресами.

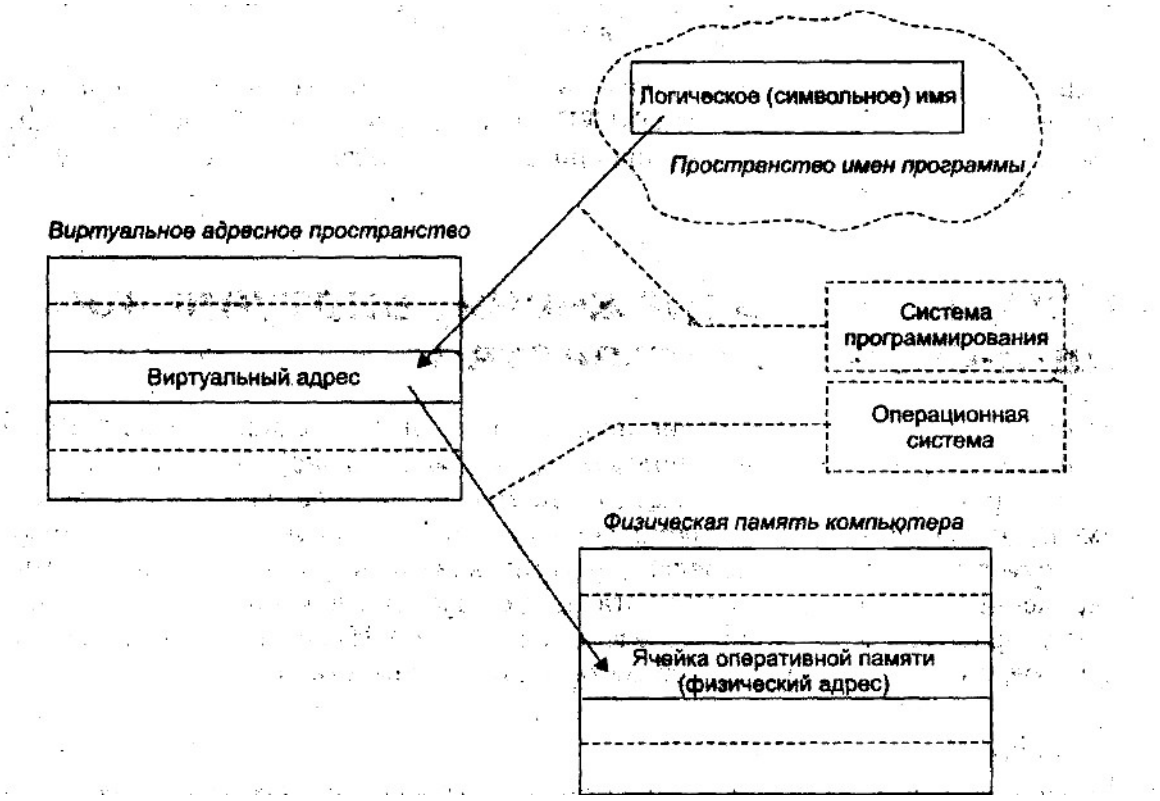

Рис. 2.5. Память и отображения

Как мы знаем, система программирования осуществляет трансляцию и компоновку программы, используя библиотечные программные модули. В результате работы системы программирования, полученные виртуальные адреса могут иметь как двоичную форму, так и символьно-двоичную, то есть некоторые программные модули (их, как правило, большинство) и их переменные получают какие-то числовые значения, а те модули, адреса для которых не могут быть сейчас определены, имеют по-прежнему символьную форму и окончательная привязка их к физическим ячейкам будет осуществлена на этапе загрузки программы в память перед ее непосредственным выполнением.

Одним из частных случаев отображения пространства имен на физическую память является тождественность виртуального адресного пространства физической памяти. При этом нет необходимости осуществлять второе отображение. В данном случае говорят, что система программирования генерирует абсолютную двоичную программу; в этой программе все двоичные адреса таковы, что программа может исполняться только в том случае, если ее виртуальные адреса будут точно соответствовать физическим. Часть программных модулей любой операционной системы обязательно должна быть абсолютными двоичными программами. Эти программы размещаются по фиксированным адресам и с их помощью уже можно впоследствии реализовывать размещение остальных программ, подготовленных системой программирования таким образом, что они могут работать на различных физических адресах (то есть на тех адресах, на которые их разместит операционная система).

Другим частным случаем этой, общей схемы трансляции адресного пространства является тождественность виртуального адресного пространства исходному пространству имен. Здесь уже отображение выполняется самой ОС, которая во время исполнения использует таблицу символьных имен. Такая схема отображения используется чрезвычайно редко, так как отображение имен на адреса необходимо выполнять для каждого вхождения имени (каждого нового имени) и особенно много времени тратится на квалификацию имен. Данную схему можно было встретить в интерпретаторах, в которых стадии трансляции и исполнения практически неразличимы. Это характерно для простейших компьютерных систем, в которых вместо операционной системы использовался встроенный интерпретатор (например, Basic).

Возможны и промежуточные варианты. В простейшем случае транслятор-компилятор генерирует относительные адреса, которые, по сути, являются виртуальными адресами с последующей настройкой программы на один из непрерывных разделов. Второе отображение осуществляется перемещающим загрузчиком. После загрузки программы виртуальный адрес теряется, и доступ выполняется непосредственно к физическим ячейкам. Более эффективное решение достигается в том случае, когда транслятор вырабатываете качестве виртуального адреса относительный адрес и, информацию о начальном адресе, а процессор, используя подготавливаемую операционной системой адресную информацию, выполняет второе отображение не один раз (при загрузке программы), а при каждом обращении к памяти.

Термин виртуальная память фактически относите» к системам, которые сохраняют виртуальные адреса во время исполнения. Т.к. 2-е отображение осуществляется в процессе исполнения задачи, то адреса физических ячеек могут изменяться. При правильном применении такие изменения могут улучшить использование памяти, избавив программиста от деталей управления ею, и даже увеличить надежность вычислений.

Если рассматривать общую схему двухэтапного отображения адресов, то с позиции соотношения объемов упомянутых адресных пространств можно отметить наличие следующих трех ситуаций:

□ объем виртуального адресного пространства программы V_V меньше объема физической памяти Vp;

□ $V_v = Vp$;

□ $V_v > Vp$.

Первая ситуация, при которой Vv< Vp, ныне практически не встречается, но тем не менее это реальное соотношение. Скажем, не так давно 16-разрядные мини-ЭВМ имели систему команд, в которых пользователи-программисты могли адресовать до 2^{16} =64K адресов (обычно в качестве адресуемой единицы выступала ячейка памяти размером с байт). А физически старшие модели этих мини-ЭВМ могли иметь объем оперативной памяти в несколько мегабайт. Обращение к памяти столь большого объема осуществлялось с помощью специальных регистров, содержимое которых складывалось с адресом операнда (или команды), извлекаемым и/или определяемым из поля операнда (или из указателя команды). Соответствующие значения в эти специальные регистры, выступающие как базовое смещение в памяти, заносила операционная система. Для одной задачи в регистр заносилось одно значение, а для второй (третьей, четвертой и т. д.) задачи, размещаемой одновременно с первой, но в другой области памяти, заносилось, соответственно, другое значение. Вся физическая память, таким образом, разбивалась на разделы объемом по 64 Кбайт, и на каждый такой раздел осуществлялось отображение своего виртуального адресного пространства.

Ситуация, когда Vv = Vp встречалась достаточно часто, особенно характерна она была для недорогих вычислительных комплексов. Для этого случая имеется большое количество методов распределения оперативной памяти.

Наконец, в наше время мы уже достигли того, что ситуация Vv > Vp встречается даже в ПК, то есть в самых распространенных и недорогих компьютерах. Теперь это самая распространенная ситуация, и для нее имеется несколько методов распределения памяти, отличающихся как сложностью, так и эффективностью.

Простое непрерывное распределение и распределение с перекрытием (оверлейные структуры)

Простое непрерывное распределение – это самая простая схема, согласно которой вся память условно может быть разделена на три части:

□ область, занимаемая операционной системой;

□ область, в которой размещается исполняемая задача;

□ незанятая ничем (свободная) область памяти.

Изначально являясь самой первой схемой, она продолжает и сегодня быть достаточно распространенной. Эта схема предполагает, что ОС не поддерживает мультипрограммирование, поэтому не возникает проблемы распределения памяти между несколькими задачами. Программные модули, необходимые для всех программ, располагаются в области самой ОС, а вся оставшаяся память может быть предоставлена задаче. Эта область памяти при этом получается непрерывной, что облегчает работу системы программирования. Поскольку в различных однотипных вычислительных комплексах может быть разный состав внешних устройств (и, соответственно, они содержат различное количество драйверов), для системных нужд могут быть отведены отличающиеся объемы оперативной памяти, и получается, что можно не привязывать жестко виртуальные адреса программы к физическому адресному пространству. Эта привязка осуществляется на этапе загрузки задачи в память.

Чтобы для задач отвести как можно больший объем памяти, операционная система строится таким образом, что постоянно в, оперативной памяти располагается только самая нужная ее часть. Эту часть ОС стали называть ядром. Остальные модули ОС могут быть обычными диск-резидентными, (или транзитивными), то есть загружаться в оперативную память только по необходимости, и после своего выполнения, вновь освобождать память.

Такая схема распределения влечет за собой два вида потерь вычислительных ресурсов ресурсов – потеря процессорного времен, потому что процессор простаивает, пока задача ожидает завершения операций ввода/вывода, и потери самой оперативной памяти, потому что далеко не каждая программа использует всю память, а режим работы в этом случае однопрограммный. Однако это очень недорогая реализация и можно отказаться от многих функций операционной системы. В частности, такая сложная проблема, как защита памяти, здесь почти не стоит.

Мы не будем рассматривать все различные варианты простого непрерывного распределения памяти, которых было очень и очень много, а ограничимся схемой распределения памяти для MS-DOS.

Если есть необходимость создать программу, логическое (и виртуальное) адресное пространство которой должно быть больше, чем свободная область памяти или даже больше, чем весь возможный объем оперативной памяти, то используется распределение с перекрытием (так называемые оверлейные структуры (перекрытие, расположение поверх чего-то)). Этот метод распределения предполагает, что вся программа может быть разбита на части – сегменты. Каждая оверлейная программа имеет одну главную часть (main) и несколько сегментов (segment), причем в памяти машины одновременно могут находиться только ее главная часть и один или несколько не перекрывающихся сегментов.

Пока в оперативной памяти располагаются выполняющиеся сегменты, остальные находятся во внешней памяти. После, того как текущий (выполняющийся) сегмент завершит свое выполнение, возможны два варианта. Либо он сам (если данный сегмент не нужно сохранить во внешней памяти в его текущем состоянии) обращается к ОС с указанием, какой сегмент должен быть загружен в память следующим. Либо он возвращает управление главному сегменту задачи (в модуль main), и уже тот обращается к ОС с указанием, какой сегмент сохранить (если это нужно), а какой сегмент загрузить в оперативную память, и вновь отдает управление одному из сегментов, располагающихся в памяти. Простейшие схемы сегментирования предполагают, что в памяти в каждый конкретный момент времени может располагаться только один сегмент (вместе с модулем main). Более сложные схемы, используемые в больших вычислительных системах, позволяют располагать по несколько сегментов. В некоторых вычислительных комплексах могли существовать отдельно сегменты кода и сегменты данных. Сегменты кода, как правило, не претерпевают изменений в процессе своего исполнения, поэтому при загрузке нового сегмента кода, на место отработавшего последний можно не

сохранять во внешней памяти, в отличие от сегментов данных, которые сохранять необходимо.

Первоначально программисты сами должны были включать в тексты своих программ соответствующие обращения к ОС (их называют вызовами) и тщательно планировать, какие сегменты могут находиться в оперативной памяти одновременно, чтобы их адресные пространства не пересекались. Однако с некоторых пор эти вызовы система программирования стала подставлять в код программы сама, автоматически, если в том возникает необходимость. Так, в известной и популярной система программирования Turbo Pascal, начиная с третьей версии, программист просто указывал, что данный модуль является оверлейным. И при обращении к нему из основной программы модуль загружался в память и ему передавалось управление. Все адреса определялись системой программирования автоматически, обращения к DOS для загрузку оверлеев тоже генерировались системой Turbo Pascal.

Распределение статическими и динамическими разделами

Для организации мультипрограммного режима необходимо обеспечить одновременное расположение в оперативной памяти нескольких задач (целиком, ил и их частями). Самая простая схема распределения памяти между несколькими задачами предполагает, что память, незанятая ядром ОС, может быть разбита на несколько непрерывных частей (зон, разделов). Разделы характеризуются именам, типом, границами (как правило, указываются, начало раздела и его длина).

Разбиение памяти на несколько непрерывных разделов может быть фиксированным (статическим), либо динамическим (то есть процесс выделения нового раздела памяти происходит непосредственно при появлении новой задачи).

1 Partition, region – раздел.

Распределение статическими и динамическими разделами

Вначале мы кратко рассмотрим статическое распределение памяти на несколько разделов.

Разделы с фиксированными границами

Разбиение всего объема оперативной памяти на несколько разделов может осуществляться единовременно (то есть в процессе генерации варианта ОС, который потом и эксплуатируется) или по мере необходимости, оператором системы. Однако и во втором случае при выполнении разбиения памяти на разделы вычислительная система более ни для каких целей в этот момент не используется. Пример разбиения памяти на несколько разделов приведен на рис. 2.6.

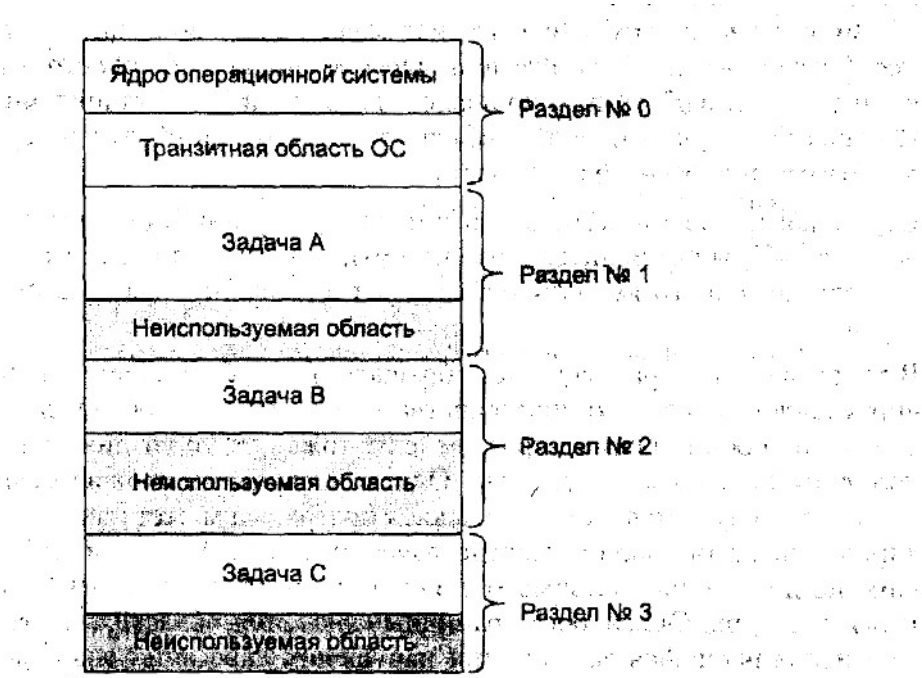

Рис. 2.6. Распределение памяти разделами с фиксированными границами

В каждом разделе в каждый момент времени может располагаться по одной программе (задаче). В этом случае по отношению к каждому разделу можно применить все те методы создания программ, которые используются для однопрограммных систем. Возможно использование оверлейных структур, что позволяет создавать большие сложные программы и в то же время поддерживать коэффициент мультипрограммирования (обычно на практике для загрузки центрального процессора для уровня 90% необходимо, чтобы коэффициент мультипрограммирования был не менее 4-5.А для того, чтобы полно использовать и остальные ресурсы системы, желательно иметь μ на уровне 10-15) (под коэффициентом мультипрограммирования (μ) понимают количество параллельно выполняемых программ) на должном уровне. Первые мультипрограммные ОС строились по этой схеме. Использовалась эта схема и много лет спустя при создании недорогих вычислительных систем, ибо она являемся несложной, и обеспечивает возможность параллельного выполнения программ. Иногда в некотором разделе размещалось по несколько небольших программ, которые постоянно в нем и находились. Такие программы назывались ОЗУ–резидентными (или просто – резидентными). Они же используются и в современных встроенных системах; правда, для них характерно, что все программы являются, резидентными и внешняя память во время работы вычислительного оборудования не используется.

При небольшом объеме памяти и, следовательно, небольшом количестве разделов увеличить количество параллельно выполняемых приложений (особенно когда эти приложения интерактивны и во время своей работы они фактически не используют процессорное время, а в основном ожидают операций ввода-вывода) можно за счет своппинга (swapping). При своппинге задача может быть целиком выгружена на магнитный диск (перемещена во внешнюю память), а на ее место загружается либо более привилегированная, либо просто готовая к выполнению другая задача, находившаяся на диске в приостановленном состоянии. При своппинге из основной памяти во внешнюю (и обратно) перемещается вся программа, а не ее отдельная часть.

Серьезная проблема, которая возникает при организации мультипрограммного режима работы вычислительной системы, – это защита как самой ОС от ошибок и преднамеренного вмешательства задач в ее работу, так и самих задач друг от друга.

В самом деле, программа может обращаться к любым ячейкам в пределах своего виртуального адресного пространства. Если система отображения памяти не содержит ошибок и в самой программе их тоже нет, то возникать ошибок при выполнении,

программы не должно. Однако в случае ошибок адресации, что не так уж и редко случается, исполняющаяся программа может начать «обработку» чужих данных или кодов с непредсказуемыми последствиями. Одной из простейших, но достаточно сильных мер является введение регистров защиты памяти. В эти регистры ОС заносят граничные значения области памяти раздела текущей исполняющейся задачи. При нарушении адресации возникает прерывание, и управление передается супервизору ОС. Обращения к ОС за необходимыми сервисами осуществляются не напрямую, а через команды программных прерываний, что обеспечивает передачу управления только в предопределенные входные точки кода ОС, и в системной режиме работы процессора, при котором регистры защиты памяти игнорируются. Таким образом, выполнение функции защиты требует введения специальных аппаратных механизмов, используемых операционной системой.

Основным недостатком такого способа распределения памяти является наличие порой достаточно больших объёма неиспользуемой памяти (см. рис. 2.6). Неиспользуемая память может быть в каждом из разделов. Поскольку разделов несколько, то и неиспользуемых областей получается несколько, поэтому такие потери стали называть фрагментацией памяти. В отдельных разделах потери памяти могут быть очень значительными, однако использовать фрагменты свободной памяти при таком, способе распределения не представляется возможным. Желание разработчиков сократить столь значительные потери привело их к следующим двум решениям:

– выделять, раздел ровно такого объёма, который нужен под текущую задачу;

– размещать задачу не в одной непрерывной области памяти, а в нескольких областях;

Второе решение реализовалось в нескольких способах организации виртуальной памяти. Мы их обсудим в следующем разделе, а сейчас кратко рассмотрим первое решение.

Разделы с подвижными границами

Чтобы избавиться от фрагментации, можно попробовать размещать в оперативной памяти задачи плотно, одну за другой, выделяя ровно столько памяти, сколько задача требует. Одной из первых ОС, реализовавшей такой способ распределения памяти, была OS MVT (мультипрограммирование с переменным числом задач). Специальный планировщик (диспетчер памяти) ведет список адресов свободной оперативной памяти. При появлении новой задачи диспетчер памяти просматривает этот список и выделяет для задачи раздел, объем которого либо равен необходимому, либо чуть больше, если память выделяется не ячейками, а некими дискретными единицами. При этом модифицируется список свободной памяти. При освобождении раздела диспетчер памяти пытается объединить освобождающийся раздел с одним из свободных участков, если таковой является смежным.

При этом список свободных участков может быть упорядочен либо по адресам, либо по объему. Выделение памяти под новый раздел может осуществляться одним из трех способов:

□ первый подходящий участок;

□ самый подходящий участок;

□ самый неподходящий участок.

В первом случае список свободных областей упорядочивается по адресам (например, по возрастанию адресов). Диспетчер памяти просматривает этот список и выделяет задаче раздел в той области, которая первой подойдёт по объему. В этом случае, если такой фрагмент имеется, то в среднем необходимо просмотреть половину списка. При освобождении раздела также необходимо просмотреть половину списка. Правили «первый подходящий» приводит к тому, что память для небольших задач преимущественно будет выделяться в области младших адресов и, следовательно, это

будет увеличивать вероятность того, что в области старших адресов будут образовываться фрагменты достаточно большого объема.

Способ «самый подходящий» предполагает, что список свободных областей упорядочен по возрастанию объема этих фрагментов. В этом случае при просмотре списка для нового раздела будет использован фрагмент свободной памяти, объём которой наиболее точно соответствует требуемому. Требуемый раздел будет определяться по-прежнему в результате просмотра в среднем половины списка.

Однако оставшийся фрагмент оказывается настолько малым, что в нем уже вряд ли удастся разместить какой-либо еще раздел и при этом этот фрагмент попадет в самое начало списка. Поэтому в целом такую дисциплину нельзя назвать эффективной.

Как ни странно, самым эффективным правилом является последнее, по которому для нового раздела выделяется «самый неподходящий» фрагмент свободной памяти. Для этой дисциплины список свободных областей упорядочивается по убыванию объема свободного фрагмента. Очевидно, что если есть подходящий фрагмент памяти, то он сразу же и будет найден, и поскольку этот фрагмент является самым большим, то, скорее всего, после выделения из него раздела памяти для задачи оставшаяся область памяти еще сможет быть использована в дальнейшем.

Однако очевидно, что при любой дисциплине обслуживания, по которой работает диспетчер памяти, из-за того, что задачи появляются и завершаются в произвольные моменты времени и при этом они имеют разные объемы, то в памяти всегда будет наблюдаться сильная фрагментация. При этом возможны ситуаций, когда из-за сильной фрагментации памяти диспетчер памяти не сможет образовать новый раздел, хотя суммарный объем свободных областей будет больше, чем необходимо для задачи. В этой ситуации, возможно, организовать так называемое «уплотнение памяти». Для уплотнения памяти все вычисления приостанавливаются, и диспетчер памяти корректирует свои списки, перемещая разделы в начало памяти (или наоборот, в область старший адресов). При определении физических адресов задачи будут участвовать новые значения базовых регистров, с помощью которых и осуществляется преобразование виртуальных адресов в физические. Недостатком этого решения является потеря времени на уплотнение и, что самое главное, невозможность при этом выполнять сами вычислительные процессы.

Данный способ распределения памяти, тем не менее применялся достаточно длительное время в нескольких операционных системах, поскольку в нем для задач выделяется непрерывное адресное пространство, а это упрощает создание систем программирования и их работу.

Сегментная, страничная и сегментно-страничная организация памяти

Методы распределения памяти, при которых задаче уже может не предоставляться сплошная (непрерывная) область памяти, называют разрывными. Идея выделять память задаче не одной сплошной областью, а фрагментами требует для своей реализации соответствующей аппаратной поддержки – нужно иметь относительную адресацию. Если указывать адрес начала текущего фрагмента программки величину смещения относительно этого начального адреса, то можно указать необходимую нам переменную или команду. Таким образом, виртуальный адрес можно представить состоящим из двух полей. Первое поле будет указывать часть программы (с которой сейчас осуществляется работа) для определения местоположения этой части в памяти, а второе поле виртуального адреса позволит найти нужную нам ячейку относительно найти нужную нам ячейку относительно найденного адреса. Программист может либо самостоятельно разбивать программу на фрагменты, либо автоматизировать эту задачу и возложить ее на систему программирования.

Сегментный способ организации виртуальной памяти

Первым среди разрывных методов распределения памяти был сегментный. Для этого метода программу необходимо разбивать на части и уже каждой такой части выделять физическую память. Естественным способом разбиение программы на части является разбиение ее на логические элементы – так называемые сегменты. В принципе каждый программой модуль (или их совокупность, если мы того пожелаем) может быть воспринят как отдельный сегмент, и вся программа тогда будет представлять собой множество сегментов. Каждый сегмент размещается в памяти как до определенной степени самостоятельная единица. Логически обращение к элементам программы в этом случае будет представляться как указание имени сегмента и смещения относительно начала этого, сегмента. Физически имя (или порядковый номер) сегмента будет соответствовать некоторому адресу, с которого этот сегмент начинается при его размещении в памяти, и смещение должна прибавляться к этому базовому адресу.

Преобразование имени сегмента в его порядковый номер осуществит система программирования, а операционная система будет размещать сегменты в память и для каждого сегмента получит информацию о его начале. Таким образом, виртуальный адрес для этого способа будет состоять из двух полей – номер сегмента и смещение относительно начала сегмента. Соответствующая иллюстрация приведена на рис. 2.7. На этом рисунке изображен случай обращения к ячейке, виртуальный адрес которой равен сегменту с номером 11 и смещением от начала этого сегмента, равным 612. Как мы видим, операционная система разместила данный сегмент в памяти, начиная с ячейки с номером 19700.

Каждый сегмент, размещаемый в памяти, имеет соответствующую информационную структуру, часто называемую дескриптором сегмента. Именно операционная система строит для каждого исполняемого процесса Соответствующую таблицу дескрипторов сегментов и при размещении каждого из сегментов в оперативной или внешней памяти в дескрипторе отмечает его текущее местоположение. Если сегмент задачи в данный момент находится в оперативной памяти, то об этом делается пометка в дескрипторе. Как правило, для этого используется «бит присутствия» P (present). В этом, случае в поле «адрес» диспетчер памяти записывает адрес физической памяти, с которого сегмент начинается, а в поле «длина сегмента» (limit) указывается количество адресуемых ячеек памяти. Это поле используется не только для того, чтобы размещать сегменты без наложения один на другой, но и для того, чтобы проконтролировать, не обращается ли код исполняющейся задачи за пределы текущего сегмента. В случае превышения длины сегмента в случае ошибок программирования мы можем говорить о нарушении адресации и с помощью введения специальных аппаратных средств генерировать сигналы прерываний, которые позволяют фиксировать (обнаруживать) такого рода ошибки.

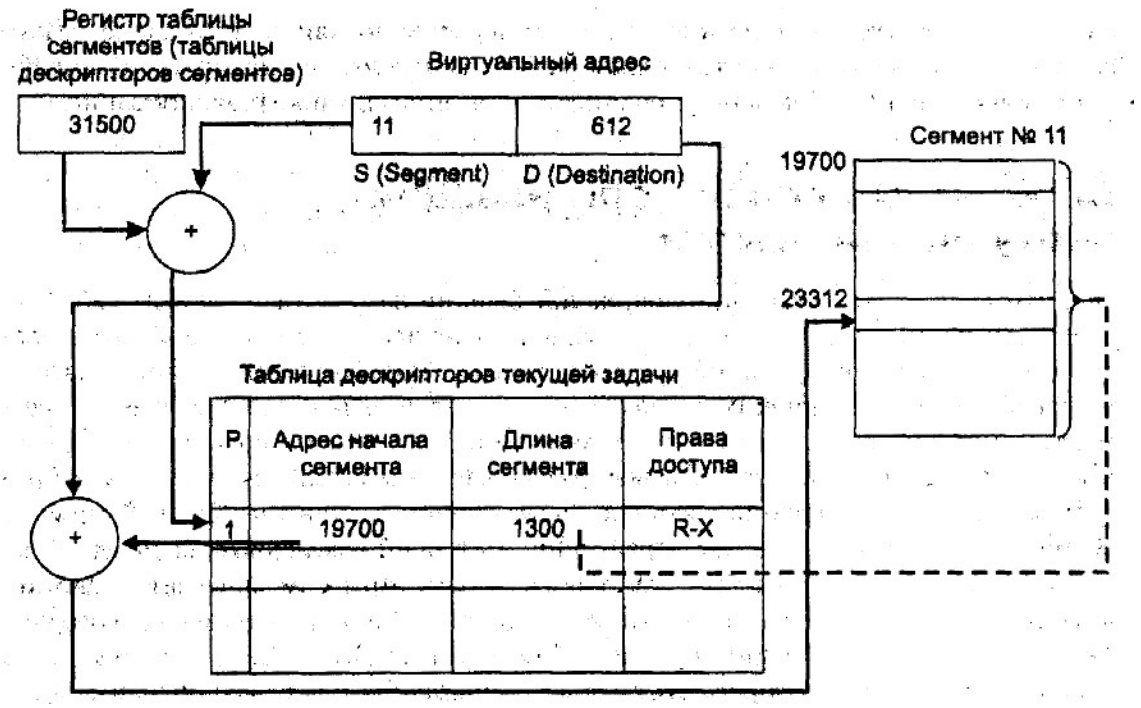

Рис. 2.7, Сегментный способ организации виртуальной памяти

Если бит present в дескрипторе указывает, что сейчас этот сегмент находится не в оперативной, а во внешней памяти (например, на винчестере), то названные поля, адреса и длины используются для указания адреса сегмента в координатах внешней памяти. Помимо информации о местоположении сегмента, в дескрипторе сегмента, как правило, содержатся данные о его типе (сегмент кода или сегмент данных), правах доступа к этому сегменту (можно или нельзя его модифицировать, предоставлять другой задаче), отметка об обращениях к данному сегменту (информация о том, как часто или как давно/недавно этот сегмент используется или не используется, на основании которой можно принять решение о том, чтобы предоставить место, занимаемое текущим сегментом, другому сегменту).

При передаче управления следующей задаче ОС должна занести в соответствующий регистр адрес таблицы дескрипторов сегментов этой задачи. Сама *таблица дескрипторов сегментов*, в свою очередь, также представляет собой сегмент данных, который обрабатывается диспетчером памяти операционной системы.

При таком подходе появляется возможность размещать в оперативной памяти не все сегменты задачи, а только те, с которыми в настоящий момент происходит работа. С одной стороны, становится возможным, чтобы общий объем виртуального адресного пространства задачи превосходил объем физической памяти компьютера, на котором эта задача будет выполняться. С другой стороны, даже если потребности в памяти не превосходят имеющуюся физическую память, появляется возможность размещать в памяти как можно больше задач. А увеличение *коэффициента мультипрограммирования* μ, как мы знаем, позволяет увеличить загрузку системы и более эффективно использовать ресурсы вычислительной системы. Очевидно, однако, что увеличивать количество задач можно только до определенного предела, ибо если в памяти не будет хватать места для часто используемых сегментов, то производительность системы резко упадет. Ведь сегмент, который сейчас находится вне оперативной памяти, для участия в вычислениях должен, перемещён в оперативную память. При этом есть свободное пространство, то необходимо всего лишь найти его во внешней памяти и загрузить в оперативную память. А если свободного места сейчас нет, то необходимо будет принять решение – на место какого из ныне присутствующих сегментов будет загружаться требуемый.

Итак, если требуемого сегмента в оперативной памяти нет, то возникает прерывание и управление передаётся через дескриптор памяти программе загрузки сегмента. Пока

происходит поиск сегмента во внешней памяти и загрузка его в оперативную, дескриптор памяти определяет подходящее для сегмента место. Возможно, что свободного места нет, и тогда принимается решение о выгрузке какого–нибудь сегмента и его перемещение во внешнюю память. Если при этом ещё остается время, то процессор передаётся другой готовой к выполнению Задаче. После загрузки необходимого сегмента процессор вновь передается задаче, вызвавшей прерывание из-за отсутствия сегмента. Всякий раз при считывании сегмента в оперативную память в таблице дескрипторов сегментов необходимо установить адрес начала сегмента и признак присутствия сегмента.

При поиске свободного места используется одна из вышеперечисленных дисциплин работы диспетчера памяти (применяются правила «первого подходящего» и «самого неподходящего» фрагментов). Если свободного фрагмента памяти достаточного объема сейчас нет, но, тем не менее, сумма этих свободных фрагментов превышает требования по памяти для нового сегмента, то в принципе может быть применено «уплотнение памяти», о котором мы уже говорили в подразделе «Разделы с фиксированными границами» при рассмотрении динамического способа разбиения памяти на разделы.

В идеальном случае размер сегмента должен быть достаточно малым, чтобы его можно было разместить в случайно освобождающихся фрагментах оперативной памяти, но достаточно большим, чтобы содержать логически законченную часть программы с тем, чтобы минимизировать межсегментные обращения.

Для решения проблемы замещения (определения того сегмента, который должен быть либо перемещен во внешнюю память, либо просто замещен новым) используются следующие дисциплины:

□ правило FIFO (first in – first out, что означает: «первый пришедший первым и выбывает»);

□ правило LRU (least recently used, что означает «последний из недавно использованных» или, иначе говоря, «дольше всего неиспользуемый»);

□ правило LFU (least frequently used, что означает: «используемый реже всех остальных»);

□ случайный (random) выбор сегмента.

Первая и последняя дисциплины являются самыми простыми в реализации, но они не учитывают, насколько часто используется тот или иной сегмент и, следовательно, диспетчер памяти может выгрузить иди расформировать тот сегмент, к которому, в самом ближайшем будущем будет обращение. Безусловно, достоверной информации о том, какой из сегментов потребуется в ближайшем будущем, в общем случае иметь нельзя, но вероятность ошибки для этих дисциплин многократно выше, чем у второй и третьей дисциплины, которые учитывают информацию об использовании сегментов.

Алгоритм FIFO ассоциирует с каждым сегментом время, когда он был помещен в память. Для замещения выбирается наиболее старый сегмент. Учет времени необязателен, когда все сегменты в памяти, связаны в FIFO – очередь и каждый помещаемый в память сегмент добавляется, в хвост этой очереди. Алгоритм учитывает только время нахождения сегмента в памяти, но не учитывает фактическое использование сегментов. Например, первые загруженные сегменты программы могут содержать переменные, используемые на протяжении работы всей программы. Это приводит к немедленному возвращению к только что замещенному сегменту.

Для реализации дисциплин LRU и LFU необходимо, чтобы процессор имел дополнительные аппаратные средства. Минимальные требования – достаточно, чтобы при обращении к дескриптору сегмента для получения физического адреса, с которого сегмент начинает располагаться в памяти, соответствующий бит обращения менял, свое значение (скажем, с нулевого, которое установила ОС в единичное). Тогда диспетчер памяти может время от времени просматривать таблицы дескрипторов исполняющихся задач и собирать для соответствующей обработки статистическую информацию об обращениях к сегментам. В результате можно составить список, упорядоченный либо по

длительности не использований (для дисциплины LRU), либо по частоте использования (для дисциплины LFU).

Важнейшей проблемой, которая возникает при организации мультипрограммного режима, является защита памяти. Для того, чтобы выполняющиеся приложения не смогли испортить саму ОС и другие вычислительные процессы, необходимо, чтобы доступ к таблицам сегментов с целью их модификации был обеспечен только для кода самой ОС. Для этого код ОС должен выполняться в некотором привилегированном режиме, из которого можно осуществлять манипуляции с дескрипторами сегментов, тогда как выход за пределы сегмента в обычной прикладной программе должен вызывать прерывание по защите памяти. Каждая прикладная задача должна иметь возможность обращаться только к своим собственным сегментам.

При использовании сегментного способа организации виртуальной памяти появляется несколько интересных возможностей. Во-первых, появляется возможность при загрузке программы на исполнение размещать ее в памяти не целиком, а «по мере необходимости». Действительно, поскольку в подавляющем большинстве случаев алгоритм, по которому работает код программы, является разветвленным, а не линейным, то в зависимости от исходных данных некоторые части программы, расположенные в самостоятельных сегментах, могут быть, и не задействованы; значит, их можно и не загружать в оперативную память.

Во-вторых, некоторые программные модули могут быть разделяемыми. Эти программные модули являются сегментами, и в этом случае относительно легко организовать доступ к таким сегментам. Сегмент с разделяемым кодом располагается в памяти в единственном экземпляре, а в нескольких таблицах дескрипторов сегментов исполняющихся задач будут находиться указатели на такие разделяемые сегменты.

Однако у сегментного способа распределения памяти есть и недостатки. Прежде всего, из рис. 2.7 видно, что для получения доступа к искомой, ячейке памяти необходимо потратить намного больше времени. Мы должны сначала найти и прочитать дескриптор сегмента, а уже потом, используя данные из него о местонахождении нужного нам сегмента, можем вычислить и конечный физический адрес. Для того чтобы уменьшить эти потери, используется кэширование – то есть те дескрипторы, с которыми мы имеем дело в данный момент, могут быть размещены в сверхоперативной памяти (специальных регистрах, размещаемых в процессоре).

Несмотря на то, что этот способ распределения памяти приводит к существенно меньшей фрагментации памяти, нежели способы с неразрывным распределением, фрагментация остается. Кроме этого, мы имеем большие потери памяти и процессорного времени на размещение и обработку дескрипторных таблиц. Ведь на каждую задачу необходимо иметь свою таблицу дескрипторов сегментов. А при определении физических адресов необходимо выполнять операции сложения.

Поэтому следующим способом разрывного размещения задач в памяти стал способ, при котором все фрагменты задачи одинакового размера и длины, кратной степени двойки, чтобы операции сложения можно было заменить операциями конкатенации (слияния). Это – страничный способ организации виртуальной памяти.

Примером использования сегментного способа организации виртуальной памяти является операционная система для ТЩ OS/2 первого поколения, которая была создана для процессора i80286. В этой ОС в полной мере использованы аппаратные средства микропроцессора, который специально проектировался для поддержки сегментного способа распределения памяти.

OS/2 v.l поддерживала распределение Памяти, при котором выделялись сегменты программы и сегменты данных. Система позволяла работать как с именованными, так и неименованными сегментами. Имена разделяемых сегментов данных имели ту же форму, что и имена файлов. Процессы получали доступ к именованным разделяемым сегментам, используя их имена в специальных системных вызовах. OS/2 v.l допускала разделение

программных сегментов приложений и подсистем, а также глобальных сегментов данных подсистем. Вообще, вся концепция систему OS/2 была построена на понятии разделения памяти: процессы почти всегда разделяют сегменты с другими процессами. В этом состояло существенное отличие от систем типа UNIX, которые обычно разделяют только реентерабельные Программные модули между процессами.

Сегменты, которые активно не использовались, могли выгружаться на жесткий диск. Система восстанавливала их, когда в этом возникала необходимость. Так как все области памяти, используемые сегментом, должны быть непрерывными, OS/2 перемещала в основной памяти сегменты, таким образом чтобы максимизировать объем свободной физической памяти. Такое размещение называется компрессией или перемещением сегментов (уплотнением памяти). Программные сегменты не выгружались, поскольку они могли просто перезагружаться с исходных дисков. Области в младших адресах физической памяти, которые использовались для запуска DOS–программ и кода самой OS/2, не участвовали в перемещении или подкачке. Кроме этого, система или прикладная программа, могли временно фиксировать сегмент в памяти с тем, чтобы гарантировать наличие буфера ввода/вывода физической памяти до тех пор, пока операция ввода/вывода не завершится.

Если в результате компрессии памяти не удавалось создать необходимое свободное пространство, то супервизор выполнял операции фонового плана для перекачки достаточного количества сегментов из физической памяти, чтобы дать возможность завершиться исходному запросу.

Механизм перекачки сегментов использовал файловую систему для перекачки данных из физической памяти и в нее. Ввиду того, что перекачка и сжатие влияют на производительность системы в целом, пользователь может сконфигурировать систему так, чтобы эти функции не выполнялись.

Было организовано в OS/2 и динамическое присоединение обслуживающих программ. Программы OS/2 используют команды удаленного вызова. Ссылки, генерируемые этими вызовами, определяются в момент загрузки самой программы или ее сегментов. Такое отсроченное определение ссылок называется динамическим присоединением. Загрузочный формат модуля OS/2 представляет совой расширение формата загрузочного модуля DOS. Он был расширен, чтобы поддерживать необходимое окружение для свопинга сегментов с динамическим присоединением. Динамическое присоединение уменьшает объем памяти для программ в OS/2, одновременно делая возможным перемещения подсистем, обслуживающих программ без необходимости повторного редактирования адресных ссылок к прикладным программам.

Страничный способ организации виртуальной памяти

Как мы уже сказали, при таком способе все фрагменты программы, на которые она разбивается (за исключением последней ее части), получаются одинаковыми. Одинаковыми полагаются и единицы памяти, которые мы предоставляем для извещения фрагментов программы. Эти одинаковые части называют страницами говорят, что память разбивается на физические страницы, а программа - на виртуальные страницы. Часть виртуальных страниц задачи размещается в оперативной памяти, а часть – во внешней. Обычно место внешней памяти, в качестве которой в абсолютном большинстве случаев выступают накопители на магнитных дисках (поскольку они относятся к быстродействующим устройствам с прямым доступом), называют файлом подкачки или страничным файлом (paging tile). Иногда этот файл называют swap-файлом, тем самым, подчеркивая, что записи этого файла – страницы – замещают друг друга в оперативной памяти. В некоторых ОС выгруженные страницы располагаются не в файле, а в специальном разделе дискового пространства. В UNIX – системах для этих целей выделяется специальный раздел, но кроме него могут быть использованы и файлы, выполняющие те же функции, если объема раздела недостаточно.

Разбиение всей оперативной памяти на страницы одинаковой величины, причем величину каждой страницы выбирается кратной степени двойки, приводит к тому, что вместо одномерного адресного пространства памяти можно говорить о двумерном. Первая координата адресного пространства – это номер страницы, а вторая координата – номер ячейки внутри выбранной страницы (его называют индексом). Таким образом, физический адрес определяется парой (Pp,i), а виртуальный адрес – парой (Pv ,i), где Pv – это номер виртуальной страницы, Pp – это номер физической страницы и i – это индекс ячейки внутри страницы. Количество битов, отводимое под индекс, определяет размер страницы, а количество битов, отводимое под номер виртуальной страницы, – объем возможной виртуальной памяти, который может пользоваться программа. Отображение, осуществляемое системой во время исполнения, сводится к отображению Pv в Pp и приписывании к полученному значению битов адреса, задаваемых величиной i. При этом нет необходимости ограничивать число виртуальных страниц числом физических, то есть не поместившиеся страницы можно разместить во внешней памяти, которая в данном случае служит расширением оперативной.

Для отображения, виртуального адресного пространства задачи на физическую память, как и в случае с сегментным способом организации, для каждой задачи необходимо иметь таблицу страниц для трансляции адресных пространств. Для описания каждой страницы диспетчер памяти ОС заводит соответствующий дескриптор, который отличается от дескриптора сегмента прежде всего тем, что в нем нет необходимости иметь поле длины– ведь все страницы имеют одинаковый размер. По номеру виртуальной страницы в таблице дескрипторов страниц текущей задачи находится соответствующий элемент (дескриптор). Если бит присутствия имеет единичное значение, значит, данная страница сейчас размещена в оперативной, а не во внешней памяти и мы в дескрипторе имеем номер физической страницы, отведенной под данную виртуальную. Если же бит присутствия равен нулю, то в дескрипторе мы будем иметь адрес виртуальной страницы, расположенной сейчас во внешней памяти. Таким образом и осуществляется трансляция виртуального адресного пространства на физическую память. Этот механизм трансляции проиллюстрирован на рис. 2.8.

Защита страничной памяти, как и в случае с сегментным механизмом, основана на контроле уровня доступа к каждой странице. Как правило, возможны следующие уровни доступа: только чтение; чтение и запись; только выполнение. В этом случае каждая страница снабжается соответствующим кодом уровня доступа. При трансформации логического адреса в физический сравнивается значение кода разрешенного уровня доступа с фактически требуемым. При их несовпадении работа программы прерывается.

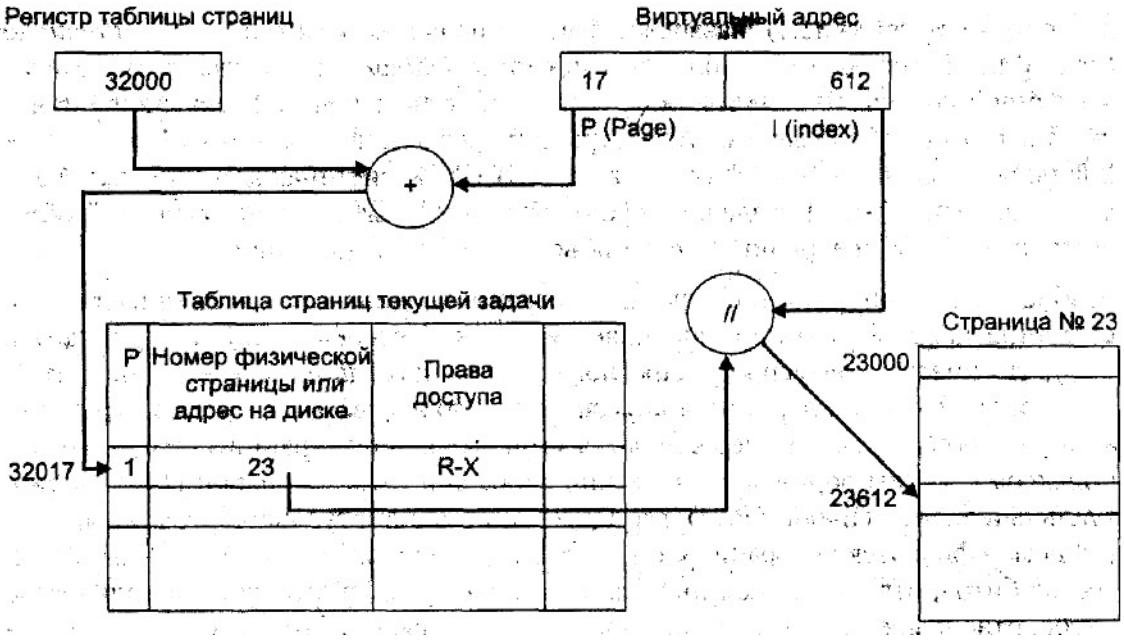

Рис. 2.8. Страничный способ организации виртуальной памяти

При обращении к виртуальной странице, не оказавшейся в данный момент в оперативной памяти, возникает прерывание, и управление передается диспетчеру памяти, который должен найти свободное место. Обычно предоставляется первая же свободная страница, рели свободной физической страницы нет, то диспетчер памяти по одной из вышеупомянутых дисциплин замещения (LRD, LFU, FIFO, random) определит страницу, подлежащую расформированию или сохранению во внешней памяти. На ее место он разместит ту новую виртуальную страницу, к которой было обращение из задачи, но ее не оказалась в оперативной памяти.

Напомним, что алгоритм выбирает для замещения ту страницу, на которую не было ссылки на протяжении наиболее длинного периода времени. Дисциплина LRU (least recently used) ассоциируют с каждой страницей время последнего ее использования. Для замещения выбирается та страница, которая дольше всех не использовалась.

Для использования, дисциплин LRU и LFU в процессоре должны быть соответствующие аппаратные средства. В дескрипторе страницы размещается бит обращения (подразумевается, что на рис. 2.8 этот бит расположен в последнем поле), и этот бит становится единичным при обращений к дескриптору.

Если объем физической памяти небольшой и даже часто требуемые страницы не удается разместить в оперативной памяти, то возникает так называемая «пробуксовка». Другими словами пробуксовка – это ситуаций, при которой загрузка нужной нам страницы вызывает перемещение во внешнюю память той страницы, с которой мы тоже активно работаем. Очевидно, что это очень плохое явление. Чтобы его не допускать, желательно увеличить объем оперативной памяти сейчас (это стало самым простым решением), уменьшить количество параллельно выполняемых задач, либо попробовать использовать более эффективные дисциплины замещения.

В абсолютном большинстве современных ОС используется дисциплина замещения страниц LRU как самая эффективная. Так, именно эта дисциплина «используется в OS/2 и Linux. Однако в такой ОС, как Windows NT, разработчики, желая сделать систему максимально независимоq от аппаратных возможностей процессора, пошли на отказ от этой дисциплины и применили правило FIFO. А для того, чтобы хоть, как-нибудь сгладить ее неэффективность, была введена «буферизация» тех страниц, которые должны быть записаны в файл подкачки на диск (В системе Windows NT файл с выгруженными вирттуальными страницами носит название PageFile.sys. Таких файлов может быть несколько. Их совокупная длина должна быть не меньше, чем объём физической памяти

комьютера плюс 11 Мб, необходимые для самой Windows NT. В системах Windows 2000 размер файла PageFile.sys намного превышает объем установленной физической памяти и часто достигает многих сотен мегабайт) или просто расформированы. Принцип буферирования прост. Прежде чем замещаемая страница действительно будет перемещена во внешнюю память или просто расформирована, она помечается как кандидат на выгрузку. Если в следующий раз произойдёт обращение к странице, находящейся в таком «буфере», то страница никуда не выгружается и уходит в конец списка FIFO. В противном случае страница действительно выгружается, а на ее место в «буфере» попадает следующий «кандидат». Величина такого «буфера» не может быть большой, поэтому эффективность страничной реализаций памяти Windows NT намного ниже, чем у вышеназванных ОС и явление пробуксовки начинается даже при существенно большей объеме оперативной памяти.

В ряде ОС с пакетным режимом работы для борьбы с пробуксовкой используется метод «рабочего множества». Рабочее множество – это множество «активных» страниц задачи за некоторый интервал, то есть тех страниц, к которым было обращение за этот интервал времени. Реально количество активных границ задачи (за интервал T) все время изменяется, и это естественно, но, тем не менее, для каждой задачи можно определить среднее количество ее активных страниц. Это среднее число активных страниц и есть рабочее множество задачи. Наблюдения за исполнением множества различных программ показали, что даже если T равно времени выполнения всей работы, то размер рабочего множества часто существенно меньше, чем общее число страниц программы. Таким образом, если ОС может определить рабочие множества исполняющихся задач, то для предотвращения пробуксовки достаточно планировать на выполнение только такое количество задач, чтобы сумма их рабочих множеств не превышала возможности системы.

Как и в случае с сегментным способом организации виртуальной памяти, страничный механизм приводит к тому, что без специальных аппаратных средств он будет существенно замедлять работу вычислительной системы. Поэтому обычно используется кэширование страничных дескрипторов. Наиболее эффективным способом кэширования является использование ассоциативного кэша. Именно такой ассоциативный кэш и создан в 32-разрядных микропроцессорах i80x86. Начиная с i80386, который поддерживает страничный способ распределения памяти, в этих микропроцессорах имеется кэш на 32 страничных дескриптора. Поскольку размер страницы в этих микропроцессорах равен 4 Кбайт, возможно быстрое обращение к 128 Кбайт памяти.

Итак, основным достоинством страничного способа распределения памяти является минимально возможная фрагментация. Поскольку на каждую задачу может приходиться по одной незаполненной странице, то становится, очевидно, что память можно использовать достаточно эффективно; этот метод организации виртуальной памяти был бы одним из самых лучших, если бы не два следующих обстоятельства.

Первое – это то, что страничная трансляция виртуальной памяти требует существенных накладных расходов. В самом деле, таблицы страниц нужно тоже размещать в памяти. Кроме этого, эти таблицы нужно обрабатывать; именно с ними работает диспетчер памяти.

Второй существенный недостаток страничной адресации заключается в том, что программы разбиваются на страницы случайно, без учета логических взаимосвязей, имеющихся в коде. Это приводит к тому, что межстраничные переходы, как правило, осуществляются чаще, нежели межсегментные, и к тому, что становится трудно организовать разделение программных модулей между выполняющимися процессами.

Для того чтобы избежать второго недостатка, постаравшись сохранить достоинства страничного способа, распределения памяти, был предложен еще один способ – сегментно-страничный. Правда, за счет дальнейшего увеличения накладных расходов на его реализацию.

Сегментно-страничный способ организации виртуальной памяти

Как и в сегментном способе распределения памяти, программа разбивается на логически запомненные части – сегменты– и виртуальный адрес содержит указание на номер соответствующего сегмента. Вторая составляющая виртуального адреса – смещение относительно начала сегмента – в свою очередь, может состоять из двух полей: виртуальной страницы и индекса. Другими словами, получается, что виртуальный адрес теперь состоит из трех компонентов: сегмент, страница, индекс. Получение физического адреса и извлечение из памяти необходимого элемента для этого способа представлено на рис. 2.9.

Из рисунка сразу видно, что этот способ организации виртуальной памяти вносит еще большую задержку доступа к памяти. Необходимо сначала вычислить адрес дескриптора сегмента и прочитать его, затем вычислить адрес элемента таблицы, страниц этого сегмента и извлечь из памяти необходимый элемент, и уже только после этого можно к номеру физической страницы приписать номер ячейки в странице (индекс). Задержка доступа к искомой ячейке, получается, по крайней мере, в три раза больше, чем при простой прямой адресации. Чтобы избежать этой неприятности, вводится кэширование, причем кэш, как правило, строится по ассоциативному принципу. Другими словами, просмотры двух таблиц в памяти могут, быть заменены одним обращением к ассоциативной памяти.

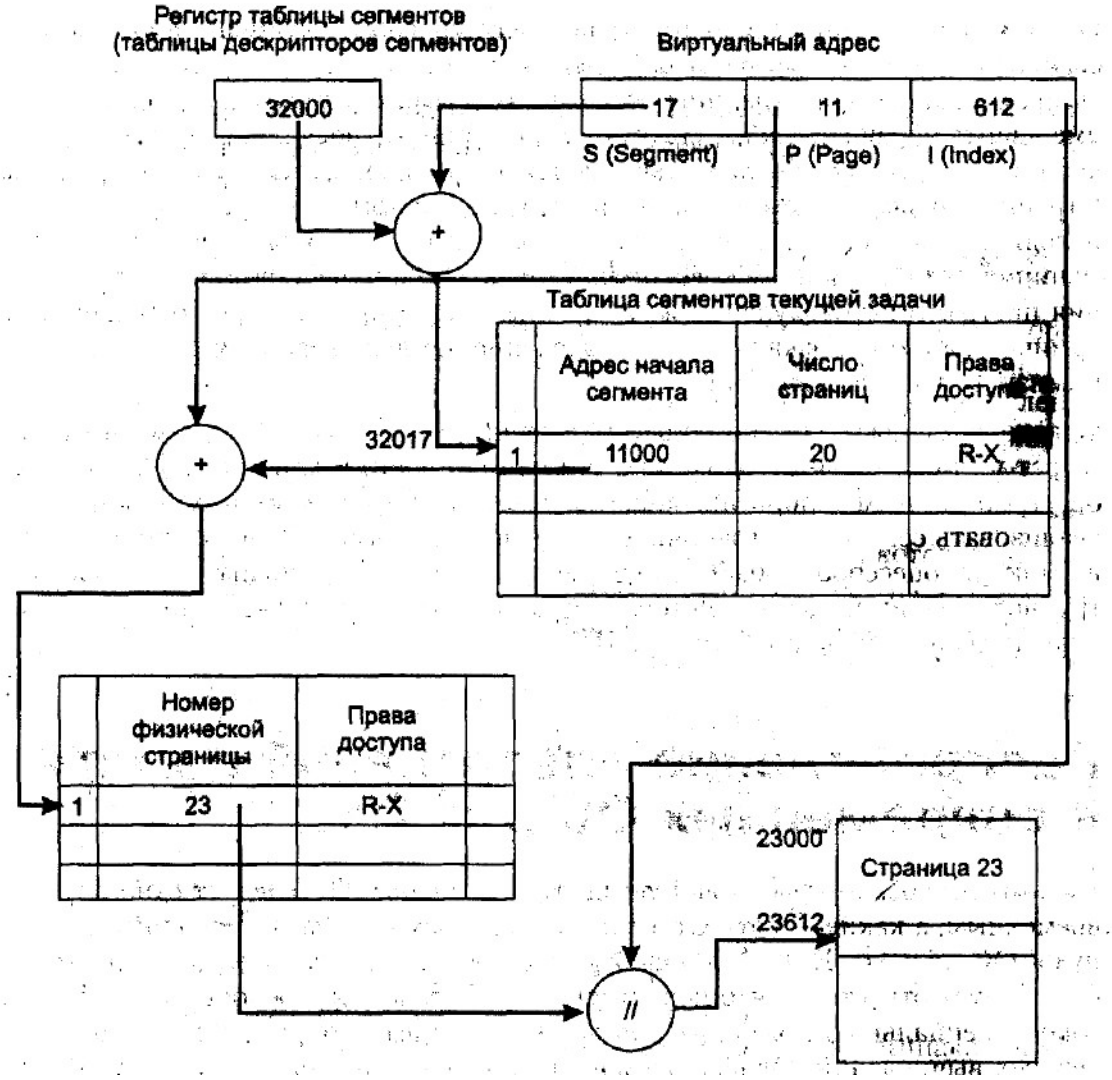

Рис. 2.9. Сегментно-страничный способ организации виртуальной памяти

Напомним, что принцип действия ассоциативного запоминающего устройства предполагает, что каждой ячейке памяти такого устройства ставится в соответствие

ячейка, в которой записывается некий ключ (признак, адрес) позволяющий однозначно идентифицировать содержимое ячейки памяти. Сопутствующую ячейку с информацией, позволяющей идентифицировать основные данные, обычно называют *полем тега*. Просмотр полей тега всех ячеек ассоциативного устройства памяти осуществляется одновременно, то есть в каждой ячейке тега есть необходимая логика, позволяющая посредством побитовой конъюнкции найти данные по их признаку за одно обращение к памяти (если они там, конечно, присутствуют). Часто поле тегов называют аргументом, а поле с данными – функцией. В качестве аргумента при доступе к ассоциативной памяти выступают номер сегмента и номер виртуальной страницы, а в качестве функции от этих аргументов получаем номер физической страницы. Остается приписать номер ячейки, в странице к полученному номеру, и мы получаем искомую команду или операнд.

Оценим достоинства сегментно-страничного способа. Разбиение программы на сегменты позволяет размещать сегменты в памяти целиком. Сегменты разбиты на страницы, все страницы сегмента загружаются в память. Это позволяет уменьшить обращения к отсутствующим страницам, поскольку вероятность выхода за пределы сегмента меньше вероятности выхода за пределы страницы. Страницы исполняемого сегмента находятся в памяти, но при этом они могут находиться не рядом друг с другом, а «россыпью», поскольку диспетчер памяти манипулирует страницами. Наличие сегментов облегчает реализацию разделения программных модулей между параллельными процессами. Возможна и динамическая компоновка задачи. А выделение памяти страницами позволяет минимизировать фрагментацию.

Однако, поскольку этот способ распределения памяти требует очень значительных затрат вычислительных ресурсов и его не так просто реализовать, используется он, редко, причем в дорогих, мощных вычислительных системах. Возможность реализовать сегментно–страничное распределение памяти заложена и в семейство микропроцессоров i80x86, однако вследствие слабой аппаратной поддержки, трудностей при создании систем программирования и операционной системы, практически он не используется в ПК.

Распределение оперативной памяти в современных ОС для ПК

Первый вопрос, который хочется задать, – это какие ОС следует относить к современным, а какие? Стоит ли в наше время изучать такую «несовременную» ОС, как MS-DOS (широко известно, что было много версий ОС, которые мы, упрощая ситуацию, относим к MS-DOS. MS-DOS – это вариант фирмы Microsoft реализации дисковой операционной системы. Фирма Microsoft прекратила разработку подобных систем, и последней их реализацией была MS-DOS 6.22. Были (и есть ныне) реализации такого рода систем и от других разработчиков. Поскольку у MS-DOS 6.22 существуют известные проблемы с 2000 годом, большой популярностью пользуется PC-DOS 7.0 от IBM)? С нашей точки зрения, прежде всего к современным ОС следует отнести те, что используют аппаратные возможности микропроцессоров, специально заложенные для организации высокопроизводительных и надежных вычислений. Однако эти ОС, как правило, очень сложны и громоздки. Они занимают большое дисковое пространство, требуют и большого объема оперативной памяти. Поэтому для решения некоторого класса задач вполне подходят и системы, использующие микропроцессоры в так называемом реальном режиме работы.

В последние годы можно встретить студентов, обучающихся специальностям, непосредственно связанным с вычислительной техникой, которые совсем не знают DOS-систем. Скорее всего, это является доказательством того, что ,такие ОС уже не являются современными. Однако достаточно часто для обслуживания компьютера необходимо выполнить простейшие программы – утилиты. Эти программы были созданы для DOS, они не требуют больших ресурсов, для их функционирования достаточно запустить MS-

DOS или аналогичную простую ОС. Однако без выполнения этих программ невозможно порой установить или загрузить иные ОС (хоть и современные, но очень сложные и громоздкие). Поэтому мы считаем правильным сделать хотя бы первичное, пусть не очень глубокое, ознакомление с MS-DOS.

Распределение оперативной памяти в MS-DOS

Как известно, MS-DOS – это однопрограммная ОС. В ней, конечно, можно организовать запуск резидентных или TSR-задач (terminate and stay resident – резидентная в памяти программа, которая благодаря изменениям в таблице векторов прерываний позволяет перехватывать прерывания и в случае обращения к ней выполнять необходимые нам действия. В целом она предназначена для выполнения только одного вычислительного процесса, поэтому распределение памяти в ней построено по самой простой схеме, которую мы уже рассматривали в разделе «Простое непрерывное распределение и распределение с перекрытием (оверлейные структуры)». Здесь мы лишь уточним некоторые характерные детали.

В IBM PG использовался 16-разрядный микропроцессор i8088, который за счет введения сегментного способа адресации позволял адресоваться к памяти объемом до 1 Мбайт. В последующих ПК (IBM PC AT, AT386 я др.) было принято решение поддерживать совместимость с первыми, поэтому при работе с DOS прежде всего рассматривают первый мегабайт. Вся эта память разделялась на несколько областей, что проиллюстрировано на рис. 2.10. На этом рисунке изображено, что памяти может быть и больше, чем 1 Мбайт.

Если не вдаваться в детали, можно сказать, что в состав MS-DOS входят следующие основные компоненты:

□ Базовая подсистема ввода/вывода – BIOS (base input-output system), включающая в себя помимо программы тестирования ПК (POST– (power on self test) – программа самотестирования при включении компьютера) обработчики прерываний, (драйверы), расположенные в постоянном запоминающей устройстве. В конечном итоге, почти все остальные модули MS-DOS обращаются к BIOS. Если и не напрямую, то через модули более высокого уровня иерархии.

□ Модуль расширения BIOS – файл IO.SYS (в других DOS-системах он может называться иначе, например, IBMIO.COM).

□ Основной, базовый модуль обработки прерываний DOS – файл MSDOS.SYS. Именно тот модуль в основном реализует работу с файловой системой. (В PC-DOS аналогичный по назначению файл называется IBMDOS.COM).

□ Командный процессор (интерпретатор команд) – файл COMMAND.COM.

□ Утилиты и драйверы, расширяющие возможности системы.

□ Программа загрузки MS-DOS – загрузочная запись (boot record), расположенная на дискете.

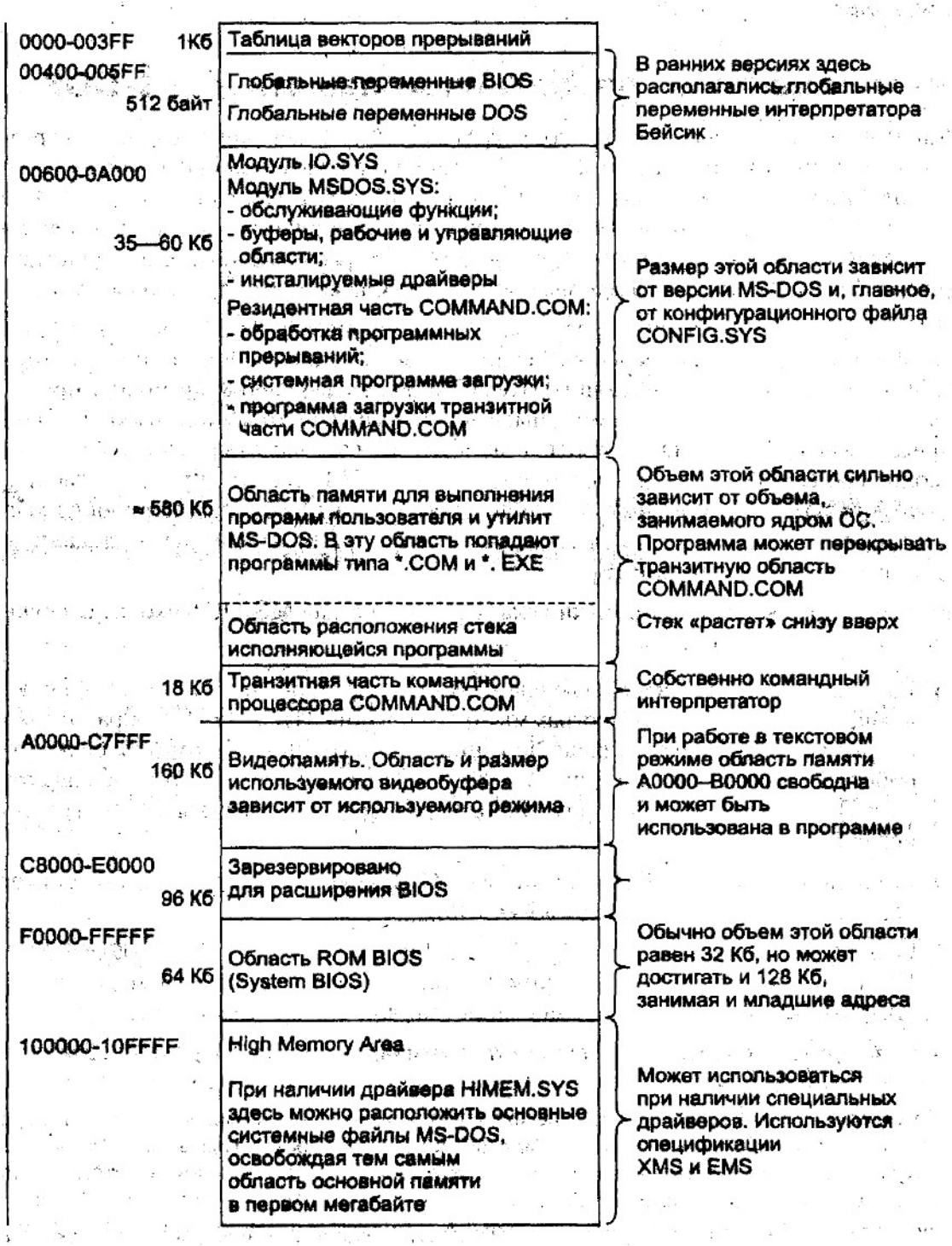

Рис. 2.10. Распределение оперативной памяти в MS-DOS

Вся память в соответствии с архитектурой IBM PC условно может быть разбита на три части.

В самых младших адресах памяти (1-е 1024 ячейки) размещается таблица векторов прерываний. Это связано с аппаратной реализацией процессора i8088, на котором была реализована ПК. В последующих процессорах (начиная с i80286) адрес таблицы прерываний определяется через содержимое соответствующего регистра, но для обеспечения полной совместимости с первым процессором при включении или аппаратном сбросе в этот регистр заносятся, нули. При желании, однако, в случае использования современных микропроцессоров i80x86 можно разместить векторы прерываний и в другой области.

Вторая часть памяти отводится для размещений программных модулей самой MS-DOS и для программ пользователя. Рассмотрим их размещение чуть ниже. Здесь, однако, заметим, что эта область памяти называется Conventional Memory (основная, стандартная память).

Наконец, третья часть адресного пространства отведена для постоянных запоминающих устройств и функционирования некоторых устройств ввода/вывода. Эта область памяти получила название UMA (upper memory areas – область верхней памяти).

В младших адресах основной памяти размещается то, что можно назвать ядром этой ОС – системные переменные, основные программные модули, блоки данных буферирования операций ввода/вывода. Для управления устройствами, драйверы которых не входят в базовую подсистему ввода/вывода, загружаются так называемые *загружаемые* (или *инсталлируемые*) драйверы. Перечень инсталлируемых драйверов определяется специальным функциональным файлом CONFIG.SYS. Загрузки расширения BIOS–файла IO.SYS–последний загрузив модуль MSDOS.SYS) считывает файл CONFIG.SYS и уже в соответствии с ним подгружаем в память необходимые драйверы. Кстати в конфигурационном файле CONFIG.SYS могут иметься и операторы указывающие количество буферов, отводимых для ускорения операций ввода/вывода, и на на кол-во файлов которые, могут обрабатываться (для работы с файлами необходимо зарезервировать, место в памяти для хранения управляющих структур, с помощью которых выполняются операции с записями файла). В случае использования микропроцессоров i80x86 и наличия в памяти драйвера HIMEM.SYS модули IO.SYS и MSDOS.SYS могут быть размещены за первого мегабайта в области, которая получила название HMA (high memory area).

Память с адресами, большими, чем 10FFFFh, может быть использована в D0S-программах при выполнении на микропроцессорах, имеющих такую возможность. Так, например, микропроцессор i80286 имел 24-разрядную шину адреса, а i80386 – уже 32-разрядную шину адреса. Но для этого, с помощью специальных драйверов необходимо переключать процессор в другой режим работы, при котором он сможет использовать адреса выше 10FFFFh. Широкое распространение получили две основные спецификации: XMS (extended memory specification) и EMS (expanded memory specification). Поскольку основные утилиты, необходимые для обслуживания ПК, как правило, не используют эти спецификации, мы не будем здесь их рассматривать.

Остальные программные модули MS-DOS (в принципе большинство из них является утилитами) оформлены как обычные исполняемые файлы. В основном они являются транзитными модулями, то есть загружаются в память только на время своей работы, хотя среди них имеются и TSR-программы.

Для того чтобы предоставить больше памяти программам пользователя, в MS-DOS применено то же решение, что и во многих других простейших ОС – командный процессор COMMAND.COM сделан состоящим из двух частей. Первая часть является резидентной, она размещается в области ядра. Вторая часть – транзитная; она размещается в области старших адресов раздела памяти, выделяемой для программ пользователя. И если программа пользователя перекрывает собой область, в которой была расположена транзитная часть командного процессора, то последний при необходимости восстанавливает в памяти свою транзитную часть, поскольку после выполнения программы она возвращает управление резидентной части COMMAND.COM.

Поскольку размер основной памяти (conventional memory) относительно небольшой, то очень часто системы программирования реализуют оверлейные структуры. Для этого в MS-DOS есть специальные вызовы.

Распределение оперативной памяти в Microsoft Windows 95/98

С точки зрения базовой архитектуры ОС Windows 95/98 они обе являются 32-разрядными, многопотоковыми ОС с вытесняющей многозадачностью. Основной пользовательский интерфейс этих ОС – графический.

Для своей загрузки, они используют операционную систему MS-DOS 7.X (MS-DOS' 98), и в случае если в файле MSDOS.SYS в файле MS-DOS.SYS в секции [Options] прописано BootGUI = 0, то процессор работает в обычном реальном режиме. Распределение памяти в MS-DOS 7.X. такое же как и в предыдущих версиях DOS. Однако при загрузке GUI интерфейса перед загрузкой ядра Windows 95/98 процессор переключается в защищенный режим работы и зачинает распределять память уже с помощью страничного механизма.

Использование так называемой плоской модели памяти, при которой все возможнее сегменты, которые может использовать Программист, совпадают друг с другом и имеют максимально возможный размер, определяемый системными соглашениями данной ОС, приводит к тому, что с точки зрения программиста память получается неструктурированной. За счет представления адреса как пары (P, i) память можно трактовать и как двумерную, то есть «плоскую», но при этом ее можно, трактовать и как линейную, и это существенно облегчает, создание системного программного обеспечения и прикладных программ с помощью соответствующих систем программирования.

Таким образом, в системе, фактически действует только страничный механизм преобразования виртуальных адресов в физические. Программы используют классическую «Small» (малую) модель памяти. Каждая прикладная программа определяется 32-битными адресами, в которых сегмент кода имеет то же значение, что и сегменты данных. Единственный сегмент программы отображается непосредственно в область виртуального линейного адресного пространства, который, в свою очередь, состоит из 4 килобайтных страниц. Каждая страница может располагаться где угодно в оперативной памяти (естественна, в том месте, куда ее разместит диспетчер памяти, который сам находится в невыгружаемой области) или может быть перемещена на диск, если не запрещено использовать страничный файл.

Младшие адреса виртуального адресного пространства совместно используются всеми процессами. Это сделано для обеспечения совместимости с драйверами устройств реального режима, резидентными программами и некоторыми 16-разрядными программами Windows. Безусловно, это плохое решение с точки зрения надёжности, поскольку оно приводит к тому, что любой процесс может непреднамеренно (или же, наоборот, специально) испортить компоненты находящиеся в этих адресах.

В Windows 95/98 каждая 32-разрядная прикладная программа выполняется в своем собственном адресном пространстве, но все они используют совместно один и тот же 32-разрядный системный код. Доступ к чужим, адресным пространствам в принципе возможен. Другими словами, виртуальные адресные пространства не используют всех аппаратных средств защиты, заложенных в микропроцессор. В результате неправильно написанная 32-разрадная прикладная программа может привести к аварийному сбою всей системы. Все 16-битовые прикладные программы Windows разделяют общее адресное пространство, поэтому они так же уязвимы; друг перед другом, как и в среде Windows 3.X.

Системный код Windows 95 размещается выше границы 2 Гбайт. В пространстве с отметками 2 и 3 Гбайт находятся системные библиотеки DLL (динамически загружаемый системный модуль), используемые несколькими программами. Заметим, что в 32-битовых микропроцессорах семейства i80x86 имеются 4-ре уровня защиты, именуемые кольцами с номерами от 0 до 3. Кольцо с номером 0 является наиболее привилегированным, то есть максимально защищенным. Компоненты системы Windows 95, относящиеся к кольцу 0, отображаются на виртуальное адресное пространство между 3 и 4 Гбайт. К этим компонентам относятся собственно ядро Windows, подсистема управления виртуальными машинами, модули файловой системы и виртуальные драйверы (VxD). Область памяти

между 2 И 4 Гбайт адресного пространства каждой 32-разрядной прикладной программы совместно используется всеми 32-разрядными прикладными программами. Такая организация позволяет обслуживать вызовы API непосредственно в адресном пространстве прикладной программы и ограничивает размер рабочего множества. Однако за это приходится расплачиваться снижением надежности. Ничто не может помешать программе, содержащей ошибку, произвести запись в адреса, принадлежащие системным DLL, и вызвать крах всей системы.

В области между 2 и 3 Гбайт также находятся все запускаемые 16-разрядные прикладные программы Windows. С целью обеспечения совместимости эти программы выполняются в совместно используемом адресном, пространстве, где они могут испортить друг друга так же, как и в Windows 3.x.

Адреса памяти ниже 4 Мбайт также отображаются в адресное пространство каждой прикладной программы и совместно используются всеми процессами. Благодаря этому становится возможной совместимость с существующими драйверами реального режима, которым, необходим доступ к этим адресам. Это делает еще одну область памяти незащищенной от случайной записи. К самым нижним 64 Кбайт этого адресного пространства 32-разрядньде прикладные программы обращаться не могут, что дает возможность перехватывать неверные указатели, но 16-разрядные программы, которые, возможно, содержат ошибки, могут записывать туда данные.

Вышеизложенную модель распределения памяти, можно проиллюстрировать с помощью рис. 2.11.

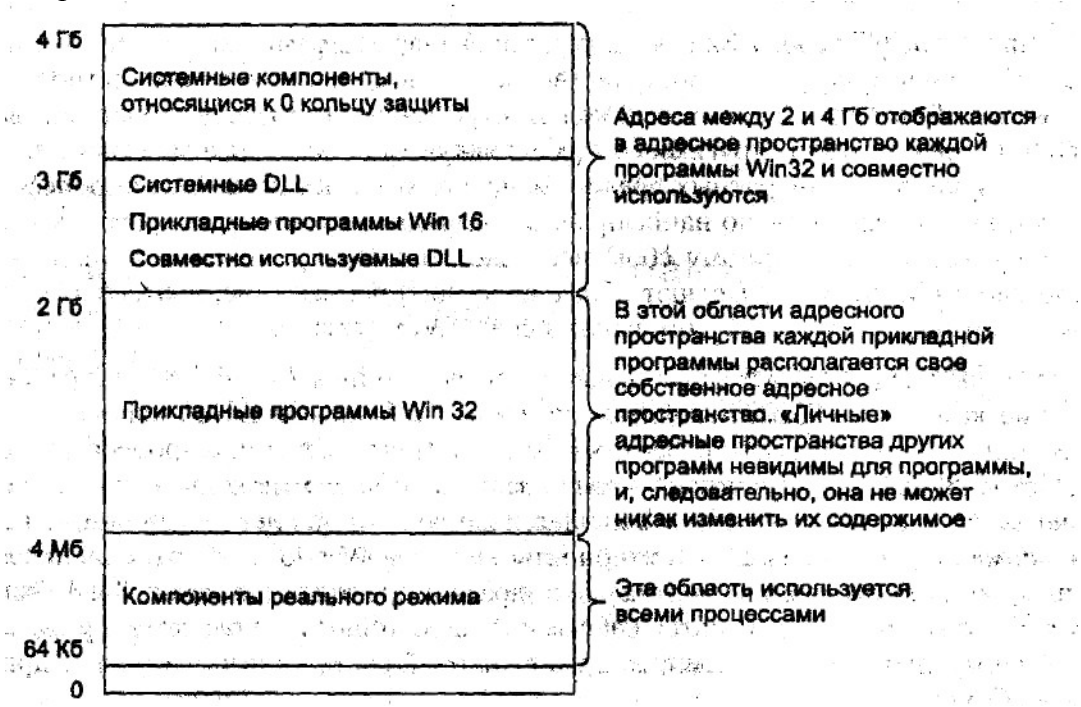

Рис. 2.11. Модель памяти ОС Windows 95/98

Минимально допустимый объем оперативной памяти, начиная с которого ОС Windows 95 может функционировать, равен 4 Мбайт, однако при таком объеме пробуксовка столь велика, что фактически работать нельзя. Страничный файл, с помощью которого реализуется механизм виртуальной памяти, по умолчанию располагается в каталоге самой Windows и имеет переменный размер. Система отслеживает его длину, увеличивая или сокращая этот файл при необходимости. Вместе с фрагментацией файла подкачки это приводит к тому, что быстродействие системы становится меньше, чем, если бы файл был фиксированного размера и располагался в смежных кластерах (был бы дефрагментирован). Сделать файл подкачки заданного размера можно либо через специально разработанный для этого апплет (Панель управления ► Система ►

Быстродействие ► Файловая система), либо просто прописав в файле SYSTEM.INI в секции [386Enh] строчки с указанием диска и имени этого файла, например:

```
PagingDrive=C:
PagingFile=C:\Pagefile.sys
MinPagingFileSize=65536
MaxPagingFileSize=262l44
```

Первая и вторая строчки указывают имя страничного файла и его размещение, а две последних – начальный и предельный размер страничного файла (значения указываются в килобайтах). Для определения необходимого минимального размера этого файла можно рекомендовать запустить программу SysMon (системный монитор) и, выбрав в качестве наблюдаемых параметров размер файла подкачки, и объем свободной памяти, оценить потребности в памяти, запуская те приложения, с которыми чаще всего приходится работать.

Распределение оперативной памяти в Microsoft Windows NT

В операционных системах Windows NT тоже используется плоская модель памяти. Заметим, что Windows NT 4.0 server практически не отличается от Windows NT 4.0. workstation разница лишь в наличии у сервера некоторых дополнительных служб, дополнительных утилит для управления доменом и несколько иных значений в настройках системного реестра. Однако схема распределения возможного виртуального адресного пространства в системах Windows NT разительно отличается от модели памяти Windows 95/98. Прежде всего, в отличие от Windows 95/98 в гораздо большей степени используется ряд серьезных аппаратных средств защиты, имеющихся в микропроцессорах, а также применено принципиально другое логическое распределение адресного пространства.

Во-первых, все системные программные модули находятся в своих собственных виртуальных адресных пространствах, и доступ к ним со стороны прикладных программ невозможен. Ядро системы и несколько драйверов работают в нулевом кольце защиты в отдельном адресном пространстве.

Во-вторых, остальные программные модули самой операционной системы, которые выступают как серверные процессы по отношению к прикладным программам (клиентам), функционируют также в своем собственном системном виртуальном адресном пространстве, невидимом для прикладных процессов. Логическое распределение адресных пространств приведено на рис. 2.12.

Прикладным программам выделяется 2 Гбайт локального (собственного) линейного (неструктурированного) адресного пространства от границы 64 Кбайт до 2 Гбайт (первые 64 Кбайт полностью недоступны). Прикладные программы изолированы друг от друга, хотя могут общаться через буфер обмена (clipboard), механизмы DDE (механизм динамического обмена данных) и OLE (механизм связи и внедрения объектов).

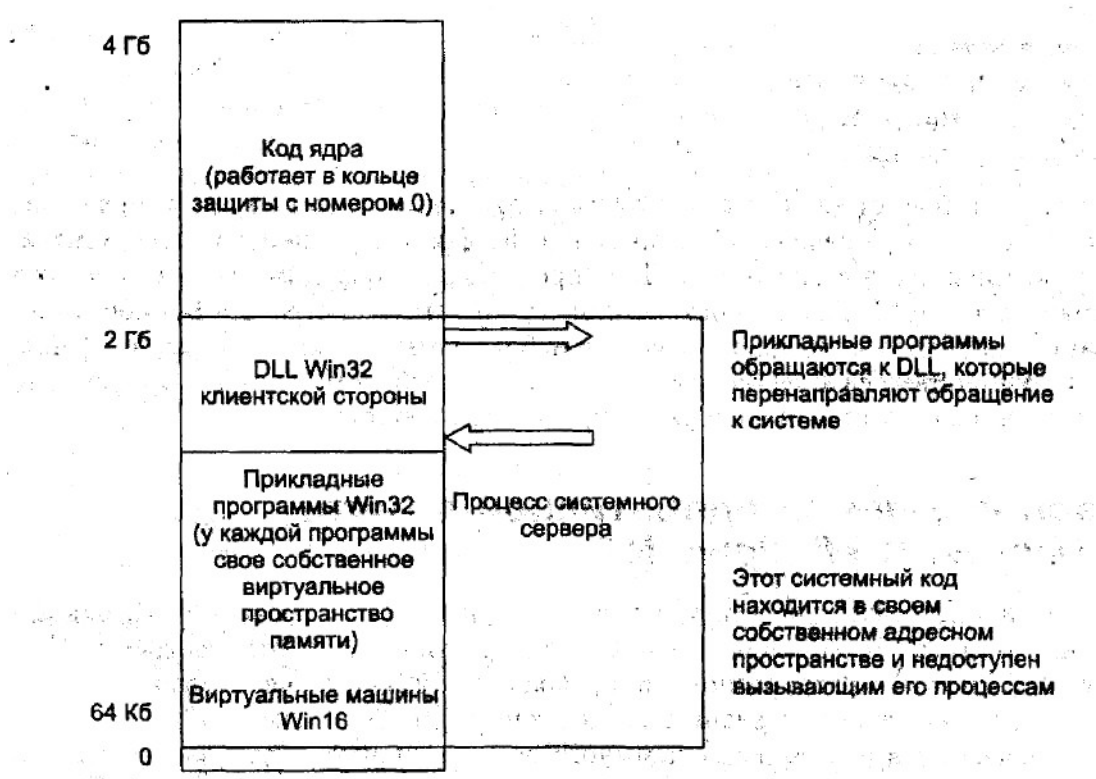

Рис. 2.12. Модель распределения виртуальной памяти в Windows NT

В верхней части каждой 2-гигабайтной области прикладной программы размещен код системных DDL кольца 3, который выполняет перенаправление вызовов в совершенно изолированное адресное пространство, где содержится уже собственно системный код. Этот системный код, выступающий как сервер-процесс (server process) проверяет значения параметров, исполняет запрошенную функцию и пересылает результата назад в адресное пространство прикладной программы. Хотя сервер-процесс сам по себе остается процессом прикладного уровня, он полностью защищен от вызывающей его прикладной программы и изолирован от нее.

Между отметками 2 и 4 Гбайт расположёны низкоуровневые системны компоненты Windows NT кольца 0, в том числе ядро, планировщик потоков и диспетчер виртуальной памяти. Системные страницы в этой области наделены привилегиями супервизора, которые задаются физическими схемами кольцевой защиты процессора. Это делает низкоуровневый системный код невидимым и недоступным для записи для программ прикладного уровня, но приводит к падению производительности во время переходов между кольцами.

Для 16-разрядных прикладных Windows-программ ОС Windows NT реализует сеансы Windows on Windows (WOW). В отличие от Windows 95/98 ОС Windows NT дает возможность выполнять 16-разрядные программы Windows индивидуально в собственных пространствах памяти или совместно в разделяемом адресном пространстве. Почти во всех случаях 16- и 32-разрядные прикладные программы Windows могут свободно взаимодействовать, используя OLE, независимо от того выполняются они в отдельной или общей памяти. Собственные прикладные программы и сеансы WOW выполняются в режиме вытесняющей многозадачности, основанной на управлении отдельными потоками. Множественные разрядные прикладные программы Windows в одном сеансе WOW выполняются в соответствии с кооперативной моделью многозадачности. Windows NT может также выполнять в многозадачном режиме несколько сеансов DOS. Поскольку Windows NT имеет полностью 32-разрядную архитектуру, не существует теоретических ограничений на ресурсы GDI (интерфейс графических устройств) и USER.

При запуске приложения создается процесс со своей информационной структурой. В рамках процесса запускается задача. При необходимости этот тред (задача) может

запустить множество других тредов (задач), которые будут выполняться параллельно в рамках одного процесса. Очевидно, что множество запущенных процессов также выполняются параллельно и каждый из процессов может представлять из себя мультизадачное приложение. Задачи (треды) в рамках одного процесса выполняются, в едином виртуальном адресном пространстве, а процессы выполняются в различных виртуальных адресных пространствах. Отображение различных виртуальных адресных пространств исполняющихся процессов на физическую память реализует сама ОС; именно корректное выполнение этой задачи гарантирует изоляцию приложений от невмешательства процессов. Для обеспечения взаимодействия между выполняющимися приложениями и между приложениями и кодом самой операционной системы используются соответствующие механизмы защиты памяти, поддерживаемые аппаратурой микропроцессора.

Процессами выделения памяти, ее резервирования, освобождения и подкачки управляет, диспетчер виртуальной памяти, Windows NT. (Windows NT virtual memory manager, VMM). В своей работе этот компонент реализует сложную стратегию учета требований к коду и данным процесса для минимизации доступа к диску, поскольку реализация виртуальной памяти часто приводит к большому количеству дисковых операций.

Каждая виртуальная страница памяти, отображаемая на физическую страницу, переносится в так называемый *страничный фрейм* (page frame). Прежде чем код или данные можно будет переместить с диска в памяти, диспетчер виртуальной памяти (модуль VMM). Должен найти или создать свободный страничный фрейм или фрейм, заполненный нулями. Заметим, что заполнение страниц нулями представляет собой одно из требований стандарта на системы безопасности уровня С2, принятого правительством США. Страничные фреймы должны заполняться нулями для того, чтобы исключить возможность использования их предыдущего содержимого другими процессами. Чтобы фрейм можно было освободить, необходимо, скопировать на диск изменения в его странице данных, и только после этого фрейм можно будет повторно использовать, Программы, как правило, не меняют страницы кода. Страницы кода, в которые программы не внесли изменений, можно удалить. Диспетчер виртуальной памяти может быстро и относительно легко удовлетворить программные прерывания типа «ошибка страницы» (page fault). Что касается аппаратных прерываний типа «ошибка страницы», то они приводят к подкачке (paging), которая снижает производительность системы. Мы уже говорили о том, что в Windows NT к большому сожалению, выбрана дисциплина FIFO для замещения страниц, а не более эффективные дисциплины LRU и LFU.

Когда процесс использует код или данные, находящиеся в физической памяти, система резервирует место для этой страницы в файле подкачки Pagefile.sys на диске. Это делается с расчетом на тот случай, что данные потребуется выгрузить на диск. Файл Pagefile.sys представляет собой зарезервированный блок дискового пространства, который используется для выгрузки страниц, помеченных как «грязные» при необходимости освобождения физической памяти. Заметим, что этот файл может быть как непрерывным; так и фрагментированным; он может быть расположен на системном диске либо на любом другом и даже на нескольких дисках. Размер этого страничного файла ограничивает объем, данных, которые могут храниться во внешней памяти при использовании механизмов виртуальной памяти. По умолчанию размер файла подкачки устанавливается равным объему физической памяти плюс 12 Мбайт, однако пользователь имеет возможность изменить его размер по своему усмотрению. Проблема нехватки виртуальной памяти часто может быть решена за счет увеличения размера файла подкачки.

В системах Windows NT 4.0 объекты, создаваемые и используемые приложениями и операционной системой, хранятся в так называемых *пулах памяти* (memory pools). Доступ к этим пулам может быть получен только в привилегированном режиме работы процессора, в котором работают компоненты операционной системы. Поэтому для того,

чтобы объекты, хранящиеся в пулах, стали видимы тредам приложений, эти треды должны переключиться в привилегированный режим.

Перемещаемый или нерезидентный пул (paged pool) содержит объекты, которые могут быть при необходимости выгружены на диск. *Неперемещаемый или резидентный пул* (nonpaged pool) содержит объекты, которые должны постоянно находиться в памяти. В частности, к такого рода объектам относятся структуры данных, используемые процедурами обработки прерываний, а также структуры, используемые для предотвращения конфликтов в мультипроцессорных системах.

Исходный размер пулов определяется объемом физической памяти, доступной Windows NT. Впоследствии размер пула устанавливается динамически и, в зависимости от работающих в системе приложений и сервисов, будет изменяться в широком диапазоне.

Вся виртуальная память в Windows NT подразделяется на классы: зарезервированную (reserved), выделенную (committed) и доступную (available).

– Зарезервированная память представляет собой набор непрерывных адресов, которое диспетчер виртуальной памяти (VMM) выделяет для процесса; но не учитывает в общей квоте памяти процесса до тех пор, пока она не будет фактически использована. Когда процессу требуется выполнить запись в память, ему выделяется нужный объем из зарезервированной памяти. Если процессу потребуется больший объем памяти, то дополнительная память может быть одновременно зарезервирована и использована, если в системе имеется доступная память.

– Память выделена, если диспетчер VMM резервирует для нее место в файле Pagefile.sys на случай, когда потребуется выгрузить содержимое памяти на диск. Объем выделенной памяти процесса характеризует фактически потребляемый им объем памяти. Выделенная память ограничивается размером файла подкачки. Предельный объем выделенной памяти в системе (commit limit) определяется тем, какой объем памяти можно выделить процессам без увеличения размеров файла подкачки. Если в системе имеется достаточный объем дискового пространства, то файл подкачки может быть увеличен и, тем самым, будет расширен предельный объем выделенной памяти.

Вся память, которая не является ни выделенной, ни зарезервированной, является доступной. К доступной относится свободная память, обнуленная память (освобожденная и заполненная нулями), а также память, находящаяся в списке ожидания (standby list), которая была удалена из рабочего набора процесса, но может быть затребована вновь.

3. Управление вводом/выводом и файловые системы

Необходимость обеспечить программам возможность осуществлять обмен данными с внешними устройствами и при этом не включать в каждую двоичную программу соответствующий двоичный код, осуществляющий собственно управление устройствами ввода/вывода, привела разработчиков к созданию системного программного обеспечения и, в частности, самих операционных систем. Программирование задач управления вводом/выводом является наиболее сложным и трудоемким, требующим очень высокой квалификации. Поэтому код, позволяющий осуществлять операции ввода/вывода, стали оформлять в виде системных библиотечных процедур; потом его стали включать не в системы программирования, а в операционную систему с тем, чтобы в каждую отдельно взятую программу его не вставлять, а только позволить обращаться к такому коду. Системы программирования стали генерировать обращения к этому системному коду ввода/вывода и осуществлять только подготовку к собственно операциям ввода-вывода, то есть автоматизировать преобразование данных к соответствующему формату, понятному устройствам, избавляя прикладных программистов от этой сложной и трудоемкой работы. Другими словами, системы программирования вставляют в машинный код необходимые библиотечные подпрограммы ввода/вывода и обращения к

тем системным программным модулям, которые, собственно, и управляют операциями обмена между оперативной памятью и внешними устройствами. Таким образом, управление вводом/выводом – это одна из основных функций любой ОС.

С одной стороны, в организации ввода/вывода в различных ОС много общего. С другой стороны, реализация ввода/вывода в ОС так сильно отличается от системы к системе, что очень нелегко выделить и описать именно основные принципы реализации этих функций. Проблема усугубляется еще тем, что в большинстве ныне используемых систем эти моменты вообще, как правило, подробно не описаны, и исключение по этому вопросу касается только системы Linux, для которой имеются комментированные исходные тексты. Детально описываются функции API, реализующие ввод/вывод. Поэтому в этом разделе, самом небольшом по объему, мы рассмотрим только основные идеи и концепции.

Поскольку внешняя память, как правило, реализуется на таких устройствах ввода/вывода, как накопители на магнитных дисках, мы также рассмотрим логическую структуру диска.

Рассмотрением наиболее распространенные файловые системы.

Основные понятия и концепции организации ввода/вывода в ОС

Как известно, ввод-вывод считается одной из самых сложных областей проектирования операционных систем, в которой сложно применить общий подход из-за изобилия частных методов. Сложность возникает из-за огромного числа устройств ввода/вывода разнообразной природы, которые должна поддерживать ОС. При этом перед создателями ОС встает очень непростая задача – не только обеспечить эффективное управление устройствами ввода/вывода, но и создать удобный и эффективный виртуальный интерфейс устройств ввода/вывода, позволяющий прикладным программистам просто считывать или сохранять данные, не обращая внимание на специфику устройств и проблемы распределения устройств между выполняющимися задачами. Система ввода/вывода, способная объединить в одной модели широкий набор устройств, должна быть универсальной. Она должна учитывать потребности существующих устройств, от простой мыши до клавиатур, принтеров, графических дисплеев, дисковых накопителей, компакт-дисков и даже сетей. С другой стороны, необходимо обеспечить доступ к устройствам ввода/вывода для множества параллельно выполняющихся задач, причем так, чтобы они как можно меньше мешали друг другу.

Поэтому самым главным является следующий принцип: любые операции по управлению вводом/выводом объявляются привилегированными и могут выполняться только кодом самой ОС. Для обеспечения этого принципа в большинстве процессоров даже вводятся режимы пользователя и супервизора. Как правило, в режиме супервизора выполнение команд ввода/вывода разрешено, а в пользовательском режиме – запрещено. Использование команд ввода/вывода в пользовательском режиме вызывает исключение, и управление через механизм прерываний передается коду ОС. Хотя возможны и более сложные системы, в которых в ряде случаев пользовательским программам разрешено непосредственное выполнение команд ввода/вывода.

Еще раз подчеркнем, что, прежде всего, мы говорим о мультипрограммных ОС, для которых существует проблема разделения ресурсов. Одним из основных видов ресурсов являются устройства ввода/вывода и соответствующее программное обеспечение, с помощью которого осуществляется управление обменом данными между внешними устройствами и оперативной памятью. Помимо разделяемых устройств ввода/вывода (эти устройства допускают разделение посредством механизма доступа) существуют неразделяемые устройства. Примерами разделяемого устройства могут служить накопитель на магнитных дисках, устройство для чтения компакт-дисков. Это устройства с прямым доступом. Примеры неразделяемых устройств – принтер, накопитель на магнитных лентах. Это устройства с последовательным доступом. Операционные системы

должны управлять и теми и другими устройствами, предоставляя возможность параллельно выполняющимся задачам использовать различные устройства ввода/вывода.

Можно назвать три основные причины, по которым нельзя разрешать каждой отдельной пользовательской программе обращаться к внешним устройствам непосредственно:

1. Необходимость разрешать возможные конфликты доступа к устройствам ввода/вывода. Например, две параллельно выполняющиеся программы пытаются вывести на печать результаты своей работы. Если не предусмотреть внешнее управление устройством печати, то в результате мы можем получить абсолютно нечитаемый текст, так как каждая программа будет время от времени выводить свои данные, которые будут перемежаться данными другой программы. Другой пример: ситуация, когда одной программе необходимо прочитать данные с некоторого сектора магнитного диска, а другой – записать результаты в другой сектор того же накопителя. Если операции ввода/вывода не будут отслеживаться каким-то третьим (внешним) процессом-арбитром, то после позиционирования магнитной головки для первого запроса может тут же появиться команда позиционирования головки для второй задачи, и обе операции ввода/вывода не смогут быть выполнены корректно.

2. Желание увеличить эффективность использования этих ресурсов. Например, у накопителя на магнитных дисках время подвода головки чтения/записи к необходимой дорожке и обращение к определенному сектору может значительно (до тысячи раз) превышать время пересылки данных. В результате, если задачи по очереди обращаются к цилиндрам, далеко отстоящим друг от друга, то полезная работа, выполняемая накопителем, может быть существенно снижена.

3. Ошибки в программах ввода/вывода могут привести к краху всех вычислительных процессов, ибо часть операций ввода/вывода осуществляется для самой операционной системы. В ряде ОС системный ввод/вывод имеет существенно более высокие привилегии, чем ввод/вывод задач пользователя. Поэтому системный код, управляющий операциями ввода/вывода, очень тщательно отлаживается и оптимизируется для повышения надежности вычислений и эффективности использования оборудования.

Итак, управление вводом/выводом осуществляется операционной системой, компонентом, который чаще всего называют супервизором ввода/вывода. В перечень основных задач, возлагаемых на супервизор, входят следующие:

– супервизор ввода/вывода получает запросы на ввод/вывод от прикладных задач и от программных модулей самой операционной системы. Эти запросы проверяются на корректность, и если запрос выполнен по спецификациям и не содержит ошибок, он обрабатывается дальше, в противном случае пользователю (задаче) выдается соответствующее диагностическое сообщение о недействительности (некорректности)запроса;

– супервизор ввода/вывода вызывает соответствующие распределители каналов и контроллеров, планирует ввод/вывод (определяет очередность предоставления устройств ввода/вывода задачам, затребовавшим их). Запрос на ввод/ вывод либо тут же выполняется, либо ставится в очередь на выполнение;

– супервизор ввода/вывода инициирует операции ввода/вывода (передает управление соответствующим драйверам) и в случае управления вводом/выводом с использованием прерываний предоставляет процессор диспетчеру задач с тем, чтобы передать его первой задаче, стоящей в очереди на выполнение;

– при получении сигналов прерываний от устройств ввода/вывода супервизор идентифицирует их (рис. 4.1) и передает управление соответствующей программе обработки прерывания (как правило, на секцию продолжения драйвера);

– супервизор ввода/вывода осуществляет передачу сообщений об ошибках, если таковые происходят в процессе управления операциями ввода/вывода;

– супервизор ввода/вывода посылает сообщения о завершении операции ввода/вывода запросившему эту операцию процессу и снимает его с состояния ожидания ввода/вывода, если процесс ожидал завершения операции.

В случае если устройство ввода/вывода является *инициативным* (инициативным называют такое устройство (обычно это датчики, внешнее устройство, а не устройство ввода/вывода), по сигналу прерывания от которого запускается соответствующая ему программа. Такая программа, с одной стороны, не является драйвером, и управлять операциями обмена данными нет необходимости. Но, с другой стороны, запуск такой программы осуществляется именно по событиям, связанным с генерацией устройством ввода/вывода соответствующего сигнала. Разница между драйверами, работающими по прерываниям, и инициативными программами заключается в статусе этих программных модулей. Драйвер является компонентом операционной системы и часто выполняется не как вычислительный процесс, а как системный объект, а инициативная программа является обычным вычислительным процессом, только его запуск осуществляется по инициативе внешнего устройства.), управление со стороны супервизора ввода/вывода будет заключаться в активизации соответствующего вычислительного процесса (перевод его в состояние готовности к выполнению). Таким образом, прикладные программы (а в общем случае – все обрабатывающие программы) не могут непосредственно связываться с устройствами ввода/ вывода независимо от использования устройств (монопольно или совместно). Установив соответствующие значения параметров в запросе на ввод/вывод, определяющих требуемую операцию и количество потребляемых ресурсов, они могут передать управление супервизору ввода/вывода, который и запускает необходимые логические и физические операции.

Упомянутый выше запрос на ввод/вывод должен удовлетворять требованиям API той операционной системы, в среде которой выполняется приложение. Параметры, указываемые в запросах на ввод/вывод, передаются не только в вызывающих последовательностях, создаваемых по спецификациям API, но и как данные, хранящиеся в соответствующих системных таблицах. Все параметры, которые будут стоять в вызывающей последовательности, поставляются компилятором и отражают требования программиста и постоянные сведения об операционной системе и архитектуре компьютера в целом. Переменные сведения о вычислительной системе (ее конфигурация, состав оборудования, состав и особенности системного программного обеспечения) содержатся в специальных системных таблицах. Процессору, каналам прямого доступа в память, контроллерам необходимо передавать конкретную двоичную информацию, с помощью которой и осуществляется управление оборудованием. Эта конкретная двоичная информация в виде кодов и данных часто готовится с помощью препроцессоров, но часть ее хранится в системных таблицах.

Режимы управления вводом/выводом

Как известно, имеются два основных режима ввода/вывода: режим обмена с опросом готовности устройства ввода/вывода и режим обмена с прерываниями. Рассмотрим рис. 3.1.

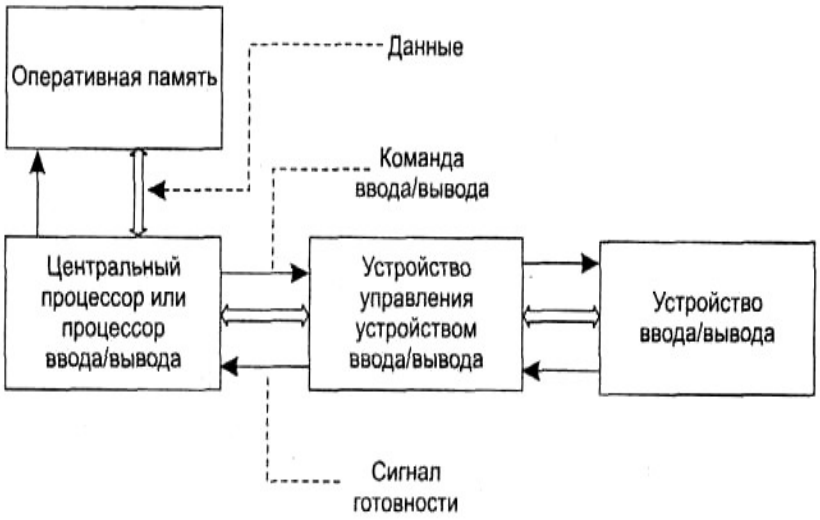

Рис. 3.1. Управление вводом-выводом

Пусть для простоты управление вводом/выводом осуществляет центральный процессор (в этом случае часто говорят о наличии программного канала обмена данными между внешними устройством и оперативной памятью, в отличие от канала прямого доступа к памяти, при котором управление вводом/выводом осуществляет специальное дополнительное оборудование; эти вопросы мы обсудим несколько позже). Центральный процессор посылает устройству управления команду выполнить некоторое действие устройству ввода/вывода. Последнее исполняет команду, транслируя сигналы, понятные центральному устройству и устройству управления в сигналы, понятные устройству ввода/вывода. Но быстродействие устройства ввода/вывода намного меньше быстродействия центрального процессора (порой на несколько порядков). Поэтому сигнал готовности (транслируемый или генерируемый устройством управления и сигнализирующий процессору о том, что команда ввода/вывода выполнена и можно выдать новую команду для продолжения обмена данными) приходится очень долго ожидать, постоянно опрашивая соответствующую линию интерфейса на наличие или отсутствие нужного сигнала. Посылать новую команду, не дождавшись сигнала готовности, сообщающего об исполнении предыдущей команды, бессмысленно. В режиме опроса готовности драйвер, управляющий процессом обмена данными с внешним устройством, как раз и выполняет в цикле команду «проверить наличие сигнала готовности». До тех пор пока сигнал готовности не появится, драйвер ничего другого не делает. При этом, естественно, нерационально используется время центрального процессора. Гораздо выгоднее, выдав команду ввода/ вывода, на время забыть об устройстве ввода/вывода и перейти на выполнение другой программы. А появление сигнала готовности трактовать как запрос на прерывание от устройства ввода/вывода. Именно эти сигналы готовности и являются сигналами запроса на прерывание.

Режим обмена с прерываниями по своей сути является режимом асинхронного управления. Для того чтобы не потерять связь с устройством (после того как процессор выдал очередную команду по управлению обменом данными и переключился на выполнение других программ), может быть запущен отсчет времени, в течение которого устройство обязательно должно выполнить команду и выдать таки сигнал запроса на прерывание. Максимальный интервал времени, в течение которого устройство ввода/вывода или его контроллер должны выдать сигнал запроса на прерывание, часто называют уставкой тайм-аута. Если это время истекло после выдачи устройству очередной команды, а устройство так и не ответило, то делается вывод о том, что связь с устройством потеряна и управлять им больше нет возможности. Пользователь и/или задача получают соответствующее диагностическое сообщение.

Драйверы, работающие в режиме прерываний, представляют собой сложный комплекс программных модулей и могут иметь несколько секций: секцию запуска, одну или несколько секций продолжения и секцию завершения.

Секция запуска инициирует операцию ввода/вывода. Эта секция запускается для включения устройства ввода/вывода либо просто для инициации очередной операции ввода/вывода.

Секция продолжения (их может быть несколько, если алгоритм управления обменом данными сложный и требуется несколько прерываний для выполнения одной логической операции) осуществляет основную работу по передаче данных. Секция продолжения, собственно говоря, и является основным обработчиком прерывания. Используемый интерфейс может потребовать для управления вводом/выводом несколько последовательностей управляющих команд, а сигнал прерывания у устройства, как правило, только один. Поэтому после выполнения очередной секции прерывания супервизор прерываний при следующем сигнале готовности должен передать управление другой секции. Это делается за счет изменения адреса обработки прерывания после выполнения очередной секции, если же имеется только одна секция прерываний, то она сама передает управление тому или иному модулю обработки.

Секция завершения обычно выключает устройство ввода/вывода либо просто завершает операцию.

Управление операциями ввода/вывода в режиме прерываний требует больших усилий со стороны системных программистов – такие программы создавать сложнее, чем те, что работают в режиме опроса готовности. Примером тому может служить ситуация с драйверами, обеспечивающими печать. Так, в ОС Windows (и Windows 9x, и Windows NT) драйвер печати через параллельный порт работает не в режиме с прерываниями, как это сделано в других ОС, а в режиме опроса готовности, что приводит к 100%-й загрузке центрального процессора на все время печати. При этом, естественно, выполняются и другие задачи, запущенные на исполнение, но исключительно за счет того, что ОС Windows реализует вытесняющую мультизадачность и время от времени прерывает процесс управления печатью и передает центральный процессор остальным задачам.

Закрепление устройств, общие устройства ввода/вывода

Как известно, многие устройства не допускают совместного использования. Прежде всего, это устройства с последовательным доступом. Такие устройства могут стать закрепленными, то есть быть предоставленными некоторому вычислительному процессу на все время жизни этого процесса. Однако это приводит к тому, что вычислительные процессы часто не могут выполняться параллельно – они ожидают освобождения устройств ввода/вывода. Для организации использования многими параллельно выполняющимися задачами устройств ввода/вывода, которые не могут быть разделяемыми, вводится понятие виртуальных устройств. Использование принципа виртуализации позволяет повысить эффективность вычислительной системы.

Вообще говоря, понятие виртуального устройства шире, нежели использование этого термина для обозначения спулинга (SPOOLing – simultaneous peripheral operation on-line, то есть имитация работы с устройством в режиме «он-лайн»). Главная задача спулинга – создать видимость параллельного разделения устройства ввода/вывода с последовательным доступом, которое фактически должно использоваться только монопольно и быть закрепленным. Например, мы уже говорили, что в случае, когда несколько приложений должны выводить на печать результаты своей работы, если разрешить каждому такому приложению печатать строку по первому же требованию, то это приведет к потоку строк, не представляющих никакой ценности. Однако можно каждому вычислительному процессу предоставлять не реальный, а виртуальный принтер, и поток выводимых символов (или управляющих кодов для их печати) сначала направлять в специальный файл1 на магнитном диске. Затем, по окончании виртуальной печати, в

соответствии с принятой дисциплиной обслуживания и приоритетами приложений выводить содержимое спул-файла на принтер. Системный процесс, который управляет спул-файлом, называется спулером (spool-reader или spool-writer).

Основные системные таблицы ввода/вывода

Каждая ОС имеет свои таблицы ввода/вывода, их состав (количество и назначение каждой таблицы) может сильно отличаться. В некоторых ОС вместо таблиц создаются списки, хотя использование статических структур данных для организации ввода/вывода, как правило, приводит к большему быстродействию. Здесь очень трудно вычленить общие составляющие, тем более что подробной документации на эту тему крайне мало, только если воспользоваться материалами ныне устаревших ОС. Тем не менее попытаемся это сделать, опираясь на идеи семейства простых, но эффективных ОС реального времени, разработанных фирмой Hewlett-Packard для своих мини-компьютеров.

Исходя из принципа управления вводом/выводом через супервизор ОС и учитывая, что драйверы устройств ввода/вывода используют механизм прерываний для установления обратной связи центральной части с внешними устройствами, можно сделать вывод о необходимости создания, по крайней мере, трех системных таблиц.

Первая таблица (или список) содержит информацию обо всех устройствах ввода/вывода, подключенных к вычислительной системе. Назовем ее условно таблицей оборудования (equipment table), а каждый элемент этой таблицы пусть называется UCB (unit control block, блок управления устройством ввода/вывода). Каждый элемент UCB таблицы оборудования, как правило, содержит следующую информацию об устройстве:

– тип устройства, его конкретная модель, символическое имя и характеристики устройства;

– как это устройство подключено (через какой интерфейс, к какому разъему, какие порты и линия запроса прерывания используются и т. д.);

– номер и адрес канала (и подканала), если такие используются для управления устройством;

– указание на драйвер, который должен управлять этим устройством, адрес секции запуска и секции продолжения драйвера;

– информация о том, используется или нет буферирование при обмене данными с этим устройством, «имя» (или просто адрес) буфера, если такой выделяется из системной области памяти;

– уставка тайм-аута и ячейки для счетчика тайм-аута;

– состояние устройства;

– поле указателя для связи задач, ожидающих устройство, и, возможно, много еще каких сведений.

Поясним перечисленное. Поскольку во многих ОС драйверы могут обладать свойством реентерабельности (напомним, это означает, что один и тот же экземпляр программного модуля может обеспечить параллельное обслуживание сразу нескольких однотипных устройств), то в элементе UCB должна храниться либо непосредственно сама информация о текущем состоянии устройства и сами переменные для реентерабельной обработки, либо указание на место, где такая информация может быть найдена. Наконец, важнейшим компонентом элемента таблицы оборудования является указатель на дескриптор той задачи, которая сейчас использует данное устройство. Если устройство свободно, то поле указателя будет иметь нулевое значение. Если же устройство уже занято и рассматриваемый указатель не нулевой, то новые запросы к устройству фиксируются посредством образования списка из дескрипторов тех задач, которые сейчас ожидают данное устройство.

Вторая таблица предназначена для реализации еще одного принципа виртуализации устройств ввода/вывода – независимости от устройства. Желательно, чтобы программист не был озабочен учетом конкретных параметров (и/или возможностей) того или иного

устройства ввода/вывода, которое установлено (или не установлено) в компьютер. Для него должны быть важны только самые общие возможности, характерные для данного класса устройств ввода/вывода, которыми он желает воспользоваться. Например, принтер должен уметь выводить (печатать) символы или графическое изображение. А накопитель на магнитных дисках – считывать или записывать по указанному адресу (в координатах C-H-S) порцию данных. Хотя чаще всего программист и не использует прямую адресацию при работе с магнитными дисками, а работает на уровне файловой системы. Однако в таком случае уже разработчики файловой системы не должны зависеть от того, накопитель какого конкретного типа и модели, а также какого производителя используется в данном конкретном компьютере (например, HDD IBM DTLA 307030, WDAC 450AA или какой-нибудь еще). Важным должен быть только сам факт существования накопителя, имеющего некоторое количество цилиндров, головок чтения/записи и секторов на дорожке магнитного диска. Упомянутые значения количества цилиндров, головок и секторов должны быть взяты из элемента таблицы оборудования. При этом для программиста также не должно иметь значения, каким образом то или иное устройство подключено к вычислительной системе, а не только какая конкретная модель устройства используется. Поэтому в запросе на ввод/вывод программист указывает именно логическое имя устройства. Действительное устройство, которое сопоставляется виртуальному (логическому), выбирается супервизором с помощью таблицы, о которой мы сейчас говорим. Итак, способ подключения устройства, его конкретная модель и соответствующий ей драйвер содержатся в уже рассмотренной таблице оборудования. Но для того, чтобы связать некоторое виртуальное устройство, использованное программистом при создании приложения с системной таблицей, отображающей информацию о том, какое конкретно устройство и каким образом подключено к компьютеру, используется вторая системная таблица. Назовем ее условно таблицей описания виртуальных логических устройств (DRT, device reference table). Назначение этой второй таблицы – установление связи между виртуальными (логическими) устройствами и реальными устройствами, описанными посредством первой таблицы оборудования. Другими словами, вторая таблица позволяет супервизору перенаправить запрос на ввод/вывод из приложения на те программные модули и структуры данных, которые (или адреса которых) хранятся в соответствующем элементе первой таблицы. Во многих многопользовательских системах такая таблица не одна, а несколько: одна общая и по одной – на каждого пользователя, что позволяет строить необходимые связи между логическими (символьными) именами устройств и реальными физическими устройствами, которые имеются в системе.

Наконец, третья таблица необходима для организации обратной связи между центральной частью и устройствами ввода/вывода. Это таблица прерываний, которая указывает для каждого сигнала запроса на прерывание тот элемент UCB, который сопоставлен данному устройству, подключенному так, что оно использует настоящую линию (сигнал) прерывания. Как системная таблица ввода/вывода, таблица прерываний может в явном виде и не присутствовать. В принципе можно сразу из основной таблицы прерываний попадать на программу обработки (драйвер), имеющей связи с элементом UCB. Важно наличие связи между сигналами прерываний и таблицей оборудования.

В современных сложных ОС имеется гораздо больше системных таблиц или списков, используемых для организации процессами управления операциями ввода/вывода. Например, одной из возможных и часто реализуемых информационных структур, сопровождающих практически каждый запрос на ввод/вывод, является блок управления данными (data control block, DCB). Назначение этого DCB – подключение препроцессоров к процессу подготовки данных на ввод/вывод, то есть учет конкретных технических характеристик и используемых преобразований. Это необходимо для того, чтобы имеющееся устройство получало не какие-то непонятные ему коды либо форматы

данных, которые не соответствуют режиму его работы, а коды, созданные специально под данное устройство и используемый в настоящий момент формат представления данных.

Взаимосвязи между описанными таблицами изображены на рис. 3.2.

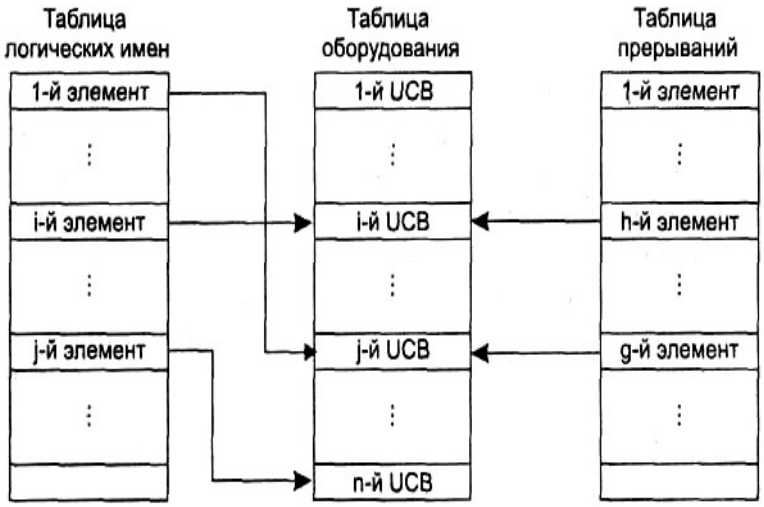

Рис. 3.2. Взаимосвязь системных таблиц ввода/вывода

Теперь нам осталось лишь еще раз, с учетом изложенных принципов и таблиц, рассмотреть процесс управления вводом/вывода с помощью рис. 3.3.

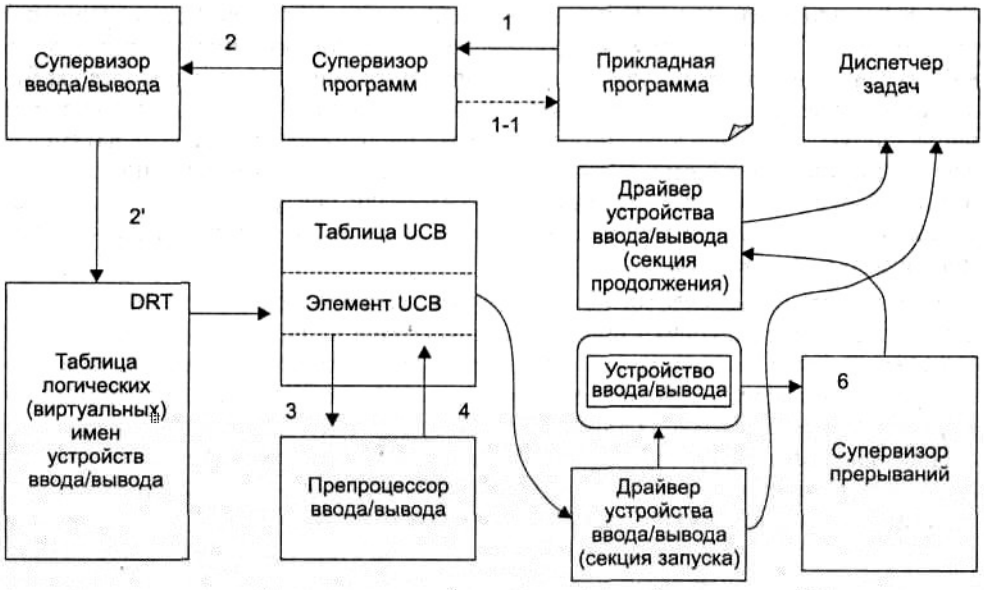

Рис. 3.3. Процесс управления вводом/выводом

Запрос на операцию ввода/вывода от выполняющейся программы поступает на супервизор (действие 1). Тот проверяет системный вызов на соответствие принятым спецификациям и в случае ошибки возвращает задаче соответствующее сообщение (действие 1-1). Если же запрос корректен, то он перенаправляется в супервизор ввода/вывода (действие 2). Последний по логическому (виртуальному) имени с помощью таблицы DRT находит соответствующий элемент UCB в таблице оборудования. Если устройство уже занято, то описатель задачи, запрос которой сейчас обрабатывается супервизором ввода/вывода, помещается в список задач, ожидающих настоящее устройство. Если же устройство свободно, то супервизор ввода/вывода определяет из UCB тип устройства и при необходимости запускает препроцессор, позволяющий получить последовательность управляющих кодов и данных, которую сможет правильно понять и отработать устройство (действие 3). Когда «программа» управления операцией ввода/вывода будет готова, супервизор ввода/вывода передаст управление соответствующему драйверу на секцию запуска (действие 4). Драйвер инициализирует операцию управления, обнуляет счетчик тайм-аута и возвращает управление супервизору

(диспетчеру задач) с тем, чтобы он поставил на процессор готовую к исполнению задачу (действие 5). Система работает своим чередом, но когда устройство ввода/вывода отработает посланную ему команду, оно выставляет сигнал запроса на прерывания, по которому через таблицу прерываний управление передается на секцию продолжения (действие 6). Получив новую команду, устройство вновь начинает ее обрабатывать, а управление процессором опять передается диспетчеру задач, и процессор продолжает полезную работу. Таким образом, получается параллельная обработка задач, на фоне которой процессор осуществляет управление операциями ввода/вывода.

Очевидно, что если имеются специальные аппаратные средства для управления вводом/выводом, снимающие эту работу с центрального процессора (речь идет о каналах прямого доступа к памяти), то в функции центрального процессора будут по-прежнему входить все только что рассмотренные шаги, за исключением последнего – непосредственного управления операциями ввода/вывода. В случае использования каналов прямого доступа к памяти последние исполняют соответствующие канальные программы и разгружают центральный процессор, избавляя его от непосредственного управления обменом данными между памятью и внешними устройствами.

При описании этой схемы мы не стали затрагивать вопросы распределения каналов, контроллеров и собственно самих устройств. Также были опущены детали получения канальных программ.

Синхронный и асинхронный ввод/вывод

Задача, выдавшая запрос на операцию ввода/вывода, переводится супервизором в состояние ожидания завершения заказанной операции. Когда супервизор получает от секции завершения сообщение о том, что операция завершилась, он переводит задачу в состояние готовности к выполнению, и она продолжает свою работу. Эта ситуация соответствует синхронному вводу/выводу. Синхронный ввод/вывод является стандартным для большинства ОС. Чтобы увеличить скорость выполнения приложений, было предложено при необходимости использовать асинхронный ввод/вывод.

Простейшим вариантом асинхронного вывода является так называемый буферированный вывод данных на внешнее устройство, при котором данные из приложения передаются не непосредственно на устройство ввода/вывода, а в специальный системный буфер. В этом случае логически операция вывода для приложения считается выполненной сразу же, и задача может не ожидать окончания действительного процесса передачи данных на устройство. Процессом реального вывода данных из системного буфера занимается супервизор ввода/ вывода. Естественно, что выделением буфера из системной области памяти занимается специальный системный процесс по указанию супервизора ввода/вывода. Итак, для рассмотренного случая вывод будет асинхронным, если, во-первых, в запросе на ввод/вывод было указано на необходимость буферирования данных, а во-вторых, если устройство ввода/вывода допускает такие асинхронные операции и это отмечено в UCB.

Можно организовать и асинхронный ввод данных. Однако для этого необходимо не только выделить область памяти для временного хранения считываемых с устройства данных и связать выделенный буфер с задачей, заказавшей операцию, но и сам запрос на операцию ввода/вывода разбить на две части (на два запроса). В первом запросе указывается операция на считывание данных, подобно тому как это делается при синхронном вводе/выводе. Однако тип (код) запроса используется другой, и в запросе указывается еще по крайней мере один дополнительный параметр – имя (код) того системного объекта, которое получает задача в ответ на запрос и которое идентифицирует выделенный буфер. Получив имя буфера (будем этот системный объект условно называть, таким образом, хотя в различных ОС для его обозначения используются и другие термины, например – класс), задача продолжает свою работу. Здесь очень важно подчеркнуть, что в результате запроса на асинхронный ввод данных задача не переводится

супервизором ввода/вывода в состояние ожидания завершения операции ввода/ вывода, а остается в состоянии выполнения или в состоянии готовности к выполнению. Через некоторое время, выполнив необходимый код, который был определен программистом, задача выдает второй запрос на завершение операции ввода/вывода. В этом втором запросе к тому же устройству, который, естественно, имеет другой код (или имя запроса), задача указывает имя системного объекта (буфера для асинхронного ввода данных) и в случае успешного завершения операции считывания данных тут же получает их из системного буфера. Если же данные еще не успели до конца переписаться с внешнего устройства в системный буфер, супервизор ввода/вывода переводит задачу в состояние ожидания завершения операции ввода/вывода, и далее все напоминает обычный синхронный ввод данных.

Обычно асинхронный ввод/вывод предоставляется в большинстве мультипрограммных ОС, особенно если ОС поддерживает мультизадачность с помощью механизма тредов. Однако если асинхронный ввод/вывод в явном виде отсутствует, его идеи можно реализовать самому, организовав для вывода данных самостоятельный поток.

Аппаратуру ввода/вывода можно рассматривать как совокупность аппаратурных процессоров, которые способны работать параллельно относительно друг друга, а также относительно центрального процессора (процессоров). На таких «процессорах» выполняются так называемые внешние процессы. Например, для внешнего устройства (устройства ввода/вывода) внешний процесс может представлять собой совокупность операций, обеспечивающих перевод печатающей головки, продвижение бумаги на одну позицию, смену цвета чернил или печать каких-то символов. Внешние процессы, используя аппаратуру ввода/вывода, взаимодействуют как между собой, так и с обычными «программными» процессами, выполняющимися на центральном процессоре. Важным при этом является то обстоятельство, что скорости выполнения внешних процессов будут существенно (порой, на порядок или больше) отличаться от скорости выполнения обычных («внутренних») процессов. Для своей нормальной работы внешние и внутренние процессы обязательно должны синхронизироваться. Для сглаживания эффекта сильного несоответствия скоростей между внутренними и внешними процессами используют упомянутое выше буферирование. Таким образом, можно говорить о системе параллельных взаимодействующих процессов.

Буферы являются критическим ресурсом в отношении внутренних (программных) и внешних процессов, которые при параллельном своем развитии информационно взаимодействуют. Через буфер (буферы) данные либо посылаются от некоторого процесса к адресуемому внешнему (операция вывода данных на внешнее устройство), либо от внешнего процесса передаются некоторому программному процессу (операция считывания данных). Введение буферирования как средства информационного взаимодействия выдвигает проблему управления этими системными буферами, которая решается средствами супервизорной части ОС. При этом на супервизор возлагаются задачи не только по выделению и освобождению буферов в системной области памяти, но и синхронизации процессов в соответствии с состоянием операций по заполнению или освобождению буферов, а также их ожидания, если свободных буферов в наличии нет, а запрос на ввод/вывод требует буферирования. Обычно супервизор ввода/вывода для решения перечисленных задач использует стандартные средства синхронизации, принятые в данной ОС. Поэтому если ОС имеет развитые средства для решения проблем параллельного выполнения взаимодействующих приложений и задач, то, как правило, она реализует и асинхронный ввод-вывод.

Кэширование операций ввода/вывода при работе с накопителями на магнитных дисках

Как известно, накопители на магнитных дисках обладают крайне низкой скоростью по сравнению с быстродействием центральной части компьютера. Разница в быстродействии отличается на несколько порядков. Например, современные процессоры за один такт работы, а они работают уже с частотами в 1 ГГц и более, могут выполнять по две операции. Таким образом, время выполнения операции (с позиции внешнего наблюдателя, не видящего конвейеризации при выполнении машинных команд, благодаря которой производительность возрастает в несколько раз) может составлять 0,5 нс (!). В то же время переход магнитной головки с дорожки на дорожку составляет несколько миллисекунд. Такие же временные интервалы имеют место и при ожидании, пока под головкой чтения/ записи не окажется нужный сектор данных. Как известно, в современных приводах средняя длительность на чтение случайным образом выбранного сектора данных составляет около 20 мс, что существенно медленнее, чем выборка команды или операнда из оперативной памяти и уж, тем более из кэша. Правда, после этого данные читаются большим пакетом (сектор, как мы уже говорили, имеет размер в 512 байтов, а при операциях с диском часто читается или записывается сразу несколько секторов). Таким образом, средняя скорость работы процессора с оперативной памятью на 2-3 порядка выше, чем средняя скорость передачи данных из внешней памяти на магнитных дисках в оперативную память.

Для того чтобы сгладить такое сильное несоответствие в производительности основных подсистем, используется буферирование и/или кэширование1 данных. Простейшим вариантом ускорения дисковых операций чтения данных можно считать использование двойного буферирования. Его суть заключается в том, что пока в один буфер заносятся данные с магнитного диска, из второго буфера ранее считанные данные могут быть прочитаны и переданы запросившей их задаче. Аналогичный процесс происходит и при записи данных. Буферирование используется во всех операционных системах, но помимо буферирования применяется и кэширование. Кэширование исключительно полезно в том случае, когда программа неоднократно читает с диска одни и те же данные. После того как они один раз будут помещены в кэш, обращений к диску больше не потребуется и скорость работы программы значительно возрастет.

Если не вдаваться в подробности, то под кэшем можно понимать некий пул буферов, которыми мы управляем с помощью соответствующего системного процесса. Если мы считываем какое-то множество секторов, содержащих записи того или иного файла, то эти данные, пройдя через кэш, там остаются (до тех пор, пока другие секторы не заменят эти буферы). Если впоследствии потребуется повторное чтение, то данные могут быть извлечены непосредственно из оперативной памяти без фактического обращения к диску. Ускорить можно и операции записи: данные помещаются в кэш, и для запросившей эту операцию задачи можно считать, что они уже фактически и записаны. Задача может продолжить свое выполнение, а системные внешние процессы через некоторое время запишут данные на диск. Это называется операцией отложенной записи (lazy write, «ленивая запись»). Если отложенная запись отключена, только одна задача может записывать на диск свои данные. Остальные приложения должны ждать своей очереди. Это ожидание подвергает информацию риску не меньшему (если не большему), чем отложенная запись, которая к тому же и более эффективна по скорости работы с диском.

Интервал времени, после которого данные будут фактически записываться, с одной стороны, желательно выбрать больше, поскольку если потребуется еще раз прочитать эти данные, то они уже и так фактически находятся в кэше. И после модификации эти данные опять же помещаются в быстродействующий кэш. С другой стороны, для большей надежности данные желательно поскорее отправить во внешнюю память, поскольку она энергонезависима и в случае какой-нибудь аварии (например, нарушения питания) данные в оперативной памяти пропадут, в то время как на магнитном диске они с большой вероятностью останутся в безопасности. Количество буферов, составляющих кэш, ограничено, поэтому возникает ситуация, когда вновь прочитанные или записываемые

новые секторы данных должны будут заменить данные в этих буферах. Возможно использование различных дисциплин, в соответствии с которыми будет назначен какой-либо буфер под вновь затребованную операцию кэширования.

Кэширование дисковых операций может быть существенно улучшено за счет введения техники упреждающего чтения (read ahead). Она основана на чтении с диска гораздо большего количества данных, чем на самом деле запросила операционная система или приложение. Когда некоторой программе требуется считать с диска только один сектор, программа кэширования читает еще и несколько дополнительных блоков данных. А операции последовательного чтения нескольких секторов фактически несущественно замедляют операцию чтения затребованного сектора с данными. Поэтому, если программа вновь обратится к диску, вероятность того, что нужные ей данные уже находятся в кэше, достаточно высока. Поскольку передача данных из одной области памяти в другую происходит во много раз быстрее, чем чтение их с диска, кэширование существенно сокращает время выполнения операций с файлами.

Итак, путь информации от диска к прикладной программе пролегает как через буфер, так и через файловый кэш. Когда приложение запрашивает с диска данные, программа кэширования перехватывает этот запрос и читает вместе с необходимыми секторами еще и несколько дополнительных. Затем она помещает в буфер требующуюся задаче информацию и ставит об этом в известность операционную систему. Операционная система сообщает задаче, что ее запрос выполнен, и данные с диска находятся в буфере. При следующем обращении приложения к диску программа кэширования прежде всего проверяет, не находятся ли уже в памяти затребованные данные. Если это так, то она копирует их в буфер; если же их в кэше нет, то запрос на чтение диска передается операционной системе. Когда задача изменяет данные в буфере, они копируются в кэш.

В ряде ОС имеется возможность указать в явном виде параметры кэширования, в то время как в других за эти параметры отвечает сама ОС. Так, например, в системе Windows NT нет возможности в явном виде управлять ни объемом файлового кэша, ни параметрами кэширования. В системах Windows 95/98 такая возможность уже имеется, но она представляет не слишком богатый выбор. Фактически мы можем указать только объем памяти, отводимый для кэширования, и объем порции данных (буфер или chunk1), из которых набирается кэш. В файле System.ini есть возможность в секции [VCACHE] прописать, например, следующие значения:

```
[vcache]
MinFileCache=4096
MaxFileCache=32768
ChunkSize=512
```

Здесь указано, что минимально под кэширование данных зарезервировано 4 Мбайт оперативной памяти, максимальный объем кэша может достигать 32 Мбайт, а размер данных, которыми манипулирует менеджер кэша, равен одному сектору.

В других ОС можно указывать больше параметров, определяющих работу подсистемы кэширования. Пример, демонстрирующий эти возможности, можно посмотреть в разделе «Файловая система HPFS».

Помимо описанных действий ОС может выполнять и работу по оптимизации перемещения головок чтения/записи данных, связанного с выполнением запросов от параллельно выполняющихся задач. Время, необходимое на получение данных с магнитного диска, складывается из времени перемещения магнитной головки на требуемый цилиндр и времени ожидания заданного сектора; временем считывания найденного сектора и затратами на передачу этих данных в оперативную память мы можем пренебречь. Таким образом, основные затраты времени уходят на поиск данных. В мультипрограммных ОС при выполнении многих задач запросы на чтение и запись данных могут идти таким потоком, что при их обслуживании образуется очередь. Если выполнять эти запросы в порядке поступления их в очередь, то вследствие случайного

характера обращений к тому или иному сектору магнитного диска мы можем иметь значительные потери времени на поиск данных. Напрашивается очевидное решение: поскольку выполнение переупорядочивания запросов с целью минимизации затрат времени на поиск данных можно выполнить очень быстро (практически этим временем можно пренебречь, учитывая разницу в быстродействии центральной части и устройств ввода/вывода), то необходимо найти метод, позволяющий перестраивать очередь запросов оптимальным образом. Изучение этой проблемы позволило найти наиболее эффективные дисциплины планирования.

Перечислим известные дисциплины, в соответствии с которыми можно перестраивать очередь запросов на операции чтения/записи данных:

– SSTF (shortest seek time – first) – с наименьшим временем поиска – первым. В соответствии с этой дисциплиной при позиционировании магнитных головок следующим выбирается запрос, для которого необходимо минимальное перемещение с цилиндра на цилиндр, даже если этот запрос не был первым в очереди на ввод/вывод. Однако для этой дисциплины характерна резкая дискриминация определенных запросов, а ведь они могут идти от высокоприоритетных задач. Обращения к диску проявляют тенденцию концентрироваться, в результате чего запросы на обращение к самым внешним и самым внутренним дорожкам могут обслуживаться существенно дольше и нет никакой гарантии обслуживания. Достоинством такой дисциплины является максимально возможная пропускная способность дисковой подсистемы.

– Scan (сканирование). По этой дисциплине головки перемещаются то в одном, то в другом «привилегированном» направлении, обслуживая «по пути» подходящие запросы. Если при перемещении головок чтения/записи более нет попутных запросов, то движение начинается в обратном направлении.

– Next-Step Scan – отличается от предыдущей дисциплины тем, что на каждом проходе обслуживаются только запросы, которые уже существовали на момент начала прохода. Новые запросы, появляющиеся в процессе перемещения головок чтения/записи, формируют новую очередь запросов, причем таким образом, чтобы их можно было оптимально обслужить на обратном ходу.

– C-Scan (циклическое сканирование). По этой дисциплине головки перемещаются циклически с самой наружной дорожки к внутренним, по пути обслуживая имеющиеся запросы, после чего вновь переносятся к наружным цилиндрам. Эту дисциплину иногда реализуют таким образом, чтобы запросы, поступающие во время текущего прямого хода головок, обслуживались не попутно, а при следующем ходе, что позволяет исключить дискриминацию запросов к самым крайним цилиндрам; она характеризуется очень малой дисперсией времени ожидания обслуживания. Эту дисциплину обслуживания часто называют «элеваторной».

Функции файловой системы ОС и иерархия данных

Напомним, что под файлом обычно понимают набор данных, организованных в виде совокупности записей одинаковой структуры. Для управления этими данными создаются соответствующие системы управления файлами. Возможность иметь дело с логическим уровнем структуры данных и операций, выполняемых над ними в процессе их обработки, предоставляет файловая система. Таким образом, файловая система – это набор спецификаций и соответствующее им программное обеспечение, которые отвечают за создание, уничтожение, организацию, чтение, запись, модификацию и перемещение файловой информации, а также за управление доступом к файлам и за управление ресурсами, которые используются файлами. Именно файловая система определяет способ организации данных на диске или на каком-нибудь ином носителе данных. В качестве примера можно привести файловую систему FAT, реализация для которой имеется в абсолютном большинстве ОС, работающих в современных ПК1.

Как правило, все современные ОС имеют соответствующие системы управления файлами. В дальнейшем постараемся различать файловую систему и систему управления файлами.

Система управления файлами является основной подсистемой в абсолютном большинстве современных операционных систем, хотя в принципе можно обходиться и без нее. Во-первых, через систему управления файлами связываются по данным все системные обрабатывающие программы. Во-вторых, с помощью этой системы решаются проблемы централизованного распределения дискового пространства и управления данными. В-третьих, благодаря использованию той или иной системы управления файлами пользователям предоставляются следующие возможности:

– создание, удаление, переименование (и другие операции) именованных наборов данных (именованных файлов) из своих программ или посредством специальных управляющих программ, реализующих функции интерфейса пользователя с его данными и активно использующих систему управления файлами;

– работа с не дисковыми периферийными устройствами как с файлами;

– обмен данными между файлами, между устройствами, между файлом и устройством (и наоборот);

– работа с файлами с помощью обращений к программным модулям системы управления файлами (часть API ориентирована именно на работу с файлами);

– защита файлов от несанкционированного доступа.

В некоторых ОС может быть несколько систем управления файлами, что обеспечивает им возможность работать с несколькими файловыми системами. Очевидно, что системы управления файлами, будучи компонентом ОС, не являются независимыми от этой ОС, поскольку они активно используют соответствующие вызовы API (application program interface, прикладной программный интерфейс). С другой стороны, системы управления файлами сами дополняют API новыми вызовами. Можно сказать, что основное назначение файловой системы и соответствующей ей системы управления файлами – организация удобного доступа к данным, организованным как файлы, то есть вместо низкоуровневого доступа к данным с указанием конкретных физических адресов нужной нам записи используется логический доступ с указанием имени файла и записи в нем.

Другими словами, термин «файловая система» определяет, прежде всего, принципы доступа к данным, организованных в файлы. Этот же термин часто используют и по отношению к конкретным файлам, расположенным на том или ином носителе данных. А термин «система управления файлами» следует употреблять по отношению к конкретной реализации файловой системы, то есть это – комплекс программных модулей, обеспечивающих работу с файлами в конкретной операционной системе.

Следует еще раз заметить, что любая система управления файлами не существует сама по себе – она разработана для работы в конкретной ОС. В качестве примера можно сказать, что всем известная файловая система FAT (file allocation table) имеет множество реализаций как система управления файлами. Так, система, получившая это название и разработанная для первых персональных компьютеров, называлась просто FAT (сейчас ее называют FAT-12). Ее разрабатывали для работы с дискетами, и некоторое время она использовалась при работе с жесткими дисками. Потом ее усовершенствовали для работы с жесткими дисками большего объема, и эта новая реализация получила название FAT-16. Это название файловой системы мы используем и по отношению к системе управления файлами самой MS-DOS. Реализацию же системы управления файлами для OS/2, которая использует основные принципы системы FAT, называют super-FAT; основное отличие – возможность поддерживать для каждого файла расширенные атрибуты. Есть версия системы управления файлами с принципами FAT и для Windows 95/98, для Windows NT и т. д. Другими словами, для работы с файлами, организованными в соответствии с некоторой файловой системой, для каждой ОС должна быть разработана соответствующая система управления файлами. Эта система управления файлами будет

работать только в той ОС, для которой она и создана; но при этом она позволит работать с файлами, созданными с помощью системы управления файлами другой ОС, работающей по тем же основным принципам файловой системы.

Структура магнитного диска (разбиение дисков на разделы)

Для того чтобы можно было загрузить с магнитного диска собственно саму ОС, а уже с ее помощью и организовать работу той или иной системы управления файлами, были приняты специальные системные соглашения о структуре диска. Расположение структуры данных, несущее информацию о логической организации диска и простейшую программу, с помощью которой можно находить и загружать программы загрузки той или иной ОС, очевидно – это самый первый сектор магнитного диска.

Как известно, информация на магнитных дисках размещается и передается блоками. Каждый такой блок называется сектором (sector), сектора расположены на концентрических дорожках поверхности диска. Каждая дорожка (track) образуется при вращении магнитного диска под зафиксированной в некотором предопределенном положении головкой чтения/записи. Накопитель на жестких магнитных дисках (НЖМД) содержит один или более дисков (в современных распространенных НЖМД часто – два или три). Однако обычно под термином «жесткий диск» понимают весь пакет магнитных дисков.

Группы дорожек (треков) одного радиуса, расположенных на поверхностях магнитных дисков, образуют так называемые цилиндры (cylinder). Современные жесткие диски могут иметь по несколько десятков тысяч цилиндров, в то время как на поверхности дискеты число дорожек (число цилиндров) ныне, как правило, составляет всего восемьдесят.

Каждый сектор состоит из поля данных и поля служебной информации, ограничивающей и идентифицирующей его. Размер сектора (точнее – емкость поля данных) устанавливается контроллером или драйвером. Пользовательский интерфейс DOS поддерживает единственный размер сектора – 512 байт. BIOS же непосредственно предоставляет возможности работы с секторами размером 128, 256, 512 или 1024 байт. Если управлять контроллером непосредственно, а не через программный интерфейс более высокого уровня (например, уровень DOS), то можно обрабатывать секторы и с другими размерами. Однако в большинстве современных ОС размер сектора выбирается равным 512 байт.

Физический адрес сектора на диске определяется с помощью трех «координат», то есть представляется триадой [c-h-s], где c – номер цилиндра (дорожки на поверхности диска, cylinder), h – номер рабочей поверхности диска (магнитной головки, head), a s – номер сектора на дорожке. Номер цилиндра c лежит в диапазоне 0. .C-1, где C – количество цилиндров. Номер рабочей поверхности диска h принадлежит диапазону 0..H-1, где H – число магнитных головок в накопителе. Номер сектора на дорожке s указывается в диапазоне 1. .S, где S – количество секторов на дорожке. Например, триада [1-0-2] адресует сектор 2 на дорожке O (обычно верхняя рабочая поверхность) цилиндра 1. В дальнейшем мы тоже будем пользоваться именно этими обозначениями.

Напомним, что обмен информацией между ОЗУ и дисками физически осуществляется только секторами. Вся совокупность физических секторов на винчестере представляет его неформатированную емкость.

Жесткий диск может быть" разбит на несколько разделов (partition), которые, в принципе затем могут использоваться либо одной ОС, либо различными ОС. Причем самым главным является то, что на каждом разделе может быть организована своя файловая система. Однако для организации даже одной-единственной файловой системы необходимо определить, по крайней мере, один раздел.

Разделы диска могут быть двух типов – primary (обычно этот термин переводят как первичный) и extended (расширенный). Максимальное число primary-разделов равно четырем. При этом на диске обязательно должен быть по крайней мере один primary-

раздел. Если primary-разделов несколько, то только один из них может быть активным. Именно загрузчику, расположенному в активном разделе, передается управление при включении компьютера и загрузке операционной системы. Остальные primary-разделы в этом случае считаются1 «невидимыми, скрытыми» (hidden).

Согласно спецификациям на одном жестком диске может быть только один ex-tended-раздел, который, в свою очередь, может быть разделен на большое количество подразделов – логических дисков (logical). В этом смысле термин «первичный» следует признать не совсем удачным переводом слова primary; можно это слово перевести и как «простейший, примитивный». В этом случае становится понятным и логичным термин extended.

Один из primary-разделов должен быть активным, именно с него должна загружаться программа загрузки операционной системы, или так называемый менеджер загрузки, назначение которого – загрузить программу загрузки ОС из какого-нибудь другого раздела, и уже с ее помощью загружать операционную систему. Поскольку до загрузки ОС система управления файлами работать не может, то следует использовать для указания упомянутых загрузчиков исключительно абсолютные адреса в формате [c-h-s].

По физическому адресу [0-0-1] на винчестере располагается главная загрузочная запись (master boot record, MBR), содержащая внесистемный загрузчик (non-system bootstrap – NSB), а также таблицу разделов (partition table, PT). Эта запись занимает ровно один сектор, она размещается в памяти, начиная с адреса 0:7C00h, после чего управление передается коду, содержащемуся в этом самом первом секторе магнитного диска. Таким образом, в самом первом (стартовом) секторе физического жесткого диска находится не обычная запись boot record, как на дискете, а master boot record.

MBR является основным средством загрузки с жесткого диска, поддерживаемым BIOS. В MBR находятся три важных элемента:

1. программа начальной загрузки (non-system bootstrap). Именно она запускается BIOS после успешной загрузки в память первого сектора с MBR. Она, очевидно, не превышает 512 байт и ее хватает только на то, чтобы загрузить следующую, чуть более сложную программу, обычно – стартовый сектор операционной системы – и передать ему управление;

2. таблица описания разделов диска (partition table). Располагается в MBR по смещению 0xlBE и занимает 64 байта;

3. сигнатура MBR. Последние два байта MBR должны содержать число AA55h. По наличию этой сигнатуры BIOS проверяет, что первый блок был загружен успешно. Сигнатура эта выбрана не случайно. Ее успешная проверка позволяет установить, что все линии передачи данных могут передавать и нули, и единицы.

Таблица *partition table* описывает размещение и характеристики имеющихся на винчестере разделов. Можно сказать, что эта таблица разделов – одна из наиболее важных структур данных на жестком диске. Если эта таблица повреждена, то не только не будет загружаться операционная система (или одна из операционных систем, установленных на винчестере), но перестанут быть доступными и данные, расположенные на винчестере, особенно если жесткий диск был разбит на несколько разделов.

Смещение (Offset)	Размер (Size) (байт)	Содержимое (Contents)
0	446	Программа анализа Partition Table и загрузки System Bootstrap с активного раздела жесткого диска
+1BEh	16	Partition 1 entry (Описатель раз дела)
+1CEh	16	Partition 2 entry
+1DEh	16	Partition 3 entry

+1EEh	16	Partition 4 entry
+1FEh	2	Сигнатура (AA55h)

Рис. 3.4. Структура MBR

Упрощенно структура MBR представлена на рис. 4.4. Из нее видно, что в начале этого сектора располагается программа анализа таблицы разделов и чтения первого сектора из активного раздела диска. Сама таблица partition table располагается в конце MBR, и для описания каждого раздела в этой таблице отводится по 16 байтов. Первым байтом в элементе раздела идет флаг активности раздела boot indicator (0 – не активен, 128 (80H) – активен). Он служит для определения, является ли раздел системным загрузочным и есть ли необходимость производить загрузку операционной системы с него при старте компьютера. Активным может быть только один раздел. За флагом активности раздела следует байт номера головки, с которой начинается раздел. За ним следует два байта, означающие соответственно номер сектора и номер цилиндра загрузочного сектора, где располагается первый сектор загрузчика операционной системы. Затем следует кодовый идентификатор System ID (длиной в один байт), указывающий на принадлежность данного раздела к той или иной операционной системе и установке на нем соответствующей файловой системы. В табл. 4.1 приведены некоторые (наиболее известные) идентификаторы.

Таблица 3.1. Сигнатуры (типы) разделов

System ID	Тип раздела	System ID	Тип раздела
00	Empty («пустой» раздел)	41	PPC PreP Boot
01	FAT12	42	SFS
02	XENIX root	4D	QNX 4.x
03	XENIX usr	4E	QNX 4.x 2nd part
04	FAT16 (<32 Мбайт)	4F	QNX 4,x 3nd part
05	Extended	50	OnTrack DM
06	FAT16	51	OnTrack DM6 Aux
07	HPFS/NTFS	52	CP/M
08	AIX	53	OnTrack DM6
09	AIX bootable	54	OnTrack DM6
0A	OS/2 Boot Manager	55	EZ Drive
0B	Win95 FAT32	56	Golden Bou
0C	Win95 FAT32 LBA	5C	Priam Edisk
0E	Win95 FAT16 LBA	61	Speed Stor
0F	Win95 Extended	64	Novell Netware
10	OPUS	65	Novell Netware
11	Hidden FAT12	75	PC/IX
12	Compaq diagnost	80	Old Minix
14	Hidden FAT16 (<32 Мбайт)	82	Linux swap
16	Hidden FAT16	83	Linux native
17	Hidden HPFS/NTFS	84	OS/2 hidden C:
18	AST Windows swap	85	Linux Extended
1B	Hidden Win95 Fat	86	NTFS volume set
1C	Hidden Win95 Fat	A5	BSD/386
1E	Hidden Win95 Fat	A6	Open BSD
24	NEC DOS	A7	Next Step

3C	Partition Magic	EB	BeOS
40	Venix 80286		

За байтом кода операционной системы расположен байт номера головки конца раздела, за которым идут два байта – номер сектора и номер цилиндра последнего сектора данного раздела. Ниже представлен формат элемента таблицы разделов.

Таблица 3.2. Формат элемента таблицы разделов

Название записи элемента Partition Table	Длина, байт
Флаг активности раздела	1
Номер головки начала раздела	1
Номер сектора и номер цилиндра загрузочного сектора раздела	2
Кодовый идентификатор операционной системы	1
Номер головки конца раздела	1
Номер сектора и цилиндра последнего сектора раздела	2
Младшее и старшее двухбайтовое слово относительного номера начального сектора	4
Младшее и старшее двухбайтовое слово размера раздела в секторах	4

Номера сектора и номер цилиндра секторов в разделах занимают по 6 и 10 бит соответственно. Ниже представлен формат записи, содержащей номера сектора и цилиндра.

Биты номера цилиндра										Биты номера секторов					
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Как мы уже сказали, загрузчик non-system bootstrap служит для поиска с помощью partition table активного раздела, копирования в оперативную память компьютера загрузчика system bootstrap из выбранного раздела и передачи ему управления, что позволяет осуществить загрузку ОС.

Вслед за сектором MBR размещаются собственно сами разделы (рис. 3.5). В процессе начальной загрузки сектора MBR, содержащего таблицу partition table, работают программные модули BIOS. Начальная загрузка считается выполненной корректно только в том случае, когда таблица разделов содержит допустимую информацию.

В MS-DOS в первичном разделе может быть сформирован только один логический диск, а в расширенном – любое их количество. Каждый логический диск «управляется» своим логическим приводом. Каждому логическому диску на винчестере соответствует своя (относительная) логическая нумерация. Физическая же адресация жесткого диска сквозная.

Первичный раздел DOS включает только системный логический диск без каких-либо дополнительных информационных структур.

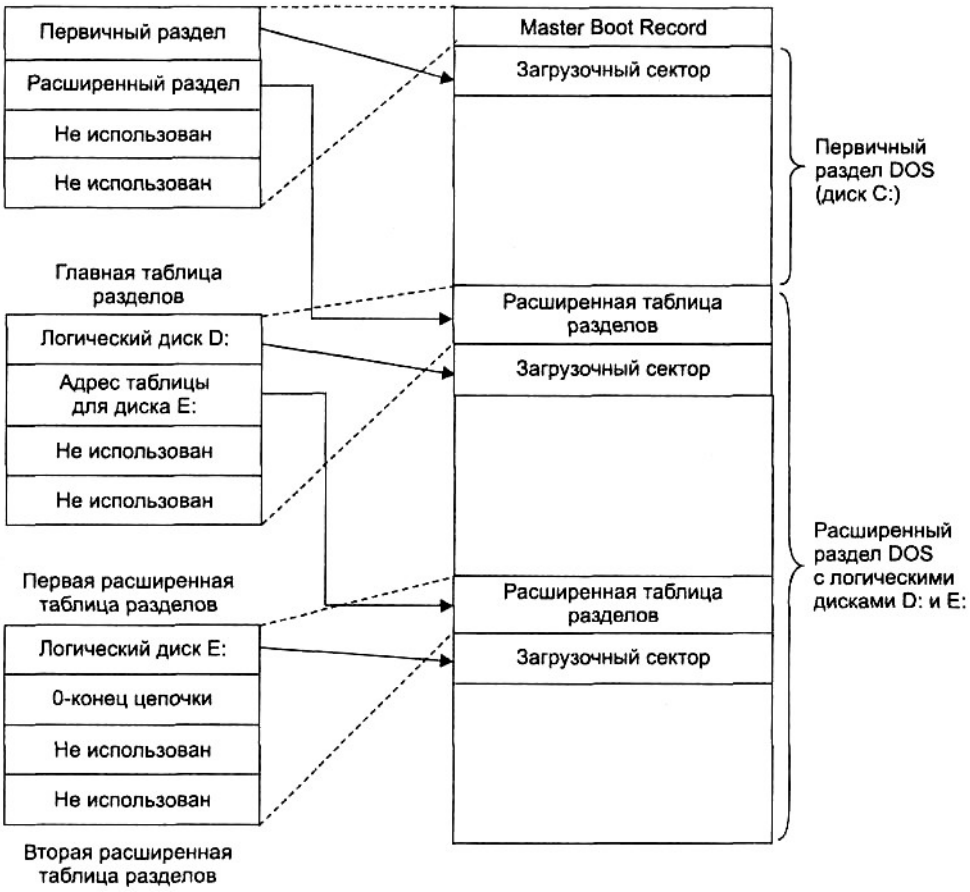

Рис. 3.5. Разбиение диска на разделы

Расширенный раздел DOS содержит вторичную запись MBR (secondary MBR, SMBR), в состав которой вместо partition table входит таблица логического диска (LDT, logical disk table), ей аналогичная. Таблица LDT описывает размещение и характеристики раздела, содержащего единственный логический диск, а также может специфицировать следующую запись SMBR. Следовательно, если в расширенном разделе DOS создано К логических дисков, то он содержит К экземпляров SMBR, связанных в список. Каждый элемент этого списка описывает соответствующий логический диск и ссылается (кроме последнего) на следующий элемент списка.

Утилиты, позволяющие разбить диск на разделы и тем самым сформировать как partition table, так и, возможно, списковую структуру из LDT (подобно тому, как это изображено на рис. 4.6), обычно называются FDISK (от form disk), хотя есть утилиты и с другим названием, умеющие выполнить разбиение диска. Напомним, что FDISK от MS-DOS позволяет создать только один primary-раздел и один extended, который, в свою очередь, предлагается разделить на несколько логических дисков. Аналогичные утилиты других ОС имеют существенно больше возможностей. Так, например, FDISK системы OS/2 позволяет создавать несколько primary-разделов, причем их можно выделять не только последовательно, начиная с первых цилиндров, но и с конца свободного дискового пространства. Это удобно, если нужно исключить из работы некоторый диапазон дискового пространства (например, из-за дефектов на поверхности магнитного диска). С помощью этой утилиты можно установить и менеджер загрузки (boot-менеджер).

Прежде чем перейти к boot-менеджерам, рассмотрим еще раз процесс загрузки ОС. Процедура начальной загрузки (bootstrap loader) вызывается как программное прерывание (BIOS INT 19h). Эта процедура определяет первое готовое устройство из списка разрешенных и доступных (гибкий или жесткий диск, в современных компьютерах это могут быть также CD-ROM, привод ZIP-drive компании iomega, сетевой адаптер или другое устройство) и пытается загрузить с него в ОЗУ короткую главную программу-загрузчик. Для винчестеров – это загрузчик non-system bootstrap из MBR, и ему передается

управление. Главный загрузчик определяет на диске активный раздел, загружает его собственный загрузчик (system bootstrap) и передает управление ему. И, наконец, этот загрузчик загружает необходимые файлы операционной системы и передает ей управление. Далее операционная система выполняет инициализацию подведомственных ей программных и аппаратных средств. Она добавляет новые сервисы, вызываемые, как правило, тоже через механизм программных прерываний, и расширяет (или заменяет) некоторые сервисы BIOS. Необходимо отметить, что в современных мультипрограммных операционных системах большинство сервисов BIOS, изначально расположенных в ПЗУ, как правило, заменяются собственными драйверами, поскольку они должны работать в режиме прерываний, а не в режиме сканирования готовности.

Соответственно рассмотренному процессу загрузки, мы каждый раз при запуске компьютера будем попадать в одну и ту же ОС. Но иногда это нас не устраивает. Так называемые boot-менеджеры (менеджеры загрузки) предназначены для того, чтобы пользователь мог выбрать среди нескольких установленных на компьютере ОС нужную и передать управление загрузчику этой выбранной ОС. Существует большое количество таких менеджеров, например, System commander. Однако этот менеджер должен располагаться в активном разделе с файловой системой FAT, что в наше время нельзя признать хорошим решением. На наш взгляд, одним из лучших долгое время был Boot manager компании IBM, входящий в состав утилит OS/2. Для его размещения создается отдельный primary-раздел, который, естественно, и является активным (он помечается как startable). Раздел для Boot-менеджера OS/2 занимает всего один цилиндр и может размещаться не только на нулевом (начальном) цилиндре, но и на последнем цилиндре. В этом разделе не организуется никакой файловой системы, поскольку обращение к менеджеру идет с использованием абсолютной адресации, а сам Boot manager представляет собой простейшую абсолютную двоичную программу. Установка Boot manager осуществляется из программы FDISK. При этом появляется возможность указать, какие разделы могут быть загружаемыми (они помечаются как bootable), то есть в каких разделах на первом логическом секторе будут размещены программы загрузки ОС. Загружаемыми могут быть как primary-, так и extended-разделы. При этом, естественно, имеется возможность указать как время на принятие решения, так и загрузку некоторой ОС «по умолчанию». Удобным является и то, что при разбиении диска на разделы можно вообще не создавать primary-разделов. Это особенно важно, если в компьютере установлено более одного дискового накопителя, либо если мы подготавливаем винчестер, который должен достаточно часто переноситься с одного компьютера на другой. Дело в том, что в большинстве ОС принято следующее правило именования логических дисков: первый логический диск обозначается литерой C:, второй – D: и т. д. При этом литеру C: получает активный primary-раздел. Однако, если к одному винчестеру в персональном компьютере подключить второй винчестер и он тоже имеет активный primary-раздел, то этот primary-раздел второго винчестера получит литеру D:, отодвигая логический диск D: первого винчестера на место диска E: (и т. д. по цепочке). Если же второй (и последующие) винчестер не имеет primary-раздела с установленной на нем файловой системой, которую данная ОС знает, то «именование» логических дисков первого винчестера не нарушается. Естественно, что логические диски второго винчестера получают литеры логических дисков вслед за дисками первого винчестера. Boot manager OS/2, создавая только логические диски на винчестере, на самом деле создает и «пустой» primary-раздел, однако этот раздел не становится активным и не получает статус логического диска. К сожалению, все более популярная в наши дни ОС Windows 2000 теперь не только снимает флаг активности с раздела, в котором размещен Boot manager (как это происходило при инсталляции любых ОС от компании Microsoft), но и физически уничтожает его двоичный код. Замена драйвера FASTFAT.SYS системы Windows 2000 на более раннюю версию (в бета-версии ОС Windows 2000 система не уничтожала код Boot manager) помогает лишь до установки Service pack. Поэтому рекомендовать этот

менеджер загрузки уже нельзя. Из последних менеджеров загрузки, пожалуй, наиболее мощным является Boot star, но его нельзя рекомендовать неподготовленным пользователям.

Одной из самых известных и до сих пор достаточно часто используемых утилит, с помощью которой можно посмотреть и отредактировать таблицу разделов, а также выполнить и другие действия, связанные с изучением и исправлением данных как на дискете, так и на винчестере, является программа Disk editor из комплекта нортоновских утилит. Следует заметить, что Disk editor, с одной стороны, очень мощное средство, и поэтому его следует использовать с большой осторожностью, а с другой стороны – эта утилита работает только в среде DOS. На сегодняшний день главным недостатком этой утилиты является ограничение на максимальный размер диска в 8 Гбайт.

Надо признать, что в последнее время появилось большое количество утилит, которые предоставляют возможность более наглядно представить разбиение диска на разделы, поскольку в них используется графический интерфейс. Эти программы успешно и корректно работают с наиболее распространенными типами разделов (разделы под FAT, FAT32, NTFS). Однако созданы они в основном для работы в среде Win32API, что часто ограничивает возможность их использования. Одной из самых известных и мощных программ для работы с разделами жесткого диска является Partition Magic фирмы Power Quest.

Файловая система FAT

Как мы уже отмечали, аббревиатура FAT (file allocation table) расшифровывается как «таблица размещения файлов». Этот термин относится к линейной таб-личной структуре со сведениями о файлах – именами файлов, их атрибутами и другими данными, определяющими местонахождение файлов (или их фрагментов) в среде FAT. Элемент FAT определяет фактическую область диска, в которой хранится начало физического файла.

В файловой системе FAT логическое дисковое пространство любого логического диска делится на две области (рис. 3.6): системную область и область данных.

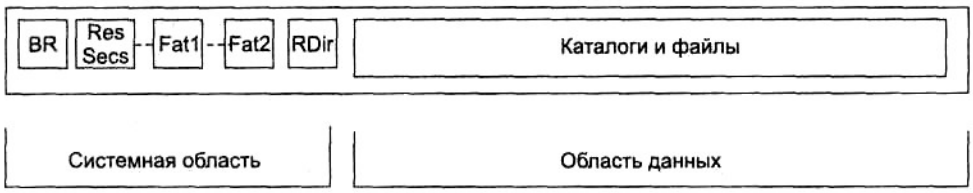

Рис. 3.6. Структура логического диска

Системная область логического диска создается и инициализируется при форматировании, а впоследствии обновляется при манипулировании файловой структурой. Область данных логического диска содержит файлы и каталоги, подчиненные корневому. Она, в отличие от системной области, доступна через пользовательский интерфейс DOS. Системная область состоит из следующих компонентов, расположенных в логическом адресном пространстве подряд:

– загрузочной записи (boot record, BR);
– зарезервированных секторов (reserved sector, ResSecs);
– таблицы размещения файлов (file allocation table, FAT);
– корневого каталога (root directory, RDir).

Таблица размещения файлов

Таблица размещения файлов является очень важной информационной структурой. Можно сказать, что она представляет собой карту (образ) области данных, в которой описывается состояние каждого участка области данных. Область данных разбивают на так называемые кластеры. Кластер представляет собой один или несколько смежных

секторов в логическом дисковом адресном пространстве (точнее – только в области данных). В таблице FAT кластеры, принадлежащие одному файлу (некорневому каталогу), связываются в цепочки. Для указания номера кластера в системе управления файлами FAT-16 используется 16-битовое слово, следовательно, можно иметь до 216 = 65536 кластеров (с номерами от 0 до 65535).

Кластер – это минимальная адресуемая единица дисковой памяти, выделяемая файлу (или некорневому каталогу). Файл или каталог занимает целое число кластеров. Последний кластер при этом может быть задействован не полностью, что приведет к заметной потере дискового пространства при большом размере кластера. На дискетах кластер занимает один или два сектора, а на жестких дисках – в зависимости от объема раздела (см. табл. 3.3).

Таблица 3.3. Соотношения между размером раздела и размером кластеров в FAT16

Емкость раздела, Мбайт	Количество секторов в кластере	Размер кластеров, Кбайт
16-127	4	2
128-255	8	4
256-511	16	8
512-1023	32	16
1024-2047	64	32

Номер кластера всегда относится к области данных диска (пространству, зарезервированному для файлов и подкаталогов). Первый допустимый номер кластера всегда начинается с 2. Номера кластеров соответствуют элементам таблицы размещения файлов.

Логическое разбиение области данных на кластеры как совокупности секторов взамен использования одиночных секторов имеет следующий смысл: прежде всего, уменьшается размер самой таблицы FAT; уменьшается возможная фрагментация файлов; ускоряется доступ к файлу, так как в несколько раз сокращается длина цепочек фрагментов дискового пространства, выделенных для него.

Однако слишком большой размер кластера ведет к неэффективному использованию области данных, особенно в случае большого количества маленьких файлов. Как мы только что заметили, в среднем на каждый файл теряется около половины кластера. Из табл. 4.3 следует, что при размере кластера в 32 сектора (объем раздела – от 512 Мбайт до 1023 Мбайт), то есть 16 Кбайт, средняя величина потерь на файл составит 8 Кбайт, и при числе файлов в несколько тысяч1 потери могут составлять более 100 Мбайт. Поэтому в современных файловых системах (к ним, прежде всего, следует отнести HPFS, NTFS, FAT32 и некоторые другие, поддерживаемые ОС семейства UNIX) размеры кластеров ограничиваются (обычно – от 512 байт до 4 Кбайт). В FAT32 проблема решается за счет того, что собственно сама FAT в этой файловой системе может содержать до 228 кластеров2. Наконец, заметим, что системы управления файлами, созданные для Windows 9x и Windows NT, могут работать с разделами размером до 4 Гбайт, на которых установлена система FAT, тогда как DOS, естественно, с такими разделами работать не сможет.

Достаточно наглядно идея файловой системы с использованием таблицы размещения файлов FAT проиллюстрирована рис. 3.7. Из этого рисунка видно, что файл с именем MYFILE.TXT размещается, начиная с восьмого кластера. Всего файл MYFILE.TXT занимает 12 кластеров. Цепочка кластеров (chain) для нашего примера может быть записана следующим образом: 8, 9, 0A, 0B, 15, 16, 17, 19, 1A, 1B, 1C, 1D. Кластер с номером 18 помечен специальным кодом F7 как плохой (bad), он не может быть использован для размещения данных. При форматировании обычно проверяется поверхность магнитного диска, и те секторы, при контрольном чтении с которых происходили ошибки, помечаются в FAT как плохие. Кластер 1D помечен кодом FF как

конечный (последний в цепочке) кластер, принадлежащий данному файлу. Свободные (незанятые) кластеры помечаются кодом 00; при выделении нового кластера для записи файла берется первый свободный кластер. Поскольку файлы на диске изменяются – удаляются, перемещаются, увеличиваются или уменьшаются, – то упомянутое правило выделения первого свободного кластера для новой порции данных приводит к фрагментации файлов, то есть данные одного файла могут располагаться не в смежных кластерах, а, порой, в очень удаленных друг от друга, образуя сложные цепочки. Естественно, что это приводит к существенному замедлению работы с файлами.

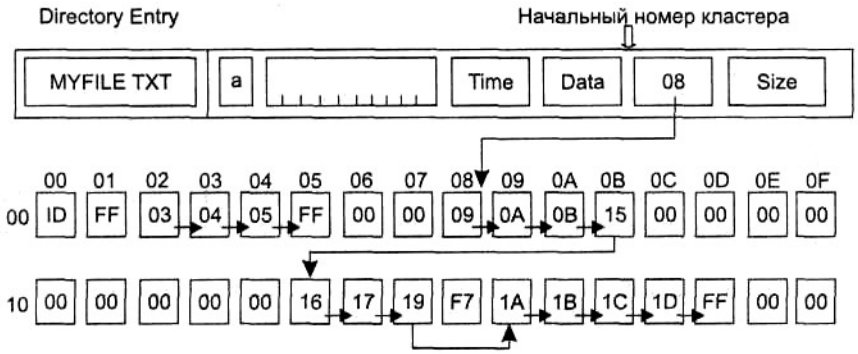

Рис. 3.7. Основная концепция FAT

Так как FAT используется при доступе к диску очень интенсивно, она обычно загружается в ОЗУ (в буфера ввода/вывода или кэш) и остается там настолько долго, насколько это возможно.

В связи с чрезвычайной важностью FAT она обычно хранится в двух идентичных экземплярах, второй из которых непосредственно следует за первым. Обновляются копии FAT одновременно. Используется же только первый экземпляр. Если он по каким-либо причинам окажется разрушенным, то произойдет обращение ко второму экземпляру. Так, например, утилита проверки и восстановления файловой структуры ScanDisk из ОС Windows 9x при обнаружении несоответствия первичной и резервной копии FAT предлагает восстановить главную таблицу, используя данные из копии.

Упомянутый корневой каталог отличается от обычного каталога тем, что он, помимо размещения в фиксированном месте логического диска, еще имеет и фиксированное число элементов. Для каждого файла и каталога в файловой системе хранится информация в соответствии со структурой, изображенной в табл. 3.4.

Таблица 3.4. Элемент каталога

Размер поля данных, байт	Содержание поля
11	Имя файла или каталога
1	Атрибуты файла
1	Резервное поле
3	Время создания
2	Дата создания
2	Дата последнего доступа
2	Зарезервировано
2	Время последней модификации
2	Дата последней модификации
2	Номер начального кластера в FAT
4	Размер файла

Структура системы файлов является иерархической. Это иллюстрируется рис. 34.8, из которого видно, что элементом каталога может быть такой файл, который сам, в свою очередь, является каталогом.

Рис. 4.8. Структура системы файлов

Структура загрузочной записи DOS

Сектор, содержащий загрузочную запись, является самым первым на логическом диске (на дискете – имеет физический адрес [0-0-1]). Boot Record состоит, как мы уже знаем, из двух частей – disk parameter block (DPB) и system bootstrap (SB). Структура блока параметров диска (DPB) служит для идентификации физического и логического форматов логического диска, а загрузчик system bootstrap играет существенную роль в процессе загрузки DOS. Эта информационная структура приведена в табл. 3.5.

Таблица 3.5. Структура загрузочной записи Boot Record для FAT 16

Смещение поля, байт	Длина поля, байт	Обозначение поля	Содержимое поля
ООН (0)	3	JUMP 3EH	Безусловный переход на начало SB
0ЗН (3)	8		Системный идентификатор
ОВН(11)	2	SectSize	Размер сектора, байт
0DH (13)	1	ClastSize	Число секторов в кластере
0ЕН (14)	2	ResSecs	Число зарезервированных секторов
ЮН (16)	1	FATcnt	Число копий FAT
1Ш(17)	2	RootSize	Максимальное число элементов Rdir
13Н (19)	2	TotSecs	Число секторов на логическом диске, если его размер не превышает 32 Мбайт; иначе 0000Н
15Н (21)	1	Media	Дескриптор носителя
16Н (22)	2	FATsize	Размер FAT, секторов
18Н (24)	2	TrkSecs	Число секторов на дорожке
1АН (26)	2	HeadCnt	Число рабочих поверхностей
1СН (28)	4	HidnSecs	Число скрытых секторов
20Н (32)	4		Число секторов на логическом диске, если его размер превышает 32 Мбайт

24H (36)	1		Тип логического диска (OOH – гибкий, 80H – жесткий)
25H (37)	1		Пусто (резерв)
26H (38)	1		Маркер с кодом 29H
27H (39)	4		Серийный номер тома
2BH (43)	11		Метка тома
36H(54)	8		Имя файловой системы
3EH(62)			System bootstrap
1FEH(510)	2		Сигнатура (слово AA55H)

Первые два байта boot record занимает JMP – команда безусловного перехода в программу SB. Третий байт содержит код 90H (NOP – нет операции). Далее располагается восьмибайтовый системный идентификатор, включающий информацию о фирме-разработчике и версии операционной системы. Затем следует DPB, а после него – SB.

Для работы с загрузочной записью удобно использовать широко известную утилиту Disk Editor из комплекта утилит Питера Нортона. Эта утилита снабжена встроенной системой подсказок и необходимой справочной информацией. Используя ее, можно сохранять, модифицировать и восстанавливать загрузочную запись, а также выполнять много других операций.

Загрузочные записи других операционных систем отличаются от рассмотренной. Так, например, в загрузочном секторе для тома с FAT32 в блоке DPB содержатся дополнительные поля, а те поля, что находятся в привычном для системы FAT16 месте, перенесены. Поэтому ОС, в которой имеется возможность работать с файловой системой FAT16, но нет системы управления файлами, понимающей спецификации FAT32, не может читать данные с томов, отформатированных под файловую систему FAT32. В загрузочном секторе для файловой системы FAT32 по-прежнему байты OOH по OAH содержат команду перехода и OEM ID, а в байтах OBH по 59H содержатся данные блока DPB. Отличие заключается именно в несколько другой структуре блока DBP; его содержимое приведено в табл. 3.6.

Таблица 3.6. Структура загрузочной записи boot record для FAT32

Смещение поля, байт	Длина поля, байт	Обозначение поля	Содержимое поля
OOH (0)	3	JUMP 3EH	Безусловный переход на начало SB
03H (3)	8		Системный идентификатор
0BH (11)	2	SectSize	Размер сектора, байт
0DH (13)	1	ClastSize	Число секторов в кластере
0EH(14)	2	ResSecs	Число зарезервированных секторов, для FAT32 равно 32
10H (16)	1	FATcnt	Число копий FAT
11H (17)	2	RootSize	0000H
13H (19)	2	TotSecs	0000H
15H (21)	1	Media	Дескриптор носителя
16H (22)	2	FATsize	0000H
18H (24)	2	TrkSecs	Число секторов на дорожке
1AH (26)	2	HeadCnt	Число рабочих поверхностей

1CH (28)	4	HidnSecs	Число скрытых секторов (располагаются перед загрузочным сектором). Используется при загрузке для вычисления абсолютного смещения корневого каталога и данных
20H (32)	4		Число секторов на логическом диске
24H (36)	4		Число секторов в таблице FAT
28H (37)	2		Расширенные флаги
2AH (38)	2		Версия файловой системы
2CH (39)	4		Номер кластера для первого кластера корневого каталога
34H (43)	2		Номер сектора с резервной копией загрузочного сектора
36H (54)	12		Зарезервировано

Файловые системы VFAT и FAT32

Одной из важнейших характеристик исходной FAT было использование имен файлов формата «8.3», в котором 8 символов отводится на указание имени файла и 3 символа – для расширения имени. К стандартной FAT (имеется в виду прежде всего реализация FAT 16) добавились еще две разновидности, используемые в широко распространенных операционных системах Microsoft (конкретно – в Windows 95 и Windows NT): VFAT (виртуальная FAT) и FAT32, используемая в одной из редакций ОС Windows 95 и Windows 98. Ныне эта файловая система (FAT32) поддерживается и такими ОС, как Windows Millennium Edition, и всеми ОС семейства Windows 2000. Имеются реализации систем управления файлами FAT32 для Windows NT и ОС Linux.

Файловая система VFAT впервые появилась в Windows for Workgroups 3.11 и была предназначена для выполнения файлового ввода/вывода в защищенном режиме. С выходом Windows 95 в VFAT добавилась поддержка длинных имен файлов (long file name, LFN). Тем не менее, VFAT сохраняет совместимость с исходным вариантом FAT; это означает, что наряду с длинными именами в ней поддерживаются имена формата «8.3», а также существует специальный механизм для преобразования имен «8.3» в длинные имена, и наоборот. Именно файловая система VFAT поддерживается исходными версиями Windows 95, Windows NT 4. При работе с VFAT крайне важно использовать файловые утилиты, поддерживающие VFAT вообще и длинные имена в частности. Дело в том, что более ранние файловые утилиты DOS запросто модифицируют то, что кажется им исходной структурой FAT. Это может привести к потере или порче длинных имен из таблицы FAT, поддерживаемой VFAT (или FAT32). Следовательно, для томов VFAT необходимо пользоваться файловыми утилитами, которые понимают и сохраняют файловую структуру VFAT.

В исходной версии Windows 95 основной файловой системой была 32-разрядная VFAT. VFAT может использовать 32-разрядные драйверы защищенного режима или 16-разрядные драйверы реального режима. При этом элементы FAT остаются 12- или 16-разрядными, поэтому на диске используется та же структура данных, что и в предыдущих реализациях FAT. VFAT обрабатывает все обращения к жесткому диску и использует 32-разрядный код для всех файловых операций с дисковыми томами.

Основными недостатками файловых систем FAT и VFAT являются большие потери на кластеризацию при больших размерах логического диска и ограничения на сам размер логического диска. Это привело к разработке новой реализации файловой системы с

использованием той же идеи использования таблицы FAT. Поэтому в Microsoft Windows 95 OEM Service Release 2 (эта версия Windows 95 часто называется Windows 95 OSR2) на смену системе VFAT пришла файловая система FAT32. FAT32 является полностью самостоятельной 32-разрядной файловой системой и содержит многочисленные усовершенствования и дополнения по сравнению с предыдущими реализациями FAT.

Принципиальное отличие заключается в том, что FAT32 намного эффективнее расходует дисковое пространство. Прежде всего, система FAT32 использует кластеры меньшего размера по сравнению с предыдущими версиями, которые ограничивались 65 535 кластерами на том (соответственно, с увеличением размера диска приходилось увеличивать и размер кластеров). Следовательно, даже для дисков размером до 8 Гбайт FAT32 может использовать 4-килобайтные кластеры. В результате по сравнению с дисками FAT16 экономится значительное дисковое пространство (в среднем 10-15 %).

FAT32 также может перемещать корневой каталог и использовать резервную копию FAT вместо стандартной. Расширенная загрузочная запись FAT32 позволяет создавать копии критических структур данных; это повышает устойчивость дисков FAT32 к нарушениям структуры FAT по сравнению с предыдущими версиями. Корневой каталог в FAT32 представлен в виде обычной цепочки кластеров. Следовательно, корневой каталог может находиться в произвольном месте диска, что снимает действовавшее ранее ограничение на размер корневого каталога (512 элементов).

Windows 95 OSR2 и Windows 98 могут работать и с разделами VFAT, созданными Windows NT. То, что говорилось ранее об использовании файловых утилит VFAT с томами VFAT, относится и к FAT32. Поскольку прежние утилиты FAT (для FAT32 в эту категорию входят обе файловые системы, FAT и VFAT) могут повредить или уничтожить важную служебную информацию, для томов FAT32 нельзя пользоваться никакими файловыми утилитами, кроме утилит FAT32.

Кроме повышения емкости FAT до величины в 4 Тбайт, файловая система FAT32 вносит ряд необходимых усовершенствований в структуру корневого каталога. Предыдущие реализации требовали, чтобы вся информация корневого каталога FAT находилась в одном дисковом кластере. При этом корневой каталог мог содержать не более 512 файлов. Необходимость представлять длинные имена и обеспечить совместимость с прежними версиями FAT привела разработчиков компании Microsoft к компромиссному решению: для представления длинного имени они стали использовать элементы каталога, в том числе и корневого. Рассмотрим способ представления в VFAT длинного имени файла (рис. 34.9).

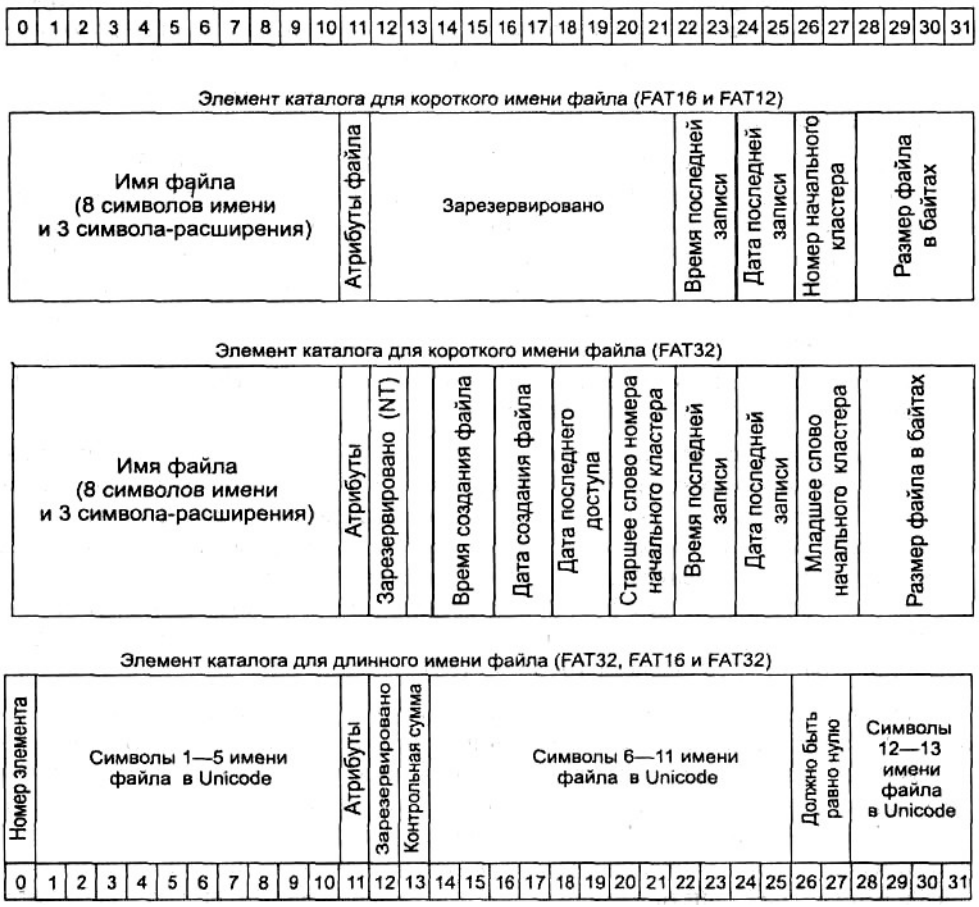

Рис. 3.9. Элементы каталогов для FAT, VFAT и FAT32.

Первые одиннадцать байтов элемента каталога DOS используются для хранения имени файла. Каждое такое имя разделяется на две части: в первых восьми байтах хранятся символы собственно имени, а в последних трех – символы так называемого расширения, с помощью которого реализуются механизмы предопределенных типов. Были введены соответствующие системные соглашения, и файлы определенного типа необходимо (желательно) именовать с оговоренным расширением. Например, исполняемые файлы с расширением COM определяют исполняемую двоичную программу с простейшей односегментной структурой1. Более сложные программы имеют расширение EXE. Определены расширения для большого количества типов файлов, и эти расширения используются для ассоциированного запуска обрабатывающих файлы программ.

Если имя файла состоит менее чем из восьми символов, то в элементе каталога оно дополняется символами пробела, чтобы полностью заполнить все восемь байтов соответствующего поля. Аналогично и расширение может содержать от нуля до трех символов. Остальные (незаполненные) позиции в элементе каталога, определяющем расширение имени файла, заполняются символами пробела. Поскольку при работе с именем файла учитываются все одиннадцать свободных мест, то необходимость в отображении точки, которая обычно вводится между именем файла и его расширением, отпадает. В элементе каталога она просто подразумевается.

В двенадцатом байте элемента каталога хранятся атрибуты файла. Шесть из восьми указанных разрядов используются DOS2. К атрибутам DOS относятся следующие:

– атрибут «архивный» (A – archive). Показывает, что файл был открыт программой таким образом, чтобы у нее была возможность изменить содержимое этого файла. DOS устанавливает этот разряд атрибута в состояние ON (включено) при открытии файла. Программы резервного копирования нередко устанавливают его в OFF (выключено) при выполнении резервного копирования файла. Если применяется подобная методика, то в

следующую создаваемую по порядку резервную копию будут добавлены только файлы с данным разрядом, установленным в состояние ON;

– атрибут каталога (D – directory). Показывает, что данный элемент каталога указывает на подкаталог, а не на файл;

– атрибут тома (V – volume). Применяется только к одному элементу каталога в корневом каталоге. В нем, собственно, и хранится имя дискового тома. Этот атрибут также применяется в случае длинных имен файлов, о чем можно будет узнать из следующего раздела;

– атрибут «системный» (S – system). Показывает, что файл является частью операционной системы или специально отмечен подобным образом прикладной программой, что иногда делается в качестве составной части метода защиты от копирования;

– атрибут «скрытый» (H – hidden). Сюда относятся, в частности, файлы с установленным в состояние ON атрибутом «системный» (S), которые не отображаются в обычном списке, выводимом по команде DIR;

– атрибут «только для чтения» (R – read only). Показывает, что данный файл не подлежит изменению. Разумеется, поскольку это лишь разряд байта, хранящегося на диске, то любая программа может изменить этот разряд, после чего DOS свободно разрешила бы изменение данного файла. Этот атрибут в основном используется для примитивной защиты от пользовательских ошибок, то есть он позволяет избежать неумышленного удаления или изменения ключевых файлов.

Следует отметить, что наличие файла, помеченного одним или более из указанных выше атрибутов, может иметь вполне определенный смысл. Например, большинство файлов, отмечаемых в качестве системных, отмечаются также признаками скрытых файлов и «только для чтения».

На дисках FAT12 или FAT16 следующие за именем десять байтов не используются. Обыкновенно они заполняются нулями и считаются резервными значениями. А на диске с файловой системой FAT32 эти 10 байт содержат самую разную информацию о файле. При этом байт, отмеченный как «зарезервировано для NT», представляет собой, как подразумевает его название, поле, не используемое в DOS или Windows 9x, но применяемое в Windows NT.

Из соображений совместимости поля, которые встречаются в элементах каталога для коротких имен формата FAT 12 и FAT 16, находятся на тех же местах и в элементах каталога для коротких имен формата FAT32. Остальные поля, которые встречаются только в элементах каталога для коротких имен формата FAT32, соответствуют зарезервированной области длиной в десять байт в элементах каталога для коротких имен форматов FAT 12 и FAT 16.

Как видно из рис. 3.9, для длинного имени файла используется несколько элементов каталога. Таким образом, появление длинных имен фактически привело к дальнейшему уменьшению количества файлов, которые могут находиться в корневом каталоге. Поскольку длинное имя может содержать до 256 символов, всего один файл с полным длинным именем занимает до 25 элементов FAT (1 для имени 8.3 и еще 24 для самого длинного имени). Количество элементов корневого каталога VFAT уменьшается до 21. Очевидно, что это не самое изящное решение, поэтому компания Microsoft советует избегать длинных имен в корневых каталогах FAT при отсутствии FAT32, у которой количество элементов каталога соответственно просто увеличено. Помните и о том, что длина полной файловой спецификации, включающей путь и имя файла (длинное или в формате 8.3), тоже ограничивается 260 символами. FAT32 успешно справляется с проблемой длинных имен в корневом каталоге, но проблема с ограничением длины полной файловой спецификации остается. По этой причине Microsoft рекомендует ограничивать длинные имена 75-80 символами, чтобы оставить достаточно места для пути (180-185 символов).

Файловая система HPFS

Сокращение HPFS расшифровывается как «High Performance File System» – высокопроизводительная файловая система. HPFS впервые появилась в OS/2 1.2 и LAN Manager. HPFS была разработана совместными усилиями лучших специалистов компании IBM и Microsoft на основе опыта IBM по созданию файловых систем MVS, VM/CMS и виртуального метода доступа. Архитектура HPFS начала создаваться как файловая система, которая сможет использовать преимущества многозадачного режима и обеспечит в будущем более эффективную и надежную работу с файлами на дисках большого объема.

HPFS была первой файловой системой для ПК, в которой была реализована поддержка длинных имен. HPFS, как FAT и многие другие файловые системы, обладает структурой каталогов, но в ней также предусмотрены автоматическая сортировка каталогов и специальные расширенные атрибуты, упрощающие реализацию безопасности файлового уровня и создание множественных имен. HPFS поддерживает те же самые атрибуты, что и файловая система FAT, по историческим причинам, но также поддерживает и новую форму file-associated, то есть информацию, называемую расширенными атрибутами (EAs). Каждый EA концептуально подобен переменной окружения. Но самым главным отличием все же являются базовые принципы хранения информации о местоположении файлов.

Принципы размещения файлов на диске, положенные в основу HPFS, увеличивают как производительность файловой системы, так и ее надежность и отказоустойчивость. Для достижения этих целей предложено несколько способов: размещение каталогов в середине дискового пространства, использование методов бинарных сбалансированных деревьев для ускорения поиска информации о файле, рассредоточение информации о местоположении записей файлов по всему диску, при том что записи каждого конкретного файла размещаются (по возможности) в смежных секторах и поблизости от данных об их местоположении. Действительно, система HPFS стремится, прежде всего, к тому, чтобы расположить файл в смежных кластерах, или, если такой возможности нет, разместить его на диске таким образом, чтобы экстенты (фрагменты) файла физически были как можно ближе друг к другу. Такой подход существенно уменьшает время позиционирования головок записи/чтения жесткого диска и время ожидания (rotational latency) – задержка между установкой головки чтения/записи на нужную дорожку диска и началом чтения данных с диска)1. Можно сказать, что файловая система HPFS имеет, по сравнению с FAT, следующие основные преимущества:

– высокая производительность;

– надежность;

– работа с расширенными атрибутами, что позволяет управлять доступом к файлам и каталогам;

– эффективное использование дискового пространства.

Все эти преимущества обусловлены структурой диска HPFS. Рассмотрим ее более подробно (рис. 3.10).

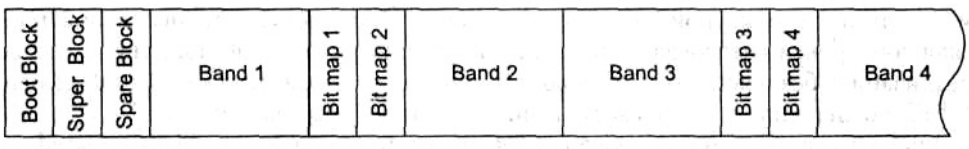

Рис. 3.10. Структура раздела HPFS

В начале диска расположено несколько управляющих блоков. Все остальное дисковое пространство в HPFS разбито на части («полосы», «ленты» из смежных секторов, в оригинале – band). Каждая такая группа данных занимает на диске пространство в 8 Мбайт и имеет свою собственную битовую карту распределения секторов. Эти битовые карты показывают, какие секторы данной полосы заняты, а какие – свободны. Каждому сектору ленты данных соответствует один бит в ее битовой карте. Если бит имеет значение 1, то соответствующий сектор занят, а если 0 – свободен.

Битовые карты двух полос располагаются на диске рядом, так же располагаются и сами полосы. То есть последовательность полос и карт выглядит следующим образом: битовая карта, битовая карта, лента с данными, лента с данными, битовая карта, битовая карта и т. д. Такое расположение «лент» позволяет непрерывно разместить на жестком диске файл размером до 16 Мбайт и в то же время не удалять от самих файлов информацию об их местонахождении. Это иллюстрируется рис. 3.10.

Очевидно, что если бы на весь диск была только одна битовая2 карта, как это сделано в FAT, то для работы с ней приходилось бы перемещать головки чтения/записи в среднем через половину диска. Именно для того, чтобы избежать этих потерь, в HPFS и разбит диск на «полосы». Получается своего рода распределенная структура данных об используемых и свободных блоках.

Дисковое пространство в HPFS выделяется не кластерами, как в FAT, а блоками. В современной реализации размер блока взят равным одному сектору, но в принципе он мог бы быть и иного размера. По сути дела, блок – это и есть кластер. Размещение файлов в таких небольших блоках позволяет более эффективно использовать пространство диска, так как непроизводительные потери свободного места составляют в среднем всего 256 байт на каждый файл. Вспомните, что чем больше размер кластера, тем больше места на диске расходуется напрасно. Например, кластер на отформатированном под FAT диске объемом от 512 до 1024 Мбайт имеет размер 16 Кбайт. Следовательно, непродуктивные потери свободного пространства на таком разделе в среднем составляют 8 Кбайт (8192 байт) на один файл, в то время как на разделе HPFS эти потери всегда будут составлять всего 256 байт на файл. Таким образом, на каждый файл экономится почти 8 Кбайт.

На рис. 4.10 показано, что помимо «лент» с записями файлов и битовых карт в томе с HPFS имеются еще три информационные структуры. Это так называемый загрузочный блок (boot block), дополнительный блок (super block) и запасной (резервный) блок (spare block). Загрузочный блок (boot block) располагается в секторах с 0 по 15; он содержит имя тома, его серийный номер, блок параметров BIOS и программу начальной загрузки. Программа начальной загрузки находит файл OS2LDR, считывает его в память и передает управление этой программе загрузки ОС, которая, в свою очередь, загружает с диска в память ядро OS/2 – OS2KRNL. И уже OS2KRNL с помощью сведений из файла CONFIG.SYS загружает в память все остальные необходимые программные модули и блоки данных.

В блоке (super block) содержится указатель на список битовых карт (bitmap block list). В этом списке перечислены все блоки на диске, в которых расположены битовые карты, используемые для обнаружения свободных секторов. Также в дополнительном блоке хранится указатель на список дефектных блоков (bad block list), указатель на группу каталогов (directory band), указатель на файловый узел (F-node) корневого каталога, а также дата последней проверки раздела программой CHKDSK. В списке дефектных блоков перечислены все поврежденные секторы (блоки) диска. Когда система обнаруживает поврежденный блок, он вносится в этот список и для хранения информации больше не используется. Кроме этого, в структуре super block содержится информация о размере «полосы». Напомним, что в текущей реализации HPFS размер «полосы» взят равным 8 Мбайт. Блок super block размещается в секторе с номером 16 логического диска, на котором установлена файловая система HPFS.

Резервный блок (spare block) содержит указатель на карту аварийного замещения (hotfix map или hotfix-areas), указатель на список свободных запасных блоков (directory emergency free block list), используемых для операций на почти переполненном диске, и ряд системных флагов и дескрипторов. Этот блок размещается в 17 секторе диска. Резервный блок обеспечивает высокую отказоустойчивость файловой системы HPFS и позволяет восстанавливать поврежденные данные на диске.

Файлы и каталоги в HPFS базируются на фундаментальном объекте, называемом F-Node. Эта структура характерна для HPFS и аналога в файловой системе FAT не имеет.

Каждый файл и каталог диска имеет свой файловый узел F-Node. Каждый объект F-Node занимает один сектор и всегда располагается поблизости от своего файла или каталога (обычно – непосредственно перед файлом или каталогом). Объект F-Node содержит длину и первые 15 символов имени файла, специальную служебную информацию, статистику по доступу к файлу, расширенные атрибуты файла и список прав доступа (или только часть этого списка, если он очень большой), ассоциативную информацию о расположении и подчинении файла и т.д. Структура распределения в F-node может принимать несколько форм в зависимости от размера каталога или файлов. HPFS просматривает файл как совокупность одного или более секторов. Из прикладной программы это не видно; файл появляется как непрерывный поток байтов. Если расширенные атрибуты слишком велики для файлового узла, то в него записывается указатель на них.

Сокращенное имя файла (в формате 8.3) используется, когда файл с длинным именем копируется или перемещается на диск с системой FAT, не допускающей подобных имен. Сокращенное имя образуется из первых 8 символов оригинального имени файла, точки и первых трех символов расширения имени, если расширение имеется. Если в имени файла присутствует несколько точек, что не противоречит правилам именования файлов в HPFS, то для расширения сокращенного имени используются три символа после самой последней из этих точек.

Так как HPFS при размещении файла на диске стремится избежать его фрагментации, то структура информации, содержащаяся в файловом узле, достаточно проста. Если файл непрерывен, то его размещение на диске описывается двумя 32-битными числами. Первое число представляет собой указатель на первый блок файла, а второе – длину экстента, то есть число следующих друг за другом блоков, принадлежащих файлу. Если файл фрагментирован, то размещение его экстентов описывается в файловом узле дополнительными парами 32-битных чисел. Фрагментация происходит, когда на диске нет непрерывного свободного участка, достаточно большого, чтобы разместить файл целиком. В этом случае файл приходится разбивать на несколько экстентов и располагать их на диске раздельно. Файловая система HPFS старается разместить экстенты фрагментиро-ванного файла как можно ближе друг к другу, чтобы сократить время позиционирования головок чтения/записи жесткого диска. Для этого HPFS использует статистику, а также старается условно резервировать хотя бы 4 килобайта места в конце файлов, которые растут. Еще один способ уменьшения фрагментирования файлов – это расположение файлов, растущих навстречу друг другу, или файлов, открытых разными тредами или процессами, в разных полосах диска.

В файловом узле можно разместить информацию максимум о восьми экстентах файла. Если файл имеет больше экстентов, то в его файловый узел записывается указатель на блок размещения (allocation block), который может содержать до 40 указателей на экстенты или, по аналогии с блоком дерева каталогов, на другие блоки размещения. Таким образом, двухуровневая структура блоков размещения может хранить информацию о 480 секторах, что позволяет работать с файлами размером до 7,68 Гбайт. На практике размер файла не может превышать 2 Гбайт, но это обусловлено текущей реализацией интерфейса прикладного программирования.

«Полоса», находящаяся в центре диска, используется для хранения каталогов. Эта полоса называется directory band. Как и все остальные «полосы», она имеет размер 8 Мбайт. Однако, если она будет полностью заполнена, HPFS начинает располагать каталоги файлов в других полосах. Расположение этой информационной структуры в середине диска значительно сокращает среднее время позиционирования головок чтения/записи. Действительно, для перемещения головок чтения/записи из произвольного места диска в его центр требуется в два раза меньше времени, чем для перемещения к краю диска, где находится корневой каталог в случае файловой системы FAT. Уже только одно это обеспечивает более высокую производительность файловой системы HPFS по

сравнению с FAT. Аналогичное замечание справедливо и для NTFS, которая тоже располагает свой master file table в начале дискового пространства, а не в его середине.

Однако существенно больший (по сравнению с размещением Directory Band в середине логического диска) вклад в производительность HPFS дает использование метода сбалансированных двоичных деревьев для хранения и поиска информации о местонахождении файлов. Как известно, в файловой системе FAT каталог имеет линейную структуру, специальным образом не упорядоченную, поэтому при поиске файла требуется последовательно просматривать его с самого начала. В HPFS структура каталога представляет собой сбалансированное дерево с записями, расположенными в алфавитном порядке (рис. 3.11). Каждая запись, входящая в состав B-Tree дерева, содержит атрибуты файла, указатель на соответствующий файловый узел, информацию о времени и дате создания файла, времени и дате последнего обновления и обращения, длине данных, содержащих расширенные атрибуты, счетчик обращений к файлу, длине имени файла и само имя, и другую информацию.

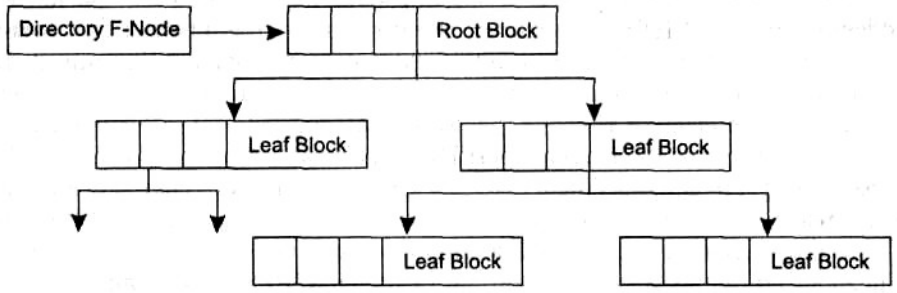

Рис. 3.11. Сбалансированное двоичное дерево

Файловая система HPFS при поиске файла в каталоге просматривает только необходимые ветви двоичного дерева (B-Tree). Такой метод во много раз эффективнее, чем последовательное чтение всех записей в каталоге, что имеет место в системе FAT. Для того чтобы найти искомый файл в каталоге (точнее, указатель на его информационную структуру F-node), организованном на принципах сбалансированных двоичных деревьев, большинство записей вообще читать не нужно. В результате для поиска информации о файле необходимо выполнить существенно меньшее количество операций чтения диска.

Действительно, если, например, каталог содержит 4096 файлов, то файловая система FAT потребует чтения в среднем 64 секторов для поиска нужного файла внутри такого каталога, в то время как HPFS осуществит чтение всего только 2-4 секторов (в среднем) и найдет искомый файл. Несложные расчеты позволяют увидеть явные преимущества HPFS над FAT. Так, например, при использовании 40 входов на блок блоки каталога дерева с двумя уровнями могут содержать 1640 входов, а каталога дерева с тремя уровнями – уже 65 640 входов. Другими словами, некоторый файл может быть найден в типичном каталоге из 65 640 файлов максимум за три обращения. Это намного лучше файловой системы FAT, где для нахождения файла нужно прочитать в худшем случае более 4000 секторов.

Размер каждого из блоков, в терминах которых выделяются каталоги в текущей реализации HPFS, равен 2 Кбайт. Размер записи, описывающей файл, зависит от размера имени файла. Если имя занимает 13 байтов (для формата 8.3), то блок из 2 Кбайт вмещает до 40 описателей файлов. Блоки связаны друг с другом посредством списковой структуры (как и описатели экстентов) для облегчения последовательного обхода.

При переименовании файлов может возникнуть так называемая перебалансировка дерева. Создание файла, переименование или стирание может приводить к каскадированию блоков каталогов. Фактически, переименование может потерпеть неудачу из-за недостатка дискового пространства, даже если файл непосредственно в размерах не увеличился. Во избежание этого «бедствия» HPFS поддерживает небольшой пул свободных блоков, которые могут использоваться при «аварии». Эта операция может

потребовать выделения дополнительных блоков на заполненном диске. Указатель на этот пул свободных блоков сохраняется в SpareBlock.

Важное значение для повышения скорости работы с файлами имеет уменьшение их фрагментации. В HPFS считается, что файл является фрагментированным, если он содержит больше одного экстента. Снижение фрагментации файлов сокращает время позиционирования и время ожидания за счет уменьшения количества перемещений головок, необходимого для доступа к данным файла. Алгоритмы работы файловой системы HPFS работают таким образом, чтобы по возможности размещать файлы в последовательных смежных секторах диска, что обеспечивает максимально быстрый доступ к данным впоследствии. В системе FAT, наоборот, запись следующей порции данных в первый же свободный кластер неизбежно приводит к фрагментации файлов. HPFS тоже, если это предоставляется возможным, записывает данные в смежные секторы диска (но не в первый попавшийся). Это позволяет несколько снизить число перемещений головок чтения/записи от дорожки к дорожке. При этом, когда данные дописываются в существующий файл, HPFS сразу же резервирует как минимум 4 Кбайт непрерывного пространства на диске. Если же часть этого пространства не потребовалась, то после закрытия файла она высвобождается для дальнейшего использования. Файловая система HPFS равномерно размещает непрерывные файлы по всему диску для того, чтобы впоследствии без фрагментации обеспечить их возможное увеличение. Если же файл не может быть увеличен без нарушения его непрерывности, HPFS опять-таки резервирует 4 Кбайт смежных блоков как можно ближе к основной части файла с целью сократить время позиционирования головок чтения/записи и время ожидания соответствующего сектора.

Очевидно, что степень фрагментации файлов на диске зависит как от числа файлов, расположенных на нем, их размеров и размеров самого диска, так и от характера и интенсивности самих дисковых операций. Незначительная фрагментация файлов практически не сказывается на быстродействии операций с файлами. Файлы, состоящие из двух-трех экстентов, практически не снижают производительность HPFS, так как эта файловая система следит за тем, чтобы области данных, принадлежащие одному и тому же файлу, располагались как можно ближе друг к другу. Файл из трех экстентов имеет только два нарушения непрерывности, и, следовательно, для его чтения потребуется всего лишь два небольших перемещения головки диска. Программы (утилиты) дефрагментации, имеющиеся для этой файловой системы, по умолчанию считают наличие двух-трех экстентов у файла нормой. Например, программа HPFSOPT из набора утилит Gamma-Tech по умолчанию не дефрагментирует файлы, состоящие из трех и менее экстентов, а файлы, которые имеют большее количестве экстентов, приводятся к 2 или 3 экстентам, если это возможно (файлы объемом в несколько десятков мегабайт всегда будут фрагментированы, ибо максимально возможный размер экстента, как вы помните, равен 8 Мбайт). Надо сказать, что практика показывает, что в среднем на диске имеется не более 2 процентов файлов, имеющих три и более экстентов. Даже общее количество фрагментированных файлов, как правило, не превышает 3 процентов. Такая ничтожная фрагментация оказывает пренебрежимо малое влияние на общую производительность системы.

Теперь кратко рассмотрим вопрос надежности хранения данных в HPFS. Любая файловая система должна обладать средствами исправления ошибок, возникающих при записи информации на диск. Система HPFS для этого использует механизм аварийного замещения (hotfix).

Если файловая система HPFS сталкивается с проблемой в процессе записи данных на диск, она выводит на экран соответствующее сообщение об ошибке. Затем HPFS сохраняет информацию, которая должна была быть записана в дефектный сектор, в одном из запасных секторов, заранее зарезервированных на этот случай. Список свободных запасных блоков хранится в резервном блоке HPFS. При обнаружении ошибки во время записи данных в нормальный блок HPFS выбирает один из свободных запасных блоков и

сохраняет эти данные в нем. Затем файловая система обновляет карту аварийного замещения в резервном блоке. Эта карта представляет собой просто пары двойных слов, каждое из которых является 32-битным номером сектора. Первый номер указывает на дефектный сектор, а второй – на тот сектор среди имеющихся запасных секторов, который был выбран для его замены. После замены дефектного сектора запасным карта аварийного замещения записывается на диск, и на экране появляется всплывающее окно, информирующее пользователя о произошедшей ошибке записи на диск. Каждый раз, когда система выполняет запись или чтение сектора диска, она просматривает карту аварийного замещения и подменяет все номера дефектных секторов номерами запасных секторов с соответствующими данными. Следует заметить, что это преобразование номеров существенно не влияет на производительность системы, так как оно выполняется только при физическом обращении к диску, но не при чтении данных из дискового кэша. Очистка карты аварийного замещения автоматически выполняется программой CHKDSK при проверке диска HPFS. Для каждого замещенного блока (сектора) программа CHKDSK выделяет новый сектор в наиболее подходящем для файла (которому принадлежат данные) месте жесткого диска. Затем программа перемещает данные из запасного блока в этот сектор и обновляет информацию о положении файла, что может потребовать новой балансировки дерева блоков размещения. После этого CHKDSK вносит поврежденный сектор в список дефектных блоков, который хранится в дополнительном блоке HPFS, и возвращает освобожденный сектор в список свободных запасных секторов резервного блока. Затем удаляет запись из карты аварийного замещения и записывает отредактированную карту на диск.

Все основные файловые объекты в HPFS, в том числе файловые узлы, блоки размещения и блоки каталогов, имеют уникальные 32-битные идентификаторы и указатели на свои родительские и дочерние блоки. Файловые узлы, кроме того, содержат сокращенное имя своего файла или каталога. Избыточность и взаимосвязь файловых структур HPFS позволяют программе CHKDSK полностью восстанавливать файловую структуру диска, последовательно анализируя все файловые узлы, блоки размещения и блоки каталогов. Руководствуясь собранной информацией, CHKDSK реконструирует файлы и каталоги, а затем заново создает битовые карты свободных секторов диска. Запуск программы CHKDSK следует осуществлять с соответствующими ключами. Так, например, один из вариантов работы этой программы позволяет найти и восстановить удаленные файлы.

HPFS относится к так называемым монтируемым файловым системам. Это означает, что она не встроена в операционную систему, а добавляется к ней при необходимости. Файловая система HPFS устанавливается оператором IFS в файле CONFIG.SYS. Этот оператор всегда помещается в первой строке данного конфигурационного файла. В приводимом далее примере оператор IFS устанавливает файловую систему HPFS с кэшем в 2 Мбайт, длиной записи кэша в 8 Кбайт и автоматической процедурой проверки дисков C и D:

```
IFS=E:\0S2\HPFS.IFS /CACHE:2048 /CRECL:4 /AUTOCHECK:CD
```

Для запуска программы управления процессом кэширования следует прописать в файле CONFIG.SYS еще одну строку:

```
RUN=E:\0S2\CACHE.EXE /Lazy:On /BufferIdle:2000 /Diskldle:4000 /MaxAge:8000 /DirtyMax:256 /ReadAhead:On
```

В этой строке включается режим отложенной («ленивой») записи, устанавливаются параметры работы этого режима, а также включается режим упреждающего чтения данных, что в целом позволяет существенно сократить количество обращений к диску и ощутимо повысить быстродействие файловой системы. Так, ключ Lazy с параметром On включает «ленивую запись», а с параметром Off – выключает. Ключ Buffer Idle определяет время в миллисекундах, в течение которого буфер кэша должен оставаться в неактивном состоянии, чтобы стало возможным осуществить запись данных из кэша на

диск. По умолчанию (то есть если не прописывать данный ключ явным образом) это время равно 500 мс. Ключ Diskldle задает время (в миллисекундах), в течение которого диск должен оставаться в неактивном состоянии, чтобы стало возможным осуществить запись данных из кэша на диск. По умолчанию это время равно 1 с. Этот параметр позволяет избежать записи из кэша на диск во время выполнения других операций с диском.

Ключ MaxAge задает время (тоже в миллисекундах), по истечении которого часто сохраняемые в кэше данные наконец помечаются как «устаревшие» и при переполнении кэша могут быть замещены новыми. По умолчанию это время равно 5 с.

Остальные подробности установки параметров и возможные значения ключей имеются в HELP-файлах, устанавливаемых вместе с операционной системой OS/2 Warp.

Наконец, следует сказать и еще об одной системе управления файлами – речь идет о реализации HPFS для работы на серверах, функционирующих под управлением OS/2. Это система управления файлами, получившая название HPFS386.IFS. Ее принципиальное отличие от системы HPFS.IFS заключается в том, что HPFS386.IFS позволяет (посредством более полного использования технологии расширенных атрибутов) организовать ограничения на доступ к файлам и каталогами с помощью соответствующих списков доступа – ACL (access control list). Эта технология, как известно, используется в файловой системе NTFS. Кроме этого, в системе HPFS386.IFS, в отличие от HPFS.IFS, нет ограничений на объем памяти, выделяемой для кэширования файловых записей. Иными словами, при наличии достаточного объема оперативной памяти объем файлового кэша может быть в несколько десятков мегабайт, в то время как для обычной HPFS.IFS этот объем не может превышать 2 Мбайт, что по сегодняшним меркам безусловно мало. Наконец, при установке режимов работы файлового кэша HPFS386.IFS есть возможность явным образом указать алгоритм кэширования. Наиболее эффективным алгоритмом можно считать так называемый «элеваторный», когда при записи данных из кэша на диск они предварительно упорядочиваются таким образом, чтобы минимизировать время, отводимое на позиционирование головок чтения/ записи. Головки чтения/записи при этом перемещаются от внешних цилиндров к внутренним и по ходу своего движения осуществляют запись и чтение данных в соответствии со специальным образом упорядочиваемым списком запросов на дисковые операции.

Приведем пример записи строк в конфигурационном файле CONFIG.SYS, которые устанавливают систему HPFS386.IFS и определяют параметры работы ее подсистемы кэширования:

```
IFS=E:\IBM386FS\HPFS386.IFS /AUTOCHECK:EGH
RUN=E:\IBM386FS\CACHE386.EXE /Labizy:On /BufferIdle:4000 /MaxAge:20000
```

Эти записи следует понимать следующим образом. При запуске операционной системы в случае обнаружения флага, означающего, что не все файлы были закрыты в процессе предыдущей работы, система управления файлами HPFS386.IFS сначала запустит программу проверки целостности файловой системы для томов E:, G: и H:. Для кэширования файлов при работе этой системы управления файлами устанавливается режим отложенной записи со временем жизни буферов до 20 с. Остальные параметры, в частности алгоритм обслуживания запросов, устанавливаются в файле HPFS386.INI, который в данном случае располагается в директории E:\IBM386FS.

Опишем кратко некоторые наиболее интересные параметры, управляющие работой кэша в этой системе управления файлами. Прежде всего отметим, что файл HPFS386.INI разбит на несколько секций. В настоящий момент рассмотрим сецию [ULTIMEDIA]:

```
[ULTIMEDIA]
QUEUESORT={ FIFO | ELEVATOR | DEFAULT | CURRENT}
QUEUEMETHOD={ PRIORITY | NOPRIORITY | DEFAULT | CURRENT}
QUEUEDEPTH={1...255|DEFAULT|CURRENT}
```

Параметр QUEUESORT задает способ ведения очереди запросов к диску. Он может принимать значения FIFO, ELEVATOR, DEFAULT и CURRENT. Если задано значение FIFO, то каждый новый запрос просто добавляется в конец очереди, то есть запросы выполняются в том порядке, в котором они поступают в систему. Однако можно упорядочить некоторое количество запросов по возрастанию номеров дорожек. Если задано значение ELEVATOR, то включается режим поддержки упорядоченной очереди запросов. При этом запросы начинают обрабатываться по алгоритму ELEVATOR (он же C-SCAN или «режим плавающей головки»). Напомним, этот алгоритм подразумевает, что головка чтения/записи сканирует диск в выбранном направлении (например, в направлении возрастания номеров дорожек), останавливаясь для выполнения запросов, находящихся на пути следования. Когда она доходит до последнего запроса, головка чтения/записи переносится на начальную дорожку и процесс обслуживания запросов продолжается.

Если для параметра QUEUESORT задано значение DEFAULT, то выбирается алгоритм по умолчанию. Сейчас это ELEVATOR. Если задано значение CURRENT, то остается в силе тот алгоритм, который был выбран DASD Manager при инициализации.

Параметр QUEUEMETHOD определяет, должны ли учитываться приоритеты запросов при построении очереди. Он может принимать значения PRIORITY, NOPRIORITY, DEFAULT и CURRENT. Если задано значение NOPRIORITY, то все запросы включаются в общую очередь, а их приоритеты игнорируются. Если задано значение PRIORITY, то модуль DASD Manager будет поддерживать несколько очередей запросов, по одной на каждый приоритет. Когда DASD Manager передает запросы на исполнение драйверу диска, он сначала выбирает запросы из самой приоритетной очереди, потом из менее приоритетной и т. д. Приоритеты назначает HPFS386, а распределены они следующим образом.

High:

1. Shutdown или экстренная запись из-за сбоя питания.
2. Страничный обмен.
3. Обычные запросы от foreground1 сессии.
4. Обычные запросы от background2 сессии. (Приоритеты 3 и 4 равны, если в файле CONFIG.SYS задан параметр RIORITY_DISK_IO=NO.)
5. Read-ahead и низкоприоритетные запросы страничного обмена (страничная предвыборка).
6. Lazy-Write и прочие запросы, не требующие немедленной реакции. Low:
7. Предвыборка.

Если для параметра QUEUEMETHOD задано значение DEFAULT, то выбирается метод по умолчанию. Сейчас это PRIORITY. Если задано значение CURRENT, то остается в силе тот метод, который был выбран DASD Manager при инициализации.

Параметр QUEUEDEPTH задает глубину просмотра очереди при выборке запросов. Он может принимать значения из диапазона (1...255), а также DEFAULT и CURRENT. Если в качестве значения параметра QUEUEDEPTH задано число, то оно определяет количество запросов, которые должны находиться в очереди дискового адаптера одновременно. Например, для SCSI-адаптеров имеет смысл поддерживать такую длину очереди, при которой они смогут загрузить все запросы в свои аппаратные структуры (tagged queue или mailbox). Если очередь запросов к адаптеру будет слишком короткой, то аппаратура будет работать с неполной загрузкой, а если она будет слишком длинной – драйвер SCSI-адаптера будет перегружен «лишними» запросами. Поэтому разумным значением для QUEUEDEPTH будет число, немного превышающее длину аппаратной очереди команд адаптера. Если для параметра QUEUEDEPTH задано значение DEFAULT, то глубина просмотра очереди определяется автоматически на основании значения, которое рекомендовано драйвером дискового адаптера. Если задано значение CURRENT,

то глубина просмотра очереди не изменяется. В текущей реализации CURRENT эквивалентно DEFAULT.

Итак, текущие умолчания для HPFS386 имеют вид:

QUEUESORT=FIFO

QUEUEMETHOD=DEFAULT

QUEUEDEPTH=2

А текущие умолчания для DASD Manager таковы:

QUEUESORT=ELEVATOR

QUEUEMETHOD=PRIORITY

QUEUEDEPTH=<зависит от адаптера диска>

Умолчания DASD Manager можно менять с помощью параметра /QF:

BASEDEV=0S2DASD.DMD /QF:{1|2|3}

где 1 - QUEUESORT = FIFO; 2 - QUEUEMETHOD = NOPRIORITY; 3 - QUEUESORT - FIFO и QUEUEMETHOD = NOPRIORITY.

Наконец, добавим еще несколько слов об устанавливаемых файловых системах (installable file systems – IFS), представляющих собой специальные «драйверы» для доступа к разделам, отформатированным под другую файловую систему. Это очень удобный и мощный механизм добавления в ОС новых файловых систем и замены одной системы управления файлами на другую. Сегодня, например, для OS/2 уже реально существуют IFS-модули для файловой системы VFAT (FAT с поддержкой длинных имен), FAT32, Ext2FS (файловая система Linux), NTFS (правда, пока только для чтения). Для работы с данными на CD-ROM имеется CDFS.1FS. Есть и FTP.IFS, позволяющая монтировать ftp-архивы как локальные диски. Механизм устанавливаемых файловых систем был перенесен и в систему Windows NT.

Файловая система NTFS (New Technology File System)

В название файловой системы NTFS входят слова «New Technology», то есть «новая технология». Действительно, NTFS содержит ряд значительных усовершенствований и изменений, существенно отличающих ее от других файловых систем. С точки зрения пользователей, файлы по-прежнему хранятся в каталогах (часто называемых «папками» или фолдерами в среде Windows). Однако в NTFS, в отличие от FAT, работа на дисках большого объема происходит намного эффективнее; имеются средства для ограничения в доступе к файлам и каталогам, введены механизмы, существенно повышающие надежность файловой системы, сняты многие ограничения на максимальное количество дисковых секторов и/или кластеров.

Основные возможности файловой системы NTFS

При проектировании системы NTFS особое внимание было уделено следующим характеристикам:

– надежность. Высокопроизводительные компьютеры и системы совместного пользования (серверы) должны обладать повышенной надежностью, которая является ключевым элементом структуры и поведения NTFS. Одним из способов увеличения надежности является введение механизма транзакций, при котором осуществляется журналирование. При журналировании файловых операций система управления файлами фиксирует в специальном служебном файле происходящие изменения. В начале операции, связанной с изменением файловой структуры, делается соответствующая пометка. Если во время операций над файлами происходит какой-нибудь сбой, то упомянутая отметка о начале операции остается указанной как незавершенная. При выполнении процедуры проверки целостности файловой системы после перезагрузки машины эти незавершенные операции будут отменены и файлы будут приведены к исходному состоянию. Если же операция изменения данных в файлах завершается нормальным образом, то в этом самом

служебном файле поддержки журналирования операция отмечается как завершенная.) файловых операций;

– *расширенная функциональность*. NTFS проектировалась с учетом возможного расширения. В ней были воплощены многие дополнительные возможности – усовершенствованная отказоустойчивость, эмуляция других файловых систем,мощная модель безопасности, параллельная обработка потоков данных и создание файловых атрибутов, определяемых пользователем;

– *поддержка POSIX*. Поскольку правительство США требовало, чтобы все закупаемые им системы хотя бы в минимальной степени соответствовали стандарту POSIX, такая возможность была предусмотрена и в NTFS. К числу базовых средств файловой системы POSIX относится необязательное использование имен файлов с учетом регистра, хранение времени последнего обращения к файлу и механизм так называемых «жестких ссылок» – альтернативных имен, позволяющих ссылаться на один и тот же файл по двум и более именам;

– *гибкость.* Модель распределения дискового пространства в NTFS отличается чрезвычайной гибкостью. Размер кластера может изменяться от 512 байт до 64 Кбайт; он представляет собой число, кратное внутреннему кванту распределения дискового пространства. NTFS также поддерживает длинные имена файлов, набор символов Unicode и альтернативные имена формата 8.3 для совместимости с FAT.

NTFS превосходно справляется с обработкой больших массивов данных и достаточно хорошо проявляет себя при работе с томами объемом от 300-400 Мбайт и выше. Максимально возможные размеры тома (и размеры файла) составляют 16 Эбайт. Количество файлов в корневом и некорневом каталогах не ограничено. Поскольку в основу структуры каталогов NTFS заложена эффективная структура данных, называемая «бинарным деревом», время поиска файлов в NTFS (в отличие от систем на базе FAT) не связано линейной зависимостью с их количеством.

Система NTFS также обладает определенными средствами самовосстановления. NTFS поддерживает различные механизмы проверки целостности системы, включая ведение журналов транзакций, позволяющих воспроизвести файловые операции записи по специальному системному журналу.

Файловая система NTFS поддерживает объектную модель безопасности NT и рассматривает все тома, каталоги и файлы как самостоятельные объекты. NTFS обеспечивает безопасность на уровне файлов; это означает, что права доступа к томам, каталогам и файлам могут зависеть от учетной записи пользователя и тех групп, к которым он принадлежит. Каждый раз, когда пользователь обращается к объекту файловой системы, его права доступа проверяются по списку разрешений данного объекта. Если пользователь обладает достаточным уровнем прав, его запрос удовлетворяется; в противном случае запрос отклоняется. Эта модель безопасности применяется как при локальной регистрации пользователей на компьютерах с NT, так и при удаленных сетевых запросах.

Наконец, помимо огромных размеров томов и файлов, система NTFS также обладает встроенными средствами сжатия, которые можно применять к отдельным файлам, целым каталогам и даже томам (и впоследствии отменять или назначать их по своему усмотрению).

Структура тома с файловой системой NTFS

Рассмотрим теперь структуру файловой системы NTFS. Мы же здесь коснемся только основных моментов.

Одним из основных понятий, используемых при работе с NTFS, является понятие тома (volume)1. Возможно также создание отказоустойчивого тома, занимающего несколько разделов, то есть использование RAID-технологии. Как и многие другие системы, NTFS делит все полезное дисковое пространство тома на кластеры – блоки данных, адресуемые

как единицы данных. NTFS поддерживает размеры кластеров от 512 байт до 64 Кбайт; стандартом же считается кластер размером 2 или 4 Кбайт.

Все дисковое пространство в NTFS делится на две неравные части (рис. 3.12). Первые 12 % диска отводятся под так называемую MFT-зону – пространство, которое может занимать, увеличиваясь в размере, главный служебный метафайл MFT2. Запись каких-либо данных в эту область невозможна. MFT-зона всегда держится пустой – это делается для того, чтобы самый главный, служебный файл (MFT) по возможности не фрагментировался при своем росте. Остальные 88 % тома представляют собой обычное пространство для хранения файлов.

Рис. 3.12. Структура тома NTFS

MFT (master file table, общая таблица файлов) представляет собой централизованный каталог всех остальных файлов диска, в том числе и себя самого. MFT поделен на записи фиксированного размера в 1 Кбайт1, и каждая запись соответствует какому-либо файлу (в общем смысле этого слова). Первые 16 файлов носят служебный характер и недоступны операционной системе – они называются метафайлами, причем самый первый метафайл – сам MFT. Эти первые 16 элементов MFT – единственная часть диска, имеющая строго фиксированное положение. Копия этих же 16 записей хранится в середине тома для надежности, поскольку они очень важны. Остальные части MFT-файла могут располагаться, как и любой другой файл, в произвольных местах диска – восстановить его положение можно с помощью его самого, «зацепившись» за самую основу – за первый элемент MFT.

Упомянутые первые 16 файлов NTFS (метафайлы) носят служебный характер; каждый из них отвечает за какой-либо аспект работы системы. Метафайлы находятся в корневом каталоге NTFS-тома. Все они начинаются с символа имени «$», хотя получить какую-либо информацию о них стандартными средствами сложно. В табл. 3.7 приведены основные известные метафайлы и их назначение. Таким образом, можно узнать, например, сколько операционная система тратит на каталогизацию тома, посмотрев размер файла SMFT.

Таблица 3.7. Метафайлы NTFS

Имя метафайла	Назначение метафайл
$MFT	Сам Master File Table
$MFTmirr	Копия первых 16 записей MFT, размещенная посередине тома
$LogFile	Файл поддержки операций журналирования
$Volume	Служебная информация – метка тома, версия файловой системы и т. д.
$AttrDef	Список стандартных атрибутов файлов на томе
$.	Корневой каталог
$Bitmap	Карта свободного места тома
$Boot	Загрузочный сектор (если раздел загрузочный)

$Quota	Файл, в котором записаны права пользователей на использование дискового пространства (этот файл начал работать лишь в Windows 2000 с системой NTFS 5.0)
$Upcase	Файл – таблица соответствия заглавных и прописных букв в именах файлов. В NTFS имена файлов записываются в Unicode (что составляет 65 тысяч различных символов) и искать большие и малые эквиваленты в данном случае – нетривиальная задача

Итак, все файлы тома упоминаются в MFT. В этой структуре хранится вся информация о файлах, за исключением собственно данных. Имя файла, размер, положение на диске отдельных фрагментов и т. д. – все это хранится в соответствующей записи. Если для информации не хватает одной записи MFT, то используется несколько записей, причем не обязательно идущих подряд. Файлы могут иметь не очень большой размер. Тогда применяется довольно удачное решение: данные файла хранятся прямо в MFT, в оставшемся от основных данных месте в пределах одной записи MFT. Файлы, занимающие сотни байт, обычно не имеют своего «физического» воплощения в основной файловой области – все данные такого файла хранятся в одном месте, в MFT.

Файл в томе с NTFS идентифицируется так называемой файловой ссылкой (File Reference), которая представляется как 64-разрядное число. Файловая ссылка состоит из номера файла, который соответствует позиции его файловой записи в MFT, и номера последовательности. Последний увеличивается всякий раз, когда данная позиция в MFT используется повторно, что позволяет файловой системе NTFS выполнять внутренние проверки целостности.

Каждый файл в NTFS представлен с помощью потоков (streams), то есть у него нет как таковых «просто данных», а есть «потоки». Для правильного понимания потока достаточно указать, что один из потоков и носит привычный нам смысл – данные файла. Но большинство атрибутов файла – это тоже потоки. Таким образом, получается, что базовая сущность у файла только одна – номер в MFT, а все остальное, включая и его потоки, – опционально. Данный подход может эффективно использоваться – например, файлу можно «прилепить» еще один поток, записав в него любые данные. В Windows 2000 таким образом записана информация об авторе и содержании файла (одна из закладок в свойствах файла, просматриваемых, например, из проводника). Интересно, что эти дополнительные потоки не видны стандартными средствами работы с файлами: наблюдаемый размер файла – это лишь размер основного потока, который содержит традиционные данные. Можно, к примеру, иметь файл нулевой длины, при стирании которого освободится 1 Гбайт свободного места – просто потому, что какая-нибудь хитрая программа или технология «прилепила» к нему дополнительный поток (альтернативные данные) такого большого размера. Но на самом деле в настоящее время потоки практически не используются, так что опасаться подобных ситуаций не следует, хотя гипотетически они возможны. Просто необходимо иметь в виду, что файл в NTFS – это более глубокое понятие, чем можно себе представить, просматривая каталоги диска.

Стандартные же атрибуты для файлов и каталогов в томе NTFS имеют фиксированные имена и коды типа, они перечислены в табл. 3.8.

Таблица 3.8. Атрибуты файлов в системе NTFS

Системный атрибут	Описание атрибута
Стандартная информация о файле	Традиционные атрибуты Read Only, Hidden, Archive, System, отметки времени, включая время создания или последней модификации, число каталогов, ссылающихся на файл
Список атрибутов	Список атрибутов, из которых состоит файл, и файловая ссылка на файловую запись и MFT, в которой расположен каждый из атрибутов. Последний используется, если файлу необходимо более одной записи в MFT

Имя файла	Имя файла в символах Unicode. Файл может иметь несколько атрибутов – имен файла, подобно тому как это имеет место в UNIX-системах. Это случается, когда имеется связь POSIX с данным файлом или если у файла есть автоматически сгенерированное имя в формате 8.3
Дескриптор защиты	Структура данных защиты (ACL), предохраняющая файл от несанкционированного доступа. Атрибут «дескриптор защиты» определяет, кто владелец файла и кто имеет доступ к нему
Данные	Собственно данные файла, его содержимое. В NTFS у файла по умолчанию есть один безымянный атрибут данных, и он может иметь дополнительные именованные атрибуты данных. У каталога нет атрибута данных по умолчанию, но он может иметь необязательные именованные атрибуты данных
Корень индекса, размещение индекса, битовая карта (только для каталогов)	Атрибуты, используемые для индексов имен файлов в больших каталогах
Расширенные атрибуты HPFS	Атрибуты, используемые для реализации расширенных атрибутов HPFS для подсистемы OS/2 и OS/2-клиентов файл-серверов Windows NT

Атрибуты файла в записях MFT расположены в порядке возрастания числовых значений кодов типа, причем некоторые типы атрибутов могут встречаться в записи более одного раза: например, если у файла есть несколько атрибутов данных или несколько имен. Обязательными для каждого файла в томе NTFS являются атрибут стандартной информации, атрибут имени файла, атрибут дескриптора защиты и атрибут данных. Остальные атрибуты могут встречаться при необходимости.

Имя файла в NTFS, в отличие от файловых систем FAT и HPFS, может содержать любые символы, включая полный набор национальных алфавитов, так как данные представлены в Unicode – 16-битном представлении, которое дает 65 535 разных символов. Максимальная длина имени файла в NTFS – 255 символов.

Большой вклад в эффективность файловой системы вносит организация каталога. Каталог в NTFS представляет собой специальный файл, хранящий ссылки на другие файлы и каталоги, создавая иерархическое строение данных на диске. Файл каталога поделен на блоки, каждый из которых содержит имя файла, базовые атрибуты и ссылку на элемент MFT, который уже предоставляет полную информацию об элементе каталога. Главный каталог диска – корневой – ничем не отличается от обычных каталогов, кроме специальной ссылки на него из начала метафайла MFT.

Внутренняя структура каталога представляет собой бинарное дерево, подобно тому как это организовано в HPFS. Кстати, при создании файловой системы NTFS разработчики решили использовать максимально возможное количество эффективных решений из HPFS. К сожалению, не было взято на вооружение разбиение всего дискового пространства на зоны, в каждой из которых хранилась бы информация об имеющихся свободных кластерах. В результате отказа от этого подхода и введения механизма транзакций скорость работы файловой системы NTFS существенно ниже скорости работы системы HPFS.

Итак, как нам теперь известно, бинарное дерево каталога располагает имена файлов таким образом, чтобы поиск файла осуществлялся с помощью получения двухзначных ответов на вопросы о положении файла. Бинарное дерево способно дать ответ на вопрос: в какой группе, относительно данного элемента, находится искомое имя – выше или ниже? Мы начинаем с такого вопроса к среднему элементу, и каждый ответ сужает зону поиска в среднем в два раза. Если представить, что файлы отсортированы по алфавиту, то ответ на вопрос осуществляется очевидным способом – сравнением начальных букв. Область

поиска, суженная в два раза, начинает исследоваться аналогичным образом, начиная опять же со среднего элемента.

Заметим, что добавлять файл в каталог в виде дерева не намного труднее, чем в линейный каталог системы FAT. Это сопоставимые по времени операции. Для того чтобы добавить новый файл в каталог, нужно сначала убедиться, что файла с таким именем там еще нет. Поэтому в системе FAT с линейной организацией записей каталога у нас появляются трудности не только с поиском файла. И это с лихвой компенсирует саму простоту добавления файла каталог.

Возможности в файловой системы NTFS

по ограничению доступа к файлам и каталогам

Благодаря наличию механизма расширенных атрибутов, в NTFS именно с их помощью реализованы ограничения в доступе к файлам и каталогам. Эти дополнительные атрибуты, использованные для ограничения в доступе к файловым объектам, назвали атрибутами безопасности. При каждом обращении к такому объекту сравнивается специальный список дискреционных прав доступа, приписанный ему, со специальным системным идентификатором, несущим информацию о том, от имени кого осуществляется текущий запрос к файлу или каталогу. Если имеется в списке необходимое разрешение, то действие выполняется, в противном случае система сообщает об отказе.

Файловая система NTFS имеет следующие разрешения, которые могут быть приписаны любому файлу и/или каталогу, и которые называются индивидуальными разрешениями. Это разрешения Read {прочитать), Write {записать), eXecute {выполнить), Delete {удалить), Change Permissions {изменить разрешения), и Take Ownership {стать владельцем). Соответствующие этим разрешениям действия можно выполнять, только если для данного пользователя или для группы (к которой он принадлежит) имеется одноименное разрешение. Другими словами, если для некоторого файла указано, что все могут его читать и исполнять, то только эти действия и можно с ним сделать, если при этом не указано, что для какой-нибудь другой группы (или отдельного пользователя) имеются другие разрешения. Комбинации этих индивидуальных разрешений и определяют те действия, которые могут быть выполнены с файлом или каталогом.

Изначально всему диску, а значит, и файлам, которые на нем создаются, присвоены все индивидуальные разрешения для группы Everyone (все). Это означает, что любой пользователь, имея полный набор индивидуальных разрешений на файлы и каталоги, может изменять их по своему усмотрению, т.е. ограничивать других пользователей в правах доступа к тому или иному объекту. Если изменить разрешения на каталог, то новые файлы, создаваемые в нем, будут получать и соответствующие разрешения: они будут наследовать разрешения своего родительского каталога.

Имеются так называемые стандартные разрешения, которые, по замыслу разработчиков, следует использовать для указания наиболее распространенных комбинаций индивидуальных разрешений. Поясним эти стандартные разрешения с помощью таблицы, расположенной на следующей странице.

Помимо этих стандартных разрешений можно использовать специальные. Они определяются как явная комбинация индивидуальных разрешений.

Фактические разрешения для пользователя, которые он будет иметь на файл или каталог, определяются как сумма разрешений, которые он получает как член нескольких групп. У этого общего правила есть исключение. Разрешение No Access (нет доступа) имеет приоритет над остальными. Оно запрещает любой доступ к файлу или каталогу, даже если пользователю, как члену другой группы, дано необходимое разрешение. Можно сказать, что это стандартное разрешение означает не отсутствие разрешений, а наложение явного запрета, и что оно отменяет для пользователя или группы все разрешения, установленные в остальных строках дискреционного списка прав доступа.

Стандартные разрешения NTFS	Соответствующие им комбинации индивидуальных разрешений NTFS	
	Для каталогов	Для файлов
No access (нет доступа)	Нет никаких разрешений	Нет никаких разрешений
List (просмотр)	Read, eXecute	Нет никаких разрешений
Read (чтение)	Read, eXecute	Read, eXecute
Add (добавление)	Write, eXecute	Нет никаких разрешений
Add & Read (чтение и добавление)	Read, Write, eXecute	Read, eXecute
Change (изменение)	Read, Write, eXecute, Delete	Read, Write, eXecute, Delete
Full Control (полный доступ)	Все разрешения	Все разрешения

Рассмотрим теперь, что происходит с правами на защищенные файлы в NTFS при их перемещении. Каталоги обычно обладают теми же разрешениями, что и находящиеся в них файлы и папки, хотя у каждого файла могут быть свои разрешения. Разрешения, которые имеются у файла, имеют приоритет над разрешениями, которые установлены на каталог, в котором находится этот файл. Например, если вы создаете каталог внутри другого каталога, для которого администраторы обладают правом полного доступа, а пользователи – правом чтения, то новый каталог унаследует эти права. То же относится и к файлам, копируемым из другого каталога или перемещаемым из другого раздела NTFS.

Если каталог или файл перемещается в другой каталог того же раздела NTFS, то атрибуты безопасности не наследуются от нового каталога. Дело в том, что при перемещении файлов в границах одного раздела NTFS изменяется только указатель местонахождения объекта, а все остальные атрибуты (включая атрибуты безопасности) остаются без изменений.

Три следующих важных правила помогут определить состояние прав доступа при перемещении или копировании объектов NTFS:

1. при перемещении файлов в границах раздела NTFS сохраняются исходные права доступа.
2. при выполнении других операций (создании или копировании файлов, а также их перемещении между разделами NTFS) наследуются права доступа родительского каталога.
3. при перемещении файлов из раздела NTFS в раздел FAT все права NTFS теряются.

Основные отличия FAT и NTFS

Если говорить о накладных расходах на хранение служебной информации, FAT отличается от NTFS большей компактностью и меньшей сложностью. В большинстве томов FAT на хранение таблицы размещения, содержащей информацию обо всех файлах тома, расходуется менее 1 Мбайт. Столь низкие накладные расходы позволяют форматировать в FAT жесткие диски малого объема и флоппи-диски. В NTFS служебные данные занимают больше места, чем в FAT. Так, каждый элемент каталога занимает 2 Кбайт. Однако это имеет и свои преимущества, так как содержимое файлов объемом 1500 байт и менее может полностью храниться в элементе каталога.

Система NTFS не может использоваться для форматирования флоппи-дисков. Не стоит пользоваться ею для форматирования разделов объемом менее 50-100 Мбайт. Относительно высокие накладные расходы приводят к тому, что для малых разделов служебные данные могут занимать до 25 % объема носителя. Корпорация Microsoft рекомендует использовать FAT для разделов объемом 256 Мбайт и менее, а NTFS – для разделов объемом 400 Мбайт и более1.

Следующий критерий сравнения – размер файлов. Разделы FAT имеют объем до 2 Гбайт, VFAT – до 4 Гбайт и FAT32 – до 4 Тбайт. Тем не менее из-за особенностей своего внутреннего строения разделы FAT лучше всего работают для разделов объемом 200 Мбайт и менее. Разделы NTFS могут достигать 16 Эбайт, однако в настоящее время из-за аппаратных и других системных причин размер файлов ограничивается 2 Тбайт.

Разделы FAT могут использоваться практически во всех операционных системах. За редкими исключениями, с разделами NTFS можно работать напрямую только из Windows NT, хотя и имеются для ряда ОС соответствующие реализации систем управления файлами для чтения файлов из томов NTFS. Так, например, утилита (драйвер) NTFSDOS позволяет читать данные NTFS на компьютере, загруженном в режиме MS-DOS. Однако полноценных реализаций для работы с NTFS вне системы Windows NT пока нет.

Разделы FAT не обеспечивают локальной безопасности. С другой стороны, разделы NTFS обеспечивают локальную безопасность как файлов, так и каталогов. Для разделов FAT могут устанавливаться общие права, связанные с общим доступом к каталогам в сети. Однако такая защита не помешает пользователю с локальным входом получить доступ к файлам своего компьютера. В отношении безопасности NTFS оказывается предпочтительным вариантом. Разделы NTFS могут запрещать или ограничивать доступ как удаленных, так и локальных пользователей. Следовательно, к защищенным файлам смогут обратиться лишь те пользователи, которым были предоставлены соответствующие права.

Напомним, что Windows NT содержит специальную утилиту CONVERT.EXE, которая преобразует тома FAT в эквивалентные тома NTFS, однако для обратного

преобразования (из NTFS в FAT) подобных утилит не существует. Чтобы выполнить такое обратное преобразование, необходимо создать раздел FAT, скопировать в него файлы из раздела NTFS и затем удалить оригиналы. Важно при этом не забывать и о том, что при копировании файлов из NTFS в FAT теряются все атрибуты безопасности NTFS (напомним, что в FAT не предусмотрены средства для определения и последующего хранения этих атрибутов).

В последнее время появилось еще одно очень важное обстоятельство, связанное с тем, что объемы дисковых механизмов намного превысили максимально допустимый размер, приемлемый для FAT, – 8,4 Гбайт. Этот предел объясняется максимально возможными значениями в адресе сектора, для которого, как мы уже знаем, отводится всего 3 байта. Поэтому в подавляющем большинстве случаев при работе в среде Windows-систем используют либо FAT32, либо NTFS. Последняя, безусловно, лучше, но она не поддерживается в широко распространенных ОС Windows 98 и ныне все более часто встречающейся Windows Millennium Edition.

4 Архитектура операционных систем, интерфейсы прикладного программирования

Несмотря на тот факт, что в наши дни уже практически никто не разрабатывает операционные системы (естественно, за исключением нескольких известных компаний, специализирующихся на этом направлении, кстати, одном из сложнейших) и все являются пользователями наиболее распространенных систем, мы все-таки рассмотрим кратко вопросы архитектуры ОС. Сделать это необходимо потому, что многие возможности и характеристики ОС определяются в значительной мере ее архитектурой.

Основные принципы построения операционных систем

Среди множества принципов, которые используются при построении ОС, перечислим несколько наиболее важных (на наш взгляд, так как в соответствующих публикациях на эту тему перечисляется существенно большее их количество).

Принцип модульности

Под модулем в общем случае понимают функционально законченный элемент системы, выполненный в соответствии с принятыми межмодульными интерфейсами. По своему определению модуль предполагает возможность без труда заменить его на другой при наличии заданных интерфейсов. Способы обособления составных частей ОС в отдельные модули могут существенно различаться, но чаще всего разделение происходит именно по функциональному признаку. В значительной степени разделение системы на модули определяется используемым методом проектирования ОС (снизу вверх или наоборот).

Особо важное значение при построении ОС имеют привилегированные, повторно входимые и реентерабельные модули, так как они позволяют более эффективно использовать ресурсы вычислительной системы. Как мы уже знаем, достижение реентерабельности реализуется различными способами. В некоторых системах реентерабельность программы получают автоматически, благодаря неизменяемости кодовых частей программ при исполнении (из-за особенностей системы команд машины), а также автоматическому распределению регистров, автоматическому отделению кодовых частей программ от данных и помещению последних в системную область памяти. Естественно, что для этого необходима соответствующая аппаратная поддержка. В других случаях это достигается программистами за счет использования специальных системных модулей.

Принцип модульности отражает технологические и эксплуатационные свойства системы. Наибольший эффект от его использования достижим в случае, когда принцип распространен одновременно на операционную систему, прикладные программы и аппаратуру.

Принцип функциональной избирательности

В ОС выделяется некоторая часть важных модулей, которые должны постоянно находиться в оперативной памяти для более эффективной организации вычислительного процесса. Эту часть в ОС называют ядром, так как это действительно основа системы. При формировании состава ядра требуется учитывать два противоречивых требования. В состав ядра должны войти наиболее часто используемые системные модули. Количество модулей должно быть таковым, чтобы объем памяти, занимаемый ядром, был бы не слишком большим. В состав ядра, как правило, входят модули по управлению системой прерываний, средства по переводу программ из состояния счета в состояние ожидания, готовности и обратно, средства по распределению таких основных ресурсов, как оперативная память и процессор. Помимо программных модулей, входящих в состав ядра и постоянно располагающихся в оперативной памяти, может быть много других системных программных модулей, которые получают название транзитных. Транзитные программные модули загружаются в оперативную память только при необходимости и в случае отсутствия свободного пространства могут быть замещены другими транзитными модулями. В качестве синонима к термину «транзитный» можно использовать термин «диск-резидентный».

Принцип генерируемости ОС

Основное положение этого принципа определяет такой способ исходного представления центральной системной управляющей программы ОС (ее ядра и основных компонентов, которые должны постоянно находиться в оперативной памяти), который позволял бы настраивать эту системную супервизорную часть, исходя из конкретной конфигурации конкретного вычислительного комплекса и круга решаемых задач. Эта процедура проводится редко, перед достаточно протяженным периодом эксплуатации ОС. Процесс генерации осуществляется с помощью специальной программы-генератора и соответствующего входного языка для этой программы, позволяющего описывать

программные возможности системы и конфигурацию машины. В результате генерации получается полная версия ОС. Сгенерированная версия ОС представляет собой совокупность системных наборов модулей и данных.

Упомянутый раньше принцип модульности положительно проявляется при генерации ОС. Он существенно упрощает настройку ОС на требуемую конфигурацию вычислительной системы. В наши дни при использовании персональных компьютеров с принципом генерируемости ОС можно столкнуться разве что только при работе с Linux. В этой UNIX-системе имеется возможность не только использовать какое-либо готовое ядро ОС, но и самому сгенерировать (скомпилировать) такое ядро, которое будет оптимальным для данного конкретного персонального компьютера и решаемых на нем задач. Кроме генерации ядра в Linux имеется возможность указать и набор подгружаемых драйверов и служб, то есть часть функций может реализовываться модулями, непосредственно входящими в ядро системы, а часть – модулями, имеющими статус подгружаемых, транзитных.

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

Принцип функциональной избыточности

Этот принцип учитывает возможность проведения одной и той же работы различными средствами. В состав ОС может входить несколько типов мониторов (модулей супервизора, управляющих тем или другим видом ресурса), различные средства организации коммуникаций между вычислительными процессами. Наличие нескольких типов мониторов, нескольких систем управления файлами позволяет пользователям быстро и наиболее адекватно адаптировать ОС к определенной конфигурации вычислительной системы, обеспечить максимально эффективную загрузку технических средств при решении конкретного класса задач, получить максимальную производительность при решении заданного класса задач.

Принцип виртуализации

Построение виртуальных ресурсов, их распределение и использование теперь используется практически в любой ОС. Этот принцип позволяет представить структуру системы в виде определенного набора планировщиков процессов и распределителей ресурсов (мониторов) и использовать единую централизованную схему распределения ресурсов.

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

Чаще виртуальная машина, предоставляемая пользователю, воспроизводит архитектуру реальной машины, но архитектурные элементы в таком представлении выступают с новыми или улучшенными характеристиками, часто упрощающими работу с системой. Характеристики могут быть произвольными, но чаще всего пользователи

желают иметь собственную «идеальную» по архитектурным характеристикам машину в следующем составе:

– единообразная по логике работы память (виртуальная) практически неограниченного объема. Среднее время доступа соизмеримо со значением этого параметра оперативной памяти. Организация работы с информацией в такой памяти производится в терминах обработки данных – в терминах работы с сегментами данных на уровне выбранного пользователем языка программирования;

– произвольное количество процессоров (виртуальных), способных работать параллельно и взаимодействовать во время работы. Способы управления процессорами, в том числе синхронизация и информационные взаимодействия, реализованы и доступны пользователям на уровне используемого языка в терминах управления процессами;

– произвольное количество внешних устройств (виртуальных), способных работать с памятью виртуальной машины параллельно или последовательно, асинхронно или синхронно по отношению к работе того или иного виртуального процессора, которые инициируют работу этих устройств. Информация, передаваемая или хранимая на виртуальных устройствах, не ограничена допустимыми размерами. Доступ к такой информации осуществляется на основе либо последовательного, либо прямого способа доступа в терминах соответствующей системы управления файлами. Предусмотрено расширение информационных структур данных, хранимых на виртуальных устройствах.

Степень приближения к «идеальной» виртуальной машине может быть большей или меньшей в каждом конкретном случае. Чем больше виртуальная машина, реализуемая средствами ОС на базе конкретной аппаратуры, приближена к «идеальной» по характеристикам машине и, следовательно, чем больше ее архитектурно-логические характеристики отличны от реально существующих, тем больше степень виртуальности у полученной пользователем машины.

Одним из аспектов виртуализации является организация возможности выполнения в данной ОС приложений, которые разрабатывались для других ОС. Другими словами, речь идет об организации нескольких операционных сред. Реализация этого принципа позволяет такой ОС иметь очень сильное преимущество перед аналогичными ОС, не имеющими такой возможности. Примером реализации принципа виртуализации может служить VDM-машина (virtual DOS machine) – защищенная подсистема, предоставляющая полную среду MS-DOS и консоль для выполнения MS-DOS приложений. Одновременно может выполняться практически произвольное число VDM-сессий. Такие VDM-машины имеются и в системах Microsoft Windows, и в OS/2.

Принцип независимости программ от внешних устройств

Этот принцип реализуется сейчас в подавляющем большинстве ОС общего применения. Мы уже говорили о нем, рассматривая принципы организации ввода/ вывода. Пожалуй, впервые наиболее последовательно данный принцип был реализован в ОС UNIX. Реализован он и в большинстве современных ОС для ПК. Напомним, этот принцип заключается в том, что связь программ с конкретными устройствами производится не на уровне трансляции программы, а в период планирования ее исполнения. В результате перекомпиляция при работе программы с новым устройством, на котором располагаются данные, не требуется.

Принцип позволяет одинаково осуществлять операции управления внешними устройствами независимо от их конкретных физических характеристик. Например, программе, содержащей операции обработки последовательного набора данных, безразлично, на каком носителе эти данные будут располагаться. Смена носителя и данных, размещаемых на них (при неизменности структурных характеристик данных), не принесет каких-либо изменений в программу, если в системе реализован принцип независимости.

Принцип совместимости

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

Необходимо разделять вопросы двоичной совместимости и совместимости на уровне исходных текстов приложений. Двоичная совместимость достигается в том случае, когда можно взять исполняемую программу и запустить ее на выполнение на другой ОС. Для этого необходимы: совместимость на уровне команд процессора, совместимость на уровне системных вызовов и даже на уровне библиотечных вызовов, если они являются динамически связываемыми.

Совместимость на уровне исходных текстов требует наличия соответствующего транслятора в составе системного программного обеспечения, а также совместимости на уровне библиотек и системных вызовов. При этом необходима перекомпиляция имеющихся исходных текстов в новый выполняемый модуль.

Гораздо сложнее достичь двоичной совместимости между процессорами, основанными на разных архитектурах. Для того чтобы один компьютер выполнял программы другого (например, программу для ПК типа IBM PC желательно выполнить на ПК типа Macintosh фирмы Apple), этот компьютер должен работать с машинными командами, которые ему изначально непонятны. В таком случае процессор типа 680x0 (или PowerPC) на Mac должен исполнять двоичный код, предназначенный для процессора i80x86. Процессор 80x86 имеет свои собственные дешифратор команд, регистры и внутреннюю архитектуру. Процессор 680x0 не понимает двоичный код 80x86, поэтому он должен выбрать каждую команду, декодировать ее, чтобы определить, для чего она предназначена, а затем выполнить эквивалентную подпрограмму, написанную для 680x0. Так как к тому же у 680x0 нет в точности таких же регистров, флагов и внутреннего арифметико-логического устройства, как в 80x86, он должен имитировать все эти элементы с использованием своих регистров или памяти. И он должен тщательно воспроизводить результаты каждой команды, что требует специально написанных подпрограмм для 680x0, гарантирующих, что состояние эмулируемых регистров и флагов после выполнения каждой команды будет в точности таким же, как и на реальном 80x86. Выходом в таких случаях является использование так называемых прикладных сред или эмуляторов. Учитывая, что основную часть программы, как правило, составляют вызовы библиотечных функций, прикладная среда имитирует библиотечные функции целиком, используя заранее написанную библиотеку функций аналогичного назначения, а остальные команды эмулирует каждую по отдельности.

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

Принцип открытой и наращиваемой ОС

Открытая ОС доступна для анализа как пользователям, так и системным специалистам, обслуживающим вычислительную систему. Наращиваемая (модифицируемая, развиваемая) ОС позволяет не только использовать возможности генерации, но и вводить в ее состав новые модули, совершенствовать существующие и т. д. Другими словами, необходимо, чтобы можно было легко внести дополнения и изменения, если это потребуется, и не нарушить целостность системы. Прекрасные возможности для расширения предоставляет подход к структурированию ОС по типу клиент–сервер с использованием микроядерной технологии. В соответствии с этим подходом ОС строится как совокупность привилегированной управляющей программы и набора непривилегированных услуг – «серверов». Основная часть ОС остается неизменной и в то же время могут быть добавлены новые серверы или улучшены старые.

Этот принцип иногда трактуют как расширяемость системы.

К открытым ОС, прежде всего, следует отнести UNIX-системы и, естественно, ОС Linux.

Принцип мобильности (переносимости)

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

Написание переносимой ОС аналогично написанию любого переносимого кода – нужно следовать некоторым правилам. Во-первых, большая часть ОС должна быть написана на языке, который имеется на всех системах, на которые планируется в дальнейшем ее переносить. Это, прежде всего, означает, что ОС должна быть написана на языке высокого уровня, предпочтительно стандартизованном, например на языке С. Программа, написанная на ассемблере, не является в общем случае переносимой. Во-вторых, важно минимизировать или, если возможно, исключить те части кода, которые непосредственно взаимодействуют с аппаратными средствами. Зависимость от аппаратуры может иметь много форм. Некоторые очевидные формы зависимости включают прямое манипулирование регистрами и другими аппаратными средствами. Наконец, если аппаратно-зависимый код не может быть полностью исключен, то он должен быть изолирован в нескольких хорошо локализуемых модулях. Аппаратно-зависимый код не должен быть распределен по всей системе. Например, можно спрятать аппаратно-зави-симую структуру в программно задаваемые данные абстрактного типа. Другие модули системы будут работать с этими данными, а не с аппаратурой, используя набор некоторых функций. Когда ОС переносится, то изменяются только эти данные и функции, которые ими манипулируют.

Введение стандартов POSIX преследовало цель обеспечить переносимость создаваемого программного обеспечения.

Принцип обеспечения безопасности вычислений

Обеспечение безопасности при выполнении вычислений является желательным свойством для любой многопользовательской системы. Правила безопасности определяют такие свойства, как защита ресурсов одного пользователя от других и установление квот по ресурсам для предотвращения захвата одним пользователем всех системных ресурсов (таких, как память).

Обеспечение защиты информации от несанкционированного доступа является обязательной функцией сетевых операционных систем. Во многих современных ОС гарантируется степень безопасности данных, соответствующая уровню С2 в системе стандартов США. Основы стандартов в области безопасности были заложены в документе «Критерии оценки надежных компьютерных систем». Этот документ, изданный Национальным центром компьютерной безопасности (NCSC – National Computer Security Center) в США в 1983 году, часто называют Оранжевой книгой.

В соответствии с требованиями Оранжевой книги безопасной считается система, которая «посредством специальных механизмов защиты контролирует доступ к информации таким образом, что только имеющие соответствующие полномочия лица или процессы, выполняющиеся от их имени, могут получить доступ на чтение, запись, создание или удаление информации».

Иерархия уровней безопасности, приведенная в Оранжевой книге, помечает низший уровень безопасности как D, а высший – как A.

В класс D попадают системы, оценка которых выявила их несоответствие требованиям всех других классов.

Основными свойствами, характерными для систем класса С, являются наличие подсистемы учета событий, связанных с безопасностью, и избирательный контроль доступа. Класс (уровень) С делится на 2 подуровня: уровень С1, обеспечивающий защиту данных от ошибок пользователей, но не от действий злоумышленников; и более строгий уровень С2. На уровне С2 должны присутствовать:

– средства секретного входа, обеспечивающие идентификацию пользователей путем ввода уникального имени и пароля перед тем, как им будет разрешен доступ к системе;

– избирательный контроль доступа, позволяющий владельцу ресурса определить, кто имеет доступ к ресурсу и что он может с ним делать. Владелец делает это путем предоставляемых прав доступа пользователю или группе пользователей;

– средства учета и наблюдения (auditing), обеспечивающие возможность обнаружить и зафиксировать важные события, связанные с безопасностью, или любые попытки создать, получить доступ или удалить системные ресурсы;

– защита памяти, заключающаяся в том, что память инициализируется перед тем, как повторно используется.

На этом уровне система не защищена от ошибок пользователя, но поведение его может быть проконтролировано по записям в журнале, оставленным средствами наблюдения и аудита.

Системы уровня В основаны на помеченных данных и распределении пользователей по категориям, то есть реализуют мандатный контроль доступа. Каждому пользователю присваивается рейтинг защиты, и он может получать доступ к данным только в соответствии с этим рейтингом. Этот уровень в отличие от уровня С защищает систему от ошибочного поведения пользователя.

Уровень А является самым высоким уровнем безопасности, он требует в дополнение ко всем требованиям уровня В выполнения формального, математически обоснованного доказательства соответствия системы требованиям безопасности.

Различные коммерческие структуры (например, банки) особо выделяют необходимость учетной службы, аналогичной той, что предлагают государственные рекомендации С2. Любая деятельность, связанная с безопасностью, может быть отслежена и тем самым учтена. Это как раз то, чего требует стандарт для систем класса С2, и что обычно нужно банкам. Однако коммерческие пользователи, как правило, не хотят расплачиваться производительностью за повышенный уровень безопасности. А-уровень безопасности занимает своими управляющими механизмами до 90 % процессорного времени, что, безусловно, в большинстве случаев уже неприемлемо. Более безопасные системы не только снижают эффективность, но и существенно ограничивают число доступных прикладных пакетов, которые соответствующим образом могут выполняться в подобной системе. Например, для ОС Solaris (версия UNIX) есть несколько тысяч приложений, а для ее аналога В-уровня – только около ста.

Микроядерные операционные системы

Микроядро – это минимальная стержневая часть операционной системы, служащая основой модульных и переносимых расширений. Существует мнение, что большинство операционных систем следующих поколений будут обладать микроядрами. Однако имеется масса разных мнений по поводу того, как следует организовывать службы операционной системы по отношению к микроядру: как проектировать драйверы устройств, чтобы добиться наибольшей эффективности, но сохранить функции драйверов максимально независимыми от аппаратуры; следует ли выполнять операции, не относящиеся к ядру, в пространстве ядра или в пространстве пользователя; стоит ли сохранять программы имеющихся подсистем (например, UNIX) или лучше отбросить все и начать с нуля.

Основная идея, заложенная в технологию микроядра, будь то собственно ОС или ее графический интерфейс, заключается в том, чтобы конструировать необходимую среду

верхнего уровня, из которой можно легко получить доступ ко всем функциональным возможностям уровня аппаратного обеспечения. При такой структуре ядро служит стартовой точкой для создания системы. Искусство разработки микроядра заключается в выборе базовых примитивов, которые должны в нем находиться для обеспечения необходимого и достаточного сервиса. В микроядре содержится и исполняется минимальное количество кода, необходимое для реализации основных системных вызовов. В число этих вызовов входят передача сообщений и организация другого общения между внешними по отношению к микроядру процессами, поддержка управления прерываниями, а также ряд некоторых других функций. Остальные функции, характерные для «обычных» (не микроядерных) ОС, обеспечиваются как модульные дополнения-процессы, взаимодействующие главным образом между собой и осуществляющие взаимодействие посредством передачи сообщений.

Микроядро является маленьким, передающим сообщения модулем системного программного обеспечения, работающим в наиболее приоритетном состоянии компьютера и поддерживающим остальную часть операционной системы, рассматриваемую как набор серверных приложений. Интерес к микроядрам возрастал по мере того, как системные разработчики реагировали на сложность современных реализаций операционных систем, и поддержан тем, что исследовательское сообщество успешно продемонстрировало реализуемость концепции микроядра.

Созданная в университете Карнеги Меллон технология микроядра Mach служит основой для многих ОС.

Исполняемые микроядром функции ограничены в целях сокращения его размеров и максимизации количества кода, работающего как прикладная программа.

Микроядро включает только те функции, которые требуются для определения набора абстрактных сред обработки для прикладных программ и для организации совместной работы приложений в обеспечении сервисов и в действии клиентами и серверами. В результате микроядро обеспечивает только пять различных типов сервисов:

1. управление виртуальной памятью;
2. задания и потоки;
3. межпроцессные коммуникации (IPC);
4. управление поддержкой ввода/вывода и прерываниями;
5. сервисы набора хоста и процессора.

Другие подсистемы и функции операционной системы, такие как системы управления файлами, поддержка внешних устройств и традиционные программные интерфейсы, размещаются в одном или более системных сервисах либо в задаче операционной системы. Эти программы работают как приложения микроядра.

Следуя концепции множественных потоков на одно задание, микроядро создает прикладную среду, обеспечивающую использование мультипроцессоров без требования, чтобы какая-то конкретная машина была мультипроцессорной: на однопроцессорной различные потоки просто выполняются в разные времена. Вся поддержка, требуемая для мультипроцессорных машин, сконцентрирована в сравнительно малом и простом микроядре.

Благодаря своим размерам и способности поддерживать стандартные сервисы программирования и характеристики в виде прикладных программ сами микроядра проще, чем ядра монолитных или модульных операционных систем. С микроядром функция операционной системы разбивается на модульные части, которые могут быть сконфигурированы целым рядом способов, позволяя строить большие системы добавлением частей к меньшим. Например, каждый аппарат -но-независимый нейтральный сервис логически отделен и может быть сконфигурирован в широком диапазоне способов. Микроядра также облегчают поддержку мультипроцессоров созданием стандартной программной среды, которая может использовать множественные процессоры в случае их наличия, однако не требует их, если их нет. Специализированный

код для мультипроцессоров ограничен самим микроядром. Более того, сети из общающихся между собой микроядер могут быть использованы для обеспечения операционной системной поддержки возникающего класса массивно параллельных машин. В некоторых случаях определенной сложностью использования микроядерного подхода на практике является замедление скорости выполнения системных вызовов при передаче сообщений через микроядро по сравнению с классическим подходом. С другой стороны, можно констатировать и обратное. Поскольку микроядра малы и имеют сравнительно мало требуемого к исполнению кода уровня ядра, они обеспечивают удобный способ поддержки характеристик реального времени, требующихся для мультимедиа, управления устройствами и высокоскоростных коммуникаций. Наконец, хорошо структурированные микроядра обеспечивают изолирующий слой для аппаратных различий, которые не маскируются применением языков программирования высокого уровня. Таким образом, они упрощают перенесение кода и увеличивают уровень его повторного использования.

Наиболее ярким представителем микроядерных ОС является ОС реального времени QNX. Микроядро QNX поддерживает только планирование и диспетчеризацию процессов, взаимодействие процессов, обработку прерываний и сетевые службы нижнего уровня. Микроядро обеспечивает всего лишь пару десятков системных вызовов, но благодаря этому оно может быть целиком размещено во внутреннем кэше даже таких процессоров, как Intel 486. Как известно, разные версии этой ОС имели и различные объемы ядер – от 8 до 46 Кбайт.

Чтобы построить минимальную систему QNX, требуется добавить к микроядру менеджер процессов, который создает процессы, управляет процессами и памятью процессов. Чтобы ОС QNX была применима не только во встроенных и бездисковых системах, нужно добавить файловую систему и менеджер устройств. Эти менеджеры исполняются вне пространства ядра, так что ядро остается небольшим.

Монолитные операционные системы

Монолитные ОС являются прямой противоположностью микроядерным ОС. В монолитной ОС, несмотря на ее возможную сильную структуризацию, очень трудно удалить один из уровней многоуровневой модульной структуры. Добавление новых функций и изменение существующих для монолитных ОС требует очень хорошего знания всей архитектуры ОС и чрезвычайно больших усилий. Поэтому более современный подход к проектированию ОС, который может быть условно назван как «клиент-серверная» технология, позволяет в большей мере и с меньшими трудозатратами реализовать перечисленные выше принципы проектирования ОС.

Модель клиент-сервер предполагает наличие программного компонента, являющегося потребителем какого-либо сервиса – клиента, и программного компонента, служащего поставщиком этого сервиса – сервера. Взаимодействие между клиентом и сервером стандартизируется, так что сервер может обслуживать клиентов, реализованных различными способами и, может быть, разными разработчиками. При этом главным требованием является использование единообразного интерфейса. Инициатором обмена обычно является клиент, который посылает запрос на обслуживание серверу, находящемуся в состоянии ожидания запроса. Один и тот же программный компонент может быть клиентом по отношению к одному виду услуг и сервером для другого вида услуг. Модель клиент-сервер является скорее удобным концептуальным средством ясного представления функций того или иного программного элемента в какой-либо ситуации, нежели технологией. Эта модель успешно применяется не только при построении ОС, но и на всех уровнях программного обеспечения и имеет в некоторых случаях более узкий, специфический смысл, сохраняя, естественно, при этом все свои общие черты.

При поддержке монолитных ОС возникает ряд проблем, связанных с тем, что все функции макроядра работают в едином адресном пространстве. Во-первых, это опасность

возникновения конфликта между различными частями ядра; во-вторых – сложность подключения к ядру новых драйверов. Преимущество микроядерной архитектуры перед монолитной заключается в том, что каждый компонент системы представляет собой самостоятельный процесс, запуск или остановка которого не отражается на работоспособности остальных процессов.

Микроядерные ОС в настоящее время разрабатываются чаще монолитных. Однако следует заметить, что использование технологии клиент-сервер – это еще не гарантия того, что ОС станет микроядерной. В качестве подтверждения можно привести пример с ОС Windows NT, которая построена на идеологии клиент-сервер, но которую тем не менее трудно назвать микроядерной. Для того чтобы согласиться с таким высказыванием, достаточно сравнить ОС QNX и ОС Windows NT.

Требования, предъявляемые к ОС реального времени

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

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

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

Примерами систем реального времени являются системы управления атомными электростанциями или какими-нибудь технологическими процессами, медицинского мониторинга, управления вооружением, космической навигации, разведки, управления лабораторными экспериментами, управления автомобильными двигателями, робототехника, телеметрические системы управления, системы антиблокировки тормозов, системы сигнализации – список в принципе бесконечен.

Иногда можно услышать из разговоров специалистов, что различают системы «мягкого» и «жесткого» реального времени. Различие между жесткой и мягкой СРВ зависит от требований к системе – система считается жесткой, если «нарушение временных ограничений не допустимо», и мягкой, если «нарушение временных ограничений нежелательно». Ведется множество дискуссий о точном смысле терминов «жесткая» и «мягкая» СРВ. Можно даже аргументировать, что мягкая СРВ не является СРВ вовсе, ибо основное требование соблюдения временных ограничений не выполнено. В действительности термин СРВ часто неправомерно применяют по отношению к быстрым системам.

Часто путают понятия СРВ и ОСРВ, а также неправильно используют атрибуты «мягкая» и «жесткая». Иногда говорят, что та или иная ОСРВ мягкая или жесткая. Нет мягких или жестких ОСРВ. ОСРВ может только служить основой для построения мягкой или жесткой СРВ. Сама по себе ОСРВ не препятствует тому, что ваша СРВ будет мягкой. Например, вы решили создать СРВ, которая должна работать через Ethernet по протоколу TCP/IP. Такая система не может быть жесткой СРВ, поскольку сама сеть Ethernet в принципе непредсказуема вследствие использования случайного метода доступа к среде передачи данных, в отличие, например, от IBM Token Ring или ARC-Net, в которых используются детерминированные методы доступа.

Итак, перечислим основные требования к ОСРВ.

Мультипрограммность и многозадачность

Требование 1. ОС должна быть мультипрограммной и многозадачной (многопоточной – multi-threaded) и активно использовать прерывания для диспетчеризации.

Как указывалось выше, ОСРВ должна быть предсказуемой. Это означает не то, что ОСРВ должна быть быстрой, а то, что максимальное время выполнения того или иного действия должно быть известно заранее и должно соответствовать требованиям приложения. Так, например, ОС Windows 3.11 даже на Pentium III 1000 MHz бесполезна для ОСРВ, ибо одно приложение может захватить управление и заблокировать систему для остальных.

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

Приоритеты задач (потоков)

Требование 2. В ОС должно существовать понятие приоритета потока.

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

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

Наследование приоритетов

Требование 3. В ОС должна существовать система наследования приоритетов.

На самом деле именно этот механизм синхронизации и тот факт, что различные треды используют одно и то же пространство памяти, отличают их от процессов. Как мы уже знаем, процессы почти не разделяют одно и то же пространство памяти, а в основном работают в своих локальных адресных пространствах. Так, например, старые версии UNIX не являются мультитредовыми (multi-threaded). «Старый» UNIX – многозадачная ОС, где задачами являются процессы (а не треды), которые сообщаются через потоки (pipes) и разделяемую память. Оба эти механизма используют файловую систему, а ее поведение – непредсказуемо.

Комбинация приоритетов тредов и разделения ресурсов между ними приводит к другому явлению – классической проблеме инверсии приоритетов. Это можно проиллюстрировать на примере, когда есть как минимум три треда. Когда тред низшего приоритета захватил ресурс, разделяемый с тредом высшего приоритета, и начал выполняться поток среднего приоритета, выполнение треда высшего приоритета будет приостановлено, пока не освободится ресурс и не отработает тред среднего приоритета. В этой ситуации время, необходимое для завершения треда высшего приоритета, зависит от нижних приоритетных уровней, – это и есть инверсия приоритетов. Ясно, что в такой ситуации трудно выдержать ограничение на время исполнения.

Чтобы устранить такие инверсии, ОСРВ должна допускать наследование приоритета, то есть повышение уровня приоритета треда до уровня треда, который его вызывает. Наследование означает, что блокирующий ресурс тред наследует приоритет треда, который он блокирует (разумеется, это справедливо лишь в том случае, если блокируемый тред имеет более высокий приоритет).

Иногда можно услышать утверждение, что в грамотно спроектированной системе такая проблема не возникает. В случае сложных систем с этим нельзя согласиться. Единственный способ решения этой проблемы состоит в увеличении приоритета треда «вручную» прежде, чем ресурс окажется заблокированным. Разумеется, это возможно в случае, когда два треда разных приоритетов претендуют на один ресурс. В общем случае решения не существует.

Синхронизация процессов и задач

Требование 4. ОС должна обеспечивать мощные, надежные и удобные механизмы синхронизации задач.

Так как задачи разделяют данные (ресурсы) и должны сообщаться друг с другом, представляется логичным, что должны существовать механизмы блокирования и коммуникации. Необходимы механизмы, гарантированно предоставляющие возможность параллельно выполняющимся задачам и процессам оперативно обмениваться сообщениями и синхросигналами. Эти системные механизмы должны быть всегда доступны процессам, требующим реального времени. Следовательно, системные ресурсы для их функционирования должны быть распределены заранее.

Предсказуемость

Требование 5. Поведение ОС должно быть известно и достаточно точно прогнозируемо.

Времена выполнения системных вызовов и временные характеристики поведения системы в различных обстоятельствах должны быть известны разработчику. Поэтому создатель ОСРВ должен приводить следующие характеристики:

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

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

– максимальное время маскирования прерываний драйверами и ОС.

Принципы построения интерфейсов операционных систем

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

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

– запуск, приостанов и снятие задачи с выполнения; О задание или изменение приоритета задачи;

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

– RPC (remote procedure call) – удаленный вызов подпрограмм.

2. Управление памятью:

– запрос на выделение блока памяти; О освобождение памяти;

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

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

3. Управление вводом/выводом:

– запрос на управление виртуальными устройствами (напомним, что управление вводом/выводом является привилегированной функцией самой ОС, и никакая из пользовательских задач не должна иметь возможности непосредственно управлять устройствами);

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

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

Поясним также, что управление GUI – частный случай задачи управления вводом/выводом, не являющийся частью ядра операционной системы, хотя в ряде случаев разработчики ОС относят функции GUI к основному системному API.

Следует отметить, что имеются два основных подхода к управлению задачами. Так, в одних системах порождаемая задача наследует все ресурсы задачи-родителя, тогда как в других системах существуют равноправные отношения, и при порождении нового процесса ресурсы для него запрашиваются у операционной системы.

Обращения к операционной системе, в соответствии с имеющимся API, может осуществляться как посредством вызова подпрограммы с передачей ей необходимых параметров, так и через механизм программных прерываний. Выбор метода реализации вызовов функций API должен определяться архитектурой платформы.

Так, например, в операционной системе MS-DOS, которая разрабатывалась для однозадачного режима (поскольку процессор i8086 не поддерживал мультипрограммирование), использовался механизм программных прерываний. При этом основной набор функций API был доступен через точку входа обработчика int 21h.

В более сложных системах имеется не одна точка входа, а множество – по количеству функций API. Так, в большинстве операционных систем используется метод вызова подпрограмм. В этом случае вызов сначала передается в модуль API (например, это библиотека времени выполнения), который и перенаправляет вызов соответствующим обработчикам программных прерываний, входящим в состав операционной системы. Использование механизма прерываний вызвано, главным образом, тем, что при этом процессор переводится в режим супервизора.

Интерфейс прикладного программирования

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

– API как интерфейс высокого уровня, принадлежащий к библиотекам RTL;

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

– прочие API.

Интерфейс прикладного программирования, как это и следует из названия, предназначен для использования прикладными программами системных ресурсов ОС и

реализуемых ею функций. API описывает совокупность функций и процедур, принадлежащих ядру или надстройкам ОС.

Итак, API представляет собой набор функций, предоставляемых системой программирования разработчику прикладной программы и ориентированных на организацию взаимодействия результирующей прикладной программы с целевой вычислительной системой. Целевая вычислительная система представляет собой совокупность программных и аппаратных средств, в окружении которых выполняется результирующая программа. Сама результирующая программа порождается системой программирования на основании кода исходной программы, созданного разработчиком, а также объектных модулей и библиотек, входящих в состав системы программирования.

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

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

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

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

Существует несколько вариантов реализации API:

– реализация на уровне ОС;

– реализация на уровне системы программирования;

– реализация на уровне внешней библиотеки процедур и функций.

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

Возможности API можно оценивать со следующих позиций:

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

– широта предоставляемых возможностей;

– зависимость прикладной программы от архитектуры целевой вычислительной системы.

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

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

Что касается двух других показателей, то в принципе нет никаких технических ограничений на их реализацию. Однако существуют организационные проблемы и узкие корпоративные интересы, тормозящие создание такого рода библиотек.

Реализация функций API на уровне ОС

При реализации функций API па уровне ОС за их выполнение ответственность несет ОС. Объектный код, выполняющий функции, либо непосредственно входит в состав ОС (или даже ядра ОС), либо поставляется в составе динамически загружаемых библиотек,

разработанных для данной ОС. Система программирования ответственна только за то, чтобы организовать интерфейс для вызова этого кода.

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

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

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

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

Хорошо известным примером API такого рода может служить набор функций, предоставляемых пользователю со стороны ОС типа Microsoft Windows – WinAPI (Windows API). Надо сказать, что даже внутри этого корпоративного API существует определенная несогласованность, которая несколько ограничивает переносимость программ между различными ОС типа Windows. Еще одним примером такого API можно считать набор сервисных функций ОС типа MS-DOS, реализованный в виде набора подпрограмм обслуживания программных прерываний.

Реализация функций API на уровне системы программирования

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

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

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

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

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

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

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

Реализация функций API с помощью внешних библиотек

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

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

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

Если говорить о переносимости исходного кода, то здесь требование только одно – используемая внешняя библиотека должна быть доступна в любой из архитектур вычислительных систем, на которые ориентирована прикладная программа. Тогда удается достигнуть переносимости. Это возможно, если используемая библиотека удовлетворяет какому-то принятому стандарту, а система программирования поддерживает этот стандарт.

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

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

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

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

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

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

Итак, нами были рассмотрены основные принципы, цели и подходы к реализации системных API. Отметим еще один очень важный момент: желательно, чтобы интерфейс прикладного программирования не зависел от системы программирования. Конечно, были одно время персональные компьютеры, у которых базовой системой программирования выступал интерпретатор с языка Basic, но это скорее исключение. Обычно базовые функции API не зависят от системы программирования и могут использоваться из любой системы программирования, хотя и с применением соответствующих правил построения вызывающих последовательностей. В то же время в ряде случаев система программирования может сама генерировать обращения к функциям API. Например, мы можем написать в программе вызов функции по запросу 256 байт памяти

unsigned char * ptr - malloc (256);

Система программирования языка С сгенерирует целую последовательность обращений. Из кода пользовательской программы будет осуществлен вызов библиотечной функции malloc, код которой расположен в RTL языка С. Библиотека времени выполнения в данном случае реализует вызов malloc уже как вызов системной функции API HeapAlloc

LPVOID HeapAlloc(
HANDLE hHeap. // handle to the private heap block - указатель на блок
DWORD dwFlags. // heap allocation control flags - свойства блока
DWORD dwBytes // number of bytes to allocate - размер блока):

Параметры выделяемого блока памяти в таком случае задаются системой программирования, и пользователь лишен возможности задавать их напрямую. С другой стороны, если это необходимо, возможно использование функций API прямо в тексте программы.

unsigned char * ptr - (LPVOID) HeapAlloc (GetProcessHeapO. 0. 256);

В этом случае программирование вызова немного усложняется, но получаемый конечный результат будет, как правило, короче и, что самое важное, будет работать эффективнее. Следует отметить, что далеко не все возможности API доступны через обращения к функциям системы программирования. Непосредственное обращение к функциям API позволяет пользователю обращаться к системным ресурсам более эффективным способом. Однако это требует знания функций API, количество которых нередко достигает нескольких сотен.

Как правило, API не стандартизированы. В каждом конкретном случае набор вызовов API определяется, прежде всего, архитектурой ОС и ее назначением. В то же время принимаются попытки стандартизировать некоторый базовый набор функций, поскольку это существенно облегчает перенос приложений с одной ОС в другую. Таким примером может служить очень известный и, пожалуй, один из самых распространенных стандартов – стандарт POSIX. В этом стандарте перечислен большой набор функций, их параметров и возвращаемых значений. Стандартизированными, согласно POSIX, являются не только обращения к API, но и файловая система, организация доступа к внешним устройствам, набор системных команд1. Использование в приложениях этого стандарта позволяет в дальнейшем легко переносить такие программы с одной ОС в другую путем простейшей перекомпиляции исходного текста.

Частным случаем попытки стандартизации API является внутренний корпоративный стандарт компании Microsoft, известный как WinAPI. Он включает в себя следующие реализации: Winl6, Win32s, Win32, WinCE. С точки зрения WinAPI (в силу ряда идеологических причин – обязательный графический «оконный» интерфейс пользователя), базовой задачей является окно. Таким образом, WinAPI изначально ориентирован на работу в графической среде. Однако базовые понятия дополнены традиционными функциями, в том числе частично поддерживается стандарт POSIX.

Платформенно-независимый интерфейс POSIX

POSIX (Portable Operating System Interface for Computer Environments) – платформонезависимый системный интерфейс для компьютерного окружения. Это стандарт IEEE, описывающий системные интерфейсы для открытых операционных систем, в том числе оболочки, утилиты и инструментарии. Помимо этого, согласно POSIX, стандартизированными являются задачи обеспечения безопасности, задачи реального времени, процессы администрирования, сетевые функции и обработка транзакций. Стандарт базируется на UNIX-системах, но допускает реализацию и в других ОС.

POSIX возник как попытка всемирно известной организации IEEE1 пропагандировать переносимость приложений в UNIX-средах путем разработки абстрактного, платформенно-независимого стандарта. Однако POSIX не ограничивается только UNIX-системами; существуют различные реализации этого стандарта в системах, которые соответствуют требованиям, предъявляемым стандартом IEEE Standard 1003.1-1990 (POSIX.1). Например, известная ОС реального времени QNX соответствует спецификациям этого стандарта, что облегчает перенос приложений в эту систему, но

UNIX-системой не является ни в каком виде, ибо ее архитектура использует абсолютно иные принципы.

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

В табл. 4.1 приведены основные направления, описываемые данными стандартами. Следует также особо отметить, что POSIX.1 предполагает язык С как основной язык описания системных функций API.

Таблица 4.1. Семейство стандартов POSIX

Стандарт	Стандарт ISO	Краткое описание
POSIX.0	Нет	Введение в стандарт открытых систем. Данный документ не является стандартом в чистом виде, а представляет собой рекомендации и краткий обзор технологий
POSIX.1	Да	Системный API (язык С)
POSIX.2	Нет	Оболочки и утилиты (одобренные IEEE)
POSIX.3	Нет	Тестирование и верификация
POSIX.4	Нет	Задачи реального времени и нити
POSIX.5	Да	Использование языка ADA применительно к стандарту POSIX.1
POSIX.6	Нет	Системная безопасность
POSIX.7	Нет	Администрирование системы
POSIX.8	Нет	Сети «Прозрачный» доступ к файлам Абстрактные сетевые интерфейсы, не зависящие от физических протоколов RPC (remote procedure calls, вызовы удаленных процедур) Связь системы с протоколозависимыми приложениями
P0SIX.9	Да	Использование языка FORTRAN применительно к стандарту POSIX.1
POSIX.10	Нет	Super-computing Application Environment Profile (AEP)
POSIX. 11	Нет	Обработка транзакций AEP
POSIX.12	Нет	Графический интерфейс пользователя (GUI)

Таким образом, программы, написанные с соблюдением данных стандартов, будут одинаково выполняться на всех POSIX-совместимых системах. Однако стандарт в некоторых случаях носит лишь рекомендательный характер. Часть стандартов описана очень строго, тогда как другая часть только поверхностно раскрывает основные требования. Нередко программные системы заявляются как POSIX-совместимые, хотя таковыми их назвать нельзя. Причины кроются в формальности подхода к реализации POSIX-интерфейса в различных ОС. На рис. 4.1 изображена типовая схема реализации строго соответствующего POSIX приложения.

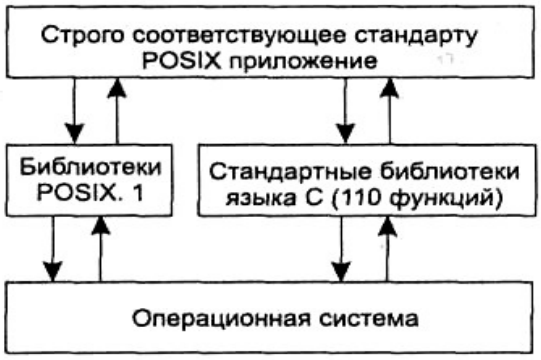

Рис. 4.1. Приложения, строго соответствующие стандарту POSIX

Из рис. 5.1 видно, что для взаимодействия с операционной системой программа использует только библиотеки POSIX.1 и стандартную библиотеку RTL языка С, в которой возможно использование лишь 110 различных функций, также описанных стандартом POSIX.1.

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

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

Пример программирования в различных API ОС

Для наглядной демонстрации принципиальных различий API наиболее популярных современных операционных систем для ПК рассмотрим простейший пример, в котором реализуется следующая задача.

Постановка задачи: необходимо подсчитать количество пробелов в текстовых файлах, имена которых должны указываться в командной строке. Рассмотрим два варианта программы, решающей эту задачу, – для Windows (с использованием WinAPI) и для Linux (POSIX API).

Поскольку нас интересует работа с параллельными задачами, пусть при выполнении программы для каждого перечисленного в командной строке файла создается свой процесс или задача (тред), который параллельно с другими процессами (тредами) производит работу по подсчету пробелов в «своем» файле. Результатом работы программы будет являться список файлов с подсчитанным количеством пробелов для каждого.

Следует обратить особое внимание на то, что приведенная ниже конкретная реализация данной задачи не является единственно возможной. В обеих рассматриваемых операционных системах существуют различные методы как работы с файловой системой, так и управления процессами. В данном случае рассматривается только один, но наиболее характерный для соответствующего API вариант.

Текст программы для Windows (WinAPI)

Для того чтобы было удобнее сравнивать эту и следующую программы, а также из-за того, что настоящая задача не требует для своего решения оконного интерфейса, в нижеприведенном тексте использованы только те вызовы API, которые не затрагивают графический интерфейс. Конечно, нынче редко какое приложение не использует возможностей GUI, но в нашем случае зато сразу можно увидеть разницу в организации параллельной работы запускаемых вычислений.

```
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>

// Название: processFile
// Описание: исполняемый код треда
// Входные параметры: lpFileName – имя файла для обработки
// Выходные параметры: нет
```

```
//
DWORD processFile(LPVOID lpFileName) {
    HANDLE handle;        // описатель файла
    DWORD numRead, total = 0;
    char buf;

    // запрос к ОС на открытие файла (только для чтения)
    handle = CreateFile( (LPCTSTR)lpFileName, GENERIC_READ,
        FILE_SHARE_READ, NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL,
        NULL);

    // цикл чтения до конца файла
    do {
           // чтение одного символа из файла
           ReadFile( handle, (LPVOID) &buf, 1, &numRead, NULL);
           if (buf == 0x20) total++;
    } while ( numRead > 0);

    fprintf( stderr, "(ThreadID: %Lu), File %s, spaces = %d\n",
        GetCurrentThreadId(), lpFileName, total);

    // закрытие файла
    CloseHandle( handle);

    return(0);
}

// Название: main
// Описание: главная программа
// Входные параметры: список имен файлов для обработки
// Выходные параметры: нет
//
int main(int argc, char *argv[]) {
int i;
DWORD pid;
HANDLE hThrd[255];    // массив ссылок на треды

    // для всех файлов, перечисленных в командной строке
    for (i = 0; i< (argc-1); i++) {
       // запуск треда – обработка одного файла
       hThrd[i] = CreateThread( NULL, 0x4000,
           (LPTHREAD_START_ROUTINE) processFile,
                           (LPVOID) argv[i+1], 0, &pid);
       fprintf( stdout, "processFile started (HND=%d)\n", hThrd[i]);
    }

    // ожидание окончания выполнения всех запущенных тредов
```

Обратите внимание, что основная программа запускает треды и ждет окончания их выполнения.

Текст программы для Linux (POSIX API)

```
    WaitForMultipleObjects( argc-1, hThrd, true, INFINITE);

    return(0);
}
```

```
#include <sys/types.h>
#include <sys/stat.h>
#include <wait.h>
#include <fcntl.h>
#include <stdio.h>

// Название: processFile
// Описание: обработка файла, подсчет кол-ва пробелов
// Входные параметры: fileName – имя файла для обработки
// Выходные параметры: кол-во пробелов в файле
//
int processFile( char *fileName) {
    int handle, numRead, total = 0;
    char buf;

    // запрос к ОС на открытие файла (только для чтения)
    handle = open( fileName, O_RDONLY);

    // цикл чтения до конца файла
    do {
        // чтение одного символа из файла
        numRead = read( handle, &buf, 1);
        if (buf == 0x20) total++;
    } while (numRead > 0);

    // закрытие файла
    close( handle);
    return( total);
}

// Название: main
// Описание: главная программа
// Входные параметры: список имен файлов для обработки
// Выходные параметры: нет
//
int main(int argc, char *argv[]) {
int i, pid, status;

 // для всех файлов, перечисленных в командной строке
 for (i = 1; i< argc; i++) {
    // запускаем дочерний процесс
    pid = fork();
    if (pid == 0) {
        // если выполняется дочерний процесс
        // вызов функции счета количества пробелов в файле
        printf( "(PID: %d), File %s, spaces = %d\n", getpid(), argv[ i], processFile(
argv[ i]));
        // выход из процесса
        exit();
    }
    // если выполняется родительский процесс
    else
    printf( "processFile started (pid=%d)\n", pid);
 }

// ожидание окончания выполнения всех запущенных процессов
 if (pid != 0) while (wait(&status)>0);
 return;
}
```

Из этого текста видно, что в этом случае все вычисления принимают статус процессов, а не тредов.

В заключение можно заметить, что очень трудно сравнивать API. При их разработке создатели, как правило, стараются реализовать полный набор основных функций, используя которые можно решать различные задачи, хотя, порой, и различными способами. Один набор будет хорош для одного набора задач, другой - для иного набора задач. Тем более что фактически у нас сейчас существенно ограниченное множество API. Причина в том, что доминируют наиболее распространенные ОС, на распространение которых в большей степени оказали влияние не достоинства или недостатки этих ОС и их API, а правильная маркетинговая политика фирм, их создавших.

6 семестр

1. Место и роль локальных сетей

1.1. Немного истории компьютерной связи

Связь на небольшие расстояния в компьютерной технике существовала еще задолго до появления первых персональных компьютеров.

К большим компьютерам (mainframes), присоединялись многочисленные терминалы (или «интеллектуальные дисплеи»). Правда, интеллекта в этих терминалах было очень мало, практически никакой обработки информации они не делали, и основная цель организации связи состояла в том, чтобы разделить интеллект («машинное время») большого мощного и дорогого компьютера между пользователями, работающими за этими терминалами. Это называлось **режимом разделения времени**, так как большой компьютер последовательно во времени решал задачи множества пользователей. В данном случае достигалось совместное использование самых дорогих в то время ресурсов, вычислительных (рис. 1.1).

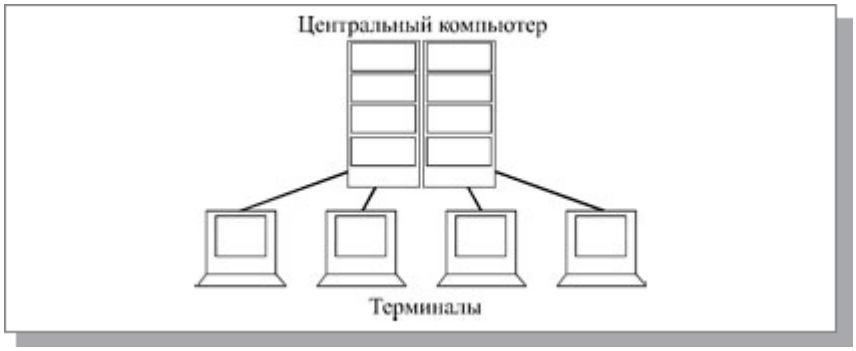

Рис. 1.1. Подключение терминалов к центральному компьютеру

Затем были созданы микропроцессоры и первые микрокомпьютеры. Появилась возможность разместить компьютер на столе у каждого пользователя, так как вычислительные, интеллектуальные ресурсы подешевели. Но зато все остальные ресурсы оставались еще довольно дорогими. А что значит голый интеллект без средств хранения информации и ее документирования? Не будешь же каждый раз после включения питания

заново набирать выполняемую программу или хранить ее в маловместительной постоянной памяти. На помощь снова пришли средства связи. Объединив несколько микрокомпьютеров, можно было организовать совместное использование ими компьютерной периферии (магнитных дисков, магнитной ленты, принтеров). При этом вся обработка информации проводилась на месте, но ее результаты передавались на централизованные ресурсы. Здесь опять же совместно использовалось самое дорогое, что есть в системе, но уже совершенно по-новому. Такой режим получил название **режима обратного разделения времени** (рис. 1.2). Как и в первом случае, средства связи снижали стоимость компьютерной системы в целом.

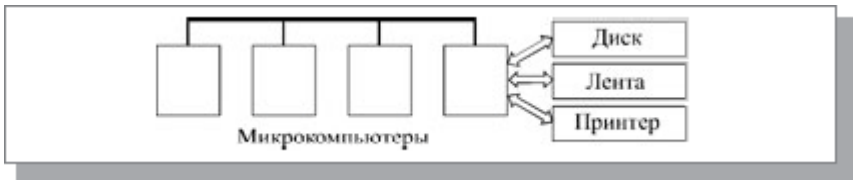

Рис. 1.2. Объединение в сеть первых микрокомпьютеров

Затем появились персональные компьютеры, которые отличались от первых микрокомпьютеров тем, что имели полный комплект достаточно развитой для полностью автономной работы периферии: магнитные диски, принтеры, не говоря уже о более совершенных средствах интерфейса пользователя (мониторы, клавиатуры, мыши и т.д.). Периферия подешевела и стала по цене вполне сравнимой с компьютером. Казалось бы, зачем теперь соединять персональные компьютеры (рис. 1.3)? Что им разделять, когда и так уже все разделено и находится на столе у каждого пользователя? Интеллекта на месте хватает, периферии тоже. Что же может дать сеть в этом случае?

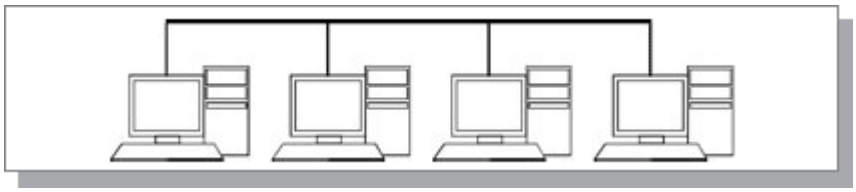

Рис. 1.3. Объединение в сеть персональных компьютеров

Самое главное – это опять же совместное использование ресурса. То самое обратное разделение времени, но уже на принципиально другом уровне. Здесь уже оно применяется не для снижения стоимости системы, а с целью более эффективного использования ресурсов имеющихся в распоряжении компьютеров. Например, сеть позволяет объединить объем дисков всех компьютеров, обеспечив доступ каждого из них к дискам всех остальных как к собственным.

Но нагляднее всего преимущества сети проявляются, в том случае, когда все пользователи активно работают с единой базой данных, запрашивая информацию из нее и занося в нее новую (например, в банке, в магазине, на складе). Никакими дискетами тут уже не обойдешься: пришлось бы целыми

днями переносить данные с каждого компьютера на все остальные, содержать целый штат курьеров. А с сетью все очень просто: любые изменения данных, произведенные с любого компьютера, тут же становятся видными и доступными всем. В этом случае особой обработки на месте обычно не требуется, и в принципе можно было бы обойтись более дешевыми терминалами (вернуться к первой рассмотренной ситуации), но персональные компьютеры имеют несравнимо более удобный интерфейс пользователя, облегчающий работу персонала. К тому же возможность сложной обработки информации на месте часто может заметно уменьшить объем передаваемых данных.

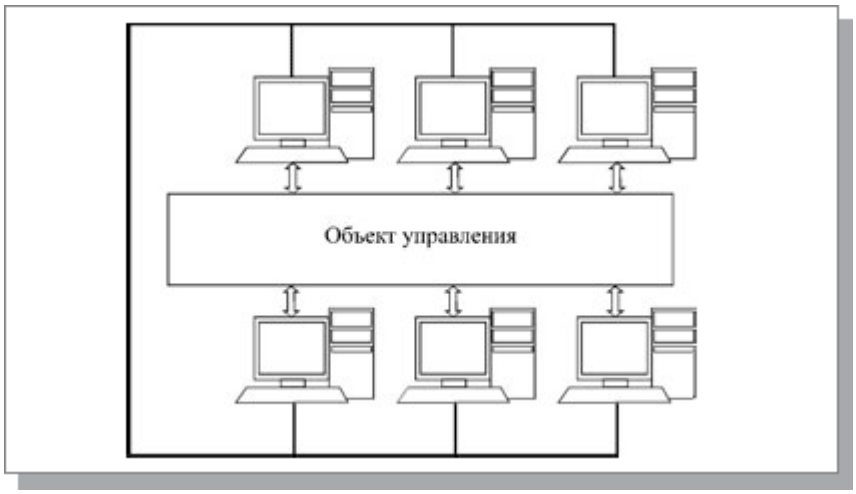

Рис. 1.4. Использование локальной сети для организации совместной работы компьютеров

Без сети также невозможно обойтись в том случае, когда необходимо обеспечить согласованную работу нескольких компьютеров. Эта ситуация чаще всего встречается, когда эти компьютеры используются не для вычислений и работы с базами данных, а в задачах управления, измерения, контроля, там, где компьютер сопрягается с теми или иными внешними устройствами (рис. 1.4). Примерами могут служить различные производственные технологические системы, а также системы управления научными установками и комплексами. Здесь сеть позволяет синхронизовать действия компьютеров, распараллелить и соответственно ускорить процесс обработки данных, то есть сложить уже не только периферийные ресурсы, но и интеллектуальную мощь.

Именно указанные преимущества локальных сетей и обеспечивают их популярность и все более широкое применение, несмотря на все неудобства, связанные с их установкой и эксплуатацией.

1.2. Определение локальной сети

Способов и средств обмена информацией за последнее время предложено множество: от простейшего переноса файлов с помощью дискеты до

всемирной компьютерной сети Интернет, способной объединить все компьютеры мира. Какое же место в этой иерархии отводится локальным сетям?

Чаще всего термин «локальные сети» или «локальные вычислительные сети» (LAN, Local Area Network) понимают буквально, то есть это такие сети, которые имеют небольшие, локальные размеры, соединяют близко расположенные компьютеры. Однако достаточно посмотреть на характеристики некоторых современных локальных сетей, чтобы понять, что такое определение не точно. Например, некоторые локальные сети легко обеспечивают связь на расстоянии нескольких десятков километров. Это уже размеры не комнаты, не здания, не близко расположенных зданий, а, может быть, даже целого города. С другой стороны, по глобальной сети (WAN, Wide Area Network или GAN, Global Area Network) вполне могут связываться компьютеры, находящиеся на соседних столах в одной комнате, но ее почему-то никто не называет локальной сетью. Близко расположенные компьютеры могут также связываться с помощью кабеля, соединяющего разъемы внешних интерфейсов (RS232-C, Centronics) или даже без кабеля по инфракрасному каналу (IrDA). Но такая связь тоже почему-то не называется локальной.

Неверно и довольно часто встречающееся определение локальной сети как малой сети, которая объединяет небольшое количество компьютеров. Действительно, как правило, локальная сеть связывает от двух до нескольких десятков компьютеров. Но предельные возможности современных локальных сетей гораздо выше: максимальное число абонентов может достигать тысячи. Называть такую сеть малой неправильно.

Некоторые авторы определяют локальную сеть как «систему для непосредственного соединения многих компьютеров». При этом подразумевается, что информация передается от компьютера к компьютеру без каких-либо посредников и по единой среде передачи. Однако говорить о единой среде передачи в современной локальной сети не приходится. Например, в пределах одной сети могут использоваться как электрические кабели различных типов (витая пара, коаксиальный кабель), так и оптоволоконные кабели. Определение передачи «без посредников» также не корректно, ведь в современных локальных сетях используются репитеры, трансиверы, концентраторы, коммутаторы, маршрутизаторы, мосты, которые порой производят довольно сложную обработку передаваемой информации. Не совсем понятно, можно ли считать их посредниками или нет, можно ли считать подобную сеть локальной.

Наверное, наиболее точно было бы определить как локальную такую сеть, которая позволяет пользователям не замечать связи. Еще можно сказать, что локальная сеть должна обеспечивать **прозрачную** связь. По сути,

компьютеры, связанные локальной сетью, объединяются, в один виртуальный компьютер, ресурсы которого могут быть доступны всем пользователям, причем этот доступ не менее удобен, чем к ресурсам, входящим непосредственно в каждый отдельный компьютер. Под удобством в данном случае понимается высокая реальная скорость доступа, скорость обмена информацией между приложениями, практически не заметная для пользователя. При таком определении становится понятно, что ни медленные глобальные сети, ни медленная связь через последовательный или параллельный порты не подпадают под понятие локальной сети.

Из данного определения следует, что скорость передачи по локальной сети обязательно должна расти по мере роста быстродействия наиболее распространенных компьютеров. Именно это и наблюдается: если еще десять лет назад вполне приемлемой считалась скорость обмена в 10 Мбит/с, то сейчас уже среднескоростной считается сеть, имеющая пропускную способность 100 Мбит/с, активно разрабатываются, а кое-где используются средства для скорости 1000 Мбит/с и даже больше. Без этого уже нельзя, иначе связь станет слишком узким местом, будет чрезмерно замедлять работу объединенного сетью виртуального компьютера, снижать удобство доступа к сетевым ресурсам.

Таким образом, главное отличие локальной сети от любой другой — высокая скорость передачи информации по сети. Но это еще не все, не менее важны и другие факторы.

В частности, принципиально необходим низкий уровень ошибок передачи, вызванных как внутренними, так и внешними факторами. Ведь даже очень быстро переданная информация, которая искажена ошибками, просто не имеет смысла, ее придется передавать еще раз. Поэтому локальные сети обязательно используют специально прокладываемые высококачественные и хорошо защищенные от помех линии связи.

Особое значение имеет и такая характеристика сети, как возможность работы с большими нагрузками, то есть с высокой интенсивностью обмена (или, как еще говорят, с большим трафиком). Ведь если механизм управления обменом, используемый в сети, не слишком эффективен, то компьютеры могут подолгу ждать своей очереди на передачу. И даже если эта передача будет производиться затем на высочайшей скорости и безошибочно, для пользователя сети такая задержка доступа ко всем сетевым ресурсам неприемлема. Ему ведь не важно, почему приходится ждать.

Механизм управления обменом может гарантированно успешно работать только в том случае, когда заранее известно, сколько компьютеров (или, как еще говорят, абонентов, узлов), допустимо подключить к сети. Иначе всегда можно включить столько абонентов, что вследствие перегрузки забуксует

любой механизм управления. Наконец, сетью можно назвать только такую систему передачи данных, которая позволяет объединять до нескольких десятков компьютеров, но никак не два, как в случае связи через стандартные порты.

Таким образом, сформулировать отличительные признаки локальной сети можно следующим образом:

- Высокая скорость передачи информации, большая пропускная способность сети. Приемлемая скорость сейчас — не менее 10 Мбит/с.
- Низкий уровень ошибок передачи (или, что то же самое, высококачественные каналы связи). Допустимая вероятность ошибок передачи данных должна быть порядка 10^{-8} — 10^{-12}.
- Эффективный, быстродействующий механизм управления обменом по сети.
- Заранее четко ограниченное количество компьютеров, подключаемых к сети.

При таком определении понятно, что глобальные сети отличаются от локальных прежде всего тем, что они рассчитаны на неограниченное число абонентов. Кроме того, они используют (или могут использовать) не слишком качественные каналы связи и сравнительно низкую скорость передачи. А механизм управления обменом в них не может быть гарантированно быстрым. В глобальных сетях гораздо важнее не качество связи, а сам факт ее существования.

Нередко выделяют еще один класс компьютерных сетей – городские, региональные сети (MAN, Metropolitan Area Network), которые обычно по своим характеристикам ближе к глобальным сетям, хотя иногда все-таки имеют некоторые черты локальных сетей, например, высококачественные каналы связи и сравнительно высокие скорости передачи. В принципе городская сеть может быть локальной со всеми ее преимуществами.

Правда, сейчас уже нельзя провести четкую границу между локальными и глобальными сетями. Большинство локальных сетей имеет выход в глобальную. Но характер передаваемой информации, принципы организации обмена, режимы доступа к ресурсам внутри локальной сети, как правило, сильно отличаются от тех, что приняты в глобальной сети. И хотя все компьютеры локальной сети в данном случае включены также и в глобальную сеть, специфики локальной сети это не отменяет. Возможность выхода в глобальную сеть остается всего лишь одним из ресурсов, разделяемых пользователями локальной сети.

По локальной сети может передаваться самая разная цифровая информация: данные, изображения, телефонные разговоры, электронные письма и т.д. Кстати, именно задача передачи изображений, особенно полноцветных динамических, предъявляет самые высокие требования к быстродействию сети. Чаще всего локальные сети используются для разделения (совместного использования) таких ресурсов, как дисковое пространство, принтеры и

выход в глобальную сеть, но это всего лишь незначительная часть тех возможностей, которые предоставляют средства локальных сетей. Например, они позволяют осуществлять обмен информацией между компьютерами разных типов. Полноценными абонентами (узлами) сети могут быть не только компьютеры, но и другие устройства, например, принтеры, плоттеры, сканеры. Локальные сети дают также возможность организовать систему параллельных вычислений на всех компьютерах сети, что многократно ускоряет решение сложных математических задач. С их помощью, как уже упоминалось, можно управлять работой технологической системы или исследовательской установки с нескольких компьютеров одновременно.

Однако сети имеют и довольно существенные недостатки, о которых всегда следует помнить:

- Сеть требует дополнительных, иногда значительных материальных затрат на покупку сетевого оборудования, программного обеспечения, на прокладку соединительных кабелей и обучение персонала.
- Сеть требует приема на работу специалиста (администратора сети), который будет заниматься контролем работы сети, ее модернизацией, управлением доступом к ресурсам, устранением возможных неисправностей, защитой информации и резервным копированием. Для больших сетей может понадобиться целая бригада администраторов.
- Сеть ограничивает возможности перемещения компьютеров, подключенных к ней, так как при этом может понадобиться перекладка соединительных кабелей.
- Сети представляют собой прекрасную среду для распространения компьютерных вирусов, поэтому вопросам защиты от них придется уделять гораздо больше внимания, чем в случае автономного использования компьютеров. Ведь достаточно инфицировать один и все компьютеры сети будут поражены.
- Сеть резко повышает опасность несанкционированного доступа к информации с целью ее кражи или уничтожения, Информационная защита требует проведения целого комплекса технических и организационных мероприятий.

Ничто не дается даром. И надо хорошо подумать, стоит ли подключать к сети все компьютеры компании, или часть из них лучше оставить автономными. Возможно, что сеть вообще не нужна, так как породит гораздо больше проблем, чем позволит решить.

Здесь же следует упомянуть о таких важнейших понятиях теории сетей, как абонент, сервер, клиент.

Абонент (узел, хост, станция) — это устройство, подключенное к сети и активно участвующее в информационном обмене. Чаще всего абонентом (узлом) сети является компьютер, но абонентом также может быть, например, сетевой принтер или другое периферийное устройство, имеющее возможность напрямую подключаться к сети. Далее в тексте книги вместо термина «абонент» для простоты будет использоваться термин «компьютер».

Сервером называется абонент (узел) сети, который предоставляет свои ресурсы другим абонентам, но сам не использует их ресурсы. Таким образом, он обслуживает сеть. Серверов в сети может быть несколько, и совсем не

обязательно, что сервер – самый мощный компьютер. **Выделенный** (dedicated) сервер — это сервер, занимающийся только сетевыми задачами. **Невыделенный** сервер может помимо обслуживания сети выполнять и другие задачи. Специфический тип сервера — это сетевой принтер.

Клиентом называется абонент сети, который только использует сетевые ресурсы, но сам свои ресурсы в сеть не отдает, то есть сеть его обслуживает, а он ей только пользуется. Компьютер-клиент также часто называют **рабочей станцией**. В принципе каждый компьютер может быть одновременно как клиентом, так и сервером.

Под сервером и клиентом часто понимают также не сами компьютеры, а работающие на них программные приложения. В этом случае то приложение, которое только отдает ресурс в сеть, является сервером, а то приложение, которое только пользуется сетевыми ресурсами – клиентом.

2. Топология локальных сетей

Под топологией (компоновкой, конфигурацией, структурой) компьютерной сети обычно понимается физическое расположение компьютеров сети друг относительно друга и способ соединения их линиями связи. Важно отметить, что понятие топологии относится, прежде всего, к локальным сетям, в которых структуру связей можно легко проследить. В глобальных сетях структура связей обычно скрыта от пользователей и не слишком важна, так как каждый сеанс связи может производиться по собственному пути.

Топология определяет требования к оборудованию, тип используемого кабеля, допустимые и наиболее удобные методы управления обменом, надежность работы, возможности расширения сети. И хотя выбирать топологию пользователю сети приходится нечасто, знать об особенностях основных топологий, их достоинствах и недостатках надо.

Существует три, базовые топологии сети:

- **Шина** (bus) — все компьютеры параллельно подключаются к одной линии связи. Информация от каждого компьютера одновременно передается всем остальным компьютерам (рис. 1.5).

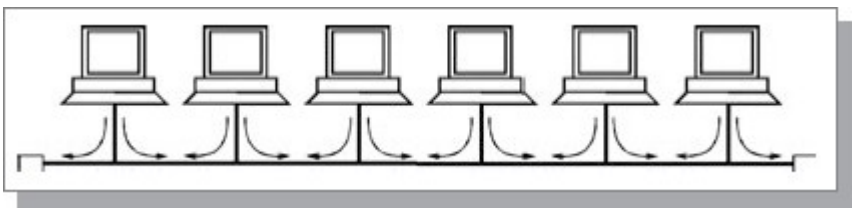

Рис. 1.5. Сетевая топология шина

- **Звезда** (star) – к одному центральному компьютеру присоединяются остальные периферийные компьютеры, причем каждый из них использует отдельную линию связи (рис. 1.6). Информация от периферийного компьютера передается только центральному компьютеру, от центрального – одному или нескольким периферийным.

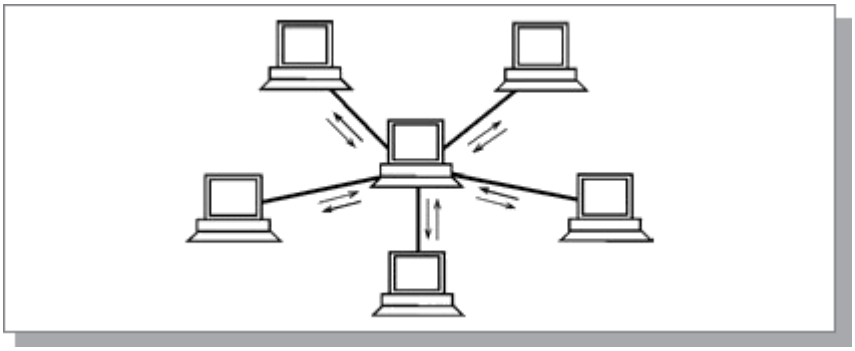

Рис. 1.6. Сетевая топология звезда

- **Кольцо** (ring) — компьютеры последовательно объединены в кольцо. Передача информации в кольце всегда производится только в одном направлении. Каждый из компьютеров передает информацию только одному компьютеру, следующему в цепочке за ним, а получает информацию только от предыдущего в цепочке компьютера (рис. 1.7).

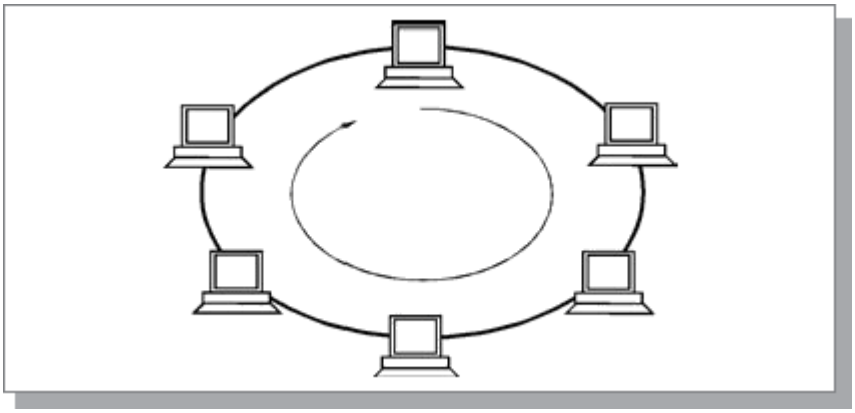

Рис. 1.7. Сетевая топология кольцо

На практике нередко используют и другие топологии локальных сетей, однако большинство сетей ориентировано именно на три базовые топологии.

Прежде чем перейти к анализу особенностей базовых сетевых топологий, необходимо выделить некоторые важнейшие факторы, влияющие на физическую работоспособность сети и непосредственно связанные с понятием топология.

- Исправность компьютеров (абонентов), подключенных к сети. В некоторых случаях поломка абонента может заблокировать работу всей сети. Иногда неисправность абонента не влияет на работу сети в целом, не мешает остальным абонентам обмениваться информацией.

- Исправность сетевого оборудования, то есть технических средств, непосредственно подключенных к сети (адаптеры, трансиверы, разъемы и т.д.). Выход из строя сетевого оборудования одного из абонентов может сказаться на всей сети, но может нарушить обмен только с одним абонентом.
- Целостность кабеля сети. При обрыве кабеля сети (например, из-за механических воздействий) может нарушиться обмен информацией во всей сети или в одной из ее частей. Для электрических кабелей столь же критично короткое замыкание в кабеле.
- Ограничение длины кабеля, связанное с затуханием распространяющегося по нему сигнала. Как известно, в любой среде при распространении сигнал ослабляется (затухает). И чем большее расстояние проходит сигнал, тем больше он затухает (рис. 1.8). Необходимо следить, чтобы длина кабеля сети не была больше предельной длины $L_{пр}$, при превышении которой затухание становится уже неприемлемым (принимающий абонент не распознает ослабевший сигнал).

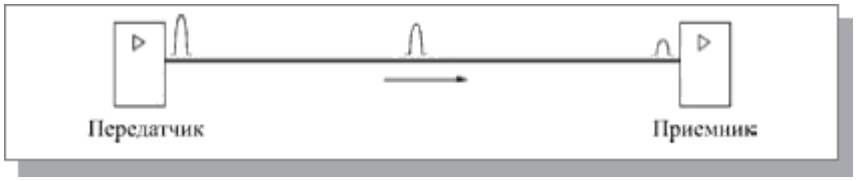

Рис. 1.8. Затухание сигнала при распространении по сети

2.1. Топология шина

Топология шина (или, как ее еще называют, общая шина) самой своей структурой предполагает идентичность сетевого оборудования компьютеров, а также равноправие всех абонентов по доступу к сети. Компьютеры в шине могут передавать только по очереди, так как линия связи в данном случае единственная. Если несколько компьютеров будут передавать информацию одновременно, она исказится в результате наложения (**конфликта**, **коллизии**). В шине всегда реализуется режим так называемого **полудуплексного** (half duplex) обмена (в обоих направлениях, но по очереди, а не одновременно).

В топологии шина отсутствует явно выраженный центральный абонент, через которого передается вся информация, это увеличивает ее надежность (ведь при отказе центра перестает функционировать вся управляемая им система). Добавление новых абонентов в шину довольно просто и обычно возможно даже во время работы сети. В большинстве случаев при использовании шины требуется минимальное количество соединительного кабеля по сравнению с другими топологиями.

Поскольку центральный абонент отсутствует, разрешение возможных конфликтов в данном случае ложится на сетевое оборудование каждого отдельного абонента. В связи с этим сетевая аппаратура при топологии шина сложнее, чем при других топологиях. Тем не менее из-за широкого распространения сетей с топологией шина (прежде всего наиболее популярной сети Ethernet) стоимость сетевого оборудования не слишком высока.

Рис. 1.9. Обрыв кабеля в сети с топологией шина

Важное преимущество шины состоит в том, что при отказе любого из компьютеров сети, исправные машины смогут нормально продолжать обмен.

Казалось бы, при обрыве кабеля получаются две вполне работоспособные шины (рис. 1.9). Однако надо учитывать, что из-за особенностей распространения электрических сигналов по длинным линиям связи необходимо предусматривать включение на концах шины специальных согласующих устройств, **терминаторов**, показанных на рис. 1.5 и 1.9 в виде прямоугольников. Без включения терминаторов сигнал отражается от конца линии и искажается так, что связь по сети становится невозможной. В случае разрыва или повреждения кабеля нарушается согласование линии связи, и прекращается обмен даже между теми компьютерами, которые остались соединенными между собой. Подробнее о согласовании будет изложено в специальном разделе книги. Короткое замыкание в любой точке кабеля шины выводит из строя всю сеть.

Отказ сетевого оборудования любого абонента в шине может вывести из строя всю сеть. К тому же такой отказ довольно трудно локализовать, поскольку все абоненты включены параллельно, и понять, какой из них вышел из строя, невозможно.

При прохождении по линии связи сети с топологией шина информационные сигналы ослабляются и никак не восстанавливаются, что накладывает жесткие ограничения на суммарную длину линий связи. Причем каждый абонент может получать из сети сигналы разного уровня в зависимости от расстояния до передающего абонента. Это предъявляет дополнительные требования к приемным узлам сетевого оборудования.

Если принять, что сигнал в кабеле сети ослабляется до предельно допустимого уровня на длине $L_{пр}$, то длина полная шины не может превышать величины $L_{пр}$. В этом смысле шина обеспечивает наименьшую длину по сравнению с другими базовыми топологиями.

Для увеличения длины сети с топологией шина часто используют несколько **сегментов** (частей сети, каждый из которых представляет собой шину), соединенных между собой с помощью специальных усилителей и восстановителей сигналов — **репитеров** или **повторителей** (на рис. 1.10 показано соединение двух сегментов, предельная длина сети в этом случае возрастает до 2 $L_{пр}$, так как каждый из сегментов может быть длиной $L_{пр}$). Однако такое наращивание длины сети не может продолжаться бесконечно.

Ограничения на длину связаны с конечной скоростью распространения сигналов по линиям связи.

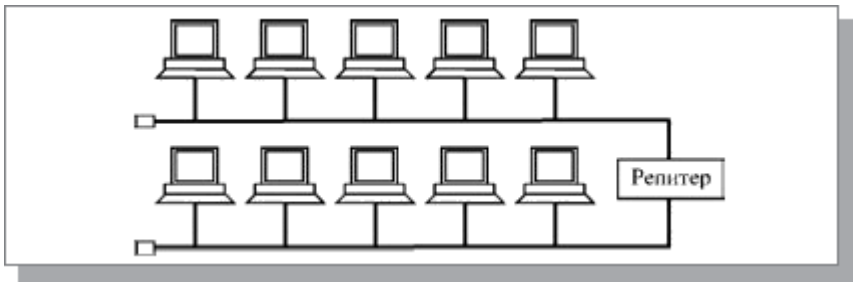

Рис. 1.10. Соединение сегментов сети типа шина с помощью репитера

2.2. Топология звезда

Звезда — это единственная топология сети с явно выделенным центром, к которому подключаются все остальные абоненты. Обмен информацией идет исключительно через центральный компьютер, на который ложится большая нагрузка, поэтому ничем другим, кроме сети, он, как правило, заниматься не может. Понятно, что сетевое оборудование центрального абонента должно быть существенно более сложным, чем оборудование периферийных абонентов. О равноправии всех абонентов (как в шине) в данном случае говорить не приходится. Обычно центральный компьютер самый мощный, именно на него возлагаются все функции по управлению обменом. Никакие конфликты в сети с топологией звезда в принципе невозможны, так как управление полностью централизовано.

Если говорить об устойчивости звезды к отказам компьютеров, то выход из строя периферийного компьютера или его сетевого оборудования никак не отражается на функционировании оставшейся части сети, зато любой отказ центрального компьютера делает сеть полностью неработоспособной. В связи с этим должны приниматься специальные меры по повышению надежности центрального компьютера и его сетевой аппаратуры.

Обрыв кабеля или короткое замыкание в нем при топологии звезда нарушает обмен только с одним компьютером, а все остальные компьютеры могут нормально продолжать работу.

В отличие от шины, в звезде на каждой линии связи находятся только два абонента: центральный и один из периферийных. Чаще всего для их соединения используется две линии связи, каждая из которых передает информацию в одном направлении, то есть на каждой линии связи имеется только один приемник и один передатчик. Это так называемая передача **точка-точка**. Все это существенно упрощает сетевое оборудование по сравнению с шиной и избавляет от необходимости применения дополнительных, внешних терминаторов.

Проблема затухания сигналов в линии связи также решается в звезде проще, чем в случае шины, ведь каждый приемник всегда получает сигнал одного уровня. Предельная длина сети с топологией звезда может быть вдвое больше, чем в шине (то есть 2 $L_{пр}$), так как каждый из кабелей, соединяющий центр с периферийным абонентом, может иметь длину $L_{пр}$.

Серьезный недостаток топологии звезда состоит в жестком ограничении количества абонентов. Обычно центральный абонент может обслуживать не более 8—16 периферийных абонентов. В этих пределах подключение новых абонентов довольно просто, но за ними оно просто невозможно. В звезде допустимо подключение вместо периферийного еще одного центрального абонента (в результате получается топология из нескольких соединенных между собой звезд).

Звезда, показанная на рис. 1.6, носит название активной или истинной звезды. Существует также топология, называемая пассивной звездой, которая только внешне похожа на звезду (рис. 1.11). В настоящее время она распространена гораздо более широко, чем активная звезда. Достаточно сказать, что она используется в наиболее популярной сегодня сети Ethernet.

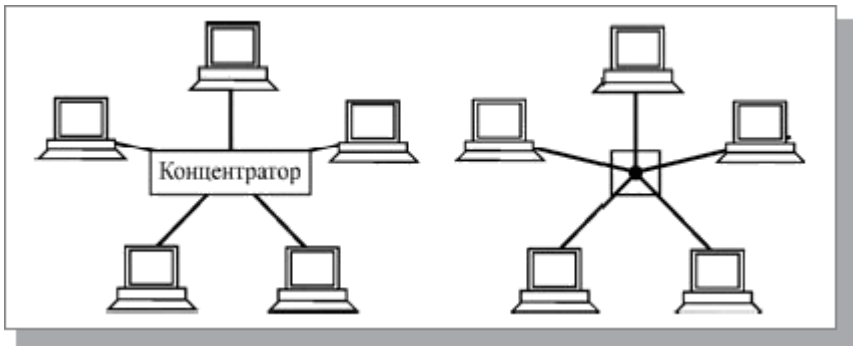

Рис. 1.11. Топология пассивная звезда и ее эквивалентная схема

В центре сети с данной топологией помещается не компьютер, а специальное устройство — концентратор или, как его еще называют, хаб (hub), которое выполняет ту же функцию, что и репитер, то есть восстанавливает приходящие сигналы и пересылает их во все другие линии связи.

Получается, что хотя схема прокладки кабелей подобна истинной или активной звезде, фактически речь идет о шинной топологии, так как информация от каждого компьютера одновременно передается ко всем остальным компьютерам, а никакого центрального абонента не существует. Безусловно, пассивная звезда дороже обычной шины, так как в этом случае требуется еще и концентратор. Однако она предоставляет целый ряд дополнительных возможностей, связанных с преимуществами звезды, в частности, упрощает обслуживание и ремонт сети. Именно поэтому в последнее время пассивная звезда все больше вытесняет истинную шину, которая считается малоперспективной топологией.

Можно выделить также промежуточный тип топологии между активной и пассивной звездой. В этом случае концентратор не только ретранслирует поступающие на него сигналы, но и производит управление обменом, однако сам в обмене не участвует (так сделано в сети 100VG-AnyLAN).

Большое достоинство звезды (как активной, так и пассивной) состоит в том, что все точки подключения собраны в одном месте. Это позволяет легко контролировать работу сети, локализовать неисправности путем простого отключения от центра тех или иных абонентов (что невозможно, например, в случае шинной топологии), а также ограничивать доступ посторонних лиц к жизненно важным для сети точкам подключения. К периферийному абоненту в случае звезды может подходить как один кабель (по которому идет передача в обоих направлениях), так и два (каждый кабель передает в одном из двух встречных направлений), причем последнее встречается гораздо чаще.

Общим недостатком для всех топологий типа звезда (как активной, так и пассивной) является значительно больший, чем при других топологиях, расход кабеля. Например, если компьютеры расположены в одну линию (как на рис. 1.5), то при выборе топологии звезда понадобится в несколько раз больше кабеля, чем при топологии шина. Это существенно влияет на стоимость сети в целом и заметно усложняет прокладку кабеля.

2.3. Топология кольцо

Кольцо — это топология, в которой каждый компьютер соединен линиями связи с двумя другими: от одного он получает информацию, а другому передает. На каждой линии связи, как и в случае звезды, работает только один передатчик и один приемник (связь типа точка-точка). Это позволяет отказаться от применения внешних терминаторов.

Важная особенность кольца состоит в том, что каждый компьютер ретранслирует (восстанавливает, усиливает) приходящий к нему сигнал, то есть выступает в роли репитера. Затухание сигнала во всем кольце не имеет никакого значения, важно только затухание между соседними компьютерами кольца. Если предельная длина кабеля, ограниченная затуханием, составляет $L_{пр}$, то суммарная длина кольца может достигать $NL_{пр}$, где N — количество компьютеров в кольце. Полный размер сети в пределе будет $NL_{пр}/2$, так как кольцо придется сложить вдвое. На практике размеры кольцевых сетей достигают десятков километров (например, в сети FDDI). Кольцо в этом отношении существенно превосходит любые другие топологии.

Четко выделенного центра при кольцевой топологии нет, все компьютеры могут быть одинаковыми и равноправными. Однако довольно часто в кольце выделяется специальный абонент, который управляет обменом или контролирует его. Понятно, что наличие такого единственного

управляющего абонента снижает надежность сети, так как выход его из строя сразу же парализует весь обмен.

Строго говоря, компьютеры в кольце не являются полностью равноправными (в отличие, например, от шинной топологии). Ведь одни из них обязательно получают информацию от компьютера, ведущего передачу в данный момент, раньше, а другие – позже. Именно на этой особенности топологии и строятся методы управления обменом по сети, специально рассчитанные на кольцо. В таких методах право на следующую передачу (или, как еще говорят, на захват сети) переходит последовательно к следующему по кругу компьютеру. Подключение новых абонентов в кольцо выполняется достаточно просто, хотя и требует обязательной остановки работы всей сети на время подключения. Как и в случае шины, максимальное количество абонентов в кольце может быть довольно велико (до тысячи и больше). Кольцевая топология обычно обладает высокой устойчивостью к перегрузкам, обеспечивает уверенную работу с большими потоками передаваемой по сети информации, так как в ней, как правило, нет конфликтов (в отличие от шины), а также отсутствует центральный абонент (в отличие от звезды), который может быть перегружен большими потоками информации.

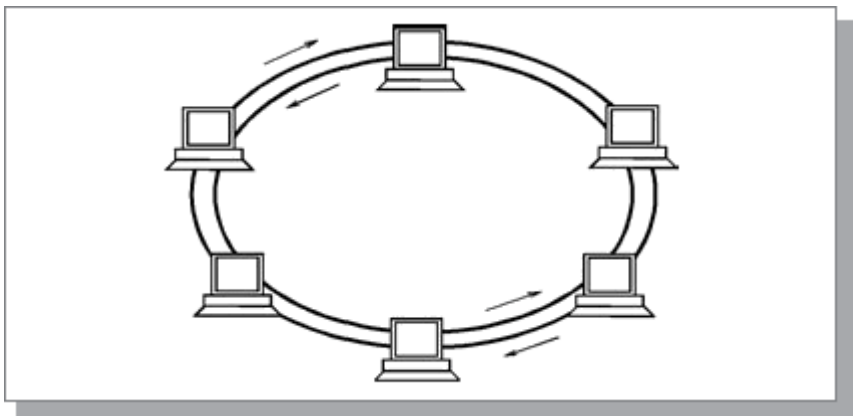

Рис. 1.12. Сеть с двумя кольцами

Сигнал в кольце проходит последовательно через все компьютеры сети, поэтому выход из строя хотя бы одного из них (или же его сетевого оборудования) нарушает работу сети в целом. Это существенный недостаток кольца.

Точно так же обрыв или короткое замыкание в любом из кабелей кольца делает работу всей сети невозможной. Из трех рассмотренных топологий кольцо наиболее уязвимо к повреждениям кабеля, поэтому в случае топологии кольца обычно предусматривают прокладку двух (или более) параллельных линий связи, одна из которых находится в резерве.

Иногда сеть с топологией кольцо выполняется на основе двух параллельных кольцевых линий связи, передающих информацию в противоположных направлениях (рис. 1.12). Цель подобного решения – увеличение (в идеале –

вдвое) скорости передачи информации по сети. К тому же при повреждении одного из кабелей сеть может работать с другим кабелем (правда, предельная скорость уменьшится).

2.4. Другие топологии

Кроме трех рассмотренных базовых топологий нередко применяется также сетевая топология дерево (tree), которую можно рассматривать как комбинацию нескольких звезд. Причем, как и в случае звезды, дерево может быть активным или истинным (рис. 1.13) и пассивным (рис. 1.14). При активном дереве в центрах объединения нескольких линий связи находятся центральные компьютеры, а при пассивном — концентраторы (хабы).

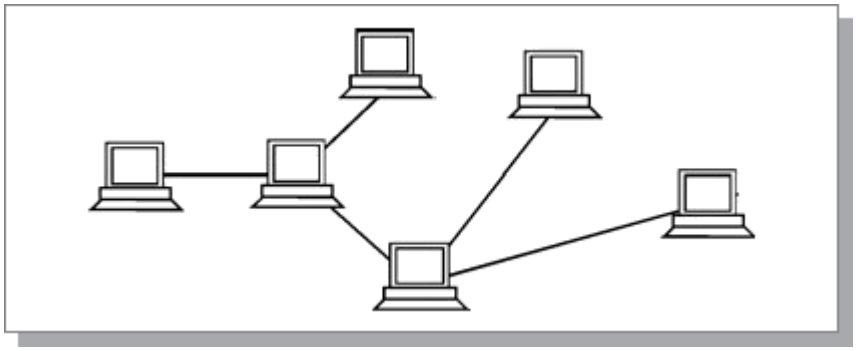

Рис. 1.13. Топология активное дерево

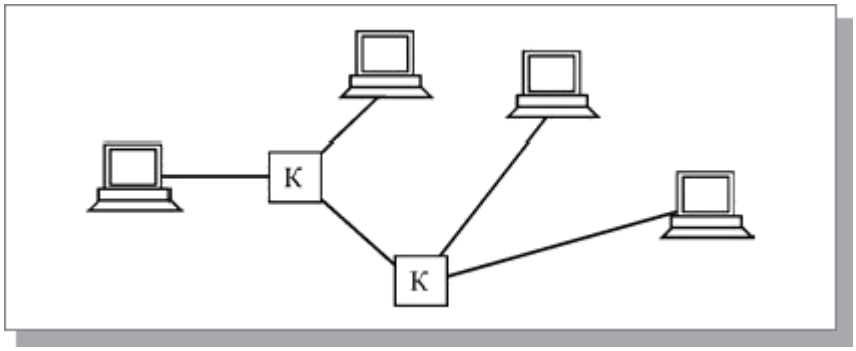

Рис. 1.14. Топология пассивное дерево. К — концентраторы

Довольно часто применяются комбинированные топологии, среди которых наиболее распространены звездно-шинная (рис. 1.15) и звездно-кольцевая (рис. 1.16).

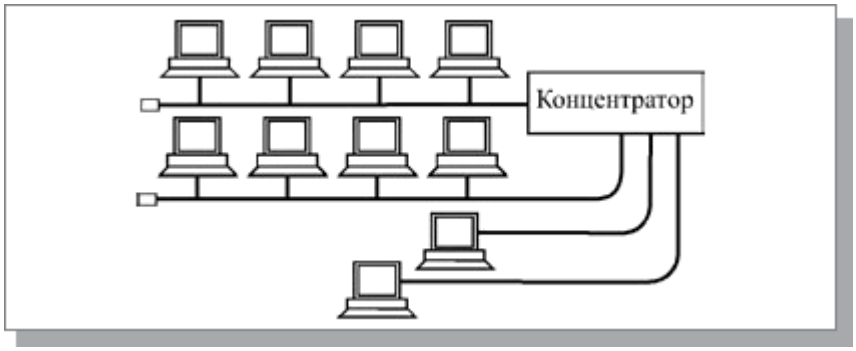

Рис. 1.15. Пример звездно-шинной топологии

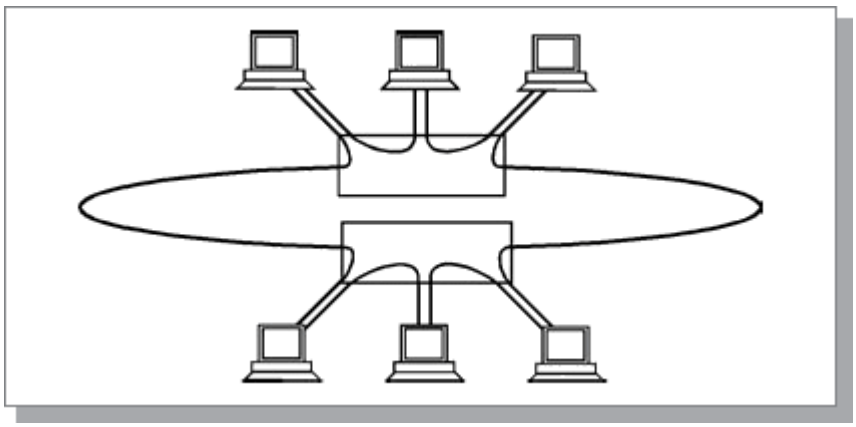

Рис. 1.16. Пример звездно-кольцевой топологии

В звездно-шинной (star-bus) топологии используется комбинация шины и пассивной звезды. К концентратору подключаются как отдельные компьютеры, так и целые шинные сегменты. На самом деле реализуется физическая топология шина, включающая все компьютеры сети. В данной топологии может использоваться и несколько концентраторов, соединенных между собой и образующих так называемую магистральную, опорную шину. К каждому из концентраторов при этом подключаются отдельные компьютеры или шинные сегменты. В результате получается звездно-шинное дерево. Таким образом, пользователь может гибко комбинировать преимущества шинной и звездной топологий, а также легко изменять количество компьютеров, подключенных к сети. С точки зрения распространения информации данная топология равноценна классической шине.

В случае звездно-кольцевой (star-ring) топологии в кольцо объединяются не сами компьютеры, а специальные концентраторы (изображенные на рис. 1.16 в виде прямоугольников), к которым в свою очередь подключаются компьютеры с помощью звездообразных двойных линий связи. В действительности все компьютеры сети включаются в замкнутое кольцо, так как внутри концентраторов линии связи образуют замкнутый контур (как показано на рис. 1.16). Данная топология дает возможность комбинировать преимущества звездной и кольцевой топологий. Например, концентраторы позволяют собрать в одно место все точки подключения кабелей сети. Если говорить о распространении информации, данная топология равноценна классическому кольцу.

В заключение надо также сказать о сеточной топологии (mesh), при которой компьютеры связываются между собой не одной, а многими линиями связи, образующими сетку (рис. 1.17).

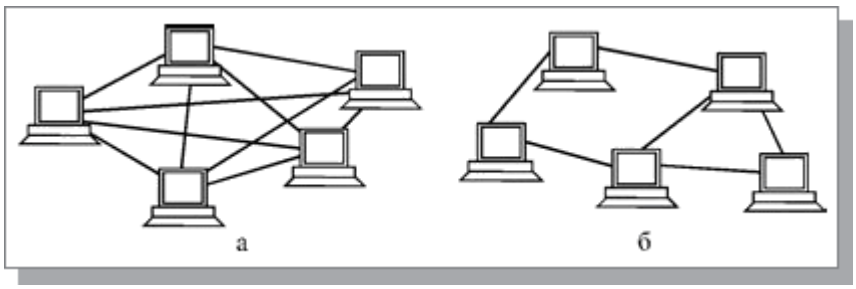

Рис. 1.17. Сеточная топология: полная (а) и частичная (б)

В полной сеточной топологии каждый компьютер напрямую связан со всеми остальными компьютерами. В этом случае при увеличении числа компьютеров резко возрастает количество линий связи. Кроме того, любое изменение в конфигурации сети требует внесения изменений в сетевую аппаратуру всех компьютеров, поэтому полная сеточная топология не получила широкого распространения.

Частичная сеточная топология предполагает прямые связи только для самых активных компьютеров, передающих максимальные объемы информации. Остальные компьютеры соединяются через промежуточные узлы. Сеточная топология позволяет выбирать маршрут для доставки информации от абонента к абоненту, обходя неисправные участки. С одной стороны, это увеличивает надежность сети, с другой же – требует существенного усложнения сетевой аппаратуры, которая должна выбирать маршрут.

2.5. Многозначность понятия топологии

Топология сети указывает не только на физическое расположение компьютеров, как часто считают, но, что гораздо важнее, на характер связей между ними, особенности распространения информации, сигналов по сети. Именно характер связей определяет степень отказоустойчивости сети, требуемую сложность сетевой аппаратуры, наиболее подходящий метод управления обменом, возможные типы сред передачи (каналов связи), допустимый размер сети (длина линий связи и количество абонентов) необходимость электрического согласования и многое другое.

Более того, физическое расположение компьютеров, соединяемых сетью, почти не влияет на выбор топологии. Как бы ни были расположены компьютеры, их можно соединить с помощью любой заранее выбранной топологии (рис. 1.18).

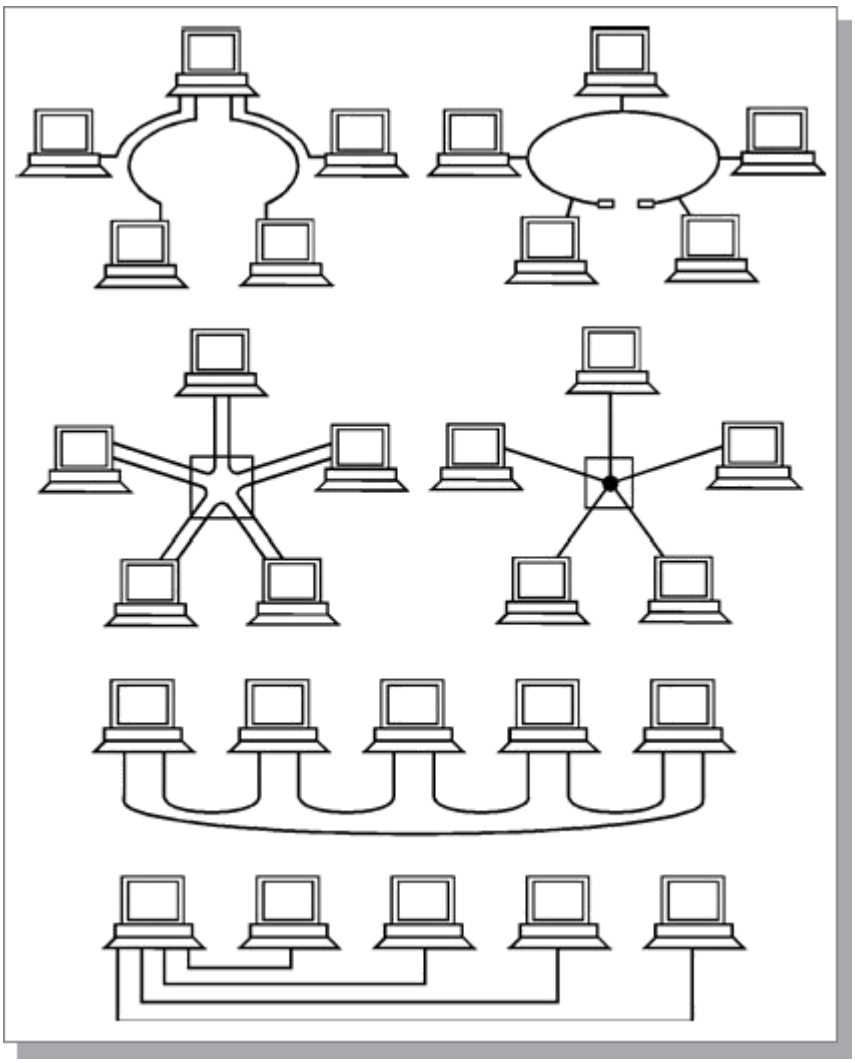

Рис. 1.18. Примеры использования разных топологий

В том случае, если соединяемые компьютеры расположены по контуру круга, они могут соединяться, как звезда или шина. Когда компьютеры расположены вокруг некоего центра, их допустимо соединить с помощью топологий шина или кольцо.

Наконец когда компьютеры расположены в одну линию, они могут соединяться звездой или кольцом. Другое дело, какова будет требуемая длина кабеля.

Строго говоря, в литературе при упоминании о топологии сети, авторы могут подразумевать четыре совершенно разные понятия, относящиеся к различным уровням сетевой архитектуры:

- Физическая топология (географическая схема расположения компьютеров и прокладки кабелей). В этом смысле, например, пассивная звезда ничем не отличается от активной, поэтому ее нередко называют просто звездой.
- Логическая топология (структура связей, характер распространения сигналов по сети). Это наиболее правильное определение топологии.
- Топология управления обменом (принцип и последовательность передачи права на захват сети между отдельными компьютерами).

- Информационная топология (направление потоков информации, передаваемой по сети).

Например, сеть с физической и логической топологией шина может в качестве метода управления использовать эстафетную передачу права захвата сети (быть в этом смысле кольцом) и одновременно передавать всю информацию через выделенный компьютер (быть в этом смысле звездой). Или сеть с логической топологией шина может иметь физическую топологию звезда (пассивная) или дерево (пассивное).

Сеть с любой физической топологией, логической топологией, топологией управления обменом может считаться звездой в смысле информационной топологии, если она построена на основе одного сервера и нескольких клиентов, общающихся только с этим сервером. В данном случае справедливы все рассуждения о низкой отказоустойчивости сети к неполадкам центра (сервера). Точно так же любая сеть может быть названа шиной в информационном смысле, если она построена из компьютеров, являющихся одновременно как серверами, так и клиентами. Такая сеть будет мало чувствительна к отказам отдельных компьютеров.

Заканчивая обзор особенностей топологий локальных сетей, необходимо отметить, что топология все-таки не является основным фактором при выборе типа сети. Гораздо важнее, например, уровень стандартизации сети, скорость обмена, количество абонентов, стоимость оборудования, выбранное программное обеспечение. Но, с другой стороны, некоторые сети позволяют использовать разные топологии на разных уровнях. Этот выбор уже целиком ложится на пользователя, который должен учитывать все перечисленные в данном разделе соображения.

2. Типы линий связи локальных сетей

Средой передачи информации называются те линии связи (или каналы связи), по которым производится обмен информацией между компьютерами. В подавляющем большинстве компьютерных сетей (особенно локальных) используются проводные или кабельные каналы связи, хотя существуют и беспроводные сети, которые сейчас находят все более широкое применение, особенно в портативных компьютерах.

Информация в локальных сетях чаще всего передается в последовательном коде, то есть бит за битом. Такая передача медленнее и сложнее, чем при использовании параллельного кода. Однако надо учитывать то, что при более быстрой параллельной передаче (по нескольким кабелям одновременно) увеличивается количество соединительных кабелей в число раз, равное количеству разрядов параллельного кода (например, в 8 раз при 8-разрядном коде). Это совсем не мелочь, как может показаться на первый взгляд. При значительных расстояниях между абонентами сети стоимость кабеля вполне

сравнима со стоимостью компьютеров и даже может превосходить ее. К тому же проложить один кабель (реже два разнонаправленных) гораздо проще, чем 8, 16 или 32. Значительно дешевле обойдется также поиск повреждений и ремонт кабеля.

Но это еще не все. Передача на большие расстояния при любом типе кабеля требует сложной передающей и приемной аппаратуры, так как при этом необходимо формировать мощный сигнал на передающем конце и детектировать слабый сигнал на приемном конце. При последовательной передаче для этого требуется всего один передатчик и один приемник. При параллельной же количество требуемых передатчиков и приемников возрастает пропорционально разрядности используемого параллельного кода. В связи с этим, даже если разрабатывается сеть незначительной длины (порядка десятка метров) чаще всего выбирают последовательную передачу.

К тому же при параллельной передаче чрезвычайно важно, чтобы длины отдельных кабелей были точно равны друг другу. Иначе в результате прохождения по кабелям разной длины между сигналами на приемном конце образуется временной сдвиг, который может привести к сбоям в работе или даже к полной неработоспособности сети. Например, при скорости передачи 100 Мбит/с и длительности бита 10 нс этот временной сдвиг не должен превышать 5-10 нс. Такую величину сдвига дает разница в длинах кабелей в 1-2 метра. При длине кабеля 1000 метров это составляет 0,1-0,2 %.

Надо отметить, что в некоторых высокоскоростных локальных сетях все-таки используют параллельную передачу по 2-4 кабелям, что позволяет при заданной скорости передачи применять более дешевые кабели с меньшей полосой пропускания. Но допустимая длина кабелей при этом не превышает сотни метров. Примером может служить сегмент 100BASE-T4 сети Fast Ethernet.

Промышленностью выпускается огромное количество типов кабелей, например, только одна крупнейшая кабельная компания Belden предлагает более 2000 их наименований. Но все кабели можно разделить на три большие группы:

- электрические (медные) кабели на основе витых пар проводов (twisted pair), которые делятся на экранированные (shielded twisted pair, STP) и неэкранированные (unshielded twisted pair, UTP);
- электрические (медные) коаксиальные кабели (coaxial cable);
- оптоволоконные кабели (fiber optic).

Каждый тип кабеля имеет свои преимущества и недостатки, так что при выборе надо учитывать как особенности решаемой задачи, так и особенности конкретной сети, в том числе и используемую топологию.

Можно выделить следующие основные параметры кабелей, принципиально важные для использования в локальных сетях:

- Полоса пропускания кабеля (частотный диапазон сигналов, пропускаемых кабелем) и затухание сигнала в кабеле. Два этих параметра тесно связаны между собой, так как с ростом частоты сигнала растет затухание сигнала. Надо выбирать кабель, который на заданной частоте сигнала имеет приемлемое затухание. Или же надо выбирать частоту сигнала, на которой затухание еще приемлемо. Затухание измеряется в децибелах и пропорционально длине кабеля.
- **Помехозащищенность** кабеля и обеспечиваемая им **секретность** передачи информации. Эти два взаимосвязанных параметра показывают, как кабель взаимодействует с окружающей средой, то есть, как он реагирует на внешние помехи, и насколько просто прослушать информацию, передаваемую по кабелю.
- **Скорость распространения сигнала** по кабелю или, обратный параметр – задержка сигнала на метр длины кабеля. Этот параметр имеет принципиальное значение при выборе длины сети. Типичные величины скорости распространения сигнала – от 0,6 до 0,8 от скорости распространения света в вакууме. Соответственно типичные величины задержек – от 4 до 5 нс/м.
- Для электрических кабелей очень важна величина **волнового сопротивления** кабеля. Волновое сопротивление важно учитывать при согласовании кабеля для предотвращения отражения сигнала от концов кабеля. Волновое сопротивление зависит от формы и взаиморасположения проводников, от технологии изготовления и материала диэлектрика кабеля. Типичные значения волнового сопротивления – от 50 до 150 Ом.

В настоящее время действует следующие стандарты на кабели:

- EIA/TIA 568 (Commercial Building Telecommunications Cabling Standard) – американский;
- ISO/IEC IS 11801 (Generic cabling for customer premises) – международный;
- CENELEC EN 50173 (Generic cabling systems) – европейский.

Эти стандарты описывают практически одинаковые кабельные системы, но отличаются терминологией и нормами на параметры. В данной работе предлагается придерживаться терминологии стандарта EIA/TIA 568.

2.1 Кабели на основе витых пар

Витые пары проводов используются в дешевых и сегодня, пожалуй, самых популярных кабелях. Кабель на основе витых пар представляет собой несколько пар скрученных попарно изолированных медных проводов в единой диэлектрической (пластиковой) оболочке. Он довольно гибкий и удобный для прокладки. Скручивание проводов позволяет свести к минимуму индуктивные наводки кабелей друг на друга и снизить влияние переходных процессов.

Обычно в кабель входит две (рис. 2.1) или четыре витые пары.

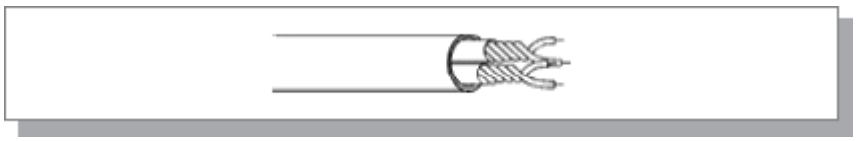

Рис. 2.1. Кабель с витыми парами

Неэкранированные витые пары характеризуются слабой защищенностью от внешних электромагнитных помех, а также от подслушивания, которое может осуществляться с целью, например, промышленного шпионажа. Причем перехват передаваемой по сети информации возможен как с помощью контактного метода (например, посредством двух иголок, воткнутых в кабель), так и с помощью бесконтактного метода, сводящегося к радиоперехвату излучаемых кабелем электромагнитных полей. Причем действие помех и величина излучения вовне увеличивается с ростом длины кабеля. Для устранения этих недостатков применяется экранирование кабелей.

В случае экранированной витой пары STP каждая из витых пар помещается в металлическую оплетку-экран для уменьшения излучений кабеля, защиты от внешних электромагнитных помех и снижения взаимного влияния пар проводов друг на друга (crosstalk – перекрестные наводки). Для того чтобы экран защищал от помех, он должен быть обязательно заземлен. Естественно, экранированная витая пара заметно дороже, чем неэкранированная. Ее использование требует специальных экранированных разъемов. Поэтому встречается она значительно реже, чем неэкранированная витая пара.

Основные достоинства неэкранированных витых пар – простота монтажа разъемов на концах кабеля, а также ремонта любых повреждений по сравнению с другими типами кабеля. Все остальные характеристики у них хуже, чем у других кабелей. Например, при заданной скорости передачи затухание сигнала (уменьшение его уровня по мере прохождения по кабелю) у них больше, чем у коаксиальных кабелей. Если учесть еще низкую помехозащищенность, то понятно, почему линии связи на основе витых пар, как правило, довольно короткие (обычно в пределах 100 метров). В настоящее время витая пара используется для передачи информации на скоростях до 1000 Мбит/с, хотя технические проблемы, возникающие при таких скоростях крайне сложны.

Согласно стандарту EIA/TIA 568, существуют пять основных и две дополнительные категории кабелей на основе неэкранированной витой пары (UTP):

- Кабель категории 1 – это обычный телефонный кабель (пары проводов не витые), по которому можно передавать только речь. Этот тип кабеля имеет большой разброс параметров (волнового сопротивления, полосы пропускания, перекрестных наводок).
- Кабель категории 2 – это кабель из витых пар для передачи данных в полосе частот до 1 МГц. Кабель не тестируется на уровень перекрестных наводок. В

настоящее время он используется очень редко. Стандарт EIA/TIA 568 не различает кабели категорий 1 и 2.

- Кабель категории 3 – это кабель для передачи данных в полосе часто до 16 МГц, состоящий из витых пар с девятью витками проводов на метр длины. Кабель тестируется на все параметры и имеет волновое сопротивление 100 Ом. Это самый простой тип кабелей, рекомендованный стандартом для локальных сетей. Еще недавно он был самым распространенным, но сейчас повсеместно вытесняется кабелем категории 5.
- Кабель категории 4 – это кабель, передающий данные в полосе частот до 20 МГц. Используется редко, так как не слишком заметно отличается от категории 3. Стандартом рекомендуется вместо кабеля категории 3 переходить сразу на кабель категории 5. Кабель категории 4 тестируется на все параметры и имеет волновое сопротивление 100 Ом. Кабель был создан для работы в сетях по стандарту IEEE 802.5.
- Кабель категории 5 – в настоящее время самый совершенный кабель, рассчитанный на передачу данных в полосе частот до 100 МГц. Состоит из витых пар, имеющих не менее 27 витков на метр длины (8 витков на фут). Кабель тестируется на все параметры и имеет волновое сопротивление 100 Ом. Рекомендуется применять его в современных высокоскоростных сетях типа Fast Ethernet и TPFDDI. Кабель категории 5 примерно на 30—50% дороже, чем кабель категории 3.
- Кабель категории 6 – перспективный тип кабеля для передачи данных в полосе частот до 200 (или 250) МГц.
- Кабель категории 7 – перспективный тип кабеля для передачи данных в полосе частот до 600 МГц.

Согласно стандарту EIA/TIA 568, полное волновое сопротивление наиболее совершенных кабелей категорий 3, 4 и 5 должно составлять 100 Ом ±15% в частотном диапазоне от 1 МГц до максимальной частоты кабеля. Требования не очень жесткие: величина волнового сопротивления может находиться в диапазоне от 85 до 115 Ом. Здесь же следует отметить, что волновое сопротивление экранированной витой пары STP по стандарту должно быть равным 150 Ом ±15%. Для согласования сопротивлений кабеля и оборудования в случае их несовпадения применяют согласующие трансформаторы (Balun). Существует также экранированная витая пара с волновым сопротивлением 100 Ом, но используется она довольно редко.

Второй важнейший параметр, задаваемый стандартом, – это максимальное затухание сигнала, передаваемого по кабелю, на разных частотах. В таблице 2.1 приведены предельные значения величины затухания в децибелах для кабелей категорий 3, 4 и 5 на расстояние 1000 футов (то есть 305 метров) при нормальной температуре окружающей среды 20°С.

Частота, МГц	Максимальное затухание, дБ		
	Категория 3	Категория 4	Категория 5
0,064	2,8	2,3	2,2
0,256	4,0	3,4	3,2
0,512	5,6	4,6	4,5
0,772	6,8	5,7	5,5
1,0	7,8	6,5	6,3
4,0	17	13	13

8,0	26	19	18
10,0	30	22	20
16,0	40	27	25
20,0	—	31	28
25,0	—	—	32
31,25	—	—	36
62,5	—	—	52
100	—	—	67

Из таблицы видно, что величины затухания на частотах, близких к предельным, для всех кабелей очень значительны. Даже на небольших расстояниях сигнал ослабляется в десятки и сотни раз, что предъявляет высокие требования к приемникам сигнала.

Еще один специфический параметр, определяемый стандартом, это величина так называемой перекрестной наводки на ближнем конце (NEXT – Near End CrossTalk). Он характеризует влияние разных проводов в кабеле друг на друга. Суть данного параметра иллюстрируется на рис. 2.2. Сигнал, передаваемый по одной из витых пар кабеля (верхняя пара), наводит индуктивную помеху на другую (нижнюю) витую пару кабеля. Две витые пары в сети обычно передают информацию в разные стороны, поэтому наиболее важна наводка на ближнем конце воспринимающей пары (нижней на рисунке), так как именно там находится приемник информации. Перекрестная наводка на дальнем конце (FEXT – Far End CrossTalk) не имеет такого большого значения.

Частота, МГц	Перекрестная наводка на ближнем конце, дБ		
	Категория 3	Категория 4	Категория 5
0,150	- 54	-68	-74
0,772	-43	-58	-64
1,0	-41	-56	-62
4,0	-32	-47	-53
8,0	-28	-42	-48
10,0	-26	-41	-47
16,0	-23	-38	-44
20,0	—	-36	-42
25,0	—	—	-41
31,25	—	—	-40
62,5	—	—	-35
100,0	—	—	-32

В таблице 2.2 представлены значения допустимой перекрестной наводки на ближнем конце для кабелей категорий 3, 4 и 5 на различных частотах сигнала. Естественно, более качественные кабели обеспечивают меньшую величину перекрестной наводки.

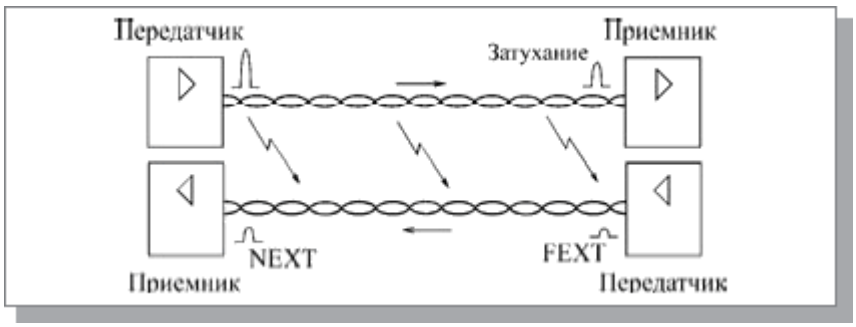

Рис. 2.2. Перекрестные помехи в кабелях на витых парах

Стандарт определяет также максимально допустимую величину рабочей емкости каждой из витых пар кабелей категории 4 и 5. Она должна составлять не более 17 нФ на 305 метров (1000 футов) при частоте сигнала 1 кГц и температуре окружающей среды 20°С.

Для присоединения витых пар используются разъемы (коннекторы) типа RJ-45, похожие на разъемы, используемые в телефонах (RJ-11), но несколько большие по размеру. Разъемы RJ-45 имеют восемь контактов вместо четырех в случае RJ-11. Присоединяются разъемы к кабелю с помощью специальных обжимных инструментов. При этом золоченые игольчатые контакты разъема прокалывают изоляцию каждого провода, входят между его жилами и обеспечивают надежное и качественное соединение. Надо учитывать, что при установке разъемов стандартом допускается расплетение витой пары кабеля на длину не более одного сантиметра.

Чаще всего витые пары используются для передачи данных в одном направлении (точка-точка), то есть в топологиях типа звезда или кольцо. Топология шина обычно ориентируется на коаксиальный кабель. Поэтому внешние терминаторы, согласующие неподключенные концы кабеля, для витых пар практически никогда не применяются.

Кабели выпускаются с двумя типами внешних оболочек:

- Кабель в поливинилхлоридной (ПВХ, PVC) оболочке дешевле и предназначен для работы в сравнительно комфортных условиях эксплуатации.
- Кабель в тефлоновой оболочке дороже и предназначен для более жестких условий эксплуатации.

Кабель в ПВХ оболочке называется еще non-plenum, а в тефлоновой – plenum. Термин plenum обозначает в данном случае пространство под фальшполом и над подвесным потолком, где удобно размещать кабели сети. Для прокладки в этих скрытых от глаз пространствах как раз удобнее кабель в тефлоновой оболочке, который, в частности, горит гораздо хуже, чем ПВХ – кабель, и не выделяет при этом ядовитых газов в большом количестве.

Еще один важный параметр любого кабеля, который жестко не определяется стандартом, но может существенно повлиять на работоспособность сети, –

это скорость распространения сигнала в кабеле или, другими словами, задержка распространения сигнала в кабеле в расчете на единицу длины.

Производители кабелей иногда указывают величину задержки на метр длины, а иногда – скорость распространения сигнала относительно скорости света (или NVP – Nominal Velocity of Propagation, как ее часто называют в документации). Связаны эти две величины простой формулой:

$$t_з = 1/(3 \times 10^{10} \times NVP)$$

где $t_з$ – величина задержки на метр длины кабеля в наносекундах. Например, если $NVP=0,65$ (65% от скорости света), то задержка $t_з$ будет равна 5,13 нс/м. Типичная величина задержки большинства современных кабелей составляет около 4—5 нс/м.

В таблице 2.3 приведены величины NVP и задержек на метр длины (в наносекундах) для некоторых типов кабеля двух самых известных компаний-производителей AT&T и Belden.

Таблица 2.3. Временные характеристики некоторых кабелей

Фирма	Марка	Категория	Оболочка	NVP	Задержка
AT&T	1010	3	non-plenum	0,67	4,98
AT&T	1041	4	non-plenum	0,70	4,76
AT&T	1061	5	non-plenum	0,70	4,76
AT&T	2010	3	plenum	0,70	4,76
AT&T	2041	4	plenum	0,75	4,44
AT&T	2061	5	plenum	0,75	4,44
Belden	1229A	3	non-plenum	0,69	4,83
Belden	1455A	4	non-plenum	0,72	4,63
Belden	1583A	5	non-plenum	0,72	4,63
Belden	1245A2	3	plenum	0,69	4,83
Belden	1457A	4	plenum	0,75	4,44
Belden	1585A	5	plenum	0,75	4,44

Стоит также отметить, что каждый из проводов, входящих в кабель на основе витых пар, как правило, имеет свой цвет изоляции, что существенно упрощает монтаж разъемов, особенно в том случае, когда концы кабеля находятся в разных комнатах, и контроль с помощью приборов затруднен.

Примером кабеля с экранированными витыми парами может служить кабель STP IBM типа 1, который включает в себя две экранированные витые пары AWG типа 22. Волновое сопротивление каждой пары составляет 150 Ом. Для этого кабеля применяются специальные разъемы, отличающиеся от разъемов для неэкранированной витой пары (например, DB9). Имеются и экранированные версии разъема RJ-45.

2.2 Коаксиальные кабели

Коаксиальный кабель представляет собой электрический кабель, состоящий из центрального медного провода и металлической оплетки (экрана), разделенных между собой слоем диэлектрика (внутренней изоляции) и помещенных в общую внешнюю оболочку (рис. 2.3).

Рис. 2.3. Коаксиальный кабель

Коаксиальный кабель до недавнего времени был очень популярен, что связано с его высокой помехозащищенностью (благодаря металлической оплетке), более широкими, чем в случае витой пары, полосами пропускания (свыше 1ГГц), а также большими допустимыми расстояниями передачи (до километра). К нему труднее механически подключиться для несанкционированного прослушивания сети, он дает также заметно меньше электромагнитных излучений вовне. Однако монтаж и ремонт коаксиального кабеля существенно сложнее, чем витой пары, а стоимость его выше (он дороже примерно в 1,5 – 3 раза). Сложнее и установка разъемов на концах кабеля. Сейчас его применяют реже, чем витую пару. Стандарт EIA/TIA-568 включает в себя только один тип коаксиального кабеля, применяемый в сети Ethernet.

Основное применение коаксиальный кабель находит в сетях с топологией типа шина. При этом на концах кабеля обязательно должны устанавливаться терминаторы для предотвращения внутренних отражений сигнала, причем один (и только один!) из терминаторов должен быть заземлен. Без заземления металлическая оплетка не защищает сеть от внешних электромагнитных помех и не снижает излучение передаваемой по сети информации во внешнюю среду. Но при заземлении оплетки в двух или более точках из строя может выйти не только сетевое оборудование, но и компьютеры, подключенные к сети. Терминаторы должны быть обязательно согласованы с кабелем, необходимо, чтобы их сопротивление равнялось волновому сопротивлению кабеля. Например, если используется 50-омный кабель, для него подходят только 50-омные терминаторы.

Реже коаксиальные кабели применяются в сетях с топологией звезда (например, пассивная звезда в сети Arcnet). В этом случае проблема согласования существенно упрощается, так как внешних терминаторов на свободных концах не требуется.

Волновое сопротивление кабеля указывается в сопроводительной документации. Чаще всего в локальных сетях применяются 50-омные (RG-58, RG-11, RG-8) и 93-омные кабели (RG-62)., Распространенные в телевизионной технике 75-омные кабели в локальных сетях не используются. Марок коаксиального кабеля немного. Он не считается особо перспективным. Не случайно в сети Fast Ethernet не предусмотрено применение коаксиальных кабелей. Однако во многих случаях классическая шинная топология (а не пассивная звезда) очень удобна. Как уже отмечалось, она не требует применения дополнительных устройств – концентраторов.

Существует два основных типа коаксиального кабеля:

- тонкий (thin) кабель, имеющий диаметр около 0,5 см, более гибкий;
- толстый (thick) кабель, диаметром около 1 см, значительно более жесткий. Он представляет собой классический вариант коаксиального кабеля, который уже почти полностью вытеснен современным тонким кабелем.

Тонкий кабель используется для передачи на меньшие расстояния, чем толстый, поскольку сигнал в нем затухает сильнее. Зато с тонким кабелем гораздо удобнее работать: его можно оперативно проложить к каждому компьютеру, а толстый требует жесткой фиксации на стене помещения. Подключение к тонкому кабелю (с помощью разъемов BNC байонетного типа) проще и не требует дополнительного оборудования. А для подключения к толстому кабелю надо использовать специальные довольно дорогие устройства, прокалывающие его оболочки и устанавливающие контакт как с центральной жилой, так и с экраном. Толстый кабель примерно вдвое дороже, чем тонкий, поэтому тонкий кабель применяется гораздо чаще.

Как и в случае витых пар, важным параметром коаксиального кабеля является тип его внешней оболочки. Точно так же в данном случае применяются как non-plenum (PVC), так и plenum кабели. Естественно, тефлоновый кабель дороже поливинилхлоридного. Обычно тип оболочки можно отличить по окраске (например, для PVC кабеля фирма Belden использует желтый цвет, а для тефлонового – оранжевый).

Типичные величины задержки распространения сигнала в коаксиальном кабеле составляют для тонкого кабеля около 5 нс/м, а для толстого – около 4,5 нс/м.

Существуют варианты коаксиального кабеля с двойным экраном (один экран расположен внутри другого и отделен от него дополнительным слоем изоляции). Такие кабели имеют лучшую помехозащищенность и защиту от прослушивания, но они немного дороже обычных.

В настоящее время считается, что коаксиальный кабель устарел, в большинстве случаев его вполне может заменить витая пара или

оптоволоконный кабель. И новые стандарты на кабельные системы уже не включают его в перечень типов кабелей.

2.3 Оптоволоконные кабели

Оптоволоконный (он же волоконно-оптический) кабель – это принципиально иной тип кабеля по сравнению с рассмотренными двумя типами электрического или медного кабеля. Информация по нему передается не электрическим сигналом, а световым. Главный его элемент – это прозрачное стекловолокно, по которому свет проходит на огромные расстояния (до десятков километров) с незначительным ослаблением.

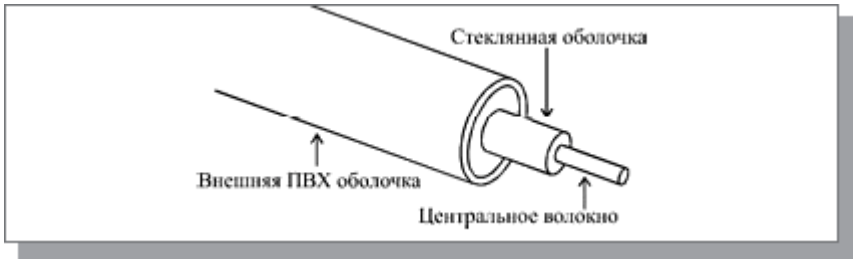

Рис. 2.4. Структура оптоволоконного кабеля

Структура оптоволоконного кабеля очень проста и похожа на структуру коаксиального электрического кабеля (рис. 2.4). Только вместо центрального медного провода здесь используется тонкое (диаметром около 1 – 10 мкм) стекловолокно, а вместо внутренней изоляции – стеклянная или пластиковая оболочка, не позволяющая свету выходить за пределы стекловолокна. В данном случае речь идет о режиме так называемого полного внутреннего отражения света от границы двух веществ с разными коэффициентами преломления (у стеклянной оболочки коэффициент преломления значительно ниже, чем у центрального волокна). Металлическая оплетка кабеля обычно отсутствует, так как экранирование от внешних электромагнитных помех здесь не требуется. Однако иногда ее все-таки применяют для механической защиты от окружающей среды (такой кабель иногда называют броневым, он может объединять под одной оболочкой несколько оптоволоконных кабелей).

Оптоволоконный кабель обладает исключительными характеристиками по помехозащищенности и секретности передаваемой информации. Никакие внешние электромагнитные помехи в принципе не способны исказить световой сигнал, а сам сигнал не порождает внешних электромагнитных излучений. Подключиться к этому типу кабеля для несанкционированного прослушивания сети практически невозможно, так как при этом нарушается целостность кабеля. Теоретически возможная полоса пропускания такого кабеля достигает величины 10^{12} Гц, то есть 1000 ГГц, что несравнимо выше, чем у электрических кабелей. Стоимость оптоволоконного кабеля постоянно

снижается и сейчас примерно равна стоимости тонкого коаксиального кабеля.

Типичная величина затухания сигнала в оптоволоконных кабелях на частотах, используемых в локальных сетях, составляет от 5 до 20 дБ/км, что примерно соответствует показателям электрических кабелей на низких частотах. Но в случае оптоволоконного кабеля при росте частоты передаваемого сигнала затухание увеличивается очень незначительно, и на больших частотах (особенно свыше 200 МГц) его преимущества перед электрическим кабелем неоспоримы, у него просто нет конкурентов.

Однако оптоволоконный кабель имеет и некоторые недостатки.

Самый главный из них – высокая сложность монтажа (при установке разъемов необходима микронная точность, от точности скола стекловолокна и степени его полировки сильно зависит затухание в разъеме). Для установки разъемов применяют сварку или склеивание с помощью специального геля, имеющего такой же коэффициент преломления света, что и стекловолокно. В любом случае для этого нужна высокая квалификация персонала и специальные инструменты. Поэтому чаще всего оптоволоконный кабель продается в виде заранее нарезанных кусков разной длины, на обоих концах которых уже установлены разъемы нужного типа. Следует помнить, что некачественная установка разъема резко снижает допустимую длину кабеля, определяемую затуханием.

Также надо помнить, что использование оптоволоконного кабеля требует специальных оптических приемников и передатчиков, преобразующих световые сигналы в электрические и обратно, что порой существенно увеличивает стоимость сети в целом.

Оптоволоконные кабели допускают разветвление сигналов (для этого производятся специальные пассивные **разветвители** (couplers) на 2—8 каналов), но, как правило, их используют для передачи данных только в одном направлении между одним передатчиком и одним приемником. Ведь любое разветвление неизбежно сильно ослабляет световой сигнал, и если разветвлений будет много, то свет может просто не дойти до конца сети. Кроме того, в разветвителе есть и внутренние потери, так что суммарная мощность сигнала на выходе меньше входной мощности.

Оптоволоконный кабель менее прочен и гибок, чем электрический. Типичная величина допустимого радиуса изгиба составляет около 10 – 20 см, при меньших радиусах изгиба центральное волокно может сломаться. Плохо переносит кабель и механическое растяжение, а также раздавливающие воздействия.

Чувствителен оптоволоконный кабель и к ионизирующим излучениям, из-за которых снижается прозрачность стекловолокна, то есть увеличивается затухание сигнала. Резкие перепады температуры также негативно сказываются на нем, стекловолокно может треснуть.

Применяют оптоволоконный кабель только в сетях с топологией звезда и кольцо. Никаких проблем согласования и заземления в данном случае не существует. Кабель обеспечивает идеальную гальваническую развязку компьютеров сети. В будущем этот тип кабеля, вероятно, вытеснит электрические кабели или, во всяком случае, сильно потеснит их. Запасы меди на планете истощаются, а сырья для производства стекла более чем достаточно.

Существуют два различных типа оптоволоконного кабеля:

- **многомодовый** или **мультимодовый** кабель, более дешевый, но менее качественный;
- **одномодовый** кабель, более дорогой, но имеет лучшие характеристики по сравнению с первым.

Суть различия между этими двумя типами сводится к разным режимам прохождения световых лучей в кабеле.

Рис. 2.5. Распространение света в одномодовом кабеле

В одномодовом кабеле практически все лучи проходят один и тот же путь, в результате чего они достигают приемника одновременно, и форма сигнала почти не искажается (рис. 2.5). Одномодовый кабель имеет диаметр центрального волокна около 1,3 мкм и передает свет только с такой же длиной волны (1,3 мкм). Дисперсия и потери сигнала при этом очень незначительны, что позволяет передавать сигналы на значительно большее расстояние, чем в случае применения многомодового кабеля. Для одномодового кабеля применяются лазерные приемопередатчики, использующие свет исключительно с требуемой длиной волны. Такие приемопередатчики пока еще сравнительно дороги и не долговечны. Однако в перспективе одномодовый кабель должен стать основным типом благодаря своим прекрасным характеристикам. К тому же лазеры имеют большее быстродействие, чем обычные светодиоды. Затухание сигнала в одномодовом кабеле составляет около 5 дБ/км и может быть даже снижено до 1 дБ/км.

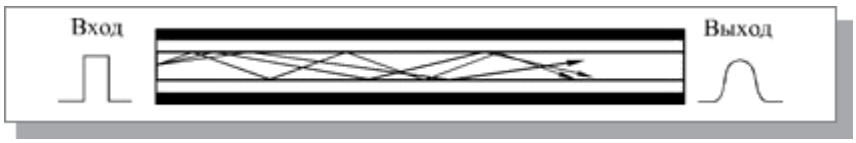

Рис. 2.6. Распространение света в многомодовом кабеле

В многомодовом кабеле траектории световых лучей имеют заметный разброс, в результате чего форма сигнала на приемном конце кабеля искажается (рис. 2.6). Центральное волокно имеет диаметр 62,5 мкм, а диаметр внешней оболочки 125 мкм (это иногда обозначается как 62,5/125). Для передачи используется обычный (не лазерный) светодиод, что снижает стоимость и увеличивает срок службы приемопередатчиков по сравнению с одномодовым кабелем. Длина волны света в многомодовом кабеле равна 0,85 мкм, при этом наблюдается разброс длин волн около 30 – 50 нм. Допустимая длина кабеля составляет 2 – 5 км. Многомодовый кабель – это основной тип оптоволоконного кабеля в настоящее время, так как он дешевле и доступнее. Затухание в многомодовом кабеле больше, чем в одномодовом и составляет 5 – 20 дБ/км.

Типичная величина задержки для наиболее распространенных кабелей составляет около 4—5 нс/м, что близко к величине задержки в электрических кабелях.

Оптоволоконные кабели, как и электрические, выпускаются в исполнении plenum и non-plenum.

2.4 Бескабельные каналы связи

Кроме кабельных каналов в компьютерных сетях иногда используются также бескабельные каналы. Их главное преимущество состоит в том, что не требуется никакой прокладки проводов (не надо делать отверстий в стенах, закреплять кабель в трубах и желобах, прокладывать его под фальшполами, над подвесными потолками или в вентиляционных шахтах, искать и устранять повреждения). К тому же компьютеры сети можно легко перемещать в пределах комнаты или здания, так как они ни к чему не привязаны.

Радиоканал использует передачу информации по радиоволнам, поэтому теоретически он может обеспечить связь на многие десятки, сотни и даже тысячи километров. Скорость передачи достигает десятков мегабит в секунду (здесь многое зависит от выбранной длины волны и способа кодирования).

Особенность радиоканала состоит в том, что сигнал свободно излучается в эфир, он не замкнут в кабель, поэтому возникают проблемы совместимости с другими источниками радиоволн (радио- и телевещательными станциями, радарами, радиолюбительскими и профессиональными передатчиками и

т.д.). В радиоканале используется передача в узком диапазоне частот и модуляция информационным сигналом сигнала несущей частоты.

Главным недостатком радиоканала является его плохая защита от прослушивания, так как радиоволны распространяются неконтролируемо. Другой большой недостаток радиоканала – слабая помехозащищенность.

Для локальных беспроводных сетей (WLAN – Wireless LAN) в настоящее время применяются подключения по радиоканалу на небольших расстояниях (обычно до 100 метров) и в пределах прямой видимости. Чаще всего используются два частотных диапазона – 2,4 ГГц и 5 ГГц. Скорость передачи – до 54 Мбит/с. Распространен вариант со скоростью 11 Мбит/с.

Сети WLAN позволяют устанавливать беспроводные сетевые соединения на ограниченной территории (обычно внутри офисного или университетского здания или в таких общественных местах, как аэропорты). Они могут использоваться во временных офисах или в других местах, где прокладка кабелей неосуществима, а также в качестве дополнения к имеющейся проводной локальной сети, призванного обеспечить пользователям возможность работать перемещаясь по зданию.

Популярная технология Wi-Fi (Wireless Fidelity) позволяет организовать связь между компьютерами числом от 2 до 15 с помощью концентратора (называемого точка доступа, Access Point, AP), или нескольких концентраторов, если компьютеров от 10 до 50. Кроме того, эта технология дает возможность связать две локальные сети на расстоянии до 25 километров с помощью мощных беспроводных мостов. Для примера на рис. 2.7 показано объединение компьютеров с помощью одной точки доступа. Важно, что многие мобильные компьютеры (ноутбуки) уже имеют встроенный контроллер Wi-Fi, что существенно упрощает их подключение к беспроводной сети.

Рис. 2.7. Объединение компьютеров с помощью технологии Wi-Fi

Радиоканал широко применяется в глобальных сетях как для наземной, так и для спутниковой связи. В этом применении у радиоканала нет конкурентов, так как радиоволны могут дойти до любой точки земного шара.

Инфракрасный канал также не требует соединительных проводов, так как использует для связи инфракрасное излучение (подобно пульту

дистанционного управления домашнего телевизора). Главное его преимущество по сравнению с радиоканалом – нечувствительность к электромагнитным помехам, что позволяет применять его, например, в производственных условиях, где всегда много помех от силового оборудования. Правда, в данном случае требуется довольно высокая мощность передачи, чтобы не влияли никакие другие источники теплового (инфракрасного) излучения. Плохо работает инфракрасная связь и в условиях сильной запыленности воздуха.

Скорости передачи информации по инфракрасному каналу обычно не превышают 5-10 Мбит/с, но при использовании инфракрасных лазеров может быть достигнута скорость более 100 Мбит/с. Секретность передаваемой информации, как и в случае радиоканала, не достигается, также, требуются сравнительно дорогие приемники и передатчики. Все это приводит к тому, что применяют инфракрасные каналы в локальных сетях довольно редко. В основном они используются для связи компьютеров с периферией (интерфейс IrDA).

Инфракрасные каналы делятся на две группы:

- Каналы прямой видимости, в которых связь осуществляется на лучах, идущих непосредственно от передатчика к приемнику. При этом связь возможна только при отсутствии препятствий между компьютерами сети. Зато протяженность канала прямой видимости может достигать нескольких километров.
- Каналы на рассеянном излучении, которые работают на сигналах, отраженных от стен, потолка, пола и других препятствий. Препятствия в данном случае не помеха, но связь может осуществляться только в пределах одного помещения.

Если говорить о возможных топологиях, то наиболее естественно все беспроводные каналы связи подходят для топологии типа шина, в которой информация передается одновременно всем абонентам. Но при использовании узконаправленной передачи и/или частотного разделения по каналам можно реализовать любые топологии (кольцо, звезда, комбинированные топологии) как на радиоканале, так и на инфракрасном канале.

3.1 Согласование, экранирование и гальваническая развязка линий связи

Как уже отмечалось, электрические линии связи (витые пары, коаксиальные кабели) требуют проведения специальных мер, без которых невозможна не только безошибочная передача данных, но и вообще любое функционирование сети. Оптоволоконные кабели решают все подобные проблемы автоматически.

Согласование электрических линий связи применяется для обеспечения нормального прохождения сигнала по длинной линии без отражений и

искажений. Следует отметить, что в локальных сетях кабель работает в режиме длинной линии даже при минимальных расстояниях между компьютерами, так как скорости передачи информации и частотный спектр сигнала очень велики.

Принцип согласования кабеля прост: на его концах необходимо установить согласующие резисторы (терминаторы) с сопротивлением, равным волновому сопротивлению используемого кабеля.

Как уже упоминалось, волновое сопротивление – это параметр данного типа кабеля, зависящий только от его устройства (сечения, количества и формы проводников, толщины и материала изоляции и т.д.). Величина волнового сопротивления обязательно указывается в сопроводительной документации на кабель и составляет обычно от 50—100 Ом для коаксиального кабеля, до 100—150 Ом для витой пары или плоского многопроводного кабеля. Точное значение волнового сопротивления легко можно измерить с помощью генератора прямоугольных импульсов и осциллографа как раз по отсутствию искажения формы передаваемого по кабелю импульса. Обычно требуется, чтобы отклонение величины согласующего резистора не превышало 10% в ту или другую сторону.

Если согласующее, нагрузочное сопротивление $R_н$ меньше волнового сопротивления кабеля $R_в$, то фронт передаваемого прямоугольного импульса на приемном конце будет затянут, если же $R_н$ больше $R_в$, то на фронте будет колебательный процесс (рис.3.1).

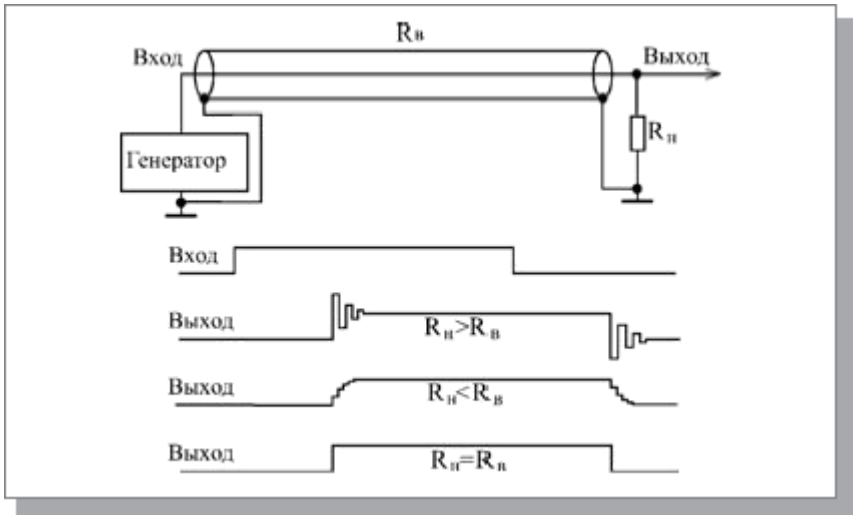

Рис. 3.1. Передача сигналов по электрическому кабелю

Сетевые адаптеры, их приемники и передатчики специально рассчитываются на работу с данным типом кабеля с известным волновым сопротивлением. Поэтому даже при идеально согласованном на концах кабеле, волновое сопротивление которого существенно отличается от стандартного, сеть, скорее всего, работать не будет или будет работать со сбоями.

Здесь же стоит упомянуть о том, что сигналы с пологими фронтами передаются по длинному электрическому кабелю лучше, чем сигналы с крутыми фронтами. Их форма значительно меньше искажается (рис. 3.2). Это связано с разницей величин затухания для разных частот (высокие частоты затухают сильнее). Меньше всего искажается форма синусоидального сигнала, он просто уменьшается по амплитуде. Для улучшения качества передачи нередко используются трапецевидные или колоколообразные импульсы (рис. 3.3), близкие по форме к полуволне синуса, для чего искусственно затягиваются или сглаживаются фронты изначальных прямоугольных сигналов.

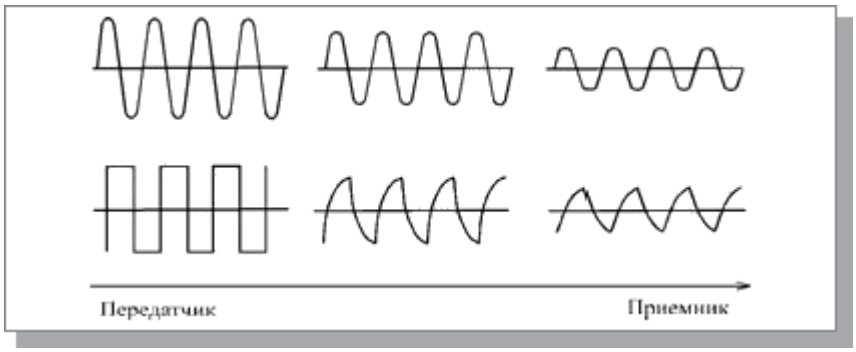

Рис. 3.2. Затухание сигналов в электрическом кабеле

Рис. 3.3. Трапециевидный и колоколообразный импульсы

Экранирование электрических линий связи применяется для снижения влияния на кабель внешних электромагнитных полей. Экран представляет собой медную или алюминиевую оболочку (плетеную или из фольги), в которую заключаются провода кабеля. Экранирование будет работать, если экран заземлен, поскольку необходимо, чтобы наведенные на него токи стекали на землю. Кроме того, экранирование заметно уменьшает и внешние излучения кабеля, что важно для обеспечения секретности передаваемой информации. Побочными полезными эффектами экранирования являются увеличение прочности кабеля и трудности с механическим подключением к кабелю для подслушивания. Экран заметно повышает стоимость кабеля, но также его механическую прочность.

Снизить влияние наведенных помех можно и без экрана, если использовать дифференциальную передачу сигнала (рис. 3.4). В этом случае передача идет по двум проводам, причем оба провода являются сигнальными. Передатчик формирует противофазные сигналы, а приемник реагирует на разность сигналов в обоих проводах. Условием согласования является равенство сопротивлений согласующих резисторов R половине волнового сопротивления кабеля Rв. Если оба провода имеют одинаковую длину и проложены рядом (в одном кабеле), то помехи действуют на оба провода

примерно одинаково, и в результате разностный сигнал между проводами практически не искажается. Именно такая дифференциальная передача применяется обычно в кабелях из витых пар. Но экранирование и в этом случае существенно улучшает помехоустойчивость.

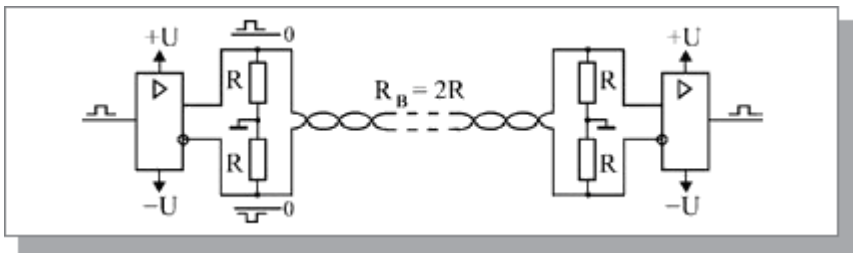

Рис. 3.4. Дифференциальная передача сигналов по витой паре

Гальваническая развязка компьютеров от сети при использовании электрического кабеля совершенно необходима. Дело в том, что по электрическим кабелям (как по сигнальным проводам, так и по экрану) могут идти не только информационные сигналы, но и так называемый выравнивающий ток, возникающий вследствие неидеальности заземления компьютеров.

Когда компьютер не заземлен, на его корпусе образуется наведенный потенциал около 110 вольт переменного тока (половина питающего напряжения). Его можно ощутить на себе, если одной рукой взяться за корпус компьютера, а другой за батарею центрального отопления или за какой-нибудь заземленный прибор.

При автономной работе компьютера отсутствие заземления, как правило, не оказывает серьезного влияния на его работу. Правда, иногда увеличивается количество сбоев в работе машины. Но при соединении нескольких территориально разнесенных компьютеров электрическим кабелем заземление становится серьезной проблемой. Если один из соединяемых компьютеров заземлен, а другой нет, то возможен выход из строя одного из них или обоих. Поэтому компьютеры крайне желательно заземлять.

В случае использования трехконтактной вилки и розетки, в которых есть нулевой провод, это получается автоматически. При двухконтактной вилке и розетке необходимо принимать специальные меры, организовывать заземление отдельным проводом большого сечения. Стоит также отметить, что в случае трехфазной сети желательно обеспечить питание всех компьютеров от одной фазы.

Но проблема осложняется еще и тем, что «земля», к которой присоединяются компьютеры, обычно далека от идеала. Теоретически заземляющие провода компьютеров должны сходиться в одной точке, соединенной короткой массивной шиной с зарытым в землю массивным проводником. Такая ситуация возможна только если компьютеры не слишком разнесены, и

заземление действительно сделано грамотно. Обычно же заземляющая шина имеет значительную длину, в результате чего стекающие по ней токи создают довольно большую разность потенциалов между ее отдельными точками. Особенно велика эта разность потенциалов в случае подключения к шине мощных и высокочастотных потребителей энергии.

Присоединенные к одной и той же шине, но в разных точках, компьютеры имеют на своих корпусах разные потенциалы (рис. 3.5). В результате по электрическому кабелю, соединяющему компьютеры, потечет выравнивающий ток (переменный с высокочастотными составляющими).

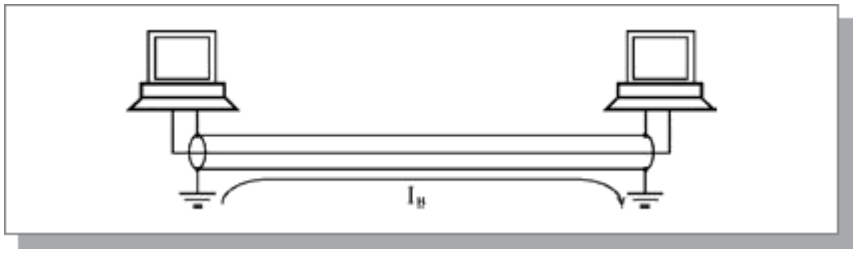

Рис. 3.5. Выравнивающий ток при отсутствии гальванической развязки

Хуже, когда компьютеры подключаются к разным шинам заземления. Выравнивающий ток может достигать в этом случае величины в несколько ампер. Подобные токи смертельно опасны для малосигнальных узлов компьютера. Кроме того выравнивающий ток существенно влияет на передаваемый сигнал, порой полностью забивая его. Даже тогда, когда сигналы передаются без участия экрана (например, по двум проводам, заключенным в экран) вследствие индуктивного действия выравнивающий ток мешает передаче информации. Именно поэтому экран всегда должен быть заземлен только в одной точке.

Однако если каждый из компьютеров самостоятельно заземлен, то заземление экрана в одной точке становится невозможным без гальванической развязки компьютеров от сети. Таким образом не должно быть связи по постоянному току между корпусом («землей») компьютера и экраном («землей») сетевого кабеля. В то же время, информационный сигнал должен передаваться из компьютера в сеть и из сети в компьютер. Для гальванической развязки обычно применяют импульсные трансформаторы, которые входят в состав сетевого оборудования (например, сетевых адаптеров). Трансформатор пропускает высокочастотные информационные сигналы, но обеспечивает полную изоляцию по постоянному току.

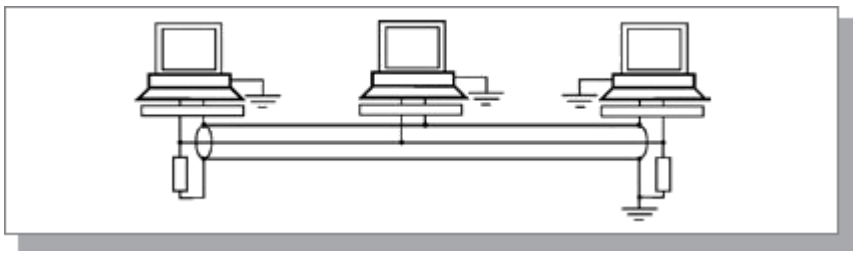

Рис. 3.6. Правильное соединение компьютеров сети (гальваническая развязка условно показана в виде прямоугольника)

Грамотное соединение компьютеров локальной сети электрическим кабелем обязательно должно включать в себя следующее (рис. 3.6):

- оконечное согласование кабеля с помощью терминаторов;
- гальваническую развязку компьютеров от сети;
- заземление каждого компьютера;
- заземление экрана (если, конечно, он есть) в одной точке.

Не стоит пренебрегать каким-либо из этих требований. Например, гальваническая развязка сетевых адаптеров часто рассчитывается на допустимое напряжение изоляции всего лишь 100 В, что при отсутствии заземления одного из компьютеров может легко привести к выходу из строя его адаптера.

Следует отметить, что для присоединения коаксиального кабеля обычно применяются разъемы в металлическом корпусе. Этот корпус не должен соединяться ни с корпусом компьютера, ни с «землей» (на плате адаптера он установлен с пластиковой изоляцией от крепежной планки). Заземление экрана кабеля сети лучше производить не через корпус компьютера, а отдельным специальным проводом, что обеспечивает лучшую надежность. Пластмассовые корпуса разъемов RJ-45 для кабелей с неэкранированными витыми парами снимают эту проблему.

Важно также учитывать, что экран кабеля, заземленный в одной точке, является радиоантенной с заземленным основанием. Он может улавливать и усиливать высокочастотные помехи с длиной волны, кратной его длине. Для снижения этого «антенного эффекта» применяется многоточечное заземление экрана по высокой частоте. В каждом сетевом адаптере «земля» сетевого кабеля соединяется с «землей» компьютера через высоковольтные керамические конденсаторы. Для примера на рис. 3.7 показана упрощенная схема гальванической развязки, применяемая в сетевых адаптерах Ethernet.

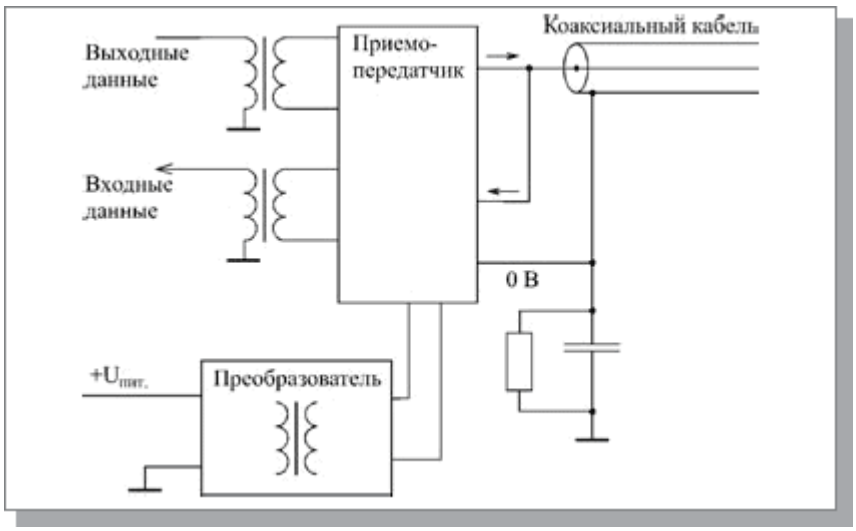

Рис. 3.7. Схема гальванической развязки в сети Ethernet

Приемопередатчик напрямую связан с кабелем сети, но гальванически развязан с помощью трансформаторов от компьютера и остальной части сетевого адаптера. Это продиктовано особенностями протокола CSMA/CD и манчестерского кода, применяемых в Ethernet. Для обеспечения полной развязки питание приемопередатчика осуществляется посредством преобразователя питающего напряжения, имеющего внутри также трансформаторную гальваническую развязку. Оплетка коаксиального кабеля соединена с общим проводом компьютера через высоковольтный конденсатор. Параллельно конденсатору включен резистор с большим сопротивлением (1 МОм), который предотвращает электрический удар пользователя при одновременном касании им оплетки кабеля (корпуса разъема) и корпуса компьютера.

В случае применения витых пар все гораздо проще. Каждая витая пара имеет развязывающие импульсные трансформаторы на обоих своих концах. Ни один из проводов витой пары не заземляется (оба они сигнальные). К тому же разъемы для витых пар имеют пластмассовый корпус.

3.2 Кодирование информации в локальных сетях

Информация в кабельных локальных сетях передается в закодированном виде, то есть каждому биту передаваемой информации соответствует свой набор уровней электрических сигналов в сетевом кабеле. Модуляция высокочастотных сигналов применяется в основном в бескабельных сетях, в радиоканалах. В кабельных сетях передача идет без модуляции или, как еще говорят, в основной полосе частот.

Правильный выбор кода позволяет повысить достоверность передачи информации, увеличить скорость передачи или снизить требования к выбору кабеля. Например, при разных кодах предельная скорость передачи по одному и тому же кабелю может отличаться в два раза. От выбранного кода

напрямую зависит также сложность сетевой аппаратуры (узлы кодирования и декодирования кода). Код должен в идеале обеспечивать хорошую синхронизацию приема, низкий уровень ошибок, работу с любой длиной передаваемых информационных последовательностей.

Некоторые коды, используемые в локальных сетях, показаны на рис. 3.8. Далее будут рассмотрены их преимущества и недостатки.

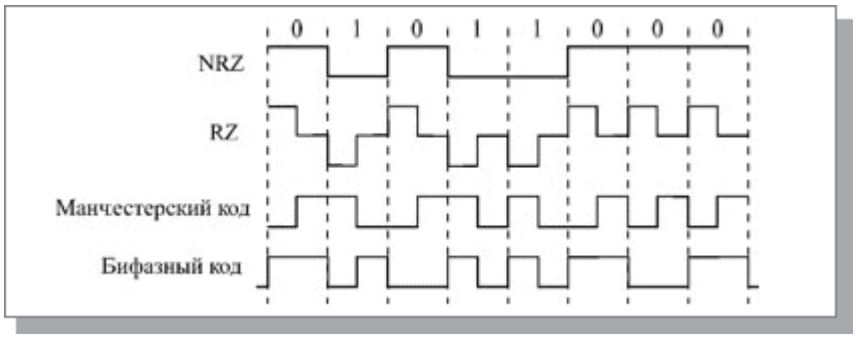

Рис. 3.8. Наиболее распространенные коды передачи информации

Код NRZ

Код NRZ (Non Return to Zero – без возврата к нулю) – это простейший код, представляющий собой обычный цифровой сигнал. Логическому нулю соответствует высокий уровень напряжения в кабеле, логической единице – низкий уровень напряжения (или наоборот, что не принципиально). Уровни могут быть разной полярности (положительной и отрицательной) или же одной полярности (положительной или отрицательной). В течение битового интервала (bit time, BT), то есть времени передачи одного бита никаких изменений уровня сигнала в кабеле не происходит.

К несомненным достоинствам кода NRZ относятся его довольно простая реализация (исходный сигнал не надо ни специально кодировать на передающем конце, ни декодировать на приемном конце), а также минимальная среди других кодов пропускная способность линии связи, требуемая при данной скорости передачи. Ведь наиболее частое изменение сигнала в сети будет при непрерывном чередовании единиц и нулей, то есть при последовательности `1010101010...`, поэтому при скорости передачи, равной 10 Мбит/с (длительность одного бита равна 100 нс) частота изменения сигнала и соответственно требуемая пропускная способность линии составит 1 / 200нс = 5 МГц (рис. 3.9).

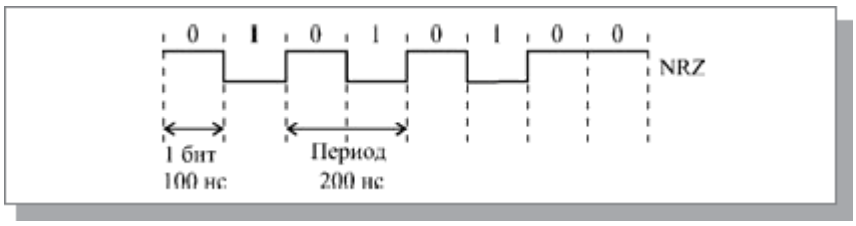

Рис. 3.9. Скорость передачи и требуемая пропускная способность при коде NRZ

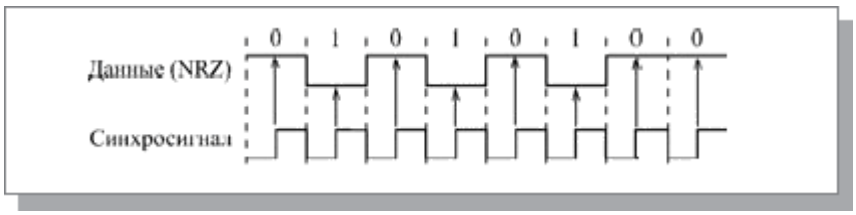

Рис. 3.10. Передача в коде NRZ с синхросигналом

Самый большой недостаток кода NRZ – это возможность потери синхронизации приемником во время приема слишком длинных блоков (пакетов) информации. Приемник может привязывать момент начала приема только к первому (стартовому) биту пакета, а в течение приема пакета он вынужден пользоваться только внутренним тактовым генератором (внутренними часами). Например, если передается последовательность нулей или последовательность единиц, то приемник может определить, где проходят границы битовых интервалов, только по внутренним часам. И если часы приемника расходятся с часами передатчика, то временной сдвиг к концу приема пакета может превысить длительность одного или даже нескольких бит. В результате произойдет потеря переданных данных. Так, при длине пакета в 10000 бит допустимое расхождение часов составит не более 0,01% даже при идеальной передаче формы сигнала по кабелю.

Во избежание потери синхронизации, можно было бы ввести вторую линию связи для синхросигнала (рис. 3.10). Но при этом требуемое количество кабеля, число приемников и передатчиков увеличивается в два раза. При большой длине сети и значительном количестве абонентов это невыгодно.

В связи с этим код NRZ используется только для передачи короткими пакетами (обычно до 1 Кбита).

Большой недостаток кода NRZ состоит еще и в том, что он может обеспечить обмен сообщениями (последовательностями, пакетами) только фиксированной, заранее обговоренной длины. Дело в том, что по принимаемой информации приемник не может определить, идет ли еще передача или уже закончилась. Для синхронизации начала приема пакета используется стартовый служебный бит, чей уровень отличается от пассивного состояния линии связи (например, пассивное состояние линии при отсутствии передачи – 0, стартовый бит – 1). Заканчивается прием после

отсчета приемником заданного количества бит последовательности (рис. 3.11).

Рис. 3.11. Определение окончания последовательности при коде NRZ

Наиболее известное применение кода NRZ – это стандарт RS232-C, последовательный порт персонального компьютера. Передача информации в нем ведется байтами (8 бит), сопровождаемыми стартовым и стоповым битами.

Три остальных кода (RZ, манчестерский код, бифазный код) принципиально отличаются от NRZ тем, что сигнал имеет дополнительные переходы (фронты) в пределах битового интервала. Это сделано для того, чтобы приемник мог подстраивать свои часы под принимаемый сигнал на каждом битовом интервале. Отслеживая фронты сигналов, приемник может точно синхронизовать прием каждого бита. В результате небольшие расхождения часов приемника и передатчика уже не имеют значения. Приемник может надежно принимать последовательности любой длины. Такие коды называются самосинхронизирующимися. Можно считать, что самосинхронизирующиеся коды несут в себе синхросигнал.

Код RZ

Код RZ (Return to Zero – с возвратом к нулю) – этот трехуровневый код получил такое название потому, что после значащего уровня сигнала в первой половине битового интервала следует возврат к некоему «нулевому», среднему уровню (например, к нулевому потенциалу). Переход к нему происходит в середине каждого битового интервала. Логическому нулю, таким образом, соответствует положительный импульс, логической единице – отрицательный (или наоборот) в первой половине битового интервала.

В центре битового интервала всегда есть переход сигнала (положительный или отрицательный), следовательно, из этого кода приемник легко может выделить синхроимпульс (строб). Возможна временная привязка не только к началу пакета, как в случае кода NRZ, но и к каждому отдельному биту, поэтому потери синхронизации не произойдет при любой длине пакета.

Еще одно важное достоинство кода RZ – простая временная привязка приема, как к началу последовательности, так и к ее концу. Приемник просто

должен анализировать, есть изменение уровня сигнала в течение битового интервала или нет. Первый битовый интервал без изменения уровня сигнала соответствует окончанию принимаемой последовательности бит (рис. 3.12). Поэтому в коде RZ можно использовать передачу последовательностями переменной длины.

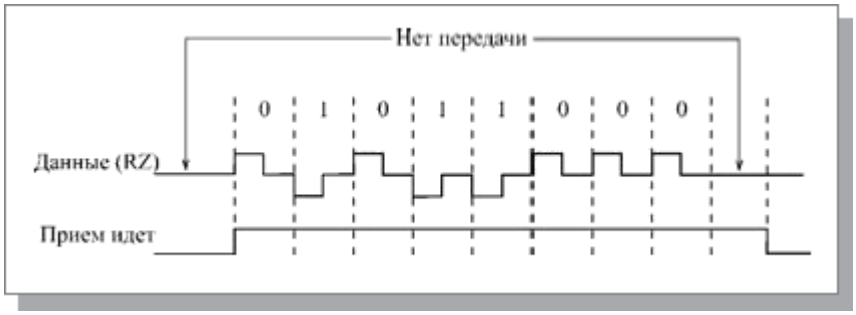

Рис. 3.12. Определение начала и конца приема при коде RZ

Недостаток кода RZ состоит в том, что для него требуется вдвое большая полоса пропускания канала при той же скорости передачи по сравнению с NRZ (так как здесь на один битовый интервал приходится два изменения уровня сигнала). Например, для скорости передачи информации 10 Мбит/с требуется пропускная способность линии связи 10 МГц, а не 5 МГц, как при коде NRZ (рис. 3.13).

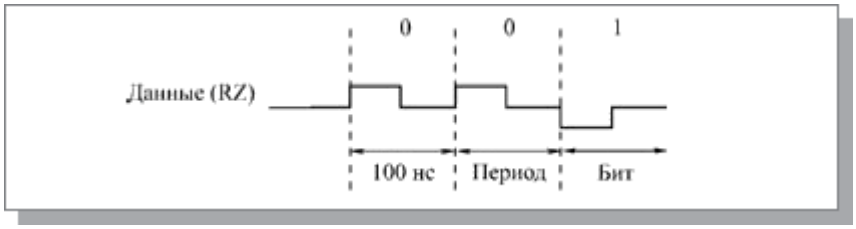

Рис. 3.13. Скорость передачи и пропускная способность при коде RZ

Другой важный недостаток – наличие трех уровней, что всегда усложняет аппаратуру как передатчика, так и приемника.

Код RZ применяется не только в сетях на основе электрического кабеля, но и в оптоволоконных сетях. Правда, в них не существует положительных и отрицательных уровней сигнала, поэтому используется три следующие уровня: отсутствие света, «средний» свет, «сильный» свет. Это очень удобно: даже когда нет передачи информации, свет все равно присутствует, что позволяет легко определить целостность оптоволоконной линии связи без дополнительных мер (рис. 3.14).

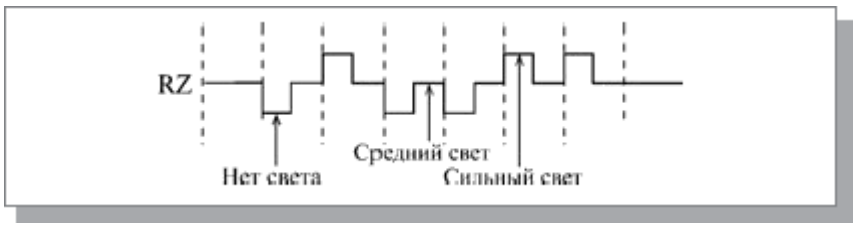

Рис. 3.14. Использование кода RZ в оптоволоконных сетях

Манчестерский код

Манчестерский код (или код Манчестер-II) получил наибольшее распространение в локальных сетях. Он также относится к самосинхронизирующимся кодам, но в отличие от RZ имеет не три, а всего два уровня, что способствует его лучшей помехозащищенности и упрощению приемных и передающих узлов. Логическому нулю соответствует положительный переход в центре битового интервала (то есть первая половина битового интервала – низкий уровень, вторая половина – высокий), а логической единице соответствует отрицательный переход в центре битового интервала (или наоборот).

Как и в RZ, обязательное наличие перехода в центре бита позволяет приемнику манчестерского кода легко выделить из пришедшего сигнала синхросигнал и передать информацию сколь угодно большими последовательностями без потерь из-за рассинхронизации. Допустимое расхождение часов приемника и передатчика может достигать 25%.

Подобно коду RZ, при использовании манчестерского кода требуется пропускная способность линии в два раза выше, чем при применении простейшего кода NRZ. Например, для скорости передачи 10 Мбит/с требуется полоса пропускания 10 МГц (рис. 3.15).

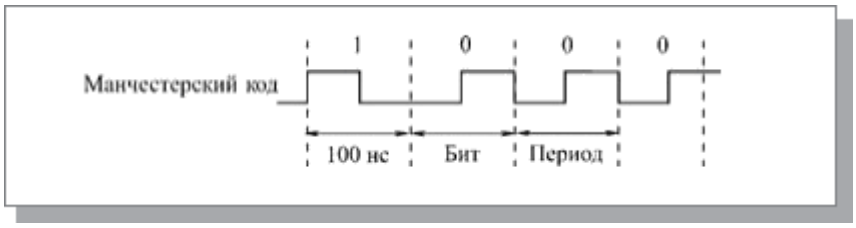

Рис. 3.15. Скорость передачи и пропускная способность при манчестерском коде

Как и при коде RZ, в данном случае приемник легко может определить не только начало передаваемой последовательности бит, но и ее конец. Если в течение битового интервала нет перехода сигнала, то прием заканчивается. В манчестерском коде можно передавать последовательности бит переменной длины (рис. 3.16). Процесс определения времени передачи называют еще контролем несущей, хотя в явном виде несущей частоты в данном случае не присутствует.

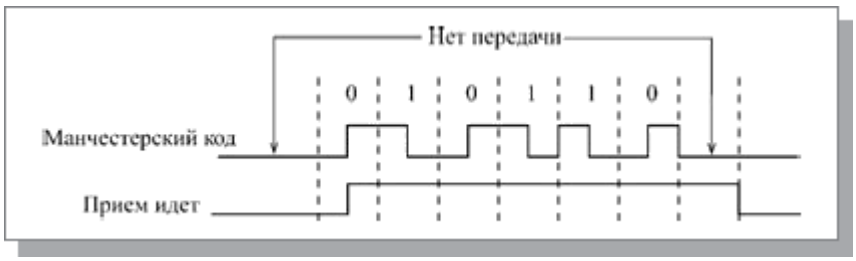

Рис. 3.16. Определение начала и конца приема при манчестерском коде

Манчестерский код используется как в электрических, так и в оптоволоконных кабелях (в последнем случае один уровень соответствует отсутствию света, а другой – его наличию).

Основное достоинство манчестерского кода – постоянная составляющая в сигнале (половину времени сигнал имеет высокий уровень, другую половину – низкий). Постоянная составляющая равна среднему значению между двумя уровнями сигнала.

Если высокий уровень имеет положительную величину, а низкий – такую же отрицательную, то постоянная составляющая равна нулю. Это дает возможность легко применять для гальванической развязки импульсные трансформаторы. При этом не требуется дополнительного источника питания для линии связи (как, например, в случае использования оптронной гальванической развязки), резко уменьшается влияние низкочастотных помех, которые не проходят через трансформатор, легко решается проблема согласования.

Если же один из уровней сигнала в манчестерском коде нулевой (как, например, в сети Ethernet), то величина постоянной составляющей в течение передачи будет равна примерно половине амплитуды сигнала. Это позволяет легко фиксировать столкновения пакетов в сети (конфликт, коллизию) по отклонению величины постоянной составляющей за установленные пределы.

Частотный спектр сигнала при манчестерском кодировании включает в себя только две частоты: при скорости передачи 10 Мбит/с это 10 МГц (соответствует передаваемой цепочке из одних нулей или из одних единиц) и 5 МГц (соответствует последовательности из чередующихся нулей и единиц: `1010101010...`). Поэтому с помощью простейших полосовых фильтров можно легко избавиться от всех других частот (помехи, наводки, шумы).

Бифазный код

Бифазный код часто рассматривают как разновидность манчестерского, так как их характеристики практически полностью совпадают.

Данный код отличается от классического манчестерского кода тем, что он не зависит от перемены мест двух проводов кабеля. Особенно это удобно в

случае, когда для связи применяется витая пара, провода которой легко перепутать. Именно этот код используется в одной из самых известных сетей Token-Ring компании IBM.

Принцип данного кода прост: в начале каждого битового интервала сигнал меняет уровень на противоположный предыдущему, а в середине единичных (и только единичных) битовых интервалов уровень изменяется еще раз. Таким образом, в начале битового интервала всегда есть переход, который используется для самосинхронизации. Как и в случае классического манчестерского кода, в частотном спектре при этом присутствует две частоты. При скорости 10 Мбит/с это частоты 10 МГц (при последовательности одних единиц: `11111111...`) и 5 МГц (при последовательности одних нулей: `00000000...`).

Имеется также еще один вариант бифазного кода (его еще называют дифференциальным манчестерским кодом). В этом коде единице соответствует наличие перехода в начале битового интервала, а нулю – отсутствие перехода в начале битового интервала (или наоборот). При этом в середине битового интервала переход имеется всегда, и именно он служит для побитовой самосинхронизации приемника. Характеристики этого варианта кода также полностью соответствуют характеристикам манчестерского кода.

Здесь же стоит упомянуть о том, что часто совершенно неправомерно считается, что единица измерения скорости передачи бод – это то же самое, что бит в секунду, а скорость передачи в бодах равняется скорости передачи в битах в секунду. Это верно только в случае кода NRZ. Скорость в бодах характеризует не количество передаваемых бит в секунду, а число изменений уровня сигнала в секунду. И при RZ или манчестерском кодах требуемая скорость в бодах оказывается вдвое выше, чем при NRZ. В бодах изменяется скорость передачи сигнала, а в битах в секунду – скорость передачи информации. Поэтому, чтобы избежать неоднозначного понимания, скорость передачи по сети лучше указывать в битах в секунду (бит/с, Кбит/с, Мбит/с, Гбит/с).

Другие коды

Все разрабатываемые в последнее время коды призваны найти компромисс между требуемой при заданной скорости передачи полосой пропускания кабеля и возможностью самосинхронизации. Разработчики стремятся сохранить самосинхронизацию, но не ценой двукратного увеличения полосы пропускания, как в рассмотренных RZ, манчестерском и бифазном кодах.

Чаще всего для этого в поток передаваемых битов добавляют биты синхронизации. Например, один бит синхронизации на 4, 5 или 6 информационных битов или два бита синхронизации на 8 информационных

битов. В действительности все обстоит несколько сложнее: кодирование не сводится к простой вставке в передаваемые данные дополнительных битов. Группы информационных битов преобразуются в передаваемые по сети группы с количеством битов на один или два больше. Приемник осуществляет обратное преобразование, восстанавливает исходные информационные биты. Довольно просто осуществляется в этом случае и обнаружение несущей частоты (детектирование передачи).

Так, например, в сети FDDI (скорость передачи 100 Мбит/с) применяется код 4B/5B, который 4 информационных бита преобразует в 5 передаваемых битов. При этом синхронизация приемника осуществляется один раз на 4 бита, а не в каждом бите, как в случае манчестерского кода. Но зато требуемая полоса пропускания увеличивается по сравнению с кодом NRZ не в два раза, а только в 1,25 раза (то есть составляет не 100 МГц, а всего лишь 62,5 МГц). По тому же принципу строятся и другие коды, в частности, 5B/6B, используемый в стандартной сети 100VG-AnyLAN, или 8B/10B, применяемый в сети Gigabit Ethernet.

В сегменте 100BASE-T4 сети Fast Ethernet использован несколько иной подход. Там применяется код 8B/6T, предусматривающий параллельную передачу трех трехуровневых сигналов по трем витым парам. Это позволяет достичь скорости передачи 100 Мбит/с на дешевых кабелях с витыми парами категории 3, имеющих полосу пропускания всего лишь16 МГц (см. табл. 2.1). Правда, это требует большего расхода кабеля и увеличения количества приемников и передатчиков. К тому же принципиально, чтобы все провода были одной длины и задержки сигнала в них не слишком различались.

Иногда уже закодированная информация подвергается дополнительному кодированию, что позволяет упростить синхронизацию на приемном конце. Наибольшее распространение для этого получили 2-уровневый код NRZI, применяемый в оптоволоконных сетях (FDDI и 100BASE-FX), а также 3-уровневый код MLT-3, используемый в сетях на витых парах (TPDDI и 100BASE-TX). Оба эти кода (рис. 3.17) не являются самосинхронизирующимися.

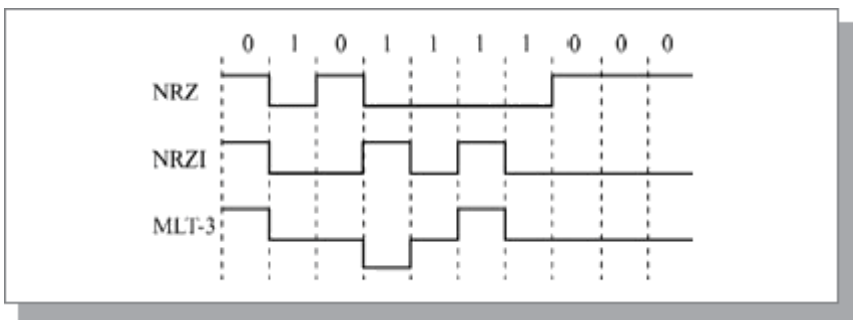

Рис. 3.17. Коды NRZI и MLT-3

Код NRZI (без возврата к нулю с инверсией единиц – Non-Return to Zero, Invert to one) предполагает, что уровень сигнала меняется на противоположный в начале единичного битового интервала и не меняется при передаче нулевого битового интервала. При последовательности единиц на границах битовых интервалов имеются переходы, при последовательности нулей – переходов нет. В этом смысле код NRZI лучше синхронизируется, чем NRZ (там нет переходов ни при последовательности нулей, ни при последовательности единиц).

Код MLT-3 (Multi-Level Transition-3) предполагает, что при передаче нулевого битового интервала уровень сигнала не меняется, а при передаче единицы – меняется на следующий уровень по такой цепочке: +U, 0, –U, 0, +U, 0, –U и т.д. Таким образом, максимальная частота смены уровней получается вчетверо меньше скорости передачи в битах (при последовательности сплошных единиц). Требуемая полоса пропускания оказывается меньше, чем при коде NRZ.

Все упомянутые в данном разделе коды предусматривают непосредственную передачу в сеть цифровых двух- или трехуровневых прямоугольных импульсов.

Однако иногда в сетях используется и другой путь – модуляция информационными импульсами высокочастотного аналогового сигнала (синусоидального). Такое аналоговое кодирование позволяет при переходе на широкополосную передачу существенно увеличить пропускную способность канала связи (в этом случае по сети можно передавать несколько бит одновременно). К тому же, как уже отмечалось, при прохождении по каналу связи аналогового сигнала (синусоидального) не искажается форма сигнала, а только уменьшается его амплитуда, а в случае цифрового сигнала форма сигнала искажается (см. рис. 3.2).

К самым простым видам аналогового кодирования относятся следующие (рис. 3.18):

- Амплитудная модуляция (AM, AM – Amplitude Modulation), при которой логической единице соответствует наличие сигнала (или сигнал большей амплитуды), а логическому нулю – отсутствие сигнала (или сигнал меньшей амплитуды). Частота сигнала при этом остается постоянной. Недостаток амплитудной модуляции состоит в том, что AM-сигнал сильно подвержен действию помех и шумов, а также предъявляет повышенные требования к затуханию сигнала в канале связи. Достоинства – простота аппаратурной реализации и узкий частотный спектр.

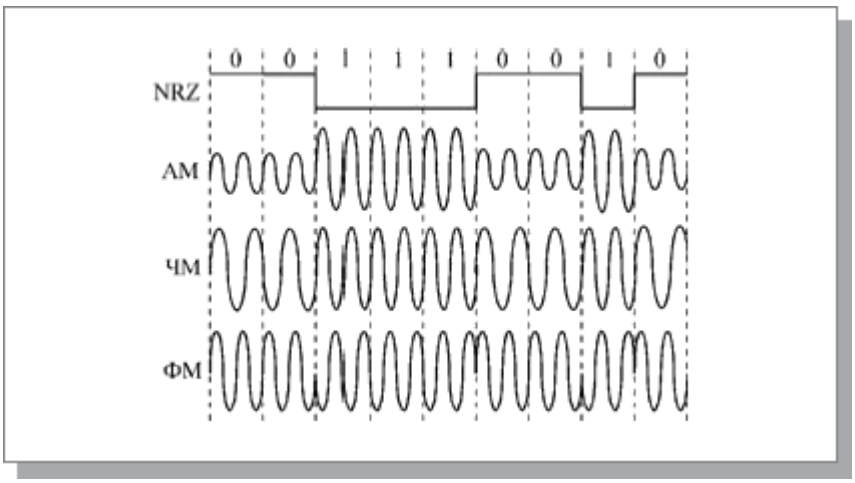

Рис. 3.18. Аналоговое кодирование цифровой информации

- Частотная модуляция (ЧМ, FM – Frequency Modulation), при которой логической единице соответствует сигнал более высокой частоты, а логическому нулю – сигнал более низкой частоты (или наоборот). Амплитуда сигнала при частотной модуляции остается постоянной, что является большим преимуществом по сравнению с амплитудной модуляцией.
- Фазовая модуляция (ФМ, PM – Phase Modulation), при которой смене логического нуля на логическую единицу и наоборот соответствует резкое изменение фазы синусоидального сигнала одной частоты и амплитуды. Важно, что амплитуда модулированного сигнала остается постоянной, как и в случае частотной модуляции.

Применяются и значительно более сложные методы модуляции, являющиеся комбинацией перечисленных простейших методов. О некоторых из них будет рассказано в главе 12.

Чаще всего аналоговое кодирование используется при передаче информации по каналу с узкой полосой пропускания, например, по телефонным линиям в глобальных сетях. Кроме того, аналоговое кодирование применяется в радиоканалах, что позволяет обеспечивать связь между многими пользователями одновременно. В локальных кабельных сетях аналоговое кодирование практически не используется из-за высокой сложности и стоимости как кодирующего, так и декодирующего оборудования.

4. Назначение пакетов и их структура

Информация в локальных сетях, как правило, передается отдельными порциями, кусками, называемыми в различных источниках пакетами (packets), кадрами (frames) или блоками. Причем предельная длина этих пакетов строго ограничена (обычно величиной в несколько килобайт). Ограничена длина пакета и снизу (как правило, несколькими десятками байт). Выбор пакетной передачи связан с несколькими важными соображениями.

Локальная сеть, как уже отмечалось, должна обеспечивать качественную, прозрачную связь всем абонентам (компьютерам) сети. Важнейшим

параметром является так называемое время доступа к сети (access time), которое определяется как временной интервал между моментом готовности абонента к передаче (когда ему есть, что передавать) и моментом начала этой передачи. Это время ожидания абонентом начала своей передачи. Естественно, оно не должно быть слишком большим, иначе величина реальной, интегральной скорости передачи информации между приложениями сильно уменьшится даже при высокоскоростной связи.

Ожидание начала передачи связано с тем, что в сети не может происходить несколько передач одновременно (во всяком случае, при топологиях шина и кольцо). Всегда есть только один передатчик и один приемник (реже – несколько приемников). В противном случае информация от разных передатчиков смешивается и искажается. В связи с этим абоненты передают свою информацию по очереди. И каждому абоненту, прежде чем начать передачу, надо дождаться своей очереди. Вот это время ожидания своей очереди и есть время доступа.

Если бы вся требуемая информация передавалась каким-то абонентом сразу, непрерывно, без разделения на пакеты, то это привело бы к монопольному захвату сети этим абонентом на довольно продолжительное время. Все остальные абоненты вынуждены были бы ждать окончания передачи всей информации, что в ряде случаев могло бы потребовать десятков секунд и даже минут (например, при копировании содержимого целого жесткого диска). С тем чтобы уравнять в правах всех абонентов, а также сделать примерно одинаковыми для всех них величину времени доступа к сети и интегральную скорость передачи информации, как раз и применяются пакеты (кадры) ограниченной длины. Важно также и то, что при передаче больших массивов информации вероятность ошибки из-за помех и сбоев довольно высока. Например, при характерной для локальных сетей величине вероятности одиночной ошибки в 10^{-8}пакет длиной 10 Кбит будет искажен с вероятностью 10^{-4}, а массив длиной 10 Мбит – уже с вероятностью 10^{-1}. К тому же выявить ошибку в массиве из нескольких мегабайт намного сложнее, чем в пакете из нескольких килобайт. А при обнаружении ошибки придется повторить передачу всего большого массива. Но и при повторной передаче большого массива снова высока вероятность ошибки, и процесс этот при слишком большом массиве может повторяться до бесконечности.

С другой стороны, сравнительно большие пакеты имеют преимущества перед очень маленькими пакетами, например, перед побайтовой (8 бит) или пословной (16 бит или 32 бита) передачей информации.

Дело в том, что каждый пакет помимо собственно данных, которые требуется передать, должен содержать некоторое количество служебной информации. Прежде всего, это адресная информация, которая определяет, от кого и кому передается данный пакет (как на почтовом конверте – адреса получателя и

отправителя). Если порция передаваемых данных будет очень маленькой (например, несколько байт), то доля служебной информации станет непозволительно высокой, что резко снизит интегральную скорость обмена информацией по сети.

Существует некоторая оптимальная длина пакета (или оптимальный диапазон длин пакетов), при которой средняя скорость обмена информацией по сети будет максимальна. Эта длина не является неизменной величиной, она зависит от уровня помех, метода управления обменом, количества абонентов сети, характера передаваемой информации, и от многих других факторов. Имеется диапазон длин, который близок к оптимуму.

Таким образом, процесс информационного обмена в сети представляет собой чередование пакетов, каждый из которых содержит информацию, передаваемую от абонента к абоненту.

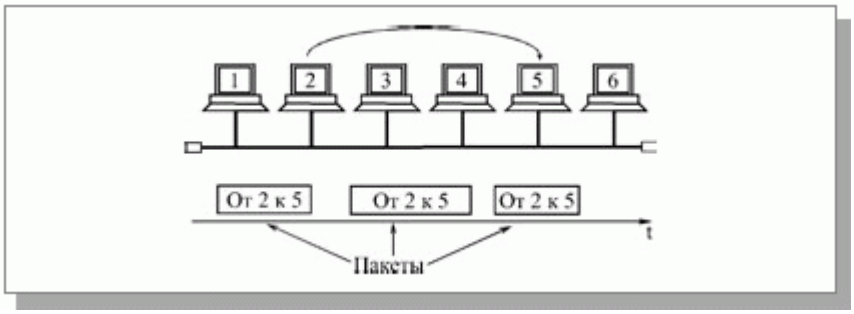

Рис. 4.1. Передача пакетов в сети между двумя абонентами

В частном случае (рис. 4.1) все эти пакеты могут передаваться одним абонентом (когда другие абоненты не хотят передавать). Но обычно в сети чередуются пакеты, посланные разными абонентами (рис. 4.2).

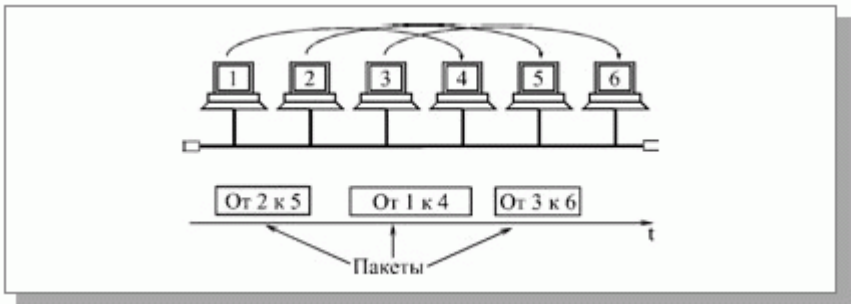

Рис. 4.2. Передача пакетов в сети между несколькими абонентами

Структура и размеры пакета в каждой сети жестко определены стандартом на данную сеть и связаны, прежде всего, с аппаратурными особенностями данной сети, выбранной топологией и типом среды передачи информации. Кроме того, эти параметры зависят от используемого протокола (порядка обмена информацией).

Но существуют некоторые общие принципы формирования структуры пакета, которые учитывают характерные особенности обмена информацией по любым локальным сетям.

Чаще всего пакет содержит в себе следующие основные поля или части (рис. 4.3):

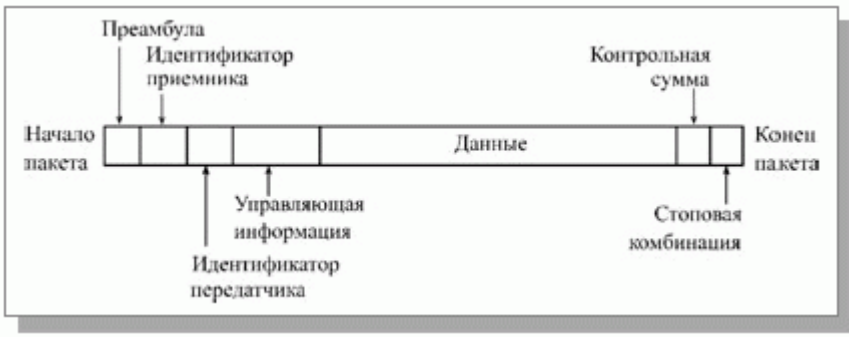

Рис. 4.3. Типичная структура пакета

- Стартовая комбинация битов или преамбула, которая обеспечивает предварительную настройку аппаратуры адаптера или другого сетевого устройства на прием и обработку пакета. Это поле может полностью отсутствовать или же сводиться к единственному стартовому биту.
- Сетевой адрес (идентификатор) принимающего абонента, то есть индивидуальный или групповой номер, присвоенный каждому принимающему абоненту в сети. Этот адрес позволяет приемнику распознать пакет, адресованный ему лично, группе, в которую он входит, или всем абонентам сети одновременно (при широком вещании).
- Сетевой адрес (идентификатор) передающего абонента, то есть индивидуальный номер, присвоенный каждому передающему абоненту. Этот адрес информирует принимающего абонента, откуда пришел данный пакет. Включение в пакет адреса передатчика необходимо в том случае, когда одному приемнику могут попеременно приходить пакеты от разных передатчиков.
- Служебная информация, которая может указывать на тип пакета, его номер, размер, формат, маршрут его доставки, на то, что с ним надо делать приемнику и т.д.
- Данные (поле данных) – это та информация, ради передачи которой используется пакет. В отличие от всех остальных полей пакета поле данных имеет переменную длину, которая, собственно, и определяет полную длину пакета. Существуют специальные управляющие пакеты, которые не имеют поля данных. Их можно рассматривать как сетевые команды. Пакеты, включающие поле данных, называются информационными пакетами. Управляющие пакеты могут выполнять функцию начала и конца сеанса связи, подтверждения приема информационного пакета, запроса информационного пакета и т.д.
- Контрольная сумма пакета – это числовой код, формируемый передатчиком по определенным правилам и содержащий в свернутом виде информацию обо всем пакете. Приемник, повторяя вычисления, сделанные передатчиком, с принятым пакетом, сравнивает их результат с контрольной суммой и делает вывод о правильности или ошибочности передачи пакета. Если пакет ошибочен, то приемник запрашивает его повторную передачу. Обычно используется циклическая контрольная сумма (CRC). Подробнее об этом рассказано в главе 7.
- Стоповая комбинация служит для информирования аппаратуры принимающего абонента об окончании пакета, обеспечивает выход аппаратуры приемника из состояния приема. Это поле может отсутствовать, если используется самосинхронизирующийся код, позволяющий определять момент окончания передачи пакета.

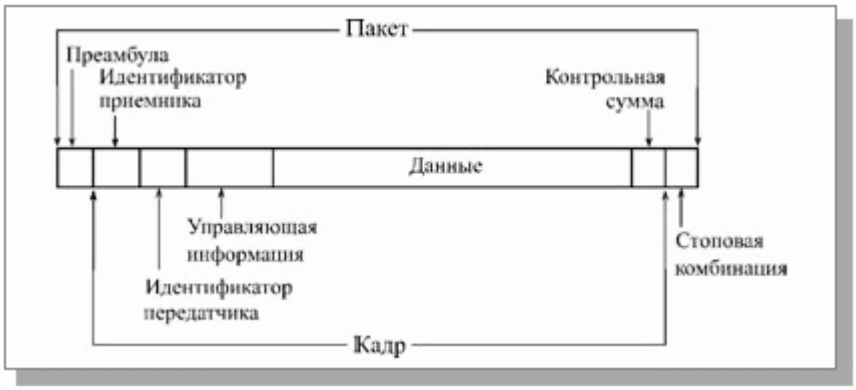

Рис. 4.4. Вложение кадра в пакет

Нередко в структуре пакета выделяют всего три поля:

- Начальное управляющее поле пакета (или заголовок пакета), то есть поле, включающее в себя стартовую комбинацию, сетевые адреса приемника и передатчика, а также служебную информацию.
- Поле данных пакета.
- Конечное управляющее поле пакета (заключение, трейлер), куда входят контрольная сумма и стоповая комбинация, а также, возможно, служебная информация.

Как уже упоминалось, помимо термина «пакет» (packet) в литературе также нередко встречается термин «кадр» (frame). Иногда под этими терминами имеется в виду одно и то же. Но иногда подразумевается, что кадр и пакет различаются. Причем единства в объяснении этих различий не наблюдается.

В некоторых источниках утверждается, что кадр вложен в пакет. В этом случае все перечисленные поля пакета кроме преамбулы и стоповой комбинации относятся к кадру (рис. 4.4). Например, в описаниях сети Ethernet говорится, что в конце преамбулы передается признак начала кадра.

В других, напротив, поддерживается мнение о том, что пакет вложен в кадр. И тогда под пакетом подразумевается только информация, содержащаяся в кадре, который передается по сети и снабжен служебными полями.

Во избежание путаницы, в данной книге термин «пакет» будет использоваться как более понятный и универсальный.

В процессе сеанса обмена информацией по сети между передающим и принимающим абонентами происходит обмен информационными и управляющими пакетами по установленным правилам, называемым протоколом обмена. Это позволяет обеспечить надежную передачу информации при любой интенсивности обмена по сети.

Пример простейшего протокола показан на рис. 4.5.

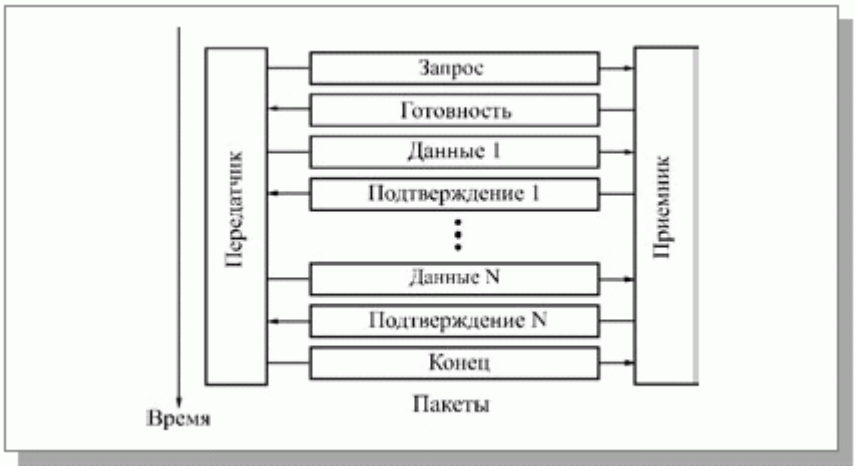

Рис. 4.5. Пример обмена пакетами при сеансе связи

Сеанс обмена начинается с запроса передатчиком готовности приемника принять данные. Для этого используется управляющий пакет «Запрос». Если приемник не готов, он отказывается от сеанса специальным управляющим пакетом. В случае, когда приемник готов, он посылает в ответ управляющий пакет «Готовность». Затем начинается собственно передача данных. При этом на каждый полученный информационный пакет приемник отвечает управляющим пакетом «Подтверждение». В случае, когда пакет данных передан с ошибками, в ответ на него приемник запрашивает повторную передачу. Заканчивается сеанс управляющим пакетом «Конец», которым передатчик сообщает о разрыве связи. Существует множество стандартных протоколов, которые используют как передачу с подтверждением (с гарантированной доставкой пакета), так и передачу без подтверждения (без гарантии доставки пакета). Подробнее о протоколах обмена будет рассказано в следующей главе.

При реальном обмене по сети применяются многоуровневые протоколы, каждый из уровней которых предполагает свою структуру пакета (адресацию, управляющую информацию, формат данных и т.д.). Ведь протоколы высоких уровней имеют дело с такими понятиями, как файл-сервер или приложение, запрашивающее данные у другого приложения, и вполне могут не иметь представления ни о типе аппаратуры сети, ни о методе управления обменом. Все пакеты более высоких уровней последовательно вкладываются в передаваемый пакет, точнее, в поле данных передаваемого пакета (рис. 4.5). Этот процесс последовательной упаковки данных для передачи называется также инкапсуляцией пакетов.

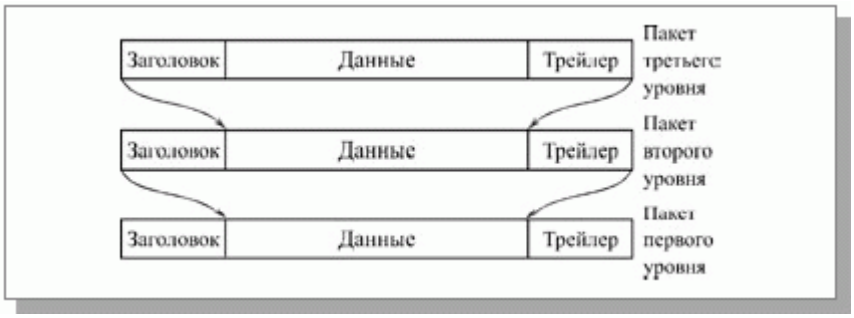

Рис. 4.6. Многоуровневая система вложения пакетов

Каждый следующий вкладываемый пакет может содержать собственную служебную информацию, располагающуюся как до данных (заголовок), так и после них (трейлер), причем ее назначение может быть различным. Безусловно, доля вспомогательной информации в пакетах при этом возрастает с каждым следующим уровнем, что снижает эффективную скорость передачи данных. Для увеличения этой скорости предпочтительнее, чтобы протоколы обмена были проще, и уровней этих протоколов было меньше. Иначе никакая скорость передачи битов не поможет, и быстрая сеть может передавать, файл дольше, чем медленная сеть, которая пользуется более простым протоколом.

Обратный процесс последовательной распаковки данных приемником называется декапсуляцией пакетов.

4.1 Адресация пакетов

Каждый абонент (узел) локальной сети должен иметь свой уникальный адрес (идентификатор или MAC-адрес), для того чтобы ему можно было адресовать пакеты. Существуют две основные системы присвоения адресов абонентам сети (точнее, сетевым адаптерам этих абонентов).

Первая система сводится к тому, что при установке сети каждому абоненту пользователь присваивает индивидуальный адрес по порядку, к примеру, от 0 до 30 или от 0 до 254. Присваивание адресов производится программно или с помощью переключателей на плате адаптера. При этом требуемое количество разрядов адреса определяется из неравенства:

$$2^n > N_{max}$$

где n – количество разрядов адреса, а N_{max} – максимально возможное количество абонентов в сети. Например, восемь разрядов адреса достаточно для сети из 255 абонентов. Один адрес (обычно `1111....11`) отводится для широковещательной передачи, то есть он используется для пакетов, адресованных всем абонентам одновременно.

Именно такой подход применен в известной сети Arcnet. Достоинства данного подхода – малый объем служебной информации в пакете, а также

простота аппаратуры адаптера, распознающей адрес пакета. Недостаток – трудоемкость задания адресов и возможность ошибки (например, двум абонентам сети может быть присвоен один и тот же адрес). Контроль уникальности сетевых адресов всех абонентов возлагается на администратора сети.

Второй подход к адресации был разработан международной организацией IEEE, занимающейся стандартизацией сетей. Именно он используется в большинстве сетей и рекомендован для новых разработок. Идея этого подхода состоит в том, чтобы присваивать уникальный сетевой адрес каждому адаптеру сети еще на этапе его изготовления. Если количество возможных адресов будет достаточно большим, то можно быть уверенным, что в любой сети по всему миру никогда не будет абонентов с одинаковыми адресами. Поэтому был выбран 48-битный формат адреса, что соответствует примерно 280 триллионам различных адресов. Понятно, что столько сетевых адаптеров никогда не будет выпущено.

С тем чтобы распределить возможные диапазоны адресов между многочисленными изготовителями сетевых адаптеров, была предложена следующая структура адреса (рис. 4.7):

- Младшие 24 разряда кода адреса называются OUA (Organizationally Unique Address) – организационно уникальный адрес. Именно их присваивает каждый из зарегистрированных производителей сетевых адаптеров. Всего возможно свыше 16 миллионов комбинаций, то есть каждый изготовитель может выпустить 16 миллионов сетевых адаптеров.
- Следующие 22 разряда кода называются OUI (Organizationally Unique Identifier) – организационно уникальный идентификатор. IEEE присваивает один или несколько OUI каждому производителю сетевых адаптеров. Это позволяет исключить совпадения адресов адаптеров от разных производителей. Всего возможно свыше 4 миллионов разных OUI, это означает, что теоретически может быть зарегистрировано 4 миллиона производителей. Вместе OUA и OUI называются UAA (Universally Administered Address) – универсально управляемый адрес или IEEE-адрес.
- Два старших разряда адреса управляющие, они определяют тип адреса, способ интерпретации остальных 46 разрядов. Старший бит I/G (Individual/Group) указывает на тип адреса. Если он установлен в 0, то индивидуальный, если в 1, то групповой (многопунктовый или функциональный). Пакеты с групповым адресом получат все имеющие этот групповой адрес сетевые адаптеры. Причем групповой адрес определяется 46 младшими разрядами. Второй управляющий бит U/L (Universal/Local) называется флажком универсального/местного управления и определяет, как был присвоен адрес данному сетевому адаптеру. Обычно он установлен в 0. Установка бита U/L в 1 означает, что адрес задан не производителем сетевого адаптера, а организацией, использующей данную сеть. Это случается довольно редко.

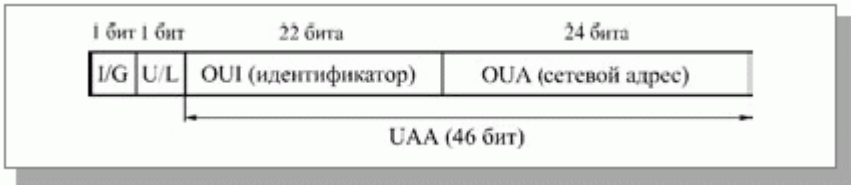

Рис. 4.7. Структура 48-битного стандартного MAC-адреса

Для широковещательной передачи (то есть передачи всем абонентам сети одновременно) применяется специально выделенный сетевой адрес, все 48 битов которого установлены в единицу. Его принимают все абоненты сети независимо от их индивидуальных и групповых адресов.

Данной системы адресов придерживаются такие популярные сети, как Ethernet, Fast Ethernet, Token-Ring, FDDI, 100VG-AnyLAN. Ее недостатки – высокая сложность аппаратуры сетевых адаптеров, а также большая доля служебной информации в передаваемом пакете (адреса источника и приемника вместе требуют уже 96 битов пакета или 12 байт).

Во многих сетевых адаптерах предусмотрен так называемый циркулярный режим. В этом режиме адаптер принимает все пакеты, приходящие к нему, независимо от значения поля адреса приемника. Такой режим используется, например, для проведения диагностики сети, измерения ее производительности, контроля ошибок передачи. При этом один компьютер принимает и контролирует все пакеты, проходящие по сети, но сам ничего не передает. В данном режиме работают сетевые адаптеры мостов и коммутаторы, которые должны обрабатывать перед ретрансляцией все пакеты, приходящие к ним.

4.2 Методы управления обменом

Сеть всегда объединяет несколько абонентов, каждый из которых имеет право передавать свои пакеты. Но, как уже отмечалось, по одному кабелю одновременно передавать два (или более) пакета нельзя, иначе может возникнуть конфликт (коллизия) который приведет к искажению либо потере обоих пакетов (или всех пакетов, участвующих в конфликте). Значит, надо каким-то образом установить очередность доступа к сети (захвата сети) всеми абонентами, желающими передавать. Это относится, прежде всего, к сетям с топологиями шина и кольцо. Точно так же при топологии звезда необходимо установить очередность передачи пакетов периферийными абонентами, иначе центральный абонент просто не сможет справиться с их обработкой.

В сети обязательно применяется тот или иной метод управления обменом (метод доступа, метод арбитража), разрешающий или предотвращающий конфликты между абонентами. От эффективности работы выбранного метода управления обменом зависит очень многое: скорость обмена информацией между компьютерами, нагрузочная способность сети (способность работать с различными интенсивностями обмена), время реакции сети на внешние события и т.д. Метод управления – это один из важнейших параметров сети.

Тип метода управления обменом во многом определяется особенностями топологии сети. Но в то же время он не привязан жестко к топологии, как нередко принято считать.

Методы управления обменом в локальных сетях делятся на две группы:

- Централизованные методы, в которых все управление обменом сосредоточенно в одном месте. Недостатки таких методов: неустойчивость к отказам центра, малая гибкость управления (центр обычно не может оперативно реагировать на все события в сети). Достоинство централизованных методов – отсутствие конфликтов, так как центр всегда предоставляет право на передачу только одному абоненту, и ему не с кем конфликтовать.
- Децентрализованные методы, в которых отсутствует центр управления. Всеми вопросами управления, в том числе предотвращением, обнаружением и разрешением конфликтов, занимаются все абоненты сети. Главные достоинства децентрализованных методов: высокая устойчивость к отказам и большая гибкость. Однако в данном случае возможны конфликты, которые надо разрешать.

Существует и другое деление методов управления обменом, относящееся, главным образом, к децентрализованным методам:

- Детерминированные методы определяют четкие правила, по которым чередуются захватывающие сеть абоненты. Абоненты имеют определенную систему приоритетов, причем приоритеты эти различны для всех абонентов. При этом, как правило, конфликты полностью исключены (или маловероятны), но некоторые абоненты могут дожидаться своей очереди на передачу слишком долго. К детерминированным методам относится, например, маркерный доступ (сети Token-Ring, FDDI), при котором право передачи передается по эстафете от абонента к абоненту.
- Случайные методыподразумевают случайное чередование передающих абонентов. При этом возможность конфликтов подразумевается, но предлагаются способы их разрешения. Случайные методы значительно хуже, чем детерминированные, работают при больших информационных потоках в сети (при большом трафике сети) и не гарантируют абоненту величину времени доступа. В то же время они обычно более устойчивы к отказам сетевого оборудования и более эффективно используют сеть при малой интенсивности обмена. Пример случайного метода – CSMA/CD (сеть Ethernet).

Для трех основных топологий характерны три наиболее типичных метода управления обменом.

Управление обменом в сети с топологией звезда

Для топологии звезда лучше всего подходит централизованный метод управления. Это связано с тем, что все информационные потоки проходят через центр, и именно этому центру логично доверить управление обменом в сети. Причем не так важно, что находится в центре звезды: компьютер (центральный абонент), как на рис. 4.6, или же специальный концентратор, управляющий обменом, но сам не участвующий в нем. В данном случае речь идет уже не о пассивной звезде (рис. 4.11), а о некой промежуточной ситуации, когда центр не является полноценным абонентом, но управляет обменом. Это, к примеру, реализовано в сети 100VG AnyLAN.

Самый простейший централизованный метод состоит в следующем.

Периферийные абоненты, желающие передать свой пакет (или, как еще говорят, имеющие заявки на передачу), посылают центру свои запросы (управляющие пакеты или специальные сигналы). Центр же предоставляет им право передачи пакета в порядке очередности, например, по их физическому расположению в звезде по часовой стрелке. После окончания передачи пакета каким-то абонентом право передавать получит следующий по порядку (по часовой стрелке) абонент, имеющий заявку на передачу (рис. 4.8). Например, если передает второй абонент, то после него имеет право на передачу третий. Если же третьему абоненту не надо передавать, то право на передачу переходит к четвертому и т.д.

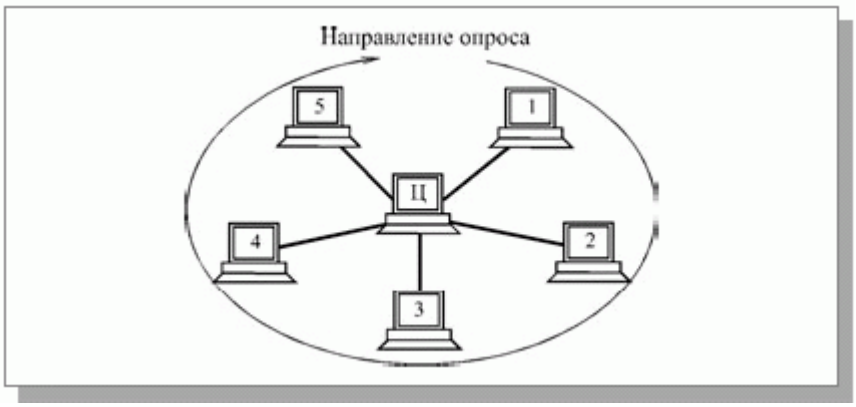

Рис. 4.8. Централизованный метод управления обменом в сети с топологией звезда

В этом случае говорят, что абоненты имеют географические приоритеты (по их физическому расположению). В каждый конкретный момент наивысшим приоритетом обладает следующий по порядку абонент, но в пределах полного цикла опроса ни один из абонентов не имеет никаких преимуществ перед другими. Никому не придется ждать своей очереди слишком долго. Максимальная величина времени доступа для любого абонента в этом случае будет равна суммарному времени передачи пакетов всех абонентов сети кроме данного. Для топологии, показанной на рис. 4.8, она составит четыре длительности пакета. Никаких столкновений пакетов при этом методе в принципе быть не может, так как все решения о доступе принимаются в одном месте.

Рассмотренный метод управления можно назвать методом с пассивным центром, так как центр пассивно прослушивает всех абонентов. Возможен и другой принцип реализации централизованного управления (его можно назвать методом с активным центром).

В этом случае центр посылает запросы о готовности передавать (управляющие пакеты или специальные сигналы) по очереди всем периферийным абонентам. Тот периферийный абонент, который хочет

передавать (первый из опрошенных) посылает ответ (или же сразу начинает свою передачу). В дальнейшем центр проводит сеанс обмена именно с ним. После окончания этого сеанса центральный абонент продолжает опрос периферийных абонентов по кругу (как на рис. 4.8). Если желает передавать центральный абонент, он передает вне очереди.

Как в первом, так и во втором случае никаких конфликтов быть не может (решение принимает единый центр, которому не с кем конфликтовать). Если все абоненты активны, и заявки на передачу поступают интенсивно, то все они будут передавать строго по очереди. Но центр должен быть исключительно надежен, иначе будет парализован весь обмен. Механизм управления не слишком гибок, так как центр работает по жестко заданному алгоритму. К тому же скорость управления невысока. Ведь даже в случае, когда передает только один абонент, ему все равно приходится ждать после каждого переданного пакета, пока центр опросит всех остальных абонентов.

Как правило, централизованные методы управления применяются в небольших сетях (с числом абонентов не более чем несколько десятков). В случае больших сетей нагрузка по управлению обменом на центр существенно возрастает.

Управление обменом в сети с топологией шина

При топологии шина также возможно централизованное управление. При этом один из абонентов («центральный») посылает по шине всем остальным («периферийным») запросы (управляющие пакеты), выясняя, кто из них хочет передать, затем разрешает передачу одному из абонентов. Абонент, получивший право на передачу, по той же шине передает свой информационный пакет тому абоненту, которому хочет. А после окончания передачи передававший абонент все по той же шине сообщает «центру», что он закончил передачу (управляющим пакетом), и «центр» снова начинает опрос (рис. 4.9).

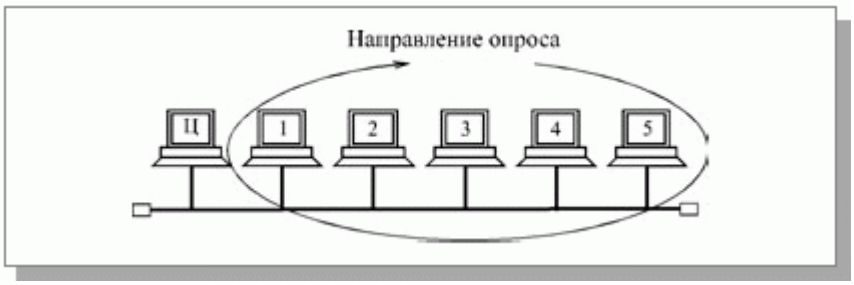

Рис. 4.9. Централизованное управление в сети с топологией шина

Преимущества и недостатки такого управления – те же самые, что и в случае централизованно управляемой звезды. Единственное отличие состоит в том, что центр здесь не пересылает информацию от одного абонента к другому, как в топологии активная звезда, а только управляет обменом.

Гораздо чаще в шине используется децентрализованное случайное управление, так как сетевые адаптеры всех абонентов в данном случае одинаковы, и именно этот метод наиболее органично подходит шине. При выборе децентрализованного управления все абоненты имеют равные права доступа к сети, то есть особенности топологии совпадают с особенностями метода управления. Решение о том, когда можно передавать свой пакет, принимается каждым абонентом на месте, исходя только из анализа состояния сети. В данном случае возникает конкуренция между абонентами за захват сети, и, следовательно, возможны конфликты между ними и искажения передаваемой информации из-за наложения пакетов.

Существует множество алгоритмов доступа или, как еще говорят, сценариев доступа, порой очень сложных. Их выбор зависит от скорости передачи в сети, длины шины, загруженности сети (интенсивности обмена или трафика сети), используемого кода передачи.

Иногда для управления доступом к шине применяется дополнительная линия связи, что позволяет упростить аппаратуру контроллеров и методы доступа, но заметно увеличивает стоимость сети за счет удвоения длины кабеля и количества приемопередатчиков. Поэтому данное решение не получило широкого распространения.

Суть всех случайных методов управления обменом довольно проста.

Если сеть свободна (то есть никто не передает своих пакетов), то абонент, желающий передавать, сразу начинает свою передачу. Время доступа в этом случае равно нулю.

Если же в момент возникновения у абонента заявки на передачу сеть занята, то абонент, желающий передавать, ждет освобождения сети. В противном случае исказятся и пропадут оба пакета. После освобождения сети абонент, желающий передавать, начинает свою передачу.

Возникновение конфликтных ситуаций (столкновений пакетов, коллизий), в результате которых передаваемая информация искажается, возможно в двух случаях.

- При одновременном начале передачи двумя или более абонентами, когда сеть свободна (рис. 4.10). Это ситуация довольно редкая, но все-таки вполне возможная.
- При одновременном начале передачи двумя или более абонентами сразу после освобождения сети (рис. 4.11). Это ситуация наиболее типична, так как за время передачи пакета одним абонентом вполне может возникнуть несколько новых заявок на передачу у других абонентов.

Существующие случайные методы управления обменом (арбитража) различаются тем, как они предотвращают возможные конфликты или же

разрешают уже возникшие. Ни один конфликт не должен нарушать обмен, все абоненты должны, в конце концов, передать свои пакеты.

В процессе развития локальных сетей было разработано несколько разновидностей случайных методов управления обменом.

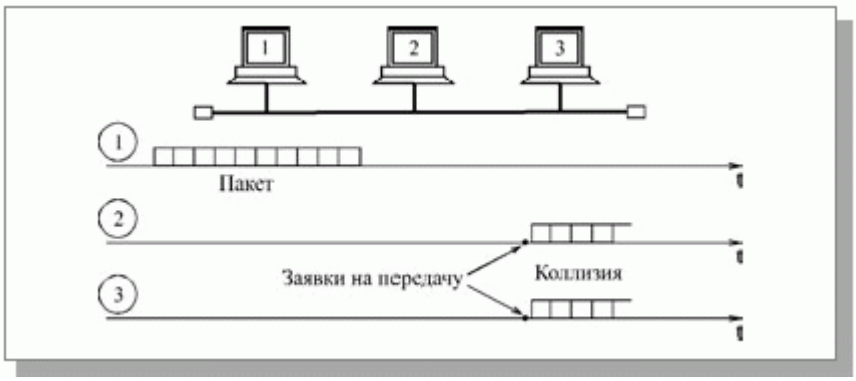

Рис. 4.10. Коллизии в случае начала передачи при свободной сети

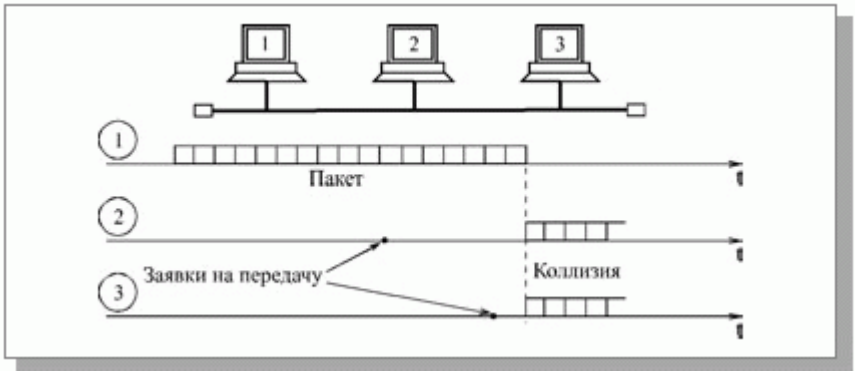

Рис. 4.11. Коллизии в случае начала передачи после освобождения сети

Например, был предложен метод, при котором не все передающие абоненты распознают коллизию, а только те, которые имеют меньшие приоритеты. Абонент с максимальным приоритетом из всех, начавших передачу, закончит передачу своего пакета без ошибок. Остальные, обнаружив коллизию, прекратят свою передачу и будут ждать освобождения сети для новой попытки. Для контроля коллизии каждый передающий абонент производит побитное сравнение передаваемой им в сеть информации и данных, присутствующих в сети. Побеждает тот абонент, заголовок пакета которого дольше других не искажается от коллизии. Этот метод, называемый децентрализованным кодовым приоритетным методом, отличается низким быстродействием и сложностью реализации.

При другом методе управления обменом каждый абонент начинает свою передачу после освобождения сети не сразу, а, выдержав свою, строго индивидуальную задержку, что предотвращает коллизии после освобождения сети и тем самым сводит к минимуму общее количество коллизий. Максимальным приоритетом в этом случае будет обладать абонент с минимальной задержкой. Столкновения пакетов возможны только тогда,

когда два и более абонентов захотели передавать одновременно при свободной сети. Этот метод, называемый децентрализованным временным приоритетным методом, хорошо работает только в небольших сетях, так как каждому абоненту нужно обеспечить свою индивидуальную задержку.

В обоих случаях имеется система приоритетов, все же данные методы относятся к случайным, так как исход конкуренции невозможно предсказать. Случайные приоритетные методы ставят абонентов в неравные условия при большой интенсивности обмена по сети, так как высокоприоритетные абоненты могут надолго заблокировать сеть для низкоприоритетных абонентов.

Чаще всего система приоритетов в методе управления обменом в шине отсутствует полностью. Именно так работает наиболее распространенный стандартный метод управления обменом CSMA/CD (Carrier Sense Multiple Access with Collision Detection – множественный доступ с контролем несущей и обнаружением коллизий), используемый в сети Ethernet. Его главное достоинство в том, что все абоненты полностью равноправны, и ни один из них не может надолго заблокировать обмен другому (как в случае наличия приоритетов). В этом методе коллизии не предотвращаются, а разрешаются.

Суть метода состоит в том, что абонент начинает передавать сразу, как только он выяснит, что сеть свободна. Если возникают коллизии, то они обнаруживаются всеми передающими абонентами. После чего все абоненты прекращают свою передачу и возобновляют попытку начать новую передачу пакета через временной интервал, длительность которого выбирается случайным образом. Поэтому повторные коллизии мало вероятны.

Еще один распространенный метод случайного доступа – CSMA/CA (Carrier Sense Multiple Access with Collision Avoidance – множественный доступ с контролем несущей и избежанием коллизий) применяющийся, например, в сети Apple LocalTalk. Абонент, желающий передавать и обнаруживший освобождение сети, передает сначала короткий управляющий пакет запроса на передачу. Затем он заданное время ждет ответного короткого управляющего пакета подтверждения запроса от абонента-приемника. Если ответа нет, передача откладывается. Если ответ получен, передается пакет. Коллизии полностью не устраняются, но в основном сталкиваются управляющие пакеты. Столкновения информационных пакетов выявляются на более высоких уровнях протокола.

Подобные методы будут хорошо работать только при не слишком большой интенсивности обмена по сети. Считается, что приемлемое качество связи обеспечивается при нагрузке не выше 30—40% (то есть когда сеть занята передачей информации примерно на 30—40% всего времени). При большей нагрузке повторные столкновения учащаются настолько, что наступает так

называемый коллапс или крах сети, представляющий собой резкое падение ее производительности.

Недостаток всех случайных методов состоит еще и в том, что они не гарантируют величину времени доступа к сети, которая зависит не только от выбора задержки между попытками передачи, но и от общей загруженности сети. Поэтому, например, в сетях, выполняющих задачи управления оборудованием (на производстве, в научных лабораториях), где требуется быстрая реакция на внешние события, сети со случайными методами управления используются довольно редко.

При любом случайном методе управления обменом, использующем детектирование коллизии (в частности, при CSMA/CD), возникает вопрос о том, какой должна быть минимальная длительность пакета, чтобы коллизию обнаружили все начавшие передавать абоненты. Ведь сигнал по любой физической среде распространяется не мгновенно, и при больших размерах сети (диаметре сети) задержка распространения может составлять десятки и сотни микросекунд. Кроме того, информацию об одновременно происходящих событиях разные абоненты получают не в одно время. С тем чтобы рассчитать минимальную длительность пакета, следует обратиться к рис. 4.12.

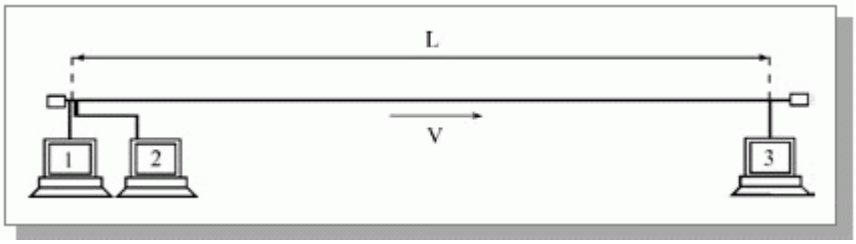

Рис. 4.12. Расчет минимальной длительности пакета

Пусть `L` – полная длина сети, `V` – скорость распространения сигнала в используемом кабеле. Допустим, абонент `1` закончил свою передачу, а абоненты `2` и `3` захотели передавать во время передачи абонента `1` и ждали освобождения сети.

После освобождения сети абонент `2` начнет передавать сразу же, так как он расположен рядом с абонентом `1`. Абонент `3` после освобождения сети узнает об этом событии и начнет свою передачу через временной интервал прохождения сигнала по всей длине сети, то есть через время `L/V`. При этом пакет от абонента `3` дойдет до абонента `2` еще через временной интервал `L/V` после начала передачи абонентом `3` (обратный путь сигнала). К этому моменту передача пакета абонентом `2` не должна закончиться, иначе абонент `2` так и не узнает о столкновении пакетов (о коллизии), в результате чего будет передан неправильный пакет.

Получается, что минимально допустимая длительность пакета в сети должна составлять $2L/V$, то есть равняться удвоенному времени распространения сигнала по полной длине сети (или по пути наибольшей длины в сети). Это время называется двойным или круговым временем задержки сигнала в сети или PDV (Path Delay Value). Этот же временной интервал можно рассматривать как универсальную меру одновременности любых событий в сети.

Стандартом на сеть задается как раз величина PDV, определяющая минимальную длину пакета, и из нее уже рассчитывается допустимая длина сети. Дело в том, что скорость распространения сигнала в сети для разных кабелей отличается. Кроме того, надо еще учитывать задержки сигнала в различных сетевых устройствах.

Отдельно следует остановиться на том, как сетевые адаптеры распознают коллизию в кабеле шины, то есть столкновение пакетов. Ведь простое побитное сравнение передаваемой абонентом информации с той, которая реально присутствует в сети, возможно только в случае самого простого кода NRZ, используемого довольно редко. При применении манчестерского кода, который обычно подразумевается в случае метода управления обменом CSMA/CD, требуется принципиально другой подход.

Как уже отмечалось, сигнал в манчестерском коде всегда имеет постоянную составляющую, равную половине размаха сигнала (если один из двух уровней сигнала нулевой). Однако в случае столкновения двух и более пакетов (при коллизии) это правило выполняться не будет. Постоянная составляющая суммарного сигнала в сети будет обязательно больше или меньше половины размаха (рис. 4.13). Ведь пакеты всегда отличаются друг от друга и к тому же сдвинуты друг относительно друга во времени. Именно по выходу уровня постоянной составляющей за установленные пределы и определяет каждый сетевой адаптер наличие коллизии в сети.

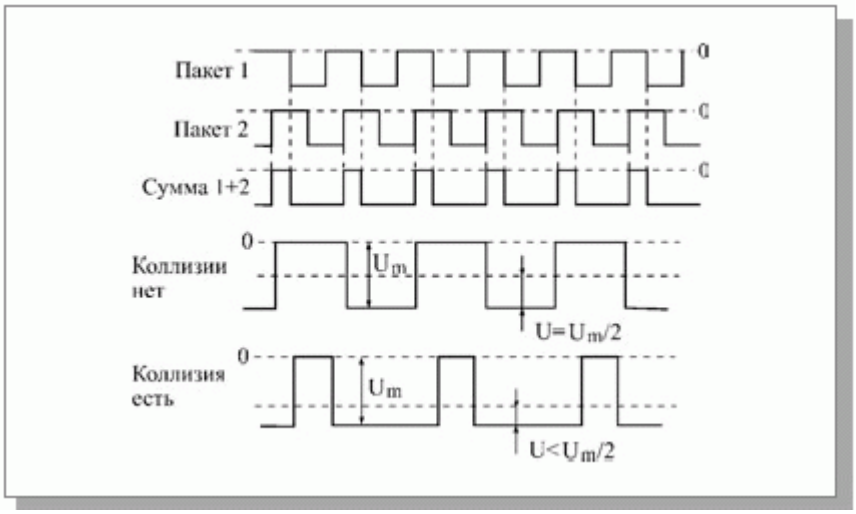

Рис. 4.13. Определение факта коллизии в шине при использовании манчестерского кода

Задача обнаружения коллизии существенно упрощается, если используется не истинная шина, а равноценная ей пассивная звезда (рис. 4.14).

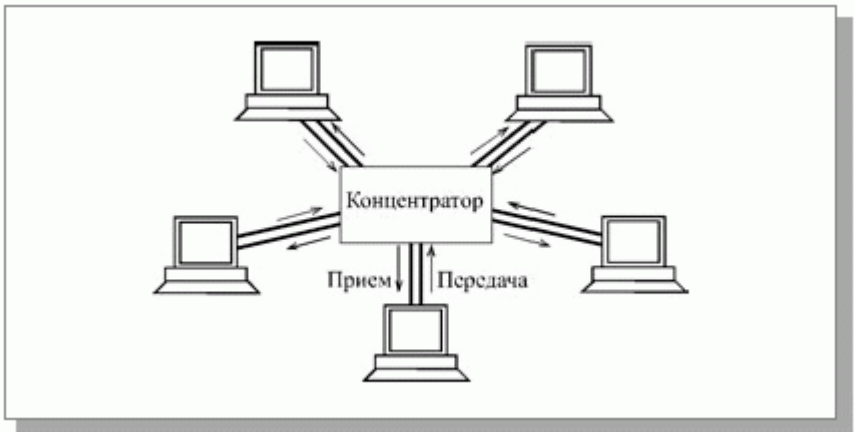

Рис. 4.14. Обнаружение коллизии в сети пассивная звезда

При этом каждый абонент соединяется с центральным концентратором, как правило, двумя кабелями, каждый из которых передает информацию в своем направлении. Во время передачи своего пакета абоненту достаточно всего лишь контролировать, не приходит ли ему в данный момент по встречному кабелю (приемному) другой пакет. Если встречный пакет приходит, то детектируется коллизия. Точно так же обнаруживает коллизии и концентратор.

Управление обменом в сети с топологией кольцо

Кольцевая топология имеет свои особенности при выборе метода управления обменом. В этом случае важно то, что любой пакет, посланный по кольцу, последовательно пройдя всех абонентов, через некоторое время возвратится в ту же точку, к тому же абоненту, который его передавал (так как топология замкнутая). Здесь нет одновременного распространения сигнала в две стороны, как в топологии шина. Как уже отмечалось, сети с топологией

кольцо бывают однонаправленными и двунаправленными. Наиболее распространены однонаправленные.

В сети с топологией кольцо можно использовать различные централизованные методы управления (как в звезде), а также методы случайного доступа (как в шине), но чаще выбирают все-таки специфические методы управления, в наибольшей степени соответствующие особенностям кольца.

Самые популярные методы управления в кольцевых сетях маркерные (эстафетные), те, которые используют маркер (эстафету) – небольшой управляющий пакет специального вида. Именно эстафетная передача маркера по кольцу позволяет передавать право на захват сети от одного абонента к другому. Маркерные методы относятся к децентрализованным и детерминированным методам управления обменом в сети. В них нет явно выраженного центра, но существует четкая система приоритетов, и потому не бывает конфликтов.

Работа маркерного метода управления в сети с топологией кольцо представлена на рис. 4.15.

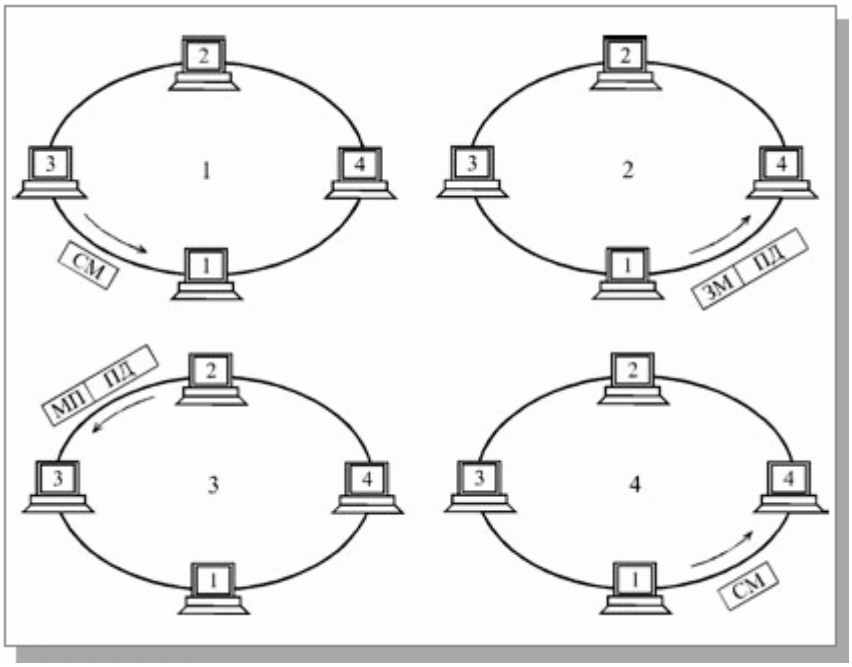

Рис. 4.15. Маркерный метод управления обменом (СМ—свободный маркер, ЗМ— занятый маркер, МП— занятый маркер с подтверждением, ПД—пакет данных)

По кольцу непрерывно ходит специальный управляющий пакет минимальной длины, маркер, предоставляющий абонентам право передавать свой пакет. Алгоритм действий абонентов:

1. Абонент 1, желающий передать свой пакет, должен дождаться прихода к нему свободного маркера. Затем он присоединяет к маркеру свой пакет, помечает маркер как занятый и отправляет эту посылку следующему по кольцу абоненту.
2. Все остальные абоненты (2, 3, 4), получив маркер с присоединенным пакетом, проверяют, им ли адресован пакет. Если пакет адресован не им, то они передают полученную посылку (маркер + пакет) дальше по кольцу.
3. Если какой-то абонент (в данном случае это абонент 3) распознает пакет как адресованный ему, то он его принимает, устанавливает в маркере бит подтверждения приема и передает посылку (маркер + пакет) дальше по кольцу.
4. Передававший абонент 1 получает свою посылку, прошедшую по всему кольцу, обратно, помечает маркер как свободный, удаляет из сети свой пакет и посылает свободный маркер дальше по кольцу. Абонент, желающий передавать, ждет этого маркера, и все повторяется снова.

Приоритет при данном методе управления получается географический, то есть право передачи после освобождения сети переходит к следующему по направлению кольца абоненту от последнего передававшего абонента. Но эта система приоритетов работает только при большой интенсивности обмена. При малой интенсивности обмена все абоненты равноправны, и время доступа к сети каждого из них определяется только положением маркера в момент возникновения заявки на передачу.

В чем-то рассматриваемый метод похож на метод опроса (централизованный), хотя явно выделенного центра здесь не существует. Однако некий центр обычно все-таки присутствует. Один из абонентов (или специальное устройство) должен следить, чтобы маркер не потерялся в процессе прохождения по кольцу (например, из-за действия помех или сбоя в работе какого-то абонента, а также из-за подключения и отключения абонентов). В противном случае механизм доступа работать не будет. Следовательно, надежность управления в данном случае снижается (выход центра из строя приводит к полной дезорганизации обмена). Существуют специальные средства для повышения надежности и восстановления центра контроля маркера.

Основное преимущество маркерного метода перед CSMA/CD состоит в гарантированной величине времени доступа. Его максимальная величина, как и при централизованном методе, составит $(N-1) \cdot t_{пк}$, где N – полное число абонентов в сети, $t_{пк}$ – время прохождения пакета по кольцу. Вообще, маркерный метод управления обменом при большой интенсивности обмена в сети (загруженность более 30—40%) гораздо эффективнее случайных методов. Он позволяет сети работать с большей нагрузкой, которая теоретически может даже приближаться к 100%.

Метод маркерного доступа используется не только в кольце (например, в сети IBM Token Ring или FDDI), но и в шине (в частности, сеть Arcnet-BUS), а также в пассивной звезде (к примеру, сеть Arcnet-STAR). В этих случаях реализуется не физическое, а логическое кольцо, то есть все абоненты

последовательно передают друг другу маркер, и эта цепочка передачи маркеров замкнута в кольцо (рис. 4.16). При этом совмещаются достоинства физической топологии шина и маркерного метода управления.

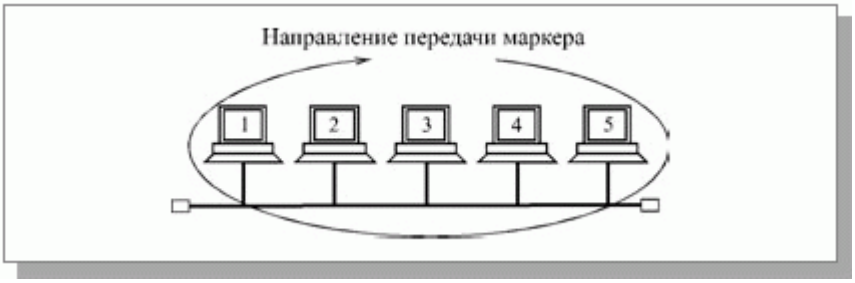

Рис. 4.16. Применение маркерного метода управления в шине

5.1 Эталонная модель OSI

Модель OSI была предложена Международной организацией стандартов ISO (International Standards Organization) в 1984 году. С тех пор ее используют (более или менее строго) все производители сетевых продуктов. Как и любая универсальная модель, OSI довольно громоздка, избыточна, и не слишком гибка. Поэтому реальные сетевые средства, предлагаемые различными фирмами, не обязательно придерживаются принятого разделения функций. Однако знакомство с моделью OSI позволяет лучше понять, что же происходит в сети.

Все сетевые функции в модели разделены на 7 уровней (рис. 5.1). При этом вышестоящие уровни выполняют более сложные, глобальные задачи, для чего используют в своих целях нижестоящие уровни, а также управляют ими. Цель нижестоящего уровня – предоставление услуг вышестоящему уровню, причем вышестоящему уровню не важны детали выполнения этих услуг. Нижестоящие уровни выполняют более простые и конкретные функции. В идеале каждый уровень взаимодействует только с теми, которые находятся рядом с ним (выше и ниже него). Верхний уровень соответствует прикладной задаче, работающему в данный момент приложению, нижний – непосредственной передаче сигналов по каналу связи.

Модель OSI относится не только к локальным сетям, но и к любым сетям связи между компьютерами или другими абонентами. В частности, функции сети Интернет также можно поделить на уровни в соответствии с моделью OSI. Принципиальные отличия локальных сетей от глобальных, с точки зрения модели OSI, наблюдаются только на нижних уровнях модели.

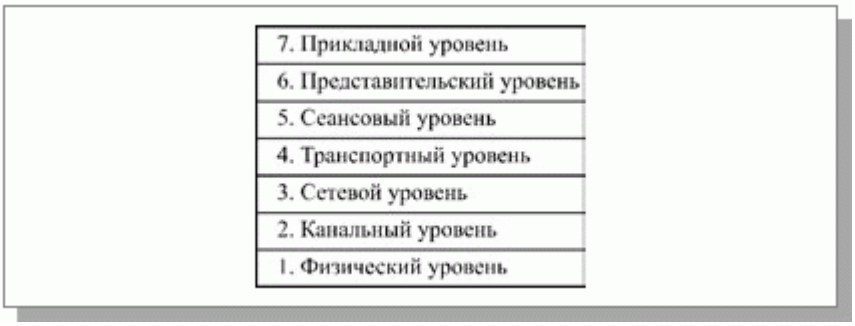

Рис. 5.1. Семь уровней модели OSI

Функции, входящие в показанные на рис. 5.1 уровни, реализуются каждым абонентом сети. При этом каждый уровень на одном абоненте работает так, как будто он имеет прямую связь с соответствующим уровнем другого абонента. Между одноименными уровнями абонентов сети существует виртуальная (логическая) связь, например, между прикладными уровнями взаимодействующих по сети абонентов. Реальную же, физическую связь (кабель, радиоканал) абоненты одной сети имеют только на самом нижнем, первом, физическом уровне. В передающем абоненте информация проходит все уровни, начиная с верхнего и заканчивая нижним. В принимающем абоненте полученная информация совершает обратный путь: от нижнего уровня к верхнему (рис. 5.2).

Рис. 5.2. Путь информации от абонента к абоненту

Данные, которые необходимо передать по сети, на пути от верхнего (седьмого) уровня до нижнего (первого) проходят процесс инкапсуляции (см. рис. 4.6). Каждый нижеследующий уровень не только производит обработку данных, приходящих с более высокого уровня, но и снабжает их своим заголовком, а также служебной информацией. Такой процесс обрастания служебной информацией продолжается до последнего (физического) уровня. На физическом уровне вся эта многооболочечная конструкция передается по кабелю приемнику. Там она проделывает обратную процедуру декапсуляции, то есть при передаче на вышестоящий уровень убирается одна из оболочек. Верхнего седьмого уровня достигают уже данные, освобожденные от всех оболочек, то есть от всей служебной информации нижестоящих уровней. При этом каждый уровень принимающего абонента производит обработку

данных, полученных с нижеследующего уровня в соответствии с убираемой им служебной информацией.

Если на пути между абонентами в сети включаются некие промежуточные устройства (например, трансиверы, репитеры, концентраторы, коммутаторы, маршрутизаторы), то и они тоже могут выполнять функции, входящие в нижние уровни модели OSI. Чем больше сложность промежуточного устройства, тем больше уровней оно захватывает. Но любое промежуточное устройство должно принимать и возвращать информацию на нижнем, физическом уровне. Все внутренние преобразования данных должны производиться дважды и в противоположных направлениях (рис. 5.3). Промежуточные сетевые устройства в отличие от полноценных абонентов (например, компьютеров) работают только на нижних уровнях и к тому же выполняют двустороннее преобразование.

Рис. 5.3. Включение промежуточных устройств между абонентами сети

Рассмотрим подробнее функции разных уровней.

- Прикладной (7) уровень (Application Layer) или уровень приложений обеспечивает услуги, непосредственно поддерживающие приложения пользователя, например, программные средства передачи файлов, доступа к базам данных, средства электронной почты, службу регистрации на сервере. Этот уровень управляет всеми остальными шестью уровнями. Например, если пользователь работает с электронными таблицами Excel и решает сохранить рабочий файл в своей директории на сетевом файл-сервере, то прикладной уровень обеспечивает перемещение файла с рабочего компьютера на сетевой диск прозрачно для пользователя.
- Представительский (6) уровень (Presentation Layer) или уровень представления данных определяет и преобразует форматы данных и их синтаксис в форму, удобную для сети, то есть выполняет функцию переводчика. Здесь же производится шифрование и дешифрирование данных, а при необходимости – и их сжатие. Стандартные форматы существуют для текстовых файлов (ASCII, EBCDIC, HTML), звуковых файлов (MIDI, MPEG, WAV), рисунков (JPEG. GIF, TIFF), видео (AVI). Все преобразования форматов делаются на представительском уровне. Если данные передаются в виде двоичного кода, то преобразования формата не требуется.
- Сеансовый (5) уровень (Session Layer) управляет проведением сеансов связи (то есть устанавливает, поддерживает и прекращает связь). Этот уровень предусматривает три режима установки сеансов: симплексный (передача данных в одном направлении), полудуплексный (передача данных поочередно в двух направлениях) и полнодуплексный (передача данных одновременно в двух

направлениях). Сеансовый уровень может также вставлять в поток данных специальные контрольные точки, которые позволяют контролировать процесс передачи при разрыве связи. Он же позволяет распознает логические имена абонентов, контролирует предоставленные им права доступа.

- Транспортный (4) уровень (Transport Layer) обеспечивает доставку пакетов без ошибок и потерь, а также в нужной последовательности. Здесь же производится разбивка на блоки передаваемых данных, помещаемые в пакеты, и восстановление принимаемых данных из пакетов. Доставка пакетов возможна как с установлением соединения (виртуального канала), так и без. Транспортный уровень является пограничным и связующим между верхними тремя, сильно зависящими от приложений, и тремя нижними уровнями, сильно привязанными к конкретной сети.
- Сетевой (3) уровень (Network Layer) отвечает за адресацию пакетов и перевод логических имен (логических адресов, например, IP-адресов или IPX-адресов) в физические сетевые MAC-адреса (и обратно). На этом же уровне решается задача выбора маршрута (пути), по которому пакет доставляется по назначению (если в сети имеется несколько маршрутов). На сетевом уровне действуют такие сложные промежуточные сетевые устройства, как маршрутизаторы.
- Канальный (2) уровень или уровень управления линией передачи (Data link Layer) отвечает за формирование пакетов (кадров) стандартного для данной сети (Ethernet, Token-Ring, FDDI) вида, включающих начальное и конечное управляющие поля. Здесь же производится управление доступом к сети, обнаруживаются ошибки передачи путем подсчета контрольных сумм, и производится повторная пересылка приемнику ошибочных пакетов. Канальный уровень делится на два подуровня: верхний LLC и нижний MAC. На канальном уровне работают такие промежуточные сетевые устройства, как, например, коммутаторы.
- Физический (1) уровень (Physical Layer) – это самый нижний уровень модели, который отвечает за кодирование передаваемой информации в уровни сигналов, принятые в используемой среде передачи, и обратное декодирование. Здесь же определяются требования к соединителям, разъемам, электрическому согласованию, заземлению, защите от помех и т.д. На физическом уровне работают такие сетевые устройства, как трансиверы, репитеры и репитерные концентраторы.

Большинство функций двух нижних уровней модели (1 и 2) обычно реализуются аппаратно (часть функций уровня 2 – программным драйвером сетевого адаптера). Именно на этих уровнях определяется скорость передачи и топология сети, метод управления обменом и формат пакета, то есть то, что имеет непосредственное отношение к типу сети, например, Ethernet, Token-Ring, FDDI, 100VG-AnyLAN. Более высокие уровни, как правило, не работают напрямую с конкретной аппаратурой, хотя уровни 3, 4 и 5 еще могут учитывать ее особенности. Уровни 6 и 7 никак не связаны с аппаратурой, замены одного типа аппаратуры на другой они не замечают.

Как уже отмечалось, в уровне 2 (канальном) нередко выделяют два подуровня (sublayers) LLC и MAC (рис. 5.4):

- Верхний подуровень (LLC – Logical Link Control) осуществляет управление логической связью, то есть устанавливает виртуальный канал связи. Строго говоря, эти функции не связаны с конкретным типом сети, но часть из них все же возлагается на аппаратуру сети (сетевой адаптер). Другая часть функций подуровня LLC выполняется программой драйвера сетевого адаптера. Подуровень LLC отвечает за взаимодействие с уровнем 3 (сетевым).

- Нижний подуровень (MAC – Media Access Control) обеспечивает непосредственный доступ к среде передачи информации (каналу связи). Он напрямую связан с аппаратурой сети. Именно на подуровне MAC осуществляется взаимодействие с физическим уровнем. Здесь производится контроль состояния сети, повторная передача пакетов заданное число раз при коллизиях, прием пакетов и проверка правильности передачи.

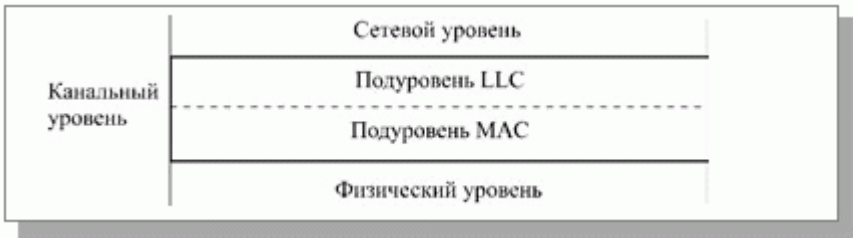

Рис. 5.4. Подуровни LLC и MAC канального уровня

Помимо модели OSI существует также модель IEEE Project 802, принятая в феврале 1980 года (отсюда и число 802 в названии), которую можно рассматривать как модификацию, развитие, уточнение модели OSI. Стандарты, определяемые этой моделью (так называемые 802-спецификации) относятся к нижним двум уровням модели OSI и делятся на двенадцать категорий, каждой из которых присвоен свой номер:

802.1 – объединение сетей с помощью мостов и коммутаторов
802.2 – управление логической связью на подуровне LLC.
802.3 – локальная сеть с методом доступа CSMA/CD и топологией шина (Ethernet).
802.4 – локальная сеть с топологией шина и маркерным доступом (Token-Bus).
802.5 – локальная сеть с топологией кольцо и маркерным доступом (Token-Ring).
802.6 – городская сеть (Metropolitan Area Network, MAN) с расстояниями между абонентами более 5 км.
802.7 – широкополосная технология передачи данных.
802.8 – оптоволоконная технология.
802.9 – интегрированные сети с возможностью передачи речи и данных.
802.10 – безопасность сетей, шифрование данных.
802.11 – беспроводная сеть по радиоканалу (WLAN – Wireless LAN).
802.12 – локальная сеть с централизованным управлением доступом по приоритетам запросов и топологией звезда (100VG-AnyLAN).

5.2. Аппаратура локальных сетей

Аппаратура локальных сетей обеспечивает реальную связь между абонентами. Выбор аппаратуры имеет важнейшее значение на этапе проектирования сети, так как стоимость аппаратуры составляет наиболее существенную часть от стоимости сети в целом, а замена аппаратуры связана

не только с дополнительными расходами, но зачастую и с трудоемкими работами. К аппаратуре локальных сетей относятся:

- кабели для передачи информации;
- разъемы для присоединения кабелей;
- согласующие терминаторы;
- сетевые адаптеры;
- репитеры;
- трансиверы;
- концентраторы;
- мосты;
- маршрутизаторы;
- шлюзы.

Сетевые адаптеры (они же контроллеры, карты, платы, интерфейсы, NIC – Network Interface Card) – это основная часть аппаратуры локальной сети. Назначение сетевого адаптера – сопряжение компьютера (или другого абонента) с сетью, то есть обеспечение обмена информацией между компьютером и каналом связи в соответствии с принятыми правилами обмена. Именно они реализуют функции двух нижних уровней модели OSI. Как правило, сетевые адаптеры выполняются в виде платы (рис. 5.5), вставляемой в слоты расширения системной магистрали (шины) компьютера (чаще всего PCI, ISA или PC-Card). Плата сетевого адаптера обычно имеет также один или несколько внешних разъемов для подключения к ней кабеля сети.

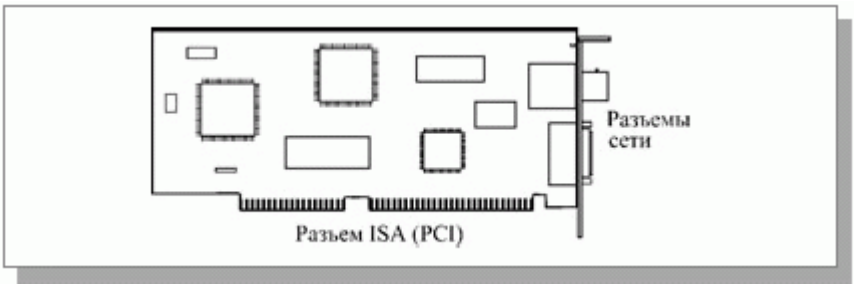

Рис. 5.5. Плата сетевого адаптера

Например, сетевые адаптеры Ethernet могут выпускаться со следующими наборами разъемов:

- TPO – разъем RJ-45 (для кабеля на витых парах по стандарту 10BASE-T).
- TPC – разъемы RJ-45 (для кабеля на витых парах 10BASE-T) и BNC (для коаксиального кабеля 10BASE2).
- TP – разъем RJ-45 (10BASE-T) и трансиверный разъем AUI.
- Combo – разъемы RJ-45 (10BASE-T), BNC (10BASE2), AUI.
- Coax – разъемы BNC, AUI.
- FL – разъем ST (для оптоволоконного кабеля 10BASE-FL).

Функции сетевого адаптера делятся на магистральные и сетевые. К магистральным относятся те функции, которые осуществляют взаимодействие адаптера с магистралью (системной шиной) компьютера (то есть опознание своего магистрального адреса, пересылка данных в

компьютер и из компьютера, выработка сигнала прерывания компьютера и т.д.). Сетевые функции обеспечивают общение адаптера с сетью.

К основным сетевым функциям адаптеров относятся:

- гальваническая развязка компьютера и кабеля локальной сети (для этого обычно используется передача сигналов через импульсные трансформаторы);
- преобразование логических сигналов в сетевые (электрические или световые) и обратно;
- кодирование и декодирование сетевых сигналов, то есть прямое и обратное преобразование сетевых кодов передачи информации (например, манчестерский код);
- опознание принимаемых пакетов (выбор из всех приходящих пакетов тех, которые адресованы данному абоненту или всем абонентам сети одновременно);
- преобразование параллельного кода в последовательный при передаче и обратное преобразование при приеме;
- буферирование передаваемой и принимаемой информации в буферной памяти адаптера;
- организация доступа к сети в соответствии с принятым методом управления обменом;
- подсчет контрольной суммы пакетов при передаче и приеме.

Типичный алгоритм взаимодействия компьютера с сетевым адаптером выглядит следующим образом.

Если компьютер хочет передать пакет, то он сначала формирует этот пакет в своей памяти, затем пересылает его в буферную память сетевого адаптера и дает команду адаптеру на передачу. Адаптер анализирует текущее состояние сети и при первой же возможности выдает пакет в сеть (выполняет управление доступом к сети). При этом он производит преобразование информации из буферной памяти в последовательный вид для побитной передачи по сети, подсчитывает контрольную сумму, кодирует биты пакета в сетевой код и через узел гальванической развязки выдает пакет в кабель сети. Буферная память в данном случае позволяет освободить компьютер от контроля состояния сети, а также обеспечить требуемый для сети темп выдачи информации.

Если по сети приходит пакет, то сетевой адаптер через узел гальванической развязки принимает биты пакета, производит их декодирование из сетевого кода и сравнивает сетевой адрес приемника из пакета со своим собственным адресом. Адрес сетевого адаптера, как правило, устанавливается производителем адаптера. Если адрес совпадает, то сетевой адаптер записывает пришедший пакет в свою буферную память и сообщает компьютеру (обычно – сигналом аппаратного прерывания) о том, что пришел пакет и его надо читать. Одновременно с записью пакета производится подсчет контрольной суммы, что позволяет к концу приема сделать вывод, имеются ли ошибки в этом пакете. Буферная память в данном случае опять же позволяет освободить компьютер от контроля сети, а также обеспечить высокую степень готовности сетевого адаптера к приему пакетов.

Чаще всего сетевые функции выполняются специальными микросхемами высокой степени интеграции, что дает возможность снизить стоимость адаптера и уменьшить площадь его платы.

Некоторые адаптеры позволяют реализовать функцию удаленной загрузки, то есть поддерживать работу в сети бездисковых компьютеров, загружающих свою операционную систему прямо из сети. Для этого в состав таких адаптеров включается постоянная память с соответствующей программой загрузки. Правда, не все сетевые программные средства поддерживают данный режим работы.

Сетевой адаптер выполняет функции первого и второго уровней модели OSI (рис. 5.6).

Рис. 5.6. Функции сетевого адаптера в модели OSI

Все остальные аппаратные средства локальных сетей (кроме адаптеров) имеют вспомогательный характер, и без них часто можно обойтись. Это сетевые промежуточные устройства.

Трансиверы или приемопередатчики (от английского TRANsmitter + reCEIVER) служат для передачи информации между адаптером и кабелем сети или между двумя сегментами (частями) сети. Трансиверы усиливают сигналы, преобразуют их уровни или преобразуют сигналы в другую форму (например, из электрической в световую и обратно). Трансиверами также часто называют встроенные в адаптер приемопередатчики.

Репитеры или повторители (repeater) выполняют более простую функцию, чем трансиверы. Они не преобразуют ни уровни сигналов, ни их физическую природу, а только восстанавливают ослабленные сигналы (их амплитуду и форму), приводя их к исходному виду. Цель такой ретрансляции сигналов состоит исключительно в увеличении длины сети (рис. 5.7).

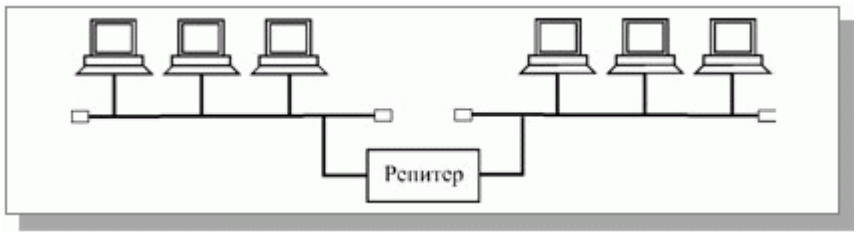

Рис. 5.7. Соединение репитером двух сегментов сети

Однако часто репитеры выполняют и некоторые другие, вспомогательные функции, например, гальваническую развязку соединяемых сегментов и оконечное согласование. Репитеры так же как трансиверы не производят никакой информационной обработки проходящих через них сигналов.

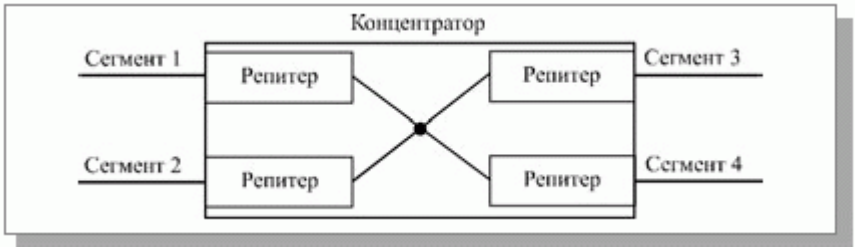

Рис. 5.8. Структура репитерного концентратора

Концентраторы (хабы, hub), как следует из их названия, служат для объединения в сеть нескольких сегментов. Концентраторы (или репитерные концентраторы) представляют собой несколько собранных в едином конструктиве репитеров, они выполняют те же функции, что и репитеры (рис. 5.8).

Преимущество подобных концентраторов по сравнению с отдельными репитерами в том, что все точки подключения собраны в одном месте, это упрощает реконфигурацию сети, контроль и поиск неисправностей. К тому же все репитеры в данном случае питаются от единого качественного источника питания.

Концентраторы иногда вмешиваются в обмен, помогая устранять некоторые явные ошибки обмена. В любом случае они работают на первом уровне модели OSI, так как имеют дело только с физическими сигналами, с битами пакета и не анализируют содержимое пакета, рассматривая пакет как единое целое (рис. 5.9). На первом же уровне работают и трансиверы, и репитеры.

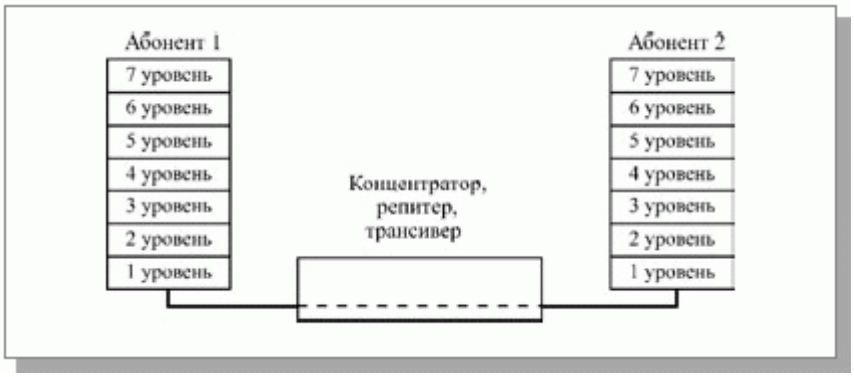

Рис. 5.9. Функции концентраторов, репитеров и трансиверов в модели OSI

Выпускаются также совсем простые концентраторы, которые соединяют сегменты сети без восстановления формы сигналов. Они не увеличивают длину сети.

Коммутаторы (свичи, коммутирующие концентраторы, switch), как и концентраторы, служат для соединения сегментов в сеть. Они также выполняют более сложные функции, производя сортировку поступающих на них пакетов.

Коммутаторы передают из одного сегмента сети в другой не все поступающие на них пакеты, а только те, которые адресованы компьютерам из другого сегмента. Пакеты, передаваемые между абонентами одного сегмента, через коммутатор не проходят. При этом сам пакет коммутатором не принимается, а только пересылается. Интенсивность обмена в сети снижается вследствие разделения нагрузки, поскольку каждый сегмент работает не только со своими пакетами, но и с пакетами, пришедшими из других сегментов.

Коммутатор работает на втором уровне модели OSI (подуровень MAC), так как анализирует MAC-адреса внутри пакета (рис. 5.10). Естественно, он выполняет и функции первого уровня.

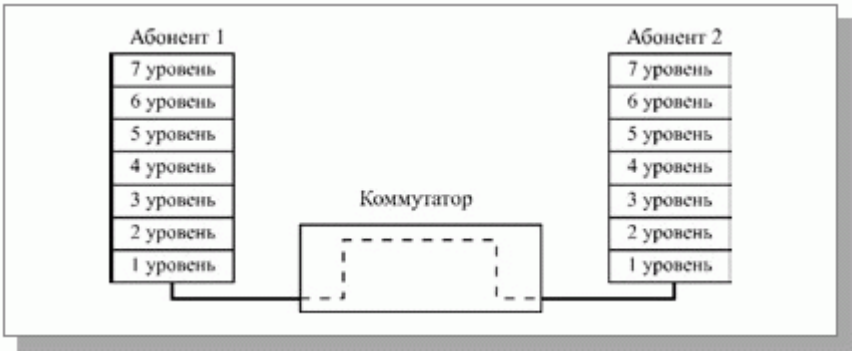

Рис. 5.10. Функции коммутаторов в модели OSI

В последнее время объем выпуска коммутаторов сильно вырос, цена на них упала, поэтому коммутаторы постепенно вытесняют концентраторы.

Мосты (bridge), маршрутизаторы (router) и шлюзы (gateway) служат для объединения в одну сеть несколько разнородных сетей с разными протоколами обмена нижнего уровня, в частности, с разными форматами пакетов, методами кодирования, скоростью передачи и т.д. В результате их применения сложная и неоднородная сеть, содержащая в себе различные сегменты, с точки зрения пользователя выглядит самой обычной сетью. Обеспечивается прозрачность сети для протоколов высокого уровня. Все они гораздо дороже, чем концентраторы, так как от них требуется довольно сложная обработка информации. Реализуются они обычно на базе компьютеров, подключенных к сети с помощью сетевых адаптеров. По сути, они представляют собой специализированные абоненты (узлы) сети.

Мосты – наиболее простые устройства, служащие для объединения сетей с разными стандартами обмена, например, Ethernet и Arcnet, или нескольких

сегментов (частей) одной и той же сети, например, Ethernet (рис. 5.11). В последнем случае мост, как и коммутатор, только разделяет нагрузку сегментов, повышая тем самым производительность сети в целом. В отличие от коммутаторов мосты принимают поступающие пакеты целиком и в случае необходимости производят их простейшую обработку. Мосты, как и коммутаторы, работают на втором уровне модели OSI (рис. 5.10), но в отличие от них могут захватывать также и верхний подуровень LLC второго уровня (для связи разнородных сетей). В последнее время мосты быстро вытесняются коммутаторами, которые становятся более функциональными.

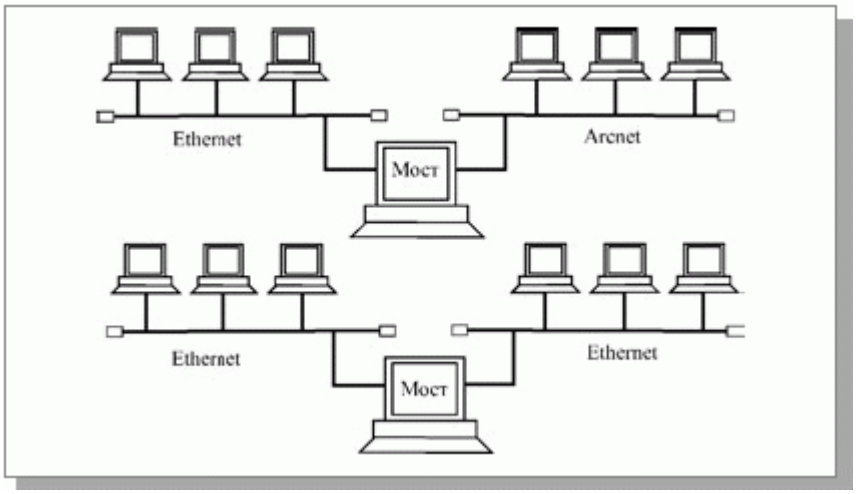

Рис. 5.11. Включение моста

Маршрутизаторы осуществляют выбор оптимального маршрута для каждого пакета с целью избежания чрезмерной нагрузки отдельных участков сети и обхода поврежденных участков. Они применяются, как правило, в сложных разветвленных сетях, имеющих несколько маршрутов между отдельными абонентами. Маршрутизаторы не преобразуют протоколы нижних уровней, поэтому они соединяют только сегменты одноименных сетей.

Маршрутизаторы работают на третьем уровне модели OSI, так как они анализируют не только MAC-адреса пакета, но и IP-адреса, то есть более глубоко проникают в инкапсулированный пакет (рис. 5.12).

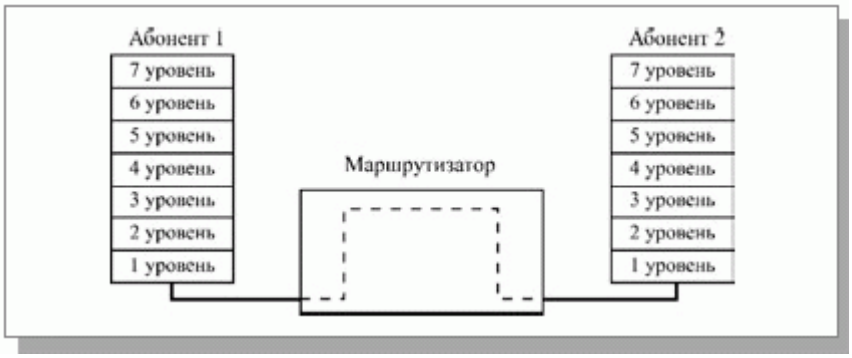

Рис. 5.12. Функции маршрутизатора в модели OSI

Существуют также гибридные маршрутизаторы (brouter), представляющие собой гибрид моста и маршрутизатора. Они выделяют пакеты, которым нужна маршрутизация и обрабатывают их как маршрутизатор, а для остальных пакетов служат обычным мостом.

Шлюзы – это устройства для соединения сетей с сильно отличающимися протоколами, например, для соединения локальных сетей с большими компьютерами или с глобальными сетями. Это самые дорогие и редко применяемые сетевые устройства. Шлюзы реализуют связь между абонентами на верхних уровнях модели OSI (с четвертого по седьмой). Соответственно, они должны выполнять и все функции нижестоящих уровней.

Подробнее промежуточные сетевые устройства будут рассмотрены в разделах, посвященных конкретным стандартным локальным сетям.

6.1. Стандартные сетевые протоколы

Протоколы – это набор правил и процедур, регулирующих порядок осуществления связи. Компьютеры, участвующие в обмене, должны работать по одним и тем же протоколам, чтобы в результате передачи вся информация восстанавливалась в первоначальном виде.

К протоколам нижних уровней относятся методы кодирования и декодирования, а также управления обменом в сети. Сейчас мы остановимся на особенностях протоколов более высоких уровней, реализуемых программно.

Связь сетевого адаптера с сетевым программным обеспечением осуществляют драйверы сетевых адаптеров. Именно благодаря драйверу компьютер может не знать никаких аппаратурных особенностей адаптера (его адресов, правил обмена с ним, его характеристик). Драйвер унифицирует, делает единообразным взаимодействие программных средств высокого уровня с любым адаптером данного класса. Сетевые драйверы, поставляемые вместе с сетевыми адаптерами, позволяют сетевым программам одинаково работать с платами разных поставщиков и даже с платами разных локальных сетей (Ethernet, Arcnet, Token-Ring и т.д.). Если говорить о стандартной модели OSI, то драйверы, как правило, выполняют функции канального уровня, хотя иногда они реализуют и часть функций сетевого уровня (рис. 6.1). Например, драйверы формируют передаваемый пакет в буферной памяти адаптера, читают из этой памяти пришедший по сети пакет, дают команду на передачу, информируют компьютер о приеме пакета.

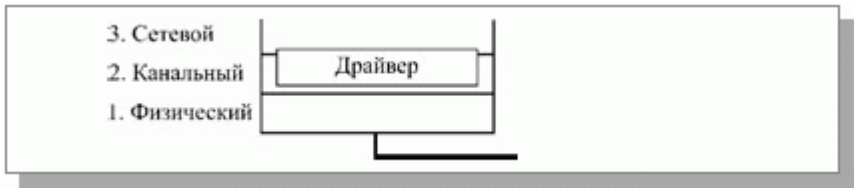

Рис. 6.1. Функции драйвера сетевого адаптера в модели OSI

Качество написания программы драйвера во многом определяет эффективность работы сети в целом. Даже при самых лучших характеристиках сетевого адаптера некачественный драйвер может резко ухудшить обмен по сети.

Прежде чем приобрести плату адаптера необходимо ознакомиться со списком совместимого оборудования (Hardware Compatibility List, HCL), который публикуют все производители сетевых ОС. Выбор этот довольно велик (например, для Microsoft Windows Server список включает более сотни драйверов сетевых адаптеров). Если в перечень HCL не входит адаптер какого-то типа, лучше его не покупать.

Протоколы высоких уровней.

Существует несколько стандартных наборов (или, как их еще называют, стеков) протоколов, получивших сейчас широкое распространение:

- набор протоколов ISO/OSI;
- IBM System Network Architecture (SNA);
- Digital DECnet;
- Novell NetWare;
- Apple AppleTalk;
- набор протоколов глобальной сети Интернет, TCP/IP.

Включение в этот список протоколов глобальной сети вполне объяснимо, поскольку модель OSI используется для любой открытой системы: на базе как локальной, так и глобальной сети или комбинации локальной и глобальной сетей.

Протоколы перечисленных наборов делятся на три основные типа:

- Прикладные протоколы (выполняющие функции трех верхних уровней модели OSI – прикладного, представительского и сеансового);
- Транспортные протоколы (реализующие функции средних уровней модели OSI – транспортного и сеансового);
- Сетевые протоколы (осуществляющие функции трех нижних уровней модели OSI).

Прикладные протоколы обеспечивают взаимодействие приложений и обмен данными между ними. Наиболее популярны:

- FTAM (File Transfer Access and Management) – протокол OSI доступа к файлам;
- X.400 – протокол CCITT для международного обмена электронной почтой;

- X.500 – протокол CCITT служб файлов и каталогов на нескольких системах;
- SMTP (Simple Mail Transfer Protocol) – протокол глобальной сети Интернет для обмена электронной почтой;
- FTP (File Transfer Protocol) – протокол глобальной сети Интернет для передачи файлов;
- SNMP (Simple Network Management Protocol) – протокол для мониторинга сети, контроля за работой сетевых компонентов и управления ими;
- Telnet – протокол глобальной сети Интернет для регистрации на удаленных серверах и обработки данных на них;
- Microsoft SMBs (Server Message Blocks, блоки сообщений сервера) и клиентские оболочки или редиректоры фирмы Microsoft;
- NCP (Novell NetWare Core Protocol) и клиентские оболочки или редиректоры фирмы Novell.

Транспортные протоколы поддерживают сеансы связи между компьютерами и гарантируют надежный обмен данными между ними. Наиболее популярные из них следующие:

- TCP (Transmission Control Protocol) – часть набора протоколов TCP/IP для гарантированной доставки данных, разбитых на последовательность фрагментов;
- SPX – часть набора протоколов IPX/SPX (Internetwork Packet Exchange/Sequential Packet Exchange) для гарантированной доставки данных, разбитых на последовательность фрагментов, предложенных компанией Novell;
- NWLink – реализация протокола IPX/SPX компании Microsoft;
- NetBEUI – (NetBIOS Extended User Interface, расширенный интерфейс NetBIOS) – устанавливает сеансы связи между компьютерами (NetBIOS) и предоставляет верхним уровням транспортные услуги (NetBEUI).

Сетевые протоколы управляют адресацией, маршрутизацией, проверкой ошибок и запросами на повторную передачу. Широко распространены следующие из них:

- IP (Internet Protocol) – TCP/IP-протокол для негарантированной передачи пакетов без установления соединений;
- IPX (Internetwork Packet Exchange) – протокол компании NetWare для негарантированной передачи пакетов и маршрутизации пакетов;
- NWLink – реализация протокола IPX/SPX компании Microsoft;
- NetBEUI – транспортный протокол, обеспечивающий услуги транспортировки данных для сеансов и приложений NetBIOS.

Все перечисленные протоколы могут быть поставлены в соответствие тем или иным уровням эталонной модели OSI. Но при этом надо учитывать, что разработчики протоколов не слишком строго придерживаются этих уровней. Например, некоторые протоколы выполняют функции, относящиеся сразу к нескольким уровням модели OSI, а другие – только часть функций одного из уровней. Это приводит к тому, что протоколы разных компаний часто оказываются несовместимы между собой. Кроме того, протоколы могут быть успешно использованы исключительно в составе своего набора протоколов (стека протоколов), который выполняет более или менее законченную группу функций. Как раз это и делает сетевую ОС «фирменной», то есть, по сути, несовместимой со стандартной моделью открытой системы OSI.

В качестве примера на рис. 6.2, 6.3 и 6.4 схематически показано соотношение протоколов, используемых популярными фирменными сетевыми ОС, и уровней стандартной модели OSI. Как видно из рисунков, практически ни на одном уровне нет четкого соответствия реального протокола какому-нибудь уровню идеальной модели. Выстраивание подобных соотношений довольно условно, так как трудно четко разграничить функции всех частей программного обеспечения. К тому же компании-производители программных средств далеко не всегда подробно описывают внутреннюю структуру продуктов.

Рис. 6.2. Соотношение уровней модели OSI и протоколов сети Интернет

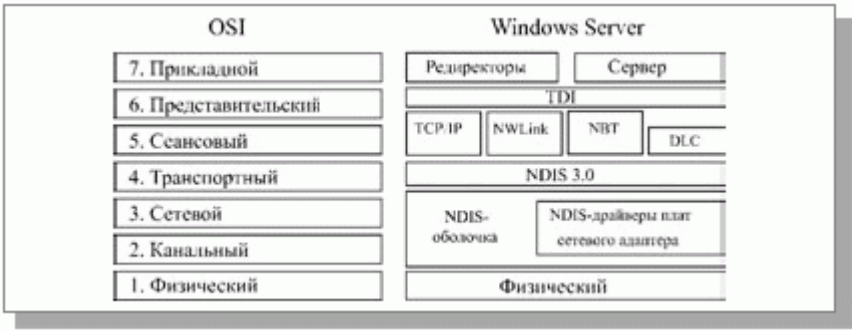

Рис. 6.3. Соотношение уровней модели OSI и протоколов ОС Windows Server

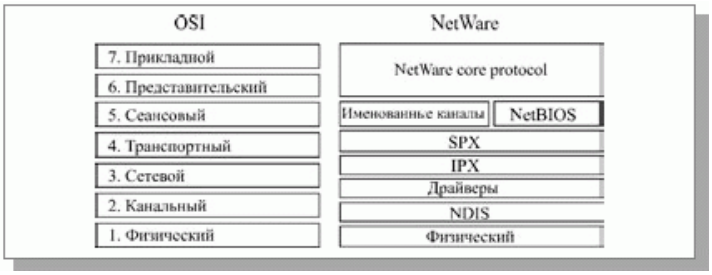

Рис. 6.4. Соотношение уровней модели OSI и протоколов операционной системы NetWare

Рассмотрим подробнее некоторые наиболее распространенные протоколы.

Модель OSI допускает два основных метода взаимодействия абонентов в сети:

- Метод взаимодействия без логического соединения (или метод дейтаграмм).
- Метод взаимодействия с логическим соединением.

Метод дейтаграмм – это простейший метод, в котором каждый пакет рассматривается как самостоятельный объект (рис. 6.5).

Рис. 6.5. Метод дейтаграмм

Пакет при этом методе передается без установления логического канала, то есть без предварительного обмена служебными пакетами для выяснения готовности приемника, а также без ликвидации логического канала, то есть без пакета подтверждения окончания передачи. Дойдет пакет до приемника или нет – неизвестно (проверка факта получения переносится на более высокие уровни).

Метод дейтаграмм предъявляет повышенные требования к аппаратуре (так как приемник всегда должен быть готов к приему пакета). Достоинства метода в том, что передатчик и приемник работают независимо друг от друга, к тому же пакеты могут накапливаться в буфере и затем передаваться вместе, можно также использовать широковещательную передачу, то есть адресовать пакет всем абонентам одновременно. Недостатки метода – это возможность потери пакетов, а также бесполезной загрузки сети пакетами в случае отсутствия или неготовности приемника.

Метод с логическим соединением (рис. 6.6, рис. 4.5) разработан позднее, чем метод дейтаграмм, и отличается усложненным порядком взаимодействия.

При этом методе пакет передается только после того, как будет установлено логическое соединение (канал) между приемником и передатчиком. Каждому информационному пакету сопутствует один или несколько служебных пакетов (установка соединения, подтверждение получения, запрос повторной передачи, разрыв соединения). Логический канал может устанавливаться на время передачи одного или нескольких пакетов.

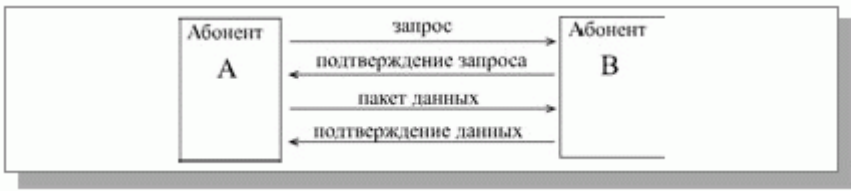

Рис. 6.6. Метод с логическим соединением

Метод с логическим соединением, более надежен, чем метод дейтаграмм, поскольку к моменту ликвидации логического канала передатчик уверен, что все его пакеты дошли до места назначения, причем дошли успешно. Не бывает при данном методе и перегрузки сети из-за бесполезных пакетов.

Недостаток метода с логическим соединением состоит в том, что довольно сложно разрешить ситуацию, когда принимающий абонент по тем или иным причинам не готов к обмену, например, из-за обрыва кабеля, отключения питания, неисправности сетевого оборудования, сбоя в компьютере. При этом требуются алгоритм обмена с повторением неподтвержденного пакета заданное количество раз, причем важен и тип неподтвержденного пакета. Не может этот метод передавать широковещательные пакеты, так как нельзя организовать логические каналы сразу со всеми абонентами.

Примеры протоколов, работающих по методу дейтаграмм— это протоколы IP и IPX.

Примеры протоколов, работающих по методу с логическим соединением – это TCP и SPX.

Именно для того, чтобы объединить достоинства обоих методов, эти протоколы используются в виде связанных наборов: TCP/IP и IPX/SPX, в которых протокол более высокого уровня (TCP, SPX), работающий на базе протокола более низкого уровня (IP, IPX), гарантирует правильную доставку пакетов в требуемом порядке.

Протоколы IPX/SPX, разработанные компанией Novell, образуют набор (стек), используемый в сетевых программных средствах довольно широко распространенных локальных сетей Novell (NetWare). Это сравнительно небольшой и быстрый протокол, поддерживающий маршрутизацию. Прикладные программы могут обращаться непосредственно к уровню IPX, например, для посылки широковещательных сообщений, но значительно чаще работают с уровнем SPX, гарантирующим быструю и надежную доставку пакетов. Если скорость не слишком важна, то прикладные программы применяют еще более высокий уровень, например, протокол NetBIOS, предоставляющий удобный сервис. Компанией Microsoft предложена своя реализация протокола IPX/SPX, называемая NWLink. Протоколы IPX/SPX и NWLink поддерживаются ОС NetWare и Windows. Выбор этих протоколов обеспечивает совместимость по сети любых абонентов с данными ОС.

Набор (стек) протоколов TCP/IP был специально разработан для глобальных сетей и для межсетевого взаимодействия. Он изначально ориентирован на низкое качество каналов связи, на большую вероятность ошибок и разрывов связей. Этот протокол принят во всемирной компьютерной сети Интернет, значительная часть абонентов которой подключается по коммутируемым линиям (то есть обычным телефонным линиям). Как и протокол IPX/SPX, протокол TCP/IP также поддерживает маршрутизацию. На его основе работают протоколы высоких уровней, такие как SMTP, FTP, SNMP. Недостаток протокола TCP/IP —более низкая скорость работы, чем у

IPX/SPX. Однако сейчас протокол TCP/IP используется и в локальных сетях, чтобы упростить согласование протоколов локальных и глобальных сетей. В настоящее время он считается основным в самых распространенных операционных системах.

В стек протоколов TCP/IP часто включают и протоколы всех верхних уровней (рис. 6.7). В этом случае, уже можно говорить о функциональной полноте стека TCP/IP.

Как протокол IPX, так и протокол IP являются самыми низкоуровневыми протоколами, поэтому они непосредственно инкапсулируют свою информацию, называемую дейтаграммой, в поле данных передаваемого по сети пакета (см. рис. 4.6). При этом в заголовок дейтаграммы входят адреса абонентов (отправителя и получателя) более высокого уровня, чем MAC-адреса, – это IPX-адреса для протокола IPX или IP-адреса для протокола IP. Эти адреса включают номера сети и узла, хоста (индивидуальный идентификатор абонента). При этом IPX-адреса (рис. 6.8) более простые, имеют всего один формат, а в IP-адрес (рис. 6.9) могут входить три формата (класса A, B и C), различающиеся значениями трех начальных битов.

Рис. 6.7. Соотношение уровней модели OSI и стека протоколов TCP/IP

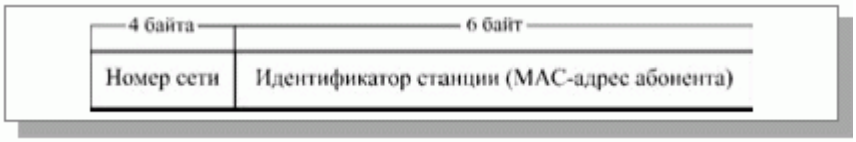

Рис. 6.8. Формат IPX- адреса

Рис. 6.9. Форматы IP-адреса

Интересно, что IP-адрес не имеет никакой связи с MAC-адресами абонентов. Номер узла в нем присваивается абоненту независимо от его MAC-адреса. В

качестве идентификатора станции IPX-адрес включает в себя полный MAC-адрес абонента.

Номер сети – это код, присвоенный каждой конкретной сети, то есть каждой широковещательной области общей, единой сети. Под широковещательной областью понимается часть сети, которая прозрачна для широковещательных пакетов, пропускает их беспрепятственно.

Протокол NetBIOS (сетевая базовая система ввода/вывода) был разработан компанией IBM для сетей IBM PC Network и IBM Token-Ring по образцу системы BIOS персонального компьютера. С тех пор этот протокол стал фактическим стандартом (официально он не стандартизован), и многие сетевые ОС содержат в себе эмулятор NetBIOS для обеспечения совместимости. Первоначально NetBIOS реализовывал сеансовый, транспортный и сетевой уровни, однако в последующих сетях на более низких уровнях используются стандартные протоколы (например, IPX/SPX), а на долю эмулятора NetBIOS остается только сеансовый уровень. NetBIOS обеспечивает более высокий уровень сервиса, чем IPX/SPX, но работает медленнее.

На основе протокола NetBIOS был разработан протокол NetBEUI, который представляет собой развитие протокола NetBIOS до транспортного уровня. Однако недостаток NetBEUI состоит в том, что он не поддерживает межсетевое взаимодействие и не обеспечивает маршрутизацию. Поэтому данный протокол используется только в простых сетях, не рассчитанных на подключение к Интернет. Сложные сети ориентируются на более универсальные протоколы TCP/IP и IPX/SPX. Протокол NetBEUI в настоящее время считается устаревшим, хотя даже в ОС Windows XP предусмотрена его поддержка, но как дополнительная опция.

Наконец, упоминавшийся уже набор протоколов OSI – это полный набор (стек) протоколов, где каждый протокол точно соответствует определенному уровню стандартной модели OSI. Набор содержит маршрутизируемые и транспортные протоколы, серии протоколов IEEE 802, протокол сеансового уровня представительского уровня и несколько протоколов прикладного уровня. Пока широкого распространения этот набор протоколов не получил, хотя он и полностью соответствует эталонной модели OSI.

6.2 Стандартные сетевые программные средства

Функции верхних уровней эталонной модели OSI выполняют сетевые программные средства. Для установки сети достаточно иметь набор сетевого оборудования, его драйверы, а также сетевое программное обеспечение. От выбора программного обеспечения зависит очень многое: допустимый размер сети, удобство использования и контроля сети, режимы доступа к

ресурсам, производительность сети в разных режимах и т.д. Хотя конечно, заменить одну программную систему на другую значительно проще, чем сменить оборудование.

С точки зрения распределения функций между компьютерами сети, все сети можно разделить на две группы:

- Одноранговые сети, состоящие из равноправных (с точки зрения доступа к сети) компьютеров.
- Сети на основе серверов, в которых существуют только выделенные (dedicated) серверы, занимающиеся исключительно сетевыми функциями. Выделенный сервер может быть единственным или их может быть несколько.

Согласно с этим, выделяют и типы программных средств, реализующих данные виды сетей.

6.2.1 Одноранговые сети

Одноранговые сети (Peer-to-Peer Network) и соответствующие программные средства, как правило, используются для объединения небольшого количества компьютеров (рис. 6.10). Каждый компьютер такой сети может одновременно являться и сервером и клиентом сети, хотя вполне допустимо назначение одного компьютера только сервером, а другого только клиентом. Принципиальна возможность совмещения функций клиента и сервера. Важно также и то, что в одноранговой сети любой сервер может быть невыделенным (non-dedicated), может не только обслуживать сеть, но и работать как автономный компьютер (правда, запросы к нему по сети сильно снижают скорость его работы). В одноранговой сети могут быть и выделенные серверы, только обслуживающие сеть.

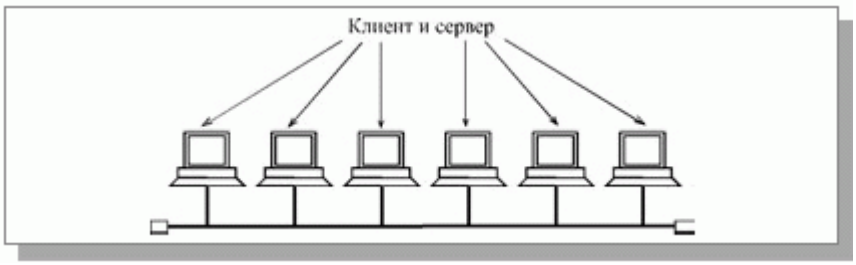

Рис. 6.10. Одноранговая сеть

Именно в данном случае наиболее правильно говорить о распределенных дисковых ресурсах, о виртуальном компьютере, а также о суммировании объемов дисков всех компьютеров сети. Если все компьютеры являются серверами, то любой файл, созданный на одном из них сразу же становится доступным всем остальным компьютерам, его не надо передавать на централизованный сервер.

Достоинством одноранговых сетей является их высокая гибкость: в зависимости от конкретной задачи сеть может использоваться очень активно

либо совсем не использоваться. Из-за большой самостоятельности компьютеров в таких сетях редко бывает ситуация перегрузки (к тому же количество компьютеров обычно невелико). Установка одноранговых сетей довольно проста, к тому же не требуются дополнительные дорогостоящие серверы. Кроме того, нет необходимости в системном администрировании, пользователи могут сами управлять своими ресурсами.

В одноранговых сетях допускается определение различных прав пользователей по доступу к сетевым ресурсам, но система разграничения прав не слишком развита. Если каждый ресурс защищен своим паролем, то пользователю приходится запоминать большое число паролей.

К недостаткам одноранговых сетей относятся также слабая система контроля и протоколирования работы сети, трудности с резервным копированием распределенной информации. К тому же выход из строя любого компьютера-сервера приводит к потере части общей информации, то есть все такие компьютеры должны быть по возможности высоконадежными. Эффективная скорость передачи информации по одноранговой сети часто оказывается недостаточной, поскольку трудно обеспечить быстродействие процессоров, большой объем оперативной памяти и высокие скорости обмена с жестким диском для всех компьютеров сети. К тому же компьютеры сети работают не только на сеть, но и решают другие задачи.

Примеры одноранговых сетевых программных средств:

- NetWare Lite компании Novell (в настоящее время уже не производится);
- LANtastic компании Artisoft (выпуск практически прекращен);
- Windows for Workgroups компании Microsoft (первая версия ОС Windows со встроенной поддержкой сети, выпущенная в 1992 году);
- Windows NT Workstation компании Microsoft;
- Windows 95... Windows XP компании Microsoft.

Первые одноранговые сетевые программные средства представляли собой сетевые оболочки, работающие под управлением DOS (например, NetWare Lite). Они перехватывали все запросы DOS, те запросы, которые вызваны обращениями к сетевым устройствам, обрабатывались и выполнялись сетевой оболочкой, а те, которые вызваны обращениями к «местным», несетевым ресурсам, возвращались обратно в DOS и обрабатывались стандартным образом.

Более поздние одноранговые сетевые программные средства уже были встроены в операционную систему Windows. Это гораздо удобнее, так как исключается этап установки сетевых программ. Поэтому сетевые оболочки сейчас уже практически не используются, хотя многие их характеристики были заметно лучше, чем у сетевых средств Windows.

Сейчас считается, что одноранговая сеть наиболее эффективна в небольших сетях (около 10 компьютеров). При значительном количестве компьютеров сетевые операции сильно замедлят работу компьютеров и создадут множество других проблем. Тем не менее, для небольшого офиса одноранговая сеть – оптимальное решение.

Самая распространенная в настоящий момент одноранговая сеть – это сеть на основе Windows XP (или более ранних версий ОС Windows).

При этом пользователь, приобретая компьютер с установленной ОС, автоматически получает и возможность выхода в сеть. Естественно, это во многих случаях гораздо удобнее, чем приобретать и устанавливать пусть даже и более совершенные продукты других фирм. К тому же пользователю не надо изучать интерфейс пользователя сетевой программы, так как он строится так же, как и интерфейс пользователя всех остальных частей ОС.

Если приобретаемый компьютер еще и имеет установленный сетевой адаптер, то построить сеть пользователю совсем просто. Надо только соединить компьютеры кабелем и настроить сетевые программы.

В Windows предусмотрена поддержка совместного использования дисков (в том числе гибких дисков и CD), а также принтеров. Имеется возможность объединения всех пользователей в рабочие группы для более удобного поиска требуемых ресурсов и организации доступа к ним. Пользователи имеют доступ к встроенной системе электронной почты. Это означает, что все пользователи сети получают возможность совместно применять многие ресурсы ОС своего компьютера.

При настройке сети пользователь должен выбрать тип сетевого протокола. По умолчанию используется протокол TCP/IP, но возможно применение IPX/SPX (NWLink), а также NetBEUI. При выборе TCP/IP можно задавать адреса IP вручную или с помощью автоматической настройки адресации (в этом случае компьютер сам присвоит себе адрес из диапазона, не используемого в Интернет).

Кроме того, надо задать индивидуальное имя компьютера и определить рабочую группу, к которой он относится.

После этого можно разрешить доступ по сети к ресурсам каждого компьютера сети, к его файлам, папкам, принтерам, сканерам, доступу в Интернет.

6.2.2. Сети на основе сервера

Сети на основе сервера (Server-based Network) применяются в тех случаях, когда в сеть должно быть объединено много пользователей. В этом случае

возможностей одноранговой сети может не хватить. Поэтому в сеть включается специализированный компьютер – сервер, который обслуживает только сеть и не решает никаких других задач (рис. 6.11). Такой сервер называется выделенным. Сервер может быть и специализирован на решении одной задачи, например, сервер печати, но чаще всего серверами выступают именно компьютеры. В сети может быть и несколько серверов, каждый из которых решает свою задачу.

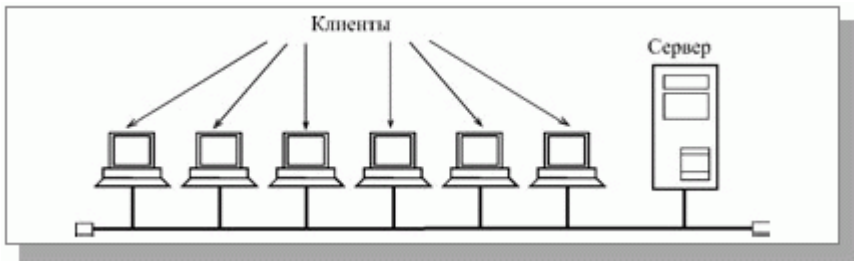

Рис. 6.11. Сеть на основе сервера

Серверы специально оптимизированы для быстрой обработки сетевых запросов на разделяемые ресурсы и для управления защитой файлов и каталогов. При больших размерах сети мощности одного сервера может оказаться недостаточно, и тогда в сеть включают несколько серверов. Серверы могут выполнять и некоторые другие задачи: сетевая печать, выход в глобальную сеть, связь с другой локальной сетью, обслуживание электронной почты и т.д. Количество пользователей сети на основе сервера может достигать нескольких тысяч. Одноранговой сетью такого размера просто невозможно было бы управлять. Кроме того, в сети на основе серверов можно легко менять количество подключаемых компьютеров, такие сети называются масштабируемыми.

В любом случае в сети на основе сервера существует четкое разделение компьютеров на клиентов (или рабочие станции) и серверы. Клиенты не могут работать как серверы, а серверы – как клиенты и как автономные компьютеры. Очевидно, что все сетевые дисковые ресурсы могут располагаться только на сервере, а клиенты могут обращаться только к серверу, но не друг к другу. Однако это не значит, что они не могут общаться между собой, просто пересылка информации от одного клиента к другому возможна только через сервер, например, через файл, доступный всем клиентам. В данном случае реализуется некоторая «логическая звезда» с сервером в центре, хотя физическая топология сети может быть любой.

Достоинством сети на основе сервера часто называют надежность. Это верно, если сам сервер действительно очень надежен. В противном случае любой отказ сервера приводит к полному параличу сети в отличие от ситуации с одноранговой сетью, где отказ одного из компьютеров не приводит к отказу всей сети. Бесспорное достоинство сети на основе сервера – высокая скорость обмена, так как сервер всегда оснащается быстрым процессором

(или даже несколькими процессорами), оперативной памятью большого объема и быстрыми жесткими дисками. Так как все ресурсы сети собраны в одном месте, возможно применение гораздо более мощных средств управления доступом, защиты данных, протоколирования обмена, чем в одноранговых сетях.

К недостаткам сети на основе сервера относятся ее громоздкость в случае небольшого количества компьютеров, зависимость всех компьютеров-клиентов от сервера, более высокая стоимость сети вследствие использования дорогого сервера. Но, говоря о стоимости, надо также учитывать, что при одном и том же объеме сетевых дисков большой диск сервера получается дешевле, чем много дисков меньшего объема, входящих в состав всех компьютеров одноранговой сети.

Примеры некоторых сетевых программных средств на основе сервера:

- NetWare компании Novell (самая распространенная сетевая ОС);
- LAN Server компании IBM (почти не используется);
- LAN Manager компании Microsoft;
- Windows NT Server компании Microsoft;
- Windows Server 2003 компании Microsoft.

На файл-сервере в данном случае устанавливается специальная сетевая ОС, рассчитанная на работу сервера. Эта сетевая ОС оптимизирована для эффективного выполнения специфических операций по организации сетевого обмена. На рабочих станциях (клиентах) может устанавливаться любая совместимая ОС, поддерживающая сеть.

Для обеспечения надежной работы сети при авариях электропитания применяется бесперебойное электропитание сервера. В данном случае это гораздо проще, чем при одноранговой сети, где желательно оснащать источниками бесперебойного питания все компьютеры сети. Для администрирования сети (то есть управления распределением ресурсов, контроля прав доступа, защиты данных, файловой системы, резервирования файлов и т.д.) в случае сети на основе сервера необходимо выделять специального человека, имеющего соответствующую квалификацию. Централизованное администрирование облегчает обслуживание сети и позволяет оперативно решать все вопросы. Особенно это важно для надежной защиты данных от несанкционированного доступа. В случае же одноранговой сети можно обойтись и без специалиста-администратора, правда, при этом все пользователи сети должны иметь хоть какое-то представление об администрировании.

Процесс установки серверной сетевой ОС гораздо сложнее, чем в случае одноранговой сети. Так, он включает в себя следующие обязательные процедуры:

- форматирование и разбиение на разделы жесткого диска компьютера-сервера;
- присвоение индивидуального имени серверу;
- присвоение имени сети;
- установка и настройка сетевого протокола;
- выбор сетевых служб;
- ввод пароля администратора.

Сетевая операционная система на базе сервера Windows Server 2003 предоставляет пользователям гораздо больше возможностей, чем в случае одноранговой сети.

Она позволяет строить сложные иерархические структуры сети на основе логических групп компьютеров (доменов, domain), наборов доменов (деревьев, tree) и наборов деревьев (леса, forest).

Домен представляет собой группу компьютеров, управляемых контроллером домена, специальным сервером. Домен использует собственную базу данных, содержащую учетные записи пользователей, и управляет собственными ресурсами, такими как принтеры и общие файлы. Каждому домену присваивается свое имя (обычно домен рассматривается как отдельная сеть со своим номером). В каждый домен может входить несколько рабочих групп, которые формируются из пользователей, решающих общую или сходные задачи. В принципе домен может включать тысячи пользователей, однако обычно домены не слишком велики, и несколько доменов объединяются в дерево доменов. Это упрощает управление сетью. Точно так же несколько деревьев может объединяться в лес, самую крупную административную структуру, поддерживаемую данной ОС.

В процессе установки Windows Server 2003 необходимо задать тип протокола сети. По умолчанию используется TCP/IP, но возможно применение NWLink (IPX/SPX).

Каждому серверу необходимо назначить роль, которую он будет выполнять в сети:

- контроллер домена (управляет работой домена);
- файловый сервер (хранит совместно используемые файлы);
- сервер печати (управляет сетевым принтером);
- Web-сервер (содержит сайт, доступный по сети Интернет или по локальной сети);
- коммуникационный сервер (обеспечивает работу электронной почты и конференций);
- сервер удаленного доступа (обеспечивает удаленный доступ).

Каждому пользователю сети необходимо присвоить свое учетное имя и пароль, а также права доступа к ресурсам (полномочия). Права доступа могут задаваться как индивидуально, так и целой рабочей группе пользователей. Windows Server 2003 обеспечивает следующие виды полномочий для папок:

- полный контроль (просмотр, чтение, запись, удаление папки, подпапок, файлов, запуск на исполнение, установка прав доступа к папке);
- изменение (просмотр, чтение, запись, удаление подпапок и файлов, запуск на исполнение);
- чтение и исполнение (просмотр, чтение, запуск на исполнение);
- просмотр содержимого папки;
- запись нового содержимого в папку;
- чтение информации из папки.

Те же самые уровни полномочий (кроме просмотра содержимого) предусмотрены и для файлов, доступных по сети.

Сетевые операционные системы NetWare компании Novell сегодня очень популярны, что объясняется их высокой производительностью, совместимостью с разными аппаратными средствами и развитой системой средств защиты данных. Компания Novell выпускает сетевые программные средства с 1979 года: несколько версий сетевых ОС на базе файловых серверов (одна из последних версий – NetWare 6 и 6.5), клиентское программное обеспечение, а также средства диагностики работы сетей. Популярные до недавнего времени сетевые оболочки одноранговых сетей, такие как NetWare Lite и Personal NetWare сейчас уже не производятся.

Отличительной особенностью сетевых программных средств Novell всегда была их открытость, то есть совместимость с ОС различных фирм: Windows, UNIX, Macintosh, OS/2. Кроме того, они всегда обеспечивали возможность работы с аппаратными средствами практически всех известных производителей. Это позволяет строить на их основе сети из разнообразных абонентов – от самых простых до самых сложных.

Все сетевые продукты NetWare допускают подключение бездисковых рабочих станций (клиентов), что позволяет при необходимости значительно снизить стоимость сети. Во всех продуктах предусмотрена поддержка сетевых мостов.

Продуктам Novell NetWare присущи и недостатки, например, их стоимость для небольших сетей оказывается достаточно высокой по сравнению с ценой продуктов других производителей. Кроме того, их установка сравнительно сложна, но они уже стали фактическим стандартом, поэтому их позиции на рынке довольно прочны.

Рассмотрим кратко особенности сетевой ОС Novell NetWare 6.5.

Как и в случае Microsoft Windows Server 2003, Novell NetWare 6.5 требует создания древовидной иерархической структуры, включающей в себя сетевые деревья, серверы, пользователей, группы и прочие объекты.

Novell NetWare 6.5 предусматривает обязательное разбиение жестких дисков с использованием собственной системы хранения файлов NSS (Novell Storage

Services), которое требует создания логических разделов (Volumes) на диске. Это позволяет серверу более эффективно решать сетевые задачи.

Для каждого сервера сети надо выбрать один из трех типов:

- Настраиваемый сервер (в частности, Web-сервер, FTP-сервер).
- Основной файловый сервер.
- Специальный сервер (например, DNS/DHCP-сервер, контролирующий сетевые адреса и имена, или сервер резервного копирования).

Кроме того, надо задать тип используемого протокола – TCP/IP или IPX/SPX.

На компьютеры-клиенты следует установить клиентское программное обеспечение. Это сравнительно простая процедура.

Каждому клиенту присваивается учетная запись, предоставляются свои права доступа к ресурсам. Клиенты могут быть объединены в рабочие группы, каждой из которых присваиваются имена и права доступа.

Предусмотрены следующие виды доступа к файлам и каталогам (папкам):

- Изменение прав доступа к каталогу или файлу;
- Просмотр каталога;
- Создание каталогов и файлов в данном каталоге;
- Удаление каталогов и файлов в данном каталоге;
- Изменение содержимого файлов;
- Любые операции над файлами каталога;
- Запись в файл.

V. МЕТОДИЧЕСКИЕ УКАЗАНИЯ К ПРАКТИЧЕСКИМ СЕМИНАРСКИМ ЗАНЯТИЯМ

Для выполнения практических семинаров студенту необходимо иметь конспект лекций. Студент знакомится с заданием и выполняет его, опираясь на конспект лекций. При выполнении задания связанного с программированием, студент разрабатывает алгоритм решения предложенной задачи, набирает текст программы (реализующей алгоритм), отлаживает и тестирует программу. При написании программы необходимо использовать конструкции языка программирования Си.

VI. ПРАКТИЧЕСКИЙ СЕМИНАР (ПЕРЕЧЕНЬ ОСНОВНЫХ ТЕМ)

Семинар #1: Введение в курс практических занятий. Знакомство с ОС UNIX.

Введение. Краткая история операционной системы UNIX, ее структура. Системные вызовы и библиотека libc. Понятия login и password. Упрощенное понятие об устройстве файловой системы в UNIX. Полные имена файлов. Понятие о текущей директории. Команда pwd. Относительные имена файлов. Домашняя директория пользователя и ее определение. Команда man - универсальный справочник. Команды cd - смены текущей директории и ls - просмотра состава директории. Команда cat и создание файла. Перенаправление ввода и вывода. Простейшие команды для работы с файлами - cp, rm, mkdir, mv. История редактирования файлов - ed, vi. Система Midnight Commander - mc. Встроенный mc редактор и редактор joe. Пользователь и группа. Команды chown и chgrp. Права доступа к файлу. Команда ls с опциями -al. Использование команд chmod и umask. Системные вызовы getuid и getgid. Компиляция программ на языке C в UNIX и их запуск.

1. Введение. Краткая история ОС UNIX, ее структура

В тексте программные конструкции, включая имена системных вызовов, стандартных функций и команды оболочки операционной системы, `выделены другим шрифтом`. В UNIX системные вызовы и команды оболочки инициируют сложные последовательности действий, затрагивая различные аспекты функционирования ОС. Подробные описания большинства используемых системных вызовов, системных функций и некоторых команд оболочки ОС вынесены из основного текста на серый фон и обведены рамкой.

Если какой-либо параметр у команды оболочки является необязательным, он будет указываться в квадратных скобках, например, `[who]`. В случае, когда возможен выбор только одного из нескольких возможных вариантов параметров, они перечисляются в фигурных скобках и разделяться вертикальной чертой, например, `{+ | - | =}`.

На первой лекции мы разобрали содержание понятия «операционная система», обсудили функции операционных систем и способы их построения. Все материалы мы будем иллюстрировать практическими примерами, связанными с использованием одной из разновидностей ОС UNIX –Linux.

Ядро ОС Linux представляет собой монолитную систему, при компиляции ядра можно разрешить динамическую загрузку и выгрузку очень многих его – модулей. В момент загрузки модуля его код загружается для исполнения в привилегированном режиме и связывается с остальной частью ядра. Внутри модуля могут использоваться любые экспортируемые ядром функции.

Свой нынешний вид эта ОС обрела в результате длительной эволюции UNIX-образных ОС. Важным в истории развития UNIX является существование двух стержневых линий эволюции – линии System V и линии BSD.

2. Системные вызовы и библиотека libc

Основной постоянно функционирующей частью ОС UNIX является ее ядро. Другие программы (системные или пользовательские) могут общаться с ядром посредством системных вызовов, которые по сути дела являются прямыми точками входа программ в ядро. При исполнении системного вызова программа пользователя временно переходит в привилегированный режим, получая доступ к данным или устройствам, которые недоступны при работе в режиме пользователя.

Реальные машинные команды, необходимые для активизации системных вызовов, естественно, отличаются от машины к машине, наряду со способом передачи параметров и результатов между вызывающей программой и ядром. Однако использование системных вызовов ничем внешне не отличается от использования других функций стандартной ANSI библиотеки языка С, например, `strlen()`, `strcpy()` и т.д. Стандартная библиотека UNIX – `libc` – обеспечивает С-интерфейс к каждому системному вызову. Это приводит к тому, что системный вызов выглядит как функция на языке С для программиста. Более того, многие известные стандартные функции, например функции для работы с файлами: `fopen()`, `fread()`, `fwrite()` при реализации в ОС UNIX будут применять различные системные вызовы.

Большинство системных вызовов, возвращающих целое значение, использует значение `-1` для оповещения о возникновении ошибки и значение большее или равное `0` – при нормальном завершении. Системные вызовы, возвращающие указатели, обычно для идентификации ошибочной ситуации пользуются значением `NULL`. Для точного определения причины ошибки С-интерфейс предоставляет глобальную переменную `errno`, описанную в файле `<errno.h>` вместе с ее возможными значениями и их краткими определениями. Анализировать значение переменной `errno` необходимо сразу после возникновения ошибочной ситуации, так как успешно завершившиеся системные вызовы не изменяют ее значения. Для получения символьной информации об ошибке на стандартном выводе программы для ошибок (по умолчанию экран терминала) может применяться стандартная UNIX-функция `perror()`.

Прототип и описание функции perror()

```
#include <stdio.h>
void perror(char *str);
```

Функция `perror()` предназначена для вывода сообщения об ошибке, соответствующего значению системной переменной `errno` на стандартный поток вывода ошибок. Функция печатает содержимое строки `str` (если параметр `str` не равен `NULL`), двоеточие, пробел и текст сообщения, соответствующий возникшей ошибке, с последующим символом перевода

строки `('\n')`.

3. Понятия login и password

ОС UNIX является многопользовательской ОС. Для обеспечения безопасной работы пользователей и целостности системы доступ к ней должен быть санкционирован. Для каждого пользователя, которому разрешен вход в систему, заводится специальное регистрационное имя – username или login и сохраняется специальный пароль – password, соответствующий этому имени. Как правило, при заведении нового пользователя начальное значение пароля для него задает системный администратор. После первого входа в систему пользователь должен изменить начальное значение пароля с помощью специальной команды.

3-1. Вход в систему и смена пароля

Если в системе установлена графическая оболочка наряду с обычными алфавитно-цифровыми терминалами, лучше всего это сделать с алфавитно-цифрового терминала или его эмулятора. На экране появляется надпись, предлагающая ввести регистрационное имя, как правило, это «`login:`». Набрав свое регистрационное имя, нажмите клавишу <Enter>. Система запросит у вас пароль, соответствующий введенному имени, выдав специальное приглашение – обычно «`Password:`». **Вводимый пароль на экране не отображается, поэтому набирайте его аккуратно!** Если все было сделано правильно, у вас на экране появится приглашение к вводу команд ОС.

Пароль, установленный системным администратором, необходимо сменить (чаще всего это команда `passwd` или `yppasswd`). В большинстве UNIX-образных систем требуется, чтобы новый пароль имел не менее шести символов и содержал, по крайней мере, две не буквы и две не цифры.

Придумайте новый пароль и хорошенько его запомните. Пароли в ОС хранятся в закодированном виде, и если вы его забыли, никто не сможет помочь вам его вспомнить. Единственное, что может сделать системный администратор, так это установить вам новый пароль.

Введите команду для смены пароля. Обычно система просит сначала набрать старый пароль, затем ввести новый и подтвердить правильность его набора.

4. Простейшие понятия об устройстве файловой системы в UNIX. Полные и относительные имена файлов

В ОС UNIX существует три базовых понятия: **«процесс»**, **«файл»** и **«пользователь»**. Понятие «процесс» характеризует динамическую сторону происходящего в ВС. Понятие «файл» характеризует статическую сторону ВС.

Из предыдущего опыта работы вы уже имеете некоторое представление о файле, как об именованном наборе данных, хранящемся где-нибудь на магнитных дисках или лентах.

Все файлы, доступные в ОС UNIX, как и в уже известных вам ОС, объединяются в древовидную логическую структуру. Файлы могут объединяться в **каталоги** или **директории**. Не существует файлов, которые не входили бы в состав какой-либо директории. Директории в свою очередь могут входить в состав других директорий. Допускается существование пустых директорий, в которые не входит ни один файл, и ни одна другая директория (рис. 1.1).

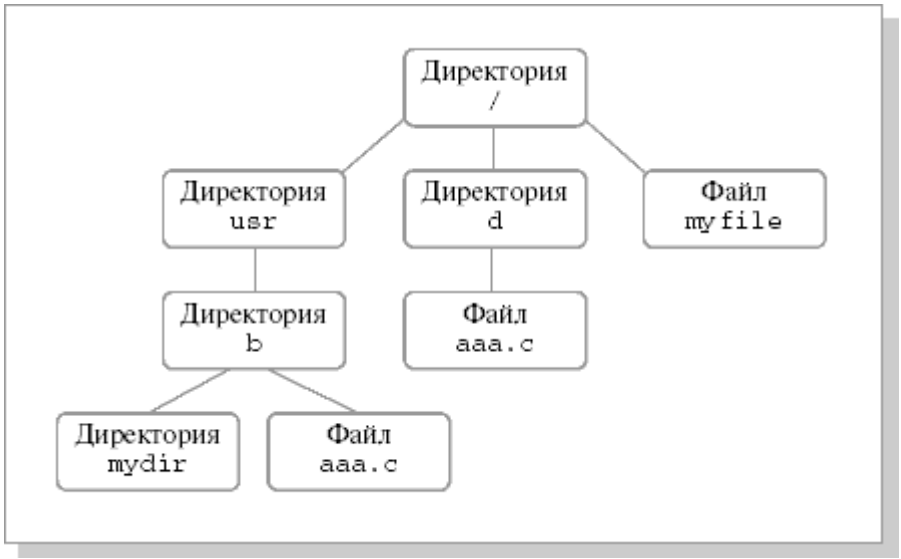

Рис. 1.1. Пример структуры файловой системы

Среди всех директорий существует только одна директория, которая не входит в состав других директорий – **корневая**. В файловой системе UNIX присутствует, по крайней мере, два типа файлов: обычные файлы, которые могут содержать тексты программ, исполняемый код, данные и т.д. – **регулярные файлы**, и директории. Каждому файлу (регулярному или директории) должно быть присвоено имя. В различных версиях ОС UNIX существуют те или иные ограничения на построение имени файла. В стандарте POSIX на интерфейс системных вызовов для ОС UNIX содержится три ограничения:

- Нельзя создавать имена большей длины, чем это предусмотрено операционной системой (для Linux – 255 символов).
- Нельзя использовать символ `NUL` (не путать с указателем `NULL!`) – он же символ с нулевым кодом, он же признак конца строки в языке С.
- Нельзя использовать символ `'/'`.

Нежелательно также применять символы «звездочка» – «`*`», «знак вопроса» – «`?`», «кавычка» – «`\"`», «апостроф» – «`\'`», «пробел» – «` `» и «обратный слэш» – «`\\`».

Единственным исключением является корневая директория, которая **всегда** имеет имя «`/`». Эта же директория представляет собой единственный файл, который должен иметь уникальное имя во всей файловой системе. Для всех остальных файлов имена должны быть уникальными только в рамках той директории, в которую они непосредственно входят. Два файла с именами «`aaa.c`», входящими в директории «`b`» и «`c`» (рис. 1.1), отличаются с помощь понятия полного имени файла.

Построив путь от корневой вершины дерева файлов к интересующему нас файлу, например, «`/ usr b aaa.c`». В этой последовательности первым будет всегда стоять имя корневой директории, а последним – имя интересующего нас файла. Отделим имена узлов друг от друга в этой записи не пробелами, а символами «`/`», за исключением имени корневой директории и следующего за ним имени («`/usr/b/aaa.c`»). Полученная запись однозначно идентифицирует файл во всей логической конструкции файловой системы. Такая запись и получила название полного имени файла.

5. Понятие о текущей директории. Команда pwd. Относительные имена файлов

Полные имена файлов могут включать в себя достаточно много имен директорий и могут быть очень длинными, с ними не всегда удобно работать. Однако, существуют такие понятия как текущая или рабочая директория и относительное имя файла.

Для каждой работающей программы в ОС, включая командный интерпретатор (`shell`), который обрабатывает вводимые команды и высвечивает приглашение к их вводу, одна из директорий в логической структуре файловой системы назначается текущей или рабочей для данной программы. Узнать, какая директория является текущей для вашего командного интерпретатора, можно с помощью команды ОС `pwd`.

Синтаксис и описание команды pwd

```
pwd
```

Команда `pwd` выводит полное имя текущей директории для работающего

командного интерпретатора.

Зная текущую директорию, мы можем проложить путь по графу файлов от текущей директории к интересующему нас файлу. Запишем последовательность узлов, которые встретятся на этом пути, следующим образом. Узел, соответствующий текущей директории, в запись не включаем. При движении по направлению к корневому каталогу каждый узел будем обозначать двумя символами «точка» – «..», а при движении по направлению от корневого каталога будем записывать имя встретившегося узла. Разделим обозначения, относящиеся к разным узлам в этой записи, символами «/». Полученную строку принято называть относительным именем файла. Относительные имена файлов меняются при смене рабочего каталога. Так, в нашем примере, если рабочий каталог – это директория «/d» , то для файла «/usr/b/aaa.c» относительным именем будет «../usr/b/aaa.c», а если рабочий каталог – это директория «/usr/b», то его относительное имя – «aaa.c».

Для полноты картины имя текущего каталога можно вставлять в относительное имя файла, обозначая текущий каталог одиночным символом «точка» – «.». Тогда наши относительные имена будут выглядеть так: «./../usr/b/aaa.c» и «./aaa.c».

Программы, запущенные с помощью командного интерпретатора, будут иметь в качестве рабочей директории его рабочую директорию, если внутри этих программ не изменить ее расположение с помощью специального системного вызова.

3-2. Домашняя директория пользователя и ее определение

Для каждого нового пользователя в системе заводится специальная директория, которая становится текущей сразу после его входа в систему. Эта директория получила название домашней директории пользователя. Воспользуйтесь командой pwd для определения своей домашней директории.

6. Команда man – универсальный справочник

Если вам требоваться информация о том, что делает та или иная команда или системный вызов, какие у них параметры и опции, для чего предназначены некоторые системные файлы, каков их формат и т.д., то для получения полной информации мы можете найти ее в руководстве по ОС UNIX - Manual. Большая часть информации в UNIX Manual доступна в интерактивном режиме с помощью утилиты man.

Пользоваться утилитой man достаточно просто – наберите команду

```
man имя
```

где имя – это имя интересующей вас команды, утилиты, системного вызова, библиотечной функции или файла. Посмотреть информацию о команде pwd.

Чтобы пролистать страницу полученного описания, если оно не поместилось на экране полностью, следует нажать клавишу <пробел>. Для прокрутки одной строки воспользуйтесь клавишей <Enter>. Вернуться на страницу назад позволит одновременное нажатие клавиш <Ctrl> и <b>. Выйти из режима просмотра информации можно с помощью клавиши <q>.

Иногда имена команд интерпретатора и системных вызовов или какие-либо еще имена совпадают. Тогда чтобы найти интересующую вас информацию, необходимо задать утилите man категорию, к которой относится эта информация (номер раздела). Например, в Linux принято следующее разделение:

1. Исполняемые файлы или команды интерпретатора.
2. Системные вызовы.
3. Библиотечные функции.
4. Специальные файлы (обычно файлы устройств).
5. Формат системных файлов и принятые соглашения.
6. Игры (обычно отсутствуют).
7. Макропакеты и утилиты – такие как сам man.
8. Команды системного администратора.
9. Подпрограммы ядра (нестандартный раздел).

Если вы знаете раздел, к которому относится информация, то утилиту man можно вызвать в Linux с дополнительным параметром

```
man номер_раздела имя
```

В других ОС этот вызов может выглядеть иначе. Для получения точной информации о разбиении на разделы, форме указания номера раздела и дополнительных возможностях утилиты man наберите команду

```
man man
```

7. Команды cd – для смены текущей директории и ls – для просмотра состава директории

Для смены текущей директории командного интерпретатора можно воспользоваться командой cd (change directory). Для этого необходимо набрать команду

```
cd имя_директории
```

где имя_директории – полное или относительное имя директории, которую вы хотите сделать текущей. Команда cd без параметров сделает текущей директорией вашу домашнюю директорию.

Просмотреть содержимое текущей или любой другой директории можно, воспользовавшись командой `ls` (от list). Если ввести ее без параметров, эта команда распечатает вам список файлов, находящихся в текущей директории. Если же в качестве параметра задать полное или относительное имя директории:

```
ls имя_директории
```

то она распечатает список файлов в указанной директории. Надо отметить, что в полученный список не войдут файлы, имена которых начинаются с символа «точка» – «.». Такие файлы обычно создаются различными системными программами для своих целей (например, для настройки). Посмотреть полный список файлов можно, дополнительно указав команде `ls` опцию `-a`, т.е. набрав ее в виде

```
ls -a
```

или

```
ls -a имя_директории
```

У команды `ls` существует и много других опций, для получения полной информации о команде `ls` воспользуйтесь утилитой `man`.

3-3. Путешествие по структуре файловой системы

Пользуясь командами `cd`, `ls` и `pwd`, попутешествуйте по структуре файловой системы и просмотрите ее содержимое. Возможно, зайти в некоторые директории или посмотреть их содержимое вам не удастся. Это связано с работой механизма защиты файлов и директорий. В конце путешествия вернитесь в свою домашнюю директорию.

8. Команда cat и создание файла. Перенаправление ввода и вывода

Для просмотра содержимого небольшого текстового файла на экране можно воспользоваться командой `cat`. Если набрать ее в виде

```
cat имя_файла
```

то на экран выплеснется все его содержимое.

Внимание! Не пытайтесь рассматривать на экране содержимое директорий – все равно не получится! Не пытайтесь просматривать содержимое неизвестных файлов, особенно если вы не знаете, текстовый он или бинарный. Вывод на экран бинарного файла может привести к непредсказуемому поведению вашего терминала.

Если даже ваш файл и текстовый, но большой, то все равно вы увидите только его последнюю страницу. Большой текстовый файл удобнее рассматривать с помощью утилиты `more`.

Команда `cat` интересна с другой точки зрения. Если мы в качестве ее параметров задать не одно имя, а имена нескольких файлов

```
cat файл1 файл2 ... файлN
```

то система выдаст на экран их содержимое в указанном порядке. Вывод команды `cat` можно перенаправить с экрана терминала в какой-нибудь файл, воспользовавшись символом перенаправления выходного потока данных – знаком «больше» – «>». Команда

```
cat файл1 файл2 ... файлN > файл_результата
```

сольет содержимое всех файлов, чьи имена стоят перед знаком «>», воедино в `файл_результата` – конкатенирует их (concatenate). Прием перенаправления выходных данных со стандартного потока вывода (экрана) в файл является стандартным для всех команд, выполняемых командным интерпретатором. Вы можете получить файл, содержащий список всех файлов текущей директории, если выполните команду `ls -a` с перенаправлением выходных данных

```
ls -a > новый_файл
```

Если имена входных файлов для команды `cat` не заданы, то она будет использовать в качестве входных данных информацию, которая вводится с клавиатуры, до тех пор, пока вы не наберете признак окончания ввода – комбинацию клавиш <CTRL> и <d>.

Таким образом, команда

```
cat > новый_файл
```

позволяет создать новый текстовый файл с именем `новый_файл` и содержимым, которое пользователь введет с клавиатуры.

Заметим, что наряду с перенаправлением выходных данных существует способ перенаправить входные данные. Если во время выполнения некоторой команды требуется ввести данные с клавиатуры, можно положить их заранее в файл, а затем перенаправить стандартный ввод этой команды с помощью знака «меньше» – «<» и следующего за ним имени файла с входными данными.

3-4. Создание файла с помощью команды cat

Убедитесь, что вы находитесь в своей домашней директории, и создайте с помощью команды `cat` новый текстовый файл. Просмотрите его содержимое.

9. Простейшие команды работы с файлами – cp, rm, mkdir, mv

Для нормальной работы с файлами необходимо не только уметь создавать файлы, просматривать их содержимое и перемещаться по логическому дереву файловой системы. Нужно уметь создавать собственные поддиректории, копировать и удалять файлы, переименовывать их.

Для создания новой поддиректории используется команда `mkdir` (сокращение от make directory). В простейшем виде команда выглядит следующим образом:

```
mkdir имя_директории
```

где `имя_директории` – полное или относительное имя создаваемой директории.

Для копирования файлов может использоваться команда `cp` (сокращение от copy). Команда `cp` умеет копировать не только отдельный файл, но и набор файлов, и даже директорию целиком вместе со всеми входящими в нее поддиректориями (рекурсивное копирование). Для задания набора файлов могут использоваться шаблоны имен файлов. Точно так же шаблон имени может быть использован и в командах переименования файлов и их удаления.

Синтаксис и описание команды cp

```
cp файл_источник файл_назначения
```

служит для копирования одного файла с именем `файл_источник` в файл с именем `файл_назначения`.

```
cp файл1 файл2 ... файлN дир_назначения
```

служит для копирования файла или файлов с именами `файл1, файл2, ... файлN` в уже существующую директорию с именем `дир_назначения` под своими именами. Вместо имен копируемых файлов могут использоваться их шаблоны.

```
cp -r дир_источник дир_назначения
```

служит для рекурсивного копирования одной директории с `именем`

дир_источник в новую директорию с именем дир_назначения. Если директория дир_назначения уже существует, то мы получаем команду cp в следующей форме

```
cp -r дир1 дир2 ... дирN дир_назначения
```

Такая команда служит для рекурсивного копирования директории или директорий с именами дир1, дир2, ... дирN в уже существующую директорию с именем дир_назначения под своими собственными именами. Вместо имен копируемых директорий могут использоваться их шаблоны.

Шаблоны имен файлов

Шаблоны имен файлов могут применяться в качестве параметра для задания набора имен файлов во многих командах операционной системы. При использовании шаблона просматривается вся совокупность имен файлов, находящихся в файловой системе, и те имена, которые удовлетворяют шаблону, включаются в набор. В общем случае шаблоны могут задаваться с использованием следующих метасимволов:

* – соответствует всем цепочкам литер, включая пустую;

? – соответствует всем одиночным литерам;

[...] – соответствует любой литере, заключенной в скобки. Пара литер, разделенных знаком минус, задает диапазон литер.

Например, шаблону *.c удовлетворяют все файлы текущей директории, чьи имена заканчиваются на .c. Шаблону [a-d]* удовлетворяют все файлы текущей директории, чьи имена начинаются с букв a, b, c, d. Существует одно ограничение на использование метасимвола * в начале имени файла, например, в случае шаблона *c. Для таких шаблонов имена файлов, начинающиеся с символа точка, считаются не удовлетворяющими шаблону.

Для удаления файлов или директорий применяется команда rm (сокращение от remove). Если вы хотите удалить один или несколько регулярных файлов, то простейший вид команды rm будет выглядеть следующим образом:

```
rm файл1 файл2 ... файлN
```

где файл1, файл2, ... файлN – полные или относительные имена регулярных файлов, которые требуется удалить. Вместо имен файлов могут использоваться их шаблоны. Если вы хотите удалить одну или несколько

директорий вместе с их содержимым (рекурсивное удаление), то к команде добавляется опция `-r`:

```
rm -r дир1 дир2 ... дирN
```

где `дир1, дир2, ... дирN` – полные или относительные имена директорий, которые нужно удалить. Вместо непосредственно имен директорий также могут использоваться их шаблоны.

Командой удаления файлов и директорий следует пользоваться с осторожностью. Удаленную информацию восстановить невозможно. Если вы системный администратор и ваша текущая директория **– это корневая директория НЕ ВЫПОЛНЯЙТЕ команду** `rm -r *`**!**

Для переименования файла или его перемещения в другой каталог применяется команда `mv` (сокращение от move). Для задания имен перемещаемых файлов в ней тоже можно использовать их шаблоны.

Синтаксис и описание команды mv

```
mv имя_источника имя_назначения
```

служит для переименования или перемещения одного файла (неважно, регулярного или директории) с именем `имя_источника` в файл с именем `имя_назначения`. При этом перед выполнением команды файла с именем `имя_назначения` существовать не должно.

```
mv имя1 имя2 ... имяN дир_назначения
```

служит для перемещения файла или файлов (регулярных файлов или директорий) с именами `имя1, имя2, ... имяN` в уже существующую директорию с именем `дир_назначения` под собственными именами. Вместо имен перемещаемых файлов могут использоваться их шаблоны.

10. История редактирования файлов – ed, vi

Когда появились первые варианты ОС UNIX, устройства ввода и отображения информации существенно отличались от существующих сегодня. На клавиатурах присутствовали только алфавитно-цифровые клавиши (не было даже клавиш курсоров), а дисплеи не предполагали экранного редактирования. Поэтому первый редактор ОС UNIX – редактор `ed` – требовал от пользователя строгого указания того, что и как будет редактироваться с помощью специальных команд. Так, например, для замены первого сочетания символов `«ra»` на `«ru»` в одиннадцатой строке редактируемого файла потребовалось бы ввести команду

```
11 s/ra/ru
```

Редактор ed, по существу, являлся построчечным редактором. Впоследствии появился экранный редактор – vi, однако и он требовал строгого указания того, что и как в текущей позиции на экране мы должны сделать, или каким образом изменить текущую позицию, с помощью специальных команд, соответствующих алфавитно-цифровым клавишам. Эти редакторы могут показаться сейчас анахронизмами, но они до сих пор входят в состав всех вариантов UNIX и иногда (например, при работе с удаленной машиной по медленному каналу связи) являются единственным средством, позволяющим удаленно редактировать файл.

3-5. Система Midnight Commander – mc. Встроенный mc редактор и редактор joe

Работа в UNIX исключительно на уровне командного интерпретатора и встроенных редакторов далека от уже привычных для нас удобств. Однако, существует разнообразные пакеты, облегчающие задачу пользователя в UNIX. К таким пакетам следует отнести Midnight Commander – аналог программ Norton Commander для DOS и FAR для Windows 9x и NT – со своим встроенным редактором, запускаемый командой `mc`, и экранный редактор `joe`. Еще больше возможностей предоставляет многофункциональный текстовый редактор emacs.

Войдите в `mc` создайте и отредактируйте ваши файлы в вами созданных директориях.

11. Пользователь и группа. Команды chown и chgrp. Права доступа к файлу

Как уже говорилось, для входа в ОС UNIX каждый пользователь должен быть зарегистрирован в ней под определенным именем. Вычислительные системы не умеют оперировать именами, поэтому каждому имени пользователя в системе соответствует некоторое числовое значение – его идентификатор – UID (user identificator).

Все пользователи в системе делятся на группы. Например, студенты одной учебной группы могут составлять отдельную группу пользователей. Группы пользователей также получают свои имена и соответствующие идентификационные номера – GID (group identificator). В одних версиях UNIX каждый пользователь может входить только в одну группу, в других – в несколько групп.

Для каждого файла, созданного в файловой системе, запоминаются имена его хозяина и группы хозяев. Заметим, что группа хозяев не обязательно должна быть группой, в которую входит хозяин. Упрощенно можно считать, что в ОС Linux при создании файла его хозяином становится пользователь,

создавший файл, а его группой хозяев – группа, к которой этот пользователь принадлежит. Впоследствии хозяин файла или системный администратор могут передать его в собственность другому пользователю или изменить его группу хозяев с помощью команд `chown` и `chgrp`.

Синтаксис и описание команды chown

```
chown owner файл1 файл2 ... файлN
```

Команда `chown` предназначена для изменения собственника (хозяина) файлов. Нового собственника файла могут назначить только предыдущий собственник файла или системный администратор.

Параметр `owner` задает нового собственника файла в символьном виде, как его `username`, или в числовом виде, как его `UID`.

Параметры `файл1, файл2, ... файлN` – это имена файлов, для которых производится изменение собственника. Вместо имен могут использоваться их шаблоны.

Синтаксис и описание команды chgrp

```
chgrp group файл1 файл2 ... файлN
```

Команда `chgrp` предназначена для изменения группы собственников (хозяев) файлов. Новую группу собственников файла могут назначить только собственник файла или системный администратор.

Параметр `group` задает новую группу собственников файла в символьном виде, как имя группы, или в числовом виде, как ее `GID`.

Параметры `файл1, файл2, ... файлN` – это имена файлов, для которых производится изменение группы собственников. Вместо имен могут использоваться их шаблоны.

Итак, для каждого файла выделяется три категории пользователей:

- Пользователь, являющийся хозяином файла;
- Пользователи, относящиеся к группе хозяев файла;
- Все остальные пользователи.

Для каждой из этих категорий хозяин файла может определить различные права доступа к файлу. Различают три вида прав доступа: право на чтение файла – `r` (от слова read), право на модификацию файла – `w` (от слова write) и право на исполнение файла — `x` (от слова execute). Для регулярных файлов

смысл этих прав совпадает с указанным выше. Для директорий он несколько иной. Право чтения для каталогов позволяет читать имена файлов, находящихся в этом каталоге (и только имена). Поскольку «исполнять» директорию бессмысленно (как, впрочем, и неисполняемый регулярный файл), право доступа на исполнение для директорий меняет смысл: наличие этого права позволяет получить дополнительную информацию о файлах, входящих в каталог (их размер, кто их хозяин, дата создания и т.д.). Без этого права вы не сможете ни читать содержимое файлов, лежащих в директории, ни модифицировать их, ни исполнять. Право на исполнение также требуется для директории, чтобы сделать ее текущей, а также для всех директорий на пути к ней. Право записи для директории позволяет изменять ее содержимое: создавать и удалять в ней файлы, переименовывать их. Отметим, что для удаления файла достаточно иметь права записи и исполнения для директории, в которую входит данный файл, независимо от прав доступа к самому файлу.

12. Команда ls с опциями -al. Использование команд chmod и umask

Получить подробную информацию о файлах в некоторой директории, включая имена хозяина, группы хозяев и права доступа, можно с помощью уже известной нам команды `ls` с опциями -`al`. В выдаче этой команды третья колонка слева содержит имена пользователей хозяев файлов, а четвертая колонка слева – имена групп хозяев файла. Самая левая колонка содержит типы файлов и права доступа к ним. Тип файла определяет первый символ в наборе символов. Если это символ '`d`', то тип файла – директория, если там стоит символ '-', то это – регулярный файл. Следующие три символа определяют права доступа для хозяина файла, следующие три – для пользователей, входящих в группу хозяев файла, и последние три – для всех остальных пользователей. Наличие символа (`r`, `w` или `x`), соответствующего праву, для некоторой категории пользователей означает, что данная категория пользователей обладает этим правом.

Вызовите команду `ls -al` для своей домашней директории и проанализируйте ее вывод.

Хозяин файла может изменять права доступа к нему, пользуясь командой `chmod`.

Синтаксис и описание команды chmod

```
chmod [who] { + | - | = } [perm] файл1 файл2 ... файлN
```

Команда `chmod` предназначена для изменения прав доступа к одному или нескольким файлам. Права доступа к файлу могут менять только собственник

(хозяин) файла или системный администратор.

Параметр `who` определяет, для каких категорий пользователей устанавливаются права доступа. Он может представлять собой один или несколько символов:

`a` – установка прав доступа для всех категорий пользователей. Если параметр `who` не задан, то по умолчанию применяется `a`. При определении прав доступа с этим значением заданные права устанавливаются с учетом значения маски создания файлов;
`u` – установка прав доступа для собственника файла;
`g` – установка прав доступа для пользователей, входящих в группу собственников файла;
`o` – установка прав доступа для всех остальных пользователей.

Операция, выполняемая над правами доступа для заданной категории пользователей, определяется одним из следующих символов:

`+` – добавление прав доступа;
`-` – отмена прав доступа;
`=` – замена прав доступа, т.е. отмена всех существовавших и добавление перечисленных.

Если параметр `perm` не определен, то все существовавшие права доступа отменяются.

Параметр `perm` определяет права доступа, которые будут добавлены, отменены или установлены взамен соответствующей командой. Он представляет собой комбинацию следующих символов или один из них:

`r` – право на чтение;
`w` – право на модификацию;
`x` – право на исполнение.

Параметры `файл1, файл2, ... файлN` – это имена файлов, для которых производится изменение прав доступа. Вместо имен могут использоваться их шаблоны.

Создайте новый файл и посмотрите на права доступа к нему, установленные системой при его создании. Чем руководствуется ОС при назначении этих прав? Она использует для этого маску создания файлов для программы, которая файл создает. Изначально для программы-оболочки она имеет некоторое значение по умолчанию.

Маска создания файлов текущего процесса

Маска создания файлов текущего процесса (umask) используется системными вызовами open() и mknod() при установке начальных прав доступа для вновь создаваемых файлов или FIFO. Младшие 9 бит маски создания файлов соответствуют правам доступа пользователя, создающего файл, группы, к которой он принадлежит, и всех остальных пользователей так, как записано ниже с применением восьмеричных значений:

0400 – право чтения для пользователя, создавшего файл;
0200 – право записи для пользователя, создавшего файл;
0100 – право исполнения для пользователя, создавшего файл;
0040 – право чтения для группы пользователя, создавшего файл;
0020 – право записи для группы пользователя, создавшего файл;
0010 – право исполнения для группы пользователя, создавшего файл;
0004 – право чтения для всех остальных пользователей;
0002 – право записи для всех остальных пользователей;
0001 – право исполнения для всех остальных пользователей.

Установление значения какого-либо бита равным 1 запрещает инициализацию соответствующего права доступа для вновь создаваемого файла. Значение маски создания файлов может изменяться с помощью системного вызова umask() или команды umask. Маска создания файлов наследуется процессом-ребенком при порождении нового процесса системным вызовом fork() и входит в состав неизменяемой части системного контекста процесса при системном вызове exec(). В результате этого наследования изменение маски с помощью команды umask окажет влияние на атрибуты доступа к вновь создаваемым файлам для всех процессов, порожденных далее командной оболочкой.

Изменить текущее значение маски для программы-оболочки или посмотреть его можно с помощью команды umask.

Если вы хотите изменить его необходимо выйти из mc, выполнить команду umask и запустить mc снова. Маска создания файлов не сохраняется между сеансами работы в системе. При новом входе в систему значение маски снова будет установлено по умолчанию.

Синтаксис и описание команды umask

```
umask [value]
```

Команда umask предназначена для изменения маски создания файлов командной оболочки или просмотра ее текущего значения. При отсутствии параметра команда выдает значение установленной маски создания файлов в

восьмеричном виде. Для установления нового значения оно задается как параметр `value` в восьмеричном виде.

13. Системные вызовы getuid и getgid. Компиляция программ на языке С в UNIX и их запуск

Узнать идентификатор пользователя, запустившего программу на исполнение, – UID и идентификатор группы, к которой он относится, – GID можно с помощью системных вызовов `getuid()` и `getgid()`, применив их внутри этой программы.

Прототипы и описание системных вызовов getuid() и getgid()

```
#include <sys/types.h>
#include <unistd.h>
uid_t getuid(void);
gid_t getgid(void);
```

Системный вызов `getuid` возвращает идентификатор пользователя для текущего процесса.

Системный вызов `getgid` возвращает идентификатор группы пользователя для текущего процесса.

Типы данных `uid_t` и `gid_t` являются синонимами для одного из целочисленных типов языка С.

Разберем, как можно откомпилировать программу на языке С и запускать ее. Для компиляции программ в Linux можно воспользоваться компилятором `gcc`.

Для того чтобы он нормально работал, необходимо, чтобы исходные файлы, содержащие текст программы, имели имена, заканчивающиеся на `.c`.

В простейшем случае откомпилировать программу можно, запуская компилятор командой

```
gcc имя_исходного_файла
```

Если программа была написана без ошибок, то компилятор создаст исполняемый файл с именем `a.out`. Изменить имя создаваемого исполняемого файла можно, задав его с помощью опции `-o`:

```
gcc имя_исходного_файла -o имя_исполняемого_файла
```

Компилятор `gcc` имеет несколько сотен возможных опций (см. UNIX Manual).

Запустить программу на исполнение можно, набрав имя исполняемого файла и нажав клавишу <Enter>.

3-6. Написание, компиляция и запуск программы с использованием системных вызовов getuid() и getgid()

Напишите, откомпилируйте и запустите программу, которая печатала бы идентификатор пользователя, запустившего программу, и идентификатор его группы.

Семинар #2: Процессы в операционной системе UNIX

Понятие процесса в UNIX, его контекст. Идентификация процесса. Состояния процесса. Краткая диаграмма состояний. Иерархия процессов. Системные вызовы getpid(), getppid(). Создание процесса в UNIX. Системный вызов fork(). Завершение процесса. Функция exit(). Параметры функции main() в языке С. Переменные среды и аргументы командной строки. Изменение пользовательского контекста процесса. Семейство функций для системного вызова exec().

1. Понятие процесса в UNIX. Его контекст

Все построение операционной системы UNIX основано на использовании концепции процессов. Контекст процесса складывается из пользовательского контекста и контекста ядра (см. рис. 2-1).

Под пользовательским контекстом процесса понимают код и данные, расположенные в адресном пространстве процесса. Все данные подразделяются на:

- инициализируемые неизменяемые данные (например, константы);
- инициализируемые изменяемые данные (все переменные, начальные значения которых присваиваются на этапе компиляции);
- неинициализируемые изменяемые данные (все статические переменные, которым не присвоены начальные значения на этапе компиляции);
- стек пользователя;
- данные, расположенные в динамически выделяемой памяти (например, с помощью функций `malloc()`, `calloc()`, `realloc()`).

Исполняемый код и инициализируемые данные составляют содержимое файла программы, который исполняется в контексте процесса. Пользовательский стек применяется при работе процесса в пользовательском режиме (user-mode).

Рис. 2.1. Контекст процесса в UNIX

Под понятием «контекст ядра» объединяются системный контекст и регистровый контекст. Мы будем выделять в контексте ядра стек ядра, который используется при работе процесса в режиме ядра (kernel mode), и данные ядра, хранящиеся в структурах, являющихся аналогом блока управления процессом – PCB. В данные ядра входят: идентификатор пользователя – `UID`, групповой идентификатор пользователя – `GID`, идентификатор процесса – PID, идентификатор родительского процесса – PPID.

2. Идентификация процесса

Каждый процесс в ОС получает уникальный идентификационный номер – PID (process identificator). При создании нового процесса операционная система пытается присвоить ему свободный номер больший, чем у процесса, созданного перед ним. Если таких свободных номеров не оказывается (например, мы достигли максимально возможного номера для процесса), то операционная система выбирает минимальный номер из всех свободных номеров. В операционной системе Linux присвоение идентификационных номеров процессов начинается с номера 0, который получает процесс *`kernel`* при старте операционной системы. Этот номер впоследствии не может быть присвоен никакому другому процессу. Максимально возможное значение для номера процесса в Linux на базе 32-разрядных процессоров Intel составляет 2^{31}-1.

3. Состояния процесса. Краткая диаграмма состояний

Модель состояний процессов в операционной системе UNIX представляет собой детализацию модели состояний, принятой нами ранее. Краткая диаграмма состояний процессов в операционной системе UNIX изображена на рисунке 2.2.

Рис. 2.2. Сокращенная диаграмма состояний процесса в UNIX

Как мы видим, состояние процесса ***исполнение*** расщепилось на два состояния: ***исполнение в режиме ядра*** и ***исполнение в режиме пользователя***. В состоянии ***исполнение в режиме пользователя*** процесс выполняет прикладные инструкции пользователя. В состоянии ***исполнение в режиме ядра*** выполняются инструкции ядра операционной системы в контексте текущего процесса (например, при обработке системного вызова или прерывания). Из состояния ***исполнение в режиме пользователя*** процесс не может непосредственно перейти в состояния ***ожидание***, ***готовность*** и ***закончил исполнение***. Такие переходы возможны только через промежуточное состояние «***исполняется в режиме ядра***». Также запрещен прямой переход из состояния ***готовность*** в состояние ***исполнение в режиме пользователя***.

4. Иерархия процессов

В ОС UNIX все процессы, кроме одного, создающегося при старте операционной системы, могут быть порождены только какими-либо другими процессами. В качестве прародителя всех остальных процессов в подобных UNIX системах могут выступать процессы с номерами `1` или `0`. В ОС Linux таким родоначальником, существующим только при загрузке системы, является процесс *`kernel`* с идентификатором `0`.

Таким образом, все процессы в UNIX связаны отношениями процесс-родитель – процесс-ребенок и образуют генеалогическое дерево процессов. Для сохранения целостности генеалогического дерева в ситуациях, когда процесс-родитель завершает свою работу до завершения выполнения процесса-ребенка, идентификатор родительского процесса в данных ядра процесса-ребенка (PPID – parent process identificator) изменяет свое значение на значение `1`, соответствующее идентификатору процесса *`init`*, время жизни которого определяет время функционирования операционной системы. Тем самым процесс *`init`* как бы усыновляет осиротевшие процессы. Наверное, логичнее было бы заменять PPID не на значение `1`, а на значение идентификатора ближайшего существующего процесса-прародителя умершего процесса-родителя, но в UNIX почему-то такая схема не реализована.

5. Системные вызовы getppid() и getpid()

Данные ядра, находящиеся в контексте ядра процесса, не могут быть прочитаны процессом непосредственно. Для получения информации о них процесс должен совершить соответствующий системный вызов. Значение идентификатора текущего процесса может быть получено с помощью системного вызова `getpid()`, а значение идентификатора родительского процесса для текущего процесса – с помощью системного вызова `getppid()`. Прототипы этих системных вызовов и соответствующие типы данных описаны в системных файлах `<sys/types.h>` и `<unistd.h>`. Системные вызовы не имеют параметров и возвращают идентификатор текущего процесса и идентификатор родительского процесса соответственно.

Прототипы и описание системных вызовов getpid() и getppid()

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpid(void);
pid_t getppid(void);
```

Системный вызов `getpid` возвращает идентификатор текущего процесса.

Системный вызов `getppid` возвращает идентификатор процесса-родителя для текущего процесса.

Тип данных `pid_t` является синонимом для одного из целочисленных типов языка C.

3-1. Написание программы с использованием getpid() и getppid()

В качестве примера использования системных вызовов `getpid()` и `getppid()` самостоятельно напишите программу, печатающую значения

PID и PPID для текущего процесса. Запустите ее несколько раз подряд. Посмотрите, как меняется идентификатор текущего процесса. Объясните наблюдаемые изменения.

6. Создание процесса в UNIX. Системный вызов fork()

В операционной системе UNIX новый процесс может быть порожден единственным способом – с помощью системного вызова fork(). При этом вновь созданный процесс будет являться практически полной копией родительского процесса. У порожденного процесса по сравнению с родительским процессом (на уровне уже полученных знаний) изменяются значения следующих параметров:

- идентификатор процесса – PID;
- идентификатор родительского процесса – PPID.

Дополнительно может измениться поведение порожденного процесса по отношению к некоторым сигналам.

Прототип и описание системного вызова для порождения нового процесса

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```

Системный вызов fork служит для создания нового процесса в операционной системе UNIX. Процесс, который инициировал системный вызов fork, принято называть родительским процессом (parent process). Вновь порожденный процесс принято называть процессом-ребенком (child process). Процесс-ребенок является почти полной копией родительского процесса. У порожденного процесса по сравнению с родительским изменяются значения следующих параметров:

- идентификатор процесса;
- идентификатор родительского процесса;
- время, оставшееся до получения сигнала SIGALRM;
- сигналы, ожидавшие доставки родительскому процессу, не будут доставляться порожденному процессу.

При однократном системном вызове возврат из него может произойти дважды: один раз в родительском процессе, а второй раз в порожденном процессе. Если создание нового процесса произошло успешно, то в порожденном процессе системный вызов вернет значение 0, а в родительском процессе – положительное значение, равное идентификатору процесса-ребенка. Если создать новый процесс не удалось, то системный вызов вернет

в инициировавший его процесс отрицательное значение.

Системный вызов `fork` является единственным способом породить новый процесс после инициализации операционной системы UNIX.

В процессе выполнения системного вызова `fork()` порождается копия родительского процесса и возвращение из системного вызова будет происходить уже как в родительском, так и в порожденном процессах. Этот системный вызов является единственным, который вызывается один раз, а при успешной работе возвращается два раза (один раз в процессе-родителе и один раз в процессе-ребенке)! После выхода из системного вызова оба процесса продолжают выполнение регулярного пользовательского кода, следующего за системным вызовом.

3-2. Прогон программы с fork() с одинаковой работой родителя и ребенка

Для иллюстрации сказанного рассмотрим следующую программу:

```
/* Программа 02-1.c – пример создания нового процесса с
   одинаковой работой процессов ребенка и родителя */
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
int main() {
    pid_t pid, ppid;
    int a = 0;
    (void)fork();

    /* При успешном создании нового процесса с этого места псевдо
       параллельно начинают работать два процесса: старый и новый */
    /* Перед выполнением следующего выражения значение
       переменной a в обоих процессах равно 0 */

    a = a+1;

    /* Узнаем идентификаторы текущего и родительского
       процесса (в каждом из процессов !!!) */

    pid = getpid();
    ppid = getppid();

    /* Печатаем значения PID, PPID и вычисленное значение
       переменной a (в каждом из процессов !!!) */
    printf("My pid = %d, my ppid = %d,
      result = %d\n", (int)pid, (int)ppid, a);
    return 0;
}
```

Программа 02-1.c – создания нового процесса с одинаковой работой процессов ребенка и родителя.

Наберите эту программу, откомпилируйте ее и запустите на исполнение (лучше всего это делать не из оболочки `mc`, так как она не очень корректно сбрасывает буферы ввода-вывода). Проанализируйте полученный результат.

6. Системный вызов fork() (продолжение)

Для того чтобы после возвращения из системного вызова `fork()` процессы могли определить, кто из них является ребенком, а кто родителем, и, соответственно, по-разному организовать свое поведение, системный вызов возвращает в них разные значения. При успешном создании нового процесса в процесс-родитель возвращается положительное значение, равное идентификатору процесса-ребенка. В процесс-ребенок же возвращается значение `0`. Если по какой-либо причине создать новый процесс не удалось, то системный вызов вернет в инициировавший его процесс значение `-1`. Таким образом, общая схема организации различной работы процесса-ребенка и процесса-родителя выглядит так:

```
pid = fork();
if(pid == -1){
    ...
    /* ошибка */
    ...
} else if (pid == 0){
    ...
    /* ребенок */
    ...
} else {
    ...
    /* родитель */
    ...
}
```

3-3. Написание, компиляция и запуск программы с использованием вызова fork() с разным поведением процессов ребенка и родителя

Измените предыдущую программу с `fork()` так, чтобы родитель и ребенок совершали разные действия (какие – все равно).

7. Завершение процесса. Функция exit()

Существует два способа корректного завершения процесса в программах, написанных на языке С. Первый способ мы использовали до сих пор: процесс корректно завершался по достижении конца функции `main()` или при выполнении оператора `return` в функции `main()`, второй способ применяется при необходимости завершить процесс в каком-либо другом месте программы. Для этого используется функция `exit()` из стандартной библиотеки функций для языка С. При выполнении этой функции происходит сброс всех частично заполненных буферов ввода-вывода с закрытием соответствующих потоков, после чего инициируется системный вызов прекращения работы процесса и перевода его в состояние ***закончил исполнение***.

Возврата из функции в текущий процесс не происходит и функция ничего не возвращает.

Значение параметра функции `exit()` – кода завершения процесса – передается ядру операционной системы и может быть затем получено процессом, породившим завершившийся процесс. На самом деле при достижении конца функции `main()` также неявно вызывается эта функция со значением параметра `0`.

Прототип и описание функции для нормального завершения процесса

```
#include <stdlib.h>
void exit(int status);
```

Функция `exit` служит для нормального завершения процесса. При выполнении этой функции происходит сброс всех частично заполненных буферов ввода-вывода с закрытием соответствующих потоков (файлов, pipe, FIFO, сокетов), после чего инициируется системный вызов прекращения работы процесса и перевода его в состояние закончил исполнение.

Возврата из функции в текущий процесс не происходит, и функция ничего не возвращает.

Значение параметра `status` – кода завершения процесса – передается ядру операционной системы и может быть затем получено процессом, породившим завершившийся процесс. При этом используются только младшие 8 бит параметра, так что для кода завершения допустимы значения от `0` до `255`. По соглашению, код завершения `0` означает безошибочное завершение процесса.

Если процесс завершает свою работу раньше, чем его родитель, и родитель явно не указал, что он не хочет получать информацию о статусе завершения порожденного процесса, то завершившийся процесс не исчезает из системы окончательно, а остается в состоянии ***закончил исполнение*** либо до завершения процесса-родителя, либо до того момента, когда родитель получит эту информацию. Процессы, находящиеся в состоянии ***закончил исполнение***, в операционной системе UNIX принято называть процессами-зомби (`zombie`, `defunct`).

8. Параметры функции main() в языке C. Переменные среды и аргументы командной строки

У функции `main()` в языке C существует три параметра, которые могут быть переданы ей ОС. Полный прототип функции `main()` выглядит следующим образом:

```
int main(int argc, char *argv[],char *envp[]);
```

Первые два параметра при запуске программы на исполнение командной строкой позволяют узнать полное содержание командной строки. Вся командная строка рассматривается как набор слов, разделенных пробелами. Через параметр `argc` передается количество слов в командной строке, которой была запущена программа. Параметр `argv` является массивом указателей на отдельные слова. Так, например, если программа была запущена командой

```
a.out 12 abcd
```

то значение параметра `argc` будет равно `3`, `argv[0]` будет указывать на имя программы — первое слово — «`a.out`», `argv[1]` — на слово «`12`», `argv[2]` — на слово «`abcd`». Так как имя программы всегда присутствует на первом месте в командной строке, то `argc` всегда больше `0`, а `argv[0]` всегда указывает на имя запущенной программы.

Анализируя в программе содержимое командной строки, мы можем предусмотреть ее различное поведение в зависимости от слов, следующих за именем программы. Таким образом, не внося изменений в текст программы, мы можем заставить ее работать по-разному от запуска к запуску. Например, компилятор `gcc`, вызванный командой `gcc 1.c` будет генерировать исполняемый файл с именем `a.out`, а при вызове командой `gcc 1.c -o 1.exe` – файл с именем `1.exe`.

Третий параметр – `envp` – является массивом указателей на параметры окружающей среды процесса. Начальные параметры окружающей среды процесса задаются в специальных конфигурационных файлах для каждого пользователя и устанавливаются при входе пользователя в систему. В дальнейшем они могут быть изменены с помощью специальных команд операционной системы UNIX. Каждый параметр имеет вид: `переменная=строка`. Такие переменные используются для изменения долгосрочного поведения процессов, в отличие от аргументов командной строки. Например, задание параметра `TERM=vt100` может говорить процессам, осуществляющим вывод на экран дисплея, что работать им придется с терминалом `vt100`. Меняя значение переменной среды `TERM`, например на `TERM=console`, мы сообщаем таким процессам, что они должны изменить свое поведение и осуществлять вывод для системной консоли.

Размер массива аргументов командной строки в функции `main()` мы получали в качестве ее параметра. Так как для массива ссылок на параметры окружающей среды такого параметра нет, то его размер определяется другим способом. Последний элемент этого массива содержит указатель `NULL`.

3-4. Написание, компиляция и запуск программы, распечатывающей аргументы командной строки и параметры среды

В качестве примера самостоятельно напишите программу, распечатывающую значения аргументов командной строки и параметров окружающей среды для текущего процесса.

9. Изменение пользовательского контекста процесса. Семейство функций для системного вызова exec()

Для изменения пользовательского контекста процесса применяется системный вызов `exec()`, который пользователь не может вызвать непосредственно. Вызов `exec()` заменяет пользовательский контекст текущего процесса на содержимое некоторого исполняемого файла и устанавливает начальные значения регистров процессора (в том числе устанавливает программный счетчик на начало загружаемой программы). Этот вызов требует для своей работы задания имени исполняемого файла, аргументов командной строки и параметров окружающей среды. Для осуществления вызова программист может воспользоваться одной из шести функций: `execlp()`, `execvp()`, `execl()` и, `execv()`, `execle()`, `execve()`, отличающихся друг от друга представлением параметров, необходимых для работы системного вызова `exec()`. Взаимосвязь перечисленных функций изображена на рис. 2–3.

Рис. 2.3. Взаимосвязь различных функций для выполнения системного вызова exec()

Прототипы и описание функций изменения пользовательского контекста процесса

```
#include <unistd.h>
int execlp(const char *file, const char *arg0, const char *argN,(char *)NULL)
int execvp(const char *file, char *argv[])
int execl(const char *path, const char *arg0, const char *argN,(char *)NULL)
int execv(const char *path, char *argv[])
int execle(const char *path, const char *arg0,
```

```
... const char *argN,(char *)NULL, char * envp[])
int execve(const char *path, char *argv[], char *envp[])
```

Для загрузки новой программы в системный контекст текущего процесса используется семейство взаимосвязанных функций, отличающихся друг от друга формой представления параметров.

Аргумент `file` является указателем на имя файла, который должен быть загружен. Аргумент `path` – это указатель на полный путь к файлу, который должен быть загружен.

Аргументы `arg0, ..., argN` представляют собой указатели на аргументы командной строки. Заметим, что аргумент `arg0` должен указывать на имя загружаемого файла. Аргумент `argv` представляет собой массив из указателей на аргументы командной строки. Начальный элемент массива должен указывать на имя загружаемой программы, а заканчиваться массив должен элементом, содержащим указатель `NULL`.

Аргумент `envp` является массивом указателей на параметры окружающей среды, заданные в виде строк «`переменная=строка`». Последний элемент этого массива должен содержать указатель `NULL`.

Поскольку вызов функции не изменяет системный контекст текущего процесса, загруженная программа унаследует от загрузившего ее процесса следующие атрибуты:

- идентификатор процесса;
- идентификатор родительского процесса;
- групповой идентификатор процесса;
- идентификатор сеанса;
- время, оставшееся до возникновения сигнала `SIGALRM`;
- текущую рабочую директорию;
- маску создания файлов;
- идентификатор пользователя;
- групповой идентификатор пользователя;
- явное игнорирование сигналов;
- таблицу открытых файлов (если для файлового дескриптора не устанавливался признак «закрыть файл при выполнении `exec()`»).

В случае успешного выполнения возврата из функций в программу, осуществившую вызов, не происходит, а управление передается загруженной программе. В случае неудачного выполнения в программу, инициировавшую вызов, возвращается отрицательное значение.

Поскольку системный контекст процесса при вызове `exec()` остается практически неизменным, большинство атрибутов процесса, доступных пользователю через системные вызовы (`PID`, `UID`, `GID`, `PPID` и другие), после запуска новой программы также не изменяется.

Важно понимать разницу между системными вызовами `fork()` и `exec()`. Системный вызов `fork()` создает новый процесс, у которого пользовательский контекст совпадает с пользовательским контекстом процесса-родителя. Системный вызов `exec()` изменяет пользовательский контекст текущего процесса, не создавая новый процесс.

3-5. Прогон программы с использованием системного вызова exec()

Для иллюстрации использования системного вызова `exec()` рассмотрим программу:

```
/* Программа 02-2.c, изменяющая пользовательский
   контекст процесса (запускающая другую программу) */

#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
int main(int argc, char *argv[],char *envp[]){

/* Мы будем запускать команду cat с аргументом командной строки
   03-2.c без изменения параметров среды, т.е. фактически выполнять
   команду "cat 03-2.c", которая должна выдать содержимое данного
   файла на экран. Для функции execle в качестве имени программы
   мы указываем ее полное имя с путем от корневой директории —/bin/cat.

   Первое слово в командной строке у нас должно совпадать с именем
   запускаемой программы. Второе слово в командной строке – это
   имя файла, содержимое которого мы хотим распечатать. */

(void) execle("/bin/cat", "/bin/cat", "03-2.c", 0, envp);

/* Сюда попадаем только при возникновении ошибки */
printf("Error on program start\n");
exit(-1);
return 0;      /* Никогда не выполняется, нужен для того,
                  чтобы компилятор не выдавал warning */
}
```

Программа 02-2.c – изменение пользовательского контекста процесса.

Откомпилируйте ее и запустите на исполнение. Поскольку при нормальной работе будет распечатываться содержимое файла с именем `02-2.c`, такой файл при запуске должен присутствовать в текущей директории (проще всего записать исходный текст программы под этим именем). Проанализируйте результат.

З-6. Написание, компиляция и запуск программы для изменения пользовательского контекста в порожденном процессе

Для закрепления полученных знаний модифицируйте программу, созданную при выполнении задания З-3 так, чтобы порожденный процесс запускал на исполнение новую (любую) программу.

Семинар #3: Организация взаимодействия процессов через pipe и FIFO в UNIX

Понятие потока ввода-вывода. Представление о работе с файлами через системные вызовы и стандартную библиотеку ввода-вывода. Понятие файлового дескриптора. Открытие файла. Системный вызов open(). Системные вызовы close(), read(), write(). Понятие pipe. Системный вызов pipe(). Организация связи через pipe между процессом-родителем и процессом-потомком. Наследование файловых дескрипторов при вызовах fork() и exec(). Особенности поведения вызовов read() и write() для pip'a. Понятие FIFO. Использование системного вызова mknod() для создания FIFO. Функция mkfifo(). Особенности поведения вызова open() при открытии FIFO.

1. Понятие о потоке ввода-вывода

Среди всех категорий средств коммуникации самыми распространенными являются каналы связи, обеспечивающие достаточно безопасное и достаточно информативное взаимодействие процессов.

Существует две модели передачи данных по каналам связи – поток ввода-вывода и сообщения. Из них более простой является потоковая модель, в которой операции передачи/приема информации не интересуются содержимым того, что передается или принимается. Вся информация в канале связи рассматривается как непрерывный поток байт, не обладающий никакой внутренней структурой. Изучению механизмов, обеспечивающих потоковую передачу данных в ОС, и посвящен этот семинар.

2. Понятие о работе с файлами через системные вызовы и стандартную библиотеку ввода-вывода для языка С

Потоковая передача информации может осуществляться не только между процессами, но и между процессом и устройством ввода-вывода, например между процессом и диском, на котором данные представляются в виде файла. Рассмотрим механизм потокового обмена между процессом и файлом.

Известны функции работы с файлами из стандартной библиотеки ввода-вывода, такие как `fopen()`, `fread()`, `fwrite()`, `fprintf()`, `fscanf()`, `fgets()` позволяют программисту получать информацию из файла или записывать ее в файл при условии, что он обладает определенными знаниями о содержимом передаваемых данных. Например, функция `fgets()` используется для ввода из файла последовательности символов, заканчивающейся

символом `'\n'` – перевод каретки, функция `fscanf()` производит ввод информации, соответствующей заданному формату. С точки зрения потоковой модели операции, определяемые функциями стандартной библиотеки ввода-вывода, не являются потоковыми операциями, так как каждая из них требует наличия некоторой структуры передаваемых данных.

В ОС UNIX эти функции представляют собой надстройку – сервисный интерфейс – над системными вызовами, осуществляющими прямые потоковые операции обмена информацией между процессом и файлом и не требующими никаких знаний о том, что она содержит. Ввести еще одно понятие – понятие файлового дескриптора.

3. Файловый дескриптор

Мы знаем, что информация о файлах, используемых процессом, входит в состав его системного контекста и хранится в его блоке управления – PCB. В ОС UNIX можно упрощенно полагать, что информация о файлах, с которыми процесс осуществляет операции потокового обмена, наряду с информацией о потоковых линиях связи, соединяющих процесс с другими процессами и устройствами ввода-вывода, хранится в некотором массиве, получившем название таблицы открытых файлов или таблицы файловых дескрипторов. Индекс элемента этого массива, соответствующий определенному потоку ввода-вывода, получил название файлового дескриптора для этого потока. Таким образом, файловый дескриптор представляет собой небольшое целое неотрицательное число, которое для текущего процесса в данный момент времени однозначно определяет некоторый действующий канал ввода-вывода. Некоторые файловые дескрипторы на этапе старта любой программы ассоциируются со стандартными потоками ввода-вывода. Так, например, файловый дескриптор 0 соответствует стандартному потоку ввода, ФД 1 – стандартному потоку вывода, ФД 2 – стандартному потоку для вывода ошибок. В нормальном интерактивном режиме работы стандартный поток ввода связывает процесс с клавиатурой, а стандартные потоки вывода и вывода ошибок – с текущим терминалом.

4. Открытие файла. Системный вызов open()

Файловый дескриптор используется в качестве параметра, описывающего поток ввода-вывода, для системных вызовов, выполняющих операции над этим потоком. Поэтому прежде чем совершать операции чтения данных из файла и записи их в файл, мы должны поместить информацию о файле в таблицу открытых файлов и определить соответствующий файловый дескриптор. Для этого применяется процедура открытия файла, осуществляемая системным вызовом `open()`.

Прототип и описание системного вызова open

```
#include <fcntl.h>
int open(char *path, int flags);
int open(char *path, int flags, int mode);
```

Системный вызов `open` предназначен для выполнения операции открытия файла и, в случае ее удачного осуществления, возвращает файловый дескриптор открытого файла (небольшое неотрицательное целое число, которое используется в дальнейшем для других операций с этим файлом).

Параметр `path` является указателем на строку, содержащую полное или относительное имя файла.

Параметр `flags` может принимать одно из следующих трех значений:

- `O_RDONLY` – если над файлом в дальнейшем будут совершаться только операции чтения;
- `O_WRONLY` – если над файлом в дальнейшем будут осуществляться только операции записи;
- `O_RDWR` – если над файлом будут осуществляться и операции чтения, и операции записи.

Каждое из этих значений может быть скомбинировано посредством операции «побитовое или (|)» с одним или несколькими флагами:

- `O_CREAT` – если файла с указанным именем не существует, он должен быть создан;
- `O_EXCL` – применяется совместно с флагом `O_CREAT`. При совместном их использовании и существовании файла с указанным именем, открытие файла не производится и констатируется ошибочная ситуация;
- `O_NDELAY` – запрещает перевод процесса в состояние ожидание при выполнении операции открытия и любых последующих операциях над этим файлом;
- `O_APPEND` – при открытии файла и перед выполнением каждой операции записи указатель текущей позиции в файле устанавливается на конец файла;
- `O_TRUNC` – если файл существует, уменьшить его размер до `0`, с сохранением существующих атрибутов файла, кроме быть может, времен последнего доступа к файлу и его последней модификации.

В некоторых версиях ОС UNIX могут применяться дополнительные значения флагов:

- `O_SYNC` – любая операция записи в файл будет блокироваться (т.е. процесс будет переведен в состояние ожидание) до тех пор, пока записанная информация не будет физически помещена на соответствующий нижележащий уровень hardware;
- `O_NOCTTY` – если имя файла относится к терминальному устройству, оно не становится управляющим терминалом процесса, даже если до этого процесс не имел управляющего терминала.

Параметр `mode` устанавливает атрибуты прав доступа различных категорий пользователей к новому файлу при его создании. Он обязателен, если среди

заданных флагов присутствует флаг `O_CREAT`, и может быть опущен в противном случае. Этот параметр задается как сумма следующих восьмеричных значений:

- `0400` – разрешено чтение для пользователя, создавшего файл;
- `0200` – разрешена запись для пользователя, создавшего файл;
- `0100` – разрешено исполнение для пользователя, создавшего файл;
- `0040` – разрешено чтение для группы пользователя, создавшего файл;
- `0020` – разрешена запись для группы пользователя, создавшего файл;
- `0010` – разрешено исполнение для группы пользователя, создавшего файл;
- `0004` – разрешено чтение для всех остальных пользователей;
- `0002` – разрешена запись для всех остальных пользователей;
- `0001` – разрешено исполнение для всех остальных пользователей.

При создании файла реально устанавливаемые права доступа получаются из стандартной комбинации параметра `mode` и маски создания файлов текущего процесса `umask`, а именно – они равны `mode & ~umask`.

При открытии файлов типа FIFO системный вызов имеет некоторые особенности поведения по сравнению с открытием файлов других типов. Если FIFO открывается только для чтения, и не задан флаг `O_NDELAY`, то процесс, осуществивший системный вызов, блокируется до тех пор, пока какой-либо другой процесс не откроет FIFO на запись. Если флаг `O_NDELAY` задан, то возвращается значение файлового дескриптора, ассоциированного с FIFO. Если FIFO открывается только для записи, и не задан флаг `O_NDELAY`, то процесс, осуществивший системный вызов, блокируется до тех пор, пока какой-либо другой процесс не откроет FIFO на чтение. Если флаг `O_NDELAY` задан, то констатируется возникновение ошибки и возвращается значение `-1`.

Возвращаемое значение

Системный вызов возвращает значение файлового дескриптора для открытого файла при нормальном завершении и значение `-1` при возникновении ошибки.

Системный вызов `open()` использует набор флагов для того, чтобы специфицировать операции, которые предполагается применять к файлу в дальнейшем или которые должны быть выполнены непосредственно в момент открытия файла. Из всего возможного набора флагов нас будут интересовать только флаги `O_RDONLY`, `O_WRONLY`, `O_RDWR`, `O_CREAT` и `O_EXCL`. Первые три флага являются взаимоисключающими: хотя бы один из них должен быть применен и наличие одного из них не допускает наличия двух других. Эти флаги описывают набор операций, которые, при успешном открытии файла, будут разрешены над файлом в дальнейшем: только чтение, только запись, чтение и запись. Как вам известно, у каждого файла существуют атрибуты прав доступа для различных категорий пользователей. Если файл с

за данным именем существует на диске, и права доступа к нему для пользователя, от имени которого работает текущий процесс, не противоречат запрошенному набору операций, то ОС сканирует таблицу открытых файлов от ее начала к концу в поисках первого свободного элемента, заполняет его и возвращает индекс этого элемента в качестве ФД открытого файла. Если файла на диске нет, не хватает прав или отсутствует свободное место в таблице открытых файлов, то констатируется возникновение ошибки.

В случае, когда мы **допускаем**, что файл на диске может отсутствовать, и хотим, чтобы он был создан, флаг для набора операций должен использоваться в комбинации с флагом `O_CREAT`. Если файл существует, то все происходит по рассмотренному выше сценарию. Если файла нет, сначала выполняется создание файла с набором прав, указанным в параметрах системного вызова. Проверка соответствия набора операций объявленным правам доступа может и не производиться (как, например, в Linux).

В случае, когда мы **требуем**, чтобы файл на диске отсутствовал и был создан в момент открытия, флаг для набора операций должен использоваться в комбинации с флагами `O_CREAT` и `O_EXCL`.

5. Системные вызовы read(), write(), close()

Для совершения потоковых операций чтения информации из файла и ее записи в файл применяются системные вызовы `read()` и `write()`.

Прототипы и описание системных вызовов read и write

```
#include <sys/types.h>
#include <unistd.h>
size_t read(int fd, void *addr, size_t nbytes);
size_t write(int fd, void *addr, size_t nbytes);
```

Системные вызовы `read` и `write` предназначены для осуществления потоковых операций ввода (чтения) и вывода (записи) информации над каналами связи, описываемыми файловыми дескрипторами, т.е. для файлов, pipe, FIFO и socket.

Параметр `fd` является файловым дескриптором созданного ранее потокового канала связи, через который будет отсылаться или получаться информация, т. е. значением, которое вернул один из системных вызовов `open()`, `pipe()` или `socket()`.

Параметр `addr` представляет собой адрес области памяти, начиная с которого будет браться информация для передачи или размещаться принятая информация.

Параметр `nbytes` для системного вызова `write` определяет количество байт,

которое должно быть передано, начиная с адреса памяти `addr`. Параметр `nbytes` для системного вызова `read` определяет количество байт, которое мы хотим получить из канала связи и разместить в памяти, начиная с адреса `addr`.

Возвращаемые значения

В случае успешного завершения системный вызов возвращает количество реально посланных/принятых байт. Это значение (большее или равное `0`) может не совпадать с заданным значением параметра `nbytes`, а быть меньше, чем оно, в силу отсутствия места на диске или в линии связи при передаче данных или отсутствия информации при ее приеме. При возникновении ошибки возвращается отрицательное значение.

Особенности поведения при работе с файлами

При работе с файлами информация записывается в файл или читается из файла, начиная с места, определяемого указателем текущей позиции в файле. Значение указателя увеличивается на количество реально прочитанных или записанных байт. При чтении информации из файла она не пропадает из него. Если системный вызов `read` возвращает значение `0`, то это означает, что файл прочитан до конца.

После завершения потоковых операций процесс должен выполнить операцию закрытия потока ввода-вывода, во время которой произойдет окончательный сброс буферов на линии связи, освободятся выделенные ресурсы ОС, и элемент таблицы открытых файлов, соответствующий файловому дескриптору, будет отмечен как свободный. За эти действия отвечает системный вызов `close()`. Надо отметить, что при завершении работы процесса с помощью явного или неявного вызова функции `exit()` происходит автоматическое закрытие всех открытых потоков ввода-вывода.

Прототип и описание системного вызова close

```
#include <unistd.h>
int close(int fd);
```

Системный вызов `close` предназначен для корректного завершения работы с файлами и другими объектами ввода-вывода, которые описываются в ОС через файловые дескрипторы: pipe, FIFO, socket.

Параметр `fd` является дескриптором соответствующего объекта, т. е. значением, которое вернул один из системных вызовов `open()`, `pipe()` или `socket()`.

Возвращаемые значения

Системный вызов возвращает значение `0` при нормальном завершении и значение `-1` при возникновении ошибки.

3-1. Прогон программы для записи информации в файл

Для иллюстрации сказанного давайте рассмотрим следующую программу:

```
/*Программа 03-1.c, иллюстрирующая использование системных вызовов
open(), write() и close() для записи информации в файл */
#include <sys/types.h>
#include <fcntl.h>
#include <stdio.h>
int main(){
    int fd;
    size_t size;
    char string[] = "Hello, world!";
    /* Обнуляем маску создания файлов текущего процесса для того,
    чтобы права доступа у создаваемого файла точно соответствовали
    параметру вызова open() */
    (void)umask(0);
    /* Попытаемся открыть файл с именем myfile в текущей директории
     только для операций вывода. Если файла не существует, попробуем
     его создать с правами доступа 0666, т.е. read-write для всех
     категорий пользователей */
    if((fd = open("myfile", O_WRONLY | O_CREAT, 0666)) < 0){
        /* Если файл открыть не удалось, печатаем об этом сообщение
        и прекращаем работу */
        printf("Can\'t open file\n");
        exit(-1);
    }
    /* Пробуем записать в файл 14 байт из нашего массива, т.е. всю
    строку "Hello, world!" вместе с признаком конца строки */
    size = write(fd, string, 14);
    if(size != 14){
        /* Если записалось меньшее количество байт, сообщаем об ошибке */
        printf("Can\'t write all string\n");
        exit(-1);
    }
    /* Закрываем файл */
    if(close(fd) < 0){
        printf("Can\'t close file\n");
    }
    return 0;
}
```

Программа 03-1.c. Использование системных вызовов open(), write() и close() для записи информации в файл

Наберите, откомпилируйте эту программу и запустите ее на исполнение. Обратите внимание на использование системного вызова `umask()` с параметром `0` для того, чтобы права доступа к созданному файлу точно соответствовали указанным в системном вызове `open()`.

3-2. Написание, компиляция и запуск программы для чтения информации из файла

Измените программу 3-1 так, чтобы она читала записанную ранее в файл информацию и печатала ее на экране. Все лишние операторы - удалить.

6. Понятие о pipe. Системный вызов pipe()

Наиболее простым способом для передачи информации с помощью потоковой модели между различными процессами или даже внутри одного процесса в ОС UNIX является `pipe` (канал, труба, конвейер).

Важное отличие pip'а от файла заключается в том, что прочитанная информация немедленно удаляется из него и не может быть прочитана повторно.

`Pipe` можно представить себе в виде трубы ограниченной емкости, расположенной внутри адресного пространства ОС, доступ к входному и выходному отверстию которой осуществляется с помощью системных вызовов. В действительности `pipe` представляет собой область памяти, недоступную пользовательским процессам напрямую, зачастую организованную в виде кольцевого буфера. По буферу при операциях чтения и записи перемещаются два указателя, соответствующие входному и выходному потокам. При этом выходной указатель никогда не может перегнать входной и наоборот. Для создания нового экземпляра такого кольцевого буфера внутри ОС используется системный вызов `pipe()`.

Прототип и описание системного вызова pipe

```
#include <unistd.h>
int pipe(int *fd);
```

Системный вызов `pipe` предназначен для создания pip'а внутри ОС.

Параметр `fd` является указателем на массив из двух целых переменных. При нормальном завершении вызова в первый элемент массива – `fd[0]` – будет занесен файловый дескриптор, соответствующий выходному потоку данных pip'а и позволяющий выполнять только операцию чтения, а во второй элемент массива – `fd[1]` – будет занесен файловый дескриптор, соответствующий входному потоку данных и позволяющий выполнять только операцию записи.

Возвращаемые значения

Системный вызов возвращает значение `0` при нормальном завершении и значение `-1` при возникновении ошибок.

В процессе работы системный вызов организует выделение области памяти под буфер и указатели и заносит информацию, соответствующую входному и выходному потокам данных, в два элемента таблицы открытых файлов, связывая тем самым с каждым pip'ом два файловых дескриптора. Для одного из них разрешена только операция чтения из pip'а, а для другого – только операция записи в `pipe`. Для выполнения этих операций мы можем использовать те же самые системные вызовы `read()` и `write()`, что и при работе с файлами. Естественно, по окончании использования входного или/и выходного потока данных, нужно закрыть соответствующий поток с помощью системного вызова `close()` для освобождения системных ресурсов. Заметим, что, когда все процессы, использующие `pipe`, закрывают все ассоциированные с ним файловые дескрипторы, ОС ликвидирует `pipe`. Таким образом, время существования pip'а в системе не может превышать время жизни процессов, работающих с ним.

3-3. Прогон программы для pipe в одном процессе

В качестве иллюстрации действий по созданию pip'а, записи в него данных, чтению из него и освобождению выделенных ресурсов может служить программа, организующая работу с pip'ом в рамках одного процесса:

```
/* Программа 03-2.c, иллюстрирующая работу с pip'ом в рамках одного
процесса */
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
int main(){
    int fd[2];
    size_t size;
    char string[] = "Hello, world!";
    char resstring[14];
    /* Попытаемся создать pipe */
    if(pipe(fd) < 0){
        /* Если создать pipe не удалось, печатаем об этом сообщение
        и прекращаем работу */
        printf("Can\'t create pipe\n");
        exit(-1);
    }
    /* Пробуем записать в pipe 14 байт из нашего массива, т.е. всю
    строку "Hello, world!" вместе с признаком конца строки */
    size = write(fd[1], string, 14);
    if(size != 14){
        /* Если записалось меньшее количество байт, сообщаем об ошибке */
        printf("Can\'t write all string\n");
        exit(-1);
    }
    /* Пробуем прочитать из pip'а 14 байт в другой массив, т.е. всю
    записанную строку */
    size = read(fd[0], resstring, 14);
    if(size < 0){
        /* Если прочитать не смогли, сообщаем об ошибке */
        printf("Can\'t read string\n");
        exit(-1);
    }
    /* Печатаем прочитанную строку */
```

```
    printf("%s\n",resstring);
    /* Закрываем входной поток*/
    if(close(fd[0]) < 0){
        printf("Can\'t close input stream\n");
    }
    /* Закрываем выходной поток*/
    if(close(fd[1]) < 0){
        printf("Can\'t close output stream\n");
    }
    return 0;
}
```

Программа 03-2.c. Иллюстрация работы с pip'ом в рамках одного процесса

Наберите программу, откомпилируйте ее и запустите на исполнение.

7. Организация связи через pipe между процессом-родителем и процессом-потомком. Наследование файловых дескрипторов при вызовах fork() и exec()

Понятно, что если бы все достоинство pip'ов сводилось к замене функции копирования из памяти в память внутри одного процесса на пересылку информации через ОС, то это того бы не стоила. Однако таблица открытых файлов наследуется процессом-ребенком при порождении нового процесса системным вызовом `fork()` и входит в состав неизменяемой части системного контекста процесса при системном вызове `exec()` (за исключением тех потоков данных, для ФД которых был специальными средствами выставлен признак, побуждающий ОС закрыть их при выполнении `exec()`). Это обстоятельство позволяет организовать передачу информации через `pipe` между родственными процессами, имеющими общего прародителя, создавшего `pipe`.

3-4. Прогон программы для организации однонаправленной связи между родственными процессами через pipe

Рассмотрим программу, осуществляющую однонаправленную связь между процессом-родителем и процессом-ребенком:

```
/* Программа 03-3.c, осуществляющая однонаправленную связь через pipe
между процессом-родителем и процессом-ребенком */
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
int main(){
    int fd[2], result;
    size_t size;
    char resstring[14];
    /* Попытаемся создать pipe */
    if(pipe(fd) < 0){
        /* Если создать pipe не удалось, печатаем об этом сообщение
        и прекращаем работу */
        printf("Can\'t create pipe\n");
        exit(-1);
```

```
    }
    /* Порождаем новый процесс */
    result = fork();
    if(result){
        /* Если создать процесс не удалось, сообщаем об этом и
        завершаем работу */
        printf("Can\'t fork child\n");
        exit(-1);
    } else if (result > 0) {
        /* Мы находимся в родительском процессе, который будет
        передавать информацию процессу-ребенку. В этом процессе
        выходной поток данных нам не понадобится, поэтому закрываем его.*/
        close(fd[0]);
        /* Пробуем записать в pipe 14 байт, т.е. всю строку
        "Hello, world!" вместе с признаком конца строки */
        size = write(fd[1], "Hello, world!", 14);
        if(size != 14){
            /* Если записалось меньшее количество байт, сообщаем
            об ошибке и завершаем работу */
            printf("Can\'t write all string\n");
            exit(-1);
        }
        /* Закрываем входной поток данных, на этом
        родитель прекращает работу */
        close(fd[1]);
        printf("Parent exit\n");
    } else {
        /* Мы находимся в порожденном процессе, который будет получать
        информацию от процесса-родителя. Он унаследовал от родителя
        таблицу открытых файлов и, зная файловые дескрипторы,
        соответствующие pip, может его использовать. В этом процессе
        входной поток данных нам не понадобится, поэтому закрываем его.*/
        close(fd[1]);
        /* Пробуем прочитать из pip'a 14 байт в массив, т.е. всю
        записанную строку */
        size = read(fd[0], resstring, 14);
        if(size < 0){
            /* Если прочитать не смогли, сообщаем об ошибке и
            завершаем работу */
            printf("Can\'t read string\n");
            exit(-1);
        }
        /* Печатаем прочитанную строку */
        printf("%s\n",resstring);
        /* Закрываем входной поток и завершаем работу */
        close(fd[0]);
    }
    return 0;
}
```

Программа 03-3.c. Однонаправленную связь через pipe между процессом-родителем и процессом-ребенком

Наберите программу, откомпилируйте ее и запустите на исполнение.

Задание: модифицируйте этот пример для связи между собой двух родственных процессов, исполняющих разные программы.

3-5. Написание, компиляция и запуск программы для организации двунаправленной связи между родственными процессами через pipe

`Pipe` служит для организации однонаправленной или симплексной связи. Если бы в предыдущем примере мы попытались организовать через `pipe` двустороннюю связь, когда процесс-родитель пишет информацию в `pipe`, предполагая, что ее получит процесс-ребенок, а затем читает информацию из pip'а, предполагая, что ее записал порожденный процесс, то могла бы возникнуть ситуация, в которой процесс-родитель прочитал бы собственную информацию, а процесс-ребенок не получил бы ничего. Для использования одного pip'а в двух направлениях необходимы специальные средства синхронизации процессов. Более простой способ организации двунаправленной связи между родственными процессами заключается в использовании двух pipe. Модифицируйте программу 3-4 для организации такой двусторонней связи, откомпилируйте ее и запустите на исполнение.

Необходимо отметить, что в некоторых UNIX-подобных системах (например, в Solaris2) реализованы полностью дуплексные pip'ы. В таких системах для обоих файловых дескрипторов, ассоциированных с pip'ом, разрешены и операция чтения, и операция записи. Однако такое поведение не характерно для pip'ов и не является переносимым.

8. Особенности поведения вызовов read() и write() для pip'а

Системные вызовы `read()` и `write()` имеют определенные особенности поведения при работе с pip'ом, связанные с его ограниченным размером, задержками в передаче данных и возможностью блокирования обменивающихся информацией процессов.

Будьте внимательны при написании программ, обменивающихся большими объемами информации через `pipe`. **Помните, что за один раз из pip'а может прочитаться меньше информации, чем вы запрашивали, и за один раз в** `pipe` **может записаться меньше информации, чем вам хотелось бы. Проверяйте значения, возвращаемые вызовами!**

Одна из особенностей поведения блокирующегося системного вызова `read()` связана с попыткой чтения из пустого pip'а. Если есть процессы, у которых этот `pipe` открыт для записи, то системный вызов блокируется и ждет появления информации. Если таких процессов нет, он вернет значение `0` без блокировки процесса. Эта особенность приводит к **необходимости закрытия ФД, ассоциированного с входным концом pip'а, в процессе, который будет использовать** `pipe` **для чтения** (`close(fd[1])` в процессе-ребенке в программе 3-4). Аналогичной особенностью поведения при

отсутствии процессов, у которых `pipe` открыт для чтения, обладает и системный вызов `write()`, с чем связана **необходимость закрытия ФД, ассоциированного с выходным концом pip'a, в процессе, который будет использовать** `pipe` **для записи** (`close(fd[0])` в процессе-родителе в той же программе).

Системные вызовы read и write (особенности поведения при работе с pipe, FIFO и socket)	
1. Системный вызов read	
Ситуация	**Поведение**
Попытка прочитать меньше байт, чем есть в наличии в канале связи.	Читает требуемое количество байт и возвращает значение, соответствующее прочитанному количеству. Прочитанная информация удаляется из канала связи.
В канале связи находится меньше байт, чем затребовано, но не нулевое количество.	Читает все, что есть в канале связи, и возвращает значение, соответствующее прочитанному количеству. Прочитанная информация удаляется из канала связи.
Попытка читать из канала связи, в котором нет информации. Блокировка вызова разрешена.	Вызов блокируется до тех пор, пока не появится информация в канале связи и пока существует процесс, который может передать в него информацию. Если информация появилась, то процесс разблокируется, и поведение вызова определяется двумя предыдущими строками таблицы. Если в канал некому передать данные (нет ни одного процесса, у которого этот канал связи открыт для записи), то вызов возвращает значение `0`. Если канал связи полностью закрывается для записи во время блокировки читающего процесса, то процесс разблокируется, и системный вызов возвращает значение `0`.
Попытка читать из канала связи, в котором нет информации. Блокировка вызова не разрешена.	Если есть процессы, у которых канал связи открыт для записи, системный вызов возвращает значение `-1` и устанавливает переменную `errno` в

	значение EAGAIN. Если таких процессов нет, системный вызов возвращает значение 0.
2. Системный вызов write	
Попытка записать в канал связи меньше байт, чем осталось до его заполнения.	Требуемое количество байт помещается в канал связи, возвращается записанное количество байт.
Попытка записать в канал связи больше байт, чем осталось до его заполнения. Блокировка вызова разрешена.	Вызов блокируется до тех пор, пока все данные не будут помещены в канал связи. Если размер буфера канала связи меньше, чем передаваемое количество информации, то вызов тем самым будет ждать, пока часть информации не будет считана из канала связи. Возвращается записанное количество байт.
Попытка записать в канал связи больше байт, чем осталось до его заполнения, но меньше, чем размер буфера канала связи. Блокировка вызова запрещена.	Системный вызов возвращает значение -1 и устанавливает переменную errno в значение EAGAIN.
В канале связи есть место. Попытка записать в канал связи больше байт, чем осталось до его заполнения, и больше, чем размер буфера канала связи. Блокировка вызова запрещена.	Записывается столько байт, сколько осталось до заполнения канала. Системный вызов возвращает количество записанных байт.
Попытка записи в канал связи, в котором нет места. Блокировка вызова не разрешена	Системный вызов возвращает значение -1 и устанавливает переменную errno в значение EAGAIN.
Попытка записи в канал связи, из которого некому больше читать, или полное закрытие канала на чтение во время блокировки системного вызова.	Если вызов был заблокирован, то он разблокируется. Процесс получает сигнал SIGPIPE. Если этот сигнал обрабатывается пользователем, то системный вызов вернет значение -1 и установит переменную errno в значение EPIPE.
Необходимо отметить дополнительную особенность системного вызова write при работе с pip'ами и FIFO. Запись информации, размер которой не превышает размер буфера, должна осуществляться атомарно – одним подряд лежащим куском. Этим объясняется ряд блокировок и ошибок в предыдущем перечне.	

Задание: определите размер `pipe` для вашей операционной системы.

9. Понятие FIFO. Использование системного вызова mknod() для создания FIFO. Функция mkfifo()

Так как доступ к информации о расположении pip'а в ОС и его состоянии может быть осуществлен только через таблицу открытых файлов процесса, создавшего `pipe`, и через унаследованные от него таблицы открытых файлов процессов-потомков. Поэтому механизм обмена информацией через `pipe` справедлив лишь для родственных процессов, имеющих общего прародителя, инициировавшего системный вызов `pipe()`, или для таких процессов и самого прародителя и не может использоваться для потокового общения с другими процессами.

Для организации потокового взаимодействия любых процессов в ОС UNIX применяется средство связи, получившее название FIFO (от **F**irst **I**nput **F**irst **O**utput) или именованный `pipe`. FIFO во всем подобен pip'у, за одним исключением: данные о расположении FIFO в адресном пространстве ядра и его состоянии процессы могут получать не через родственные связи, а через файловую систему. Для этого при создании именованного pip'а на диске заводится файл специального типа, обращаясь к которому процессы могут получить интересующую их информацию. Для создания FIFO используется системный вызов `mknod()` или функция `mkfifo()`.

Следует отметить, что при их работе не происходит действительного выделения области адресного пространства ОС под именованный `pipe`**, а только заводится файл-метка, существование которой позволяет осуществить реальную организацию** FIFO **в памяти при его открытии с помощью системного вызова** `open()`**.**

После открытия именованный `pipe` ведет себя точно так же, как и неименованный. Для дальнейшей работы с ним применяются системные вызовы `read()`, `write()` и `close()`. Время существования FIFO в адресном пространстве ядра операционной системы, как и в случае с pip'ом, не может превышать время жизни последнего из использовавших его процессов. Когда все процессы, работающие с FIFO, закрывают все файловые дескрипторы, ассоциированные с ним, система освобождает ресурсы, выделенные под FIFO. Вся непрочитанная информация теряется. В то же время файл-метка остается на диске и может использоваться для новой реальной организации FIFO в дальнейшем.

Использование системного вызова mknod для создания FIFO

Прототип и описание системного вызова

```
#include <sys/stat.h>
#include <unistd.h>
int mknod(char *path, int mode, int dev);
```

Параметр `dev` мы будем всегда задавать равным `0`.

Параметр `path` является указателем на строку, содержащую полное или относительное имя файла, который будет являться меткой FIFO на диске. Для успешного создания FIFO файла с таким именем перед вызовом существовать не должно.

Параметр `mode` устанавливает атрибуты прав доступа различных категорий пользователей к FIFO. Этот параметр задается как результат побитовой операции «или» значения `S_IFIFO`, указывающего, что системный вызов должен создать FIFO, и некоторой суммы следующих восьмеричных значений:

- `0400` – разрешено чтение для пользователя, создавшего FIFO;
- `0200` – разрешена запись для пользователя, создавшего FIFO;
- `0040` – разрешено чтение для группы пользователя, создавшего FIFO;
- `0020` – разрешена запись для группы пользователя, создавшего FIFO;
- `0004` – разрешено чтение для всех остальных пользователей;
- `0002` – разрешена запись для всех остальных пользователей.

При создании FIFO реально устанавливаемые права доступа получаются из стандартной комбинации параметра `mode` и маски создания файлов текущего процесса `umask`, а именно – они равны `(0777 & mode) & ~umask`.

Возвращаемые значения

При успешном создании FIFO системный вызов возвращает значение `0`, при неуспешном – отрицательное значение.

Прототип и описание функции mkfifo

```
#include <sys/stat.h>
#include <unistd.h>
int mkfifo(char *path, int mode);
```

Функция `mkfifo` предназначена для создания FIFO в операционной системе.

Параметр `path` является указателем на строку, содержащую полное или относительное имя файла, который будет являться меткой FIFO на диске. Для успешного создания FIFO файла с таким именем перед вызовом функции не должно существовать.

Возвращаемые значения

При успешном создании FIFO функция возвращает значение `0`, при неуспешном – отрицательное значение.

Важно понимать, что файл типа FIFO не служит для размещения на диске информации, которая записывается в именованный pipe. Эта информация располагается внутри адресного пространства ОС, а файл является только меткой, создающей предпосылки для ее размещения.

Не пытайтесь просмотреть содержимое этого файла с помощью Midnight Commander (mc)!!! Это приведет к его глубокому зависанию!

10. Особенности поведения вызова open() при открытии FIFO

Системные вызовы read() и write() при работе с FIFO имеют те же особенности поведения, что и при работе с pip'ом. Системный вызов open() при открытии FIFO также ведет себя несколько иначе, чем при открытии других типов файлов, что связано с возможностью блокирования выполняющих его процессов. Если FIFO открывается только для чтения, и флаг O_NDELAY не задан, то процесс, осуществивший системный вызов, блокируется до тех пор, пока какой-либо другой процесс не откроет FIFO на запись. Если флаг O_NDELAY задан, то возвращается значение файлового дескриптора, ассоциированного с FIFO. Если FIFO открывается только для записи, и флаг O_NDELAY не задан, то процесс, осуществивший системный вызов, блокируется до тех пор, пока какой-либо другой процесс не откроет FIFO на чтение. Если флаг O_NDELAY задан, то констатируется возникновение ошибки и возвращается значение -1. Задание флага O_NDELAY в параметрах системного вызова open() приводит и к тому, что процессу, открывшему FIFO, запрещается блокировка при выполнении последующих операций чтения из этого потока данных и записи в него.

3-6. Прогон программы с FIFO в родственных процессах

Для иллюстрации взаимодействия процессов через FIFO рассмотрим такую программу:

```
/* Программа 03-4.c, осуществляющая однонаправленную связь через
FIFO между процессом-родителем и процессом-ребенком */
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
int main(){
    int fd, result;
    size_t size;
    char resstring[14];
    char name[]="aaa.fifo";
    /* Обнуляем маску создания файлов текущего процесса для того,
    чтобы права доступа у создаваемого FIFO точно соответствовали
    параметру вызова mknod() */
    (void)umask(0);
```

```
    /* Попытаемся создать FIFO с именем aaa.fifo в текущей директории */
    if(mknod(name, S_IFIFO | 0666, 0) < 0){
        /* Если создать FIFO не удалось, печатаем об этом
        сообщение и прекращаем работу */
        printf("Can\'t create FIFO\n");
        exit(-1);
    }
    /* Порождаем новый процесс */
    if((result = fork()) < 0){
        /* Если создать процесс не удалось, сообщаем об этом и
        завершаем работу */
        printf("Can\'t fork child\n");
        exit(-1);
    } else if (result > 0) {
        /* Мы находимся в родительском процессе, который будет
        передавать информацию процессу-ребенку. В этом процессе
        открываем FIFO на запись.*/
        if((fd = open(name, O_WRONLY)) < 0){
            /* Если открыть FIFO не удалось, печатаем об этом
            сообщение и прекращаем работу */
            printf("Can\'t open FIFO for writing\n");
            exit(-1);
        }
        /* Пробуем записать в FIFO 14 байт, т.е. всю строку
        "Hello, world!" вместе с признаком конца строки */
        size = write(fd, "Hello, world!", 14);
        if(size != 14){
        /* Если записалось меньшее количество байт, то сообщаем
        об ошибке и завершаем работу */
            printf("Can\'t write all string to FIFO\n");
            exit(-1);
        }
        /* Закрываем входной поток данных и на этом родитель
        прекращает работу */
        close(fd);
        printf("Parent exit\n");
    } else {
        /* Мы находимся в порожденном процессе, который будет получать
        информацию от процесса-родителя. Открываем FIFO на чтение.*/
        if((fd = open(name, O_RDONLY)) < 0){
            /* Если открыть FIFO не удалось, печатаем об этом
            сообщение и прекращаем работу */
            printf("Can\'t open FIFO for reading\n");
            exit(-1);
        }
        /* Пробуем прочитать из FIFO 14 байт в массив, т.е.
        всю записанную строку */
        size = read(fd, resstring, 14);
        if(size < 0){
            /* Если прочитать не смогли, сообщаем об ошибке
            и завершаем работу */
            printf("Can\'t read string\n");
            exit(-1);
        }
        /* Печатаем прочитанную строку */
        printf("%s\n",resstring);
        /* Закрываем входной поток и завершаем работу */
        close(fd);
    }
    return 0;
}
```

Программа 03-4.c. Однонаправленная связь через FIFO между процессом-родителем и процессом-ребенком

Наберите программу, откомпилируйте ее и запустите на исполнение. В этой программе информацией между собой обмениваются процесс-родитель и процесс-ребенок. Заметим, что повторный ее запуск приведет к ошибке при попытке создания FIFO, так как файл с заданным именем уже существует. Здесь нужно либо удалять его перед каждым прогоном программы с диска вручную, либо после первого запуска модифицировать исходный текст, исключив из него все, связанное с системным вызовом `mknod()`.

3-7. Написание, компиляция и запуск программы с FIFO в неродственных процессах

Для закрепления полученных знаний напишите на базе предыдущего примера две программы, одна из которых пишет информацию в FIFO, а вторая – читает из него, так чтобы между ними не было ярко выраженных родственных связей (т.е. чтобы ни одна из них не была потомком другой).

3-8. Неработающий пример для связи процессов на различных компьютерах

Если у вас есть возможность, найдите два компьютера, имеющих разделяемую файловую систему (например, смонтированную с помощью NFS), и запустите на них программы из предыдущего раздела так, чтобы каждая программа работала на своем компьютере, а FIFO создавалось на разделяемой файловой системе. Хотя оба процесса видят один и тот же файл с типом FIFO, взаимодействия между ними не происходит, так как они функционируют в физически разных адресных пространствах и пытаются открыть FIFO внутри различных операционных систем.

Семинар #4: Средства System V IPC. Организация работы с разделяемой памятью в UNIX. Понятие нитей исполнения (thread)

Преимущества и недостатки потокового обмена данными. Понятие System V IPC. Пространство имен. Адресация в System V IPC. Функция ftok(). Дескрипторы System V IPC. Разделяемая память в UNIX. Системные вызовы shmget(), shmat(), shmdt(). Команды ipc и ipcrm. Использование системного вызова shmctl() для освобождения ресурса. Разделяемая память и системные вызовы fork(), exec() и функция exit(). Понятие о нити исполнения (thread) в UNIX. Идентификатор нити исполнения. Функция pthread_self(). Создание и завершение thread'a. Функции pthread_create(), pthread_exit(), pthread_join(). Необходимость синхронизации процессов и нитей исполнения, использующих общую память.

1. Преимущества и недостатки потокового обмена данными

На предыдущем семинаре мы познакомились с механизмами, обеспечивающими потоковую передачу данных между процессами в операционной системе UNIX, а именно с pip'ами и FIFO. Потоковые механизмы достаточно просты в реализации и удобны для использования, но имеют ряд существенных недостатков:

- Операции чтения и записи не анализируют содержимое передаваемых данных. Процесс, прочитавший 20 байт из потока, не может сказать, были ли они записаны одним процессом или несколькими, записывались ли они за один раз или было, например, выполнено 4 операции записи по 5 байт. Данные в потоке никак не интерпретируются системой. Если требуется какая-либо интерпретация данных, то передающий и принимающий процессы должны заранее согласовать свои действия и уметь осуществлять ее самостоятельно.
- Для передачи информации от одного процесса к другому требуется, как минимум, две операции копирования данных: первый раз – из адресного пространства передающего процесса в системный буфер, второй раз – из системного буфера в адресное пространство принимающего процесса.
- Процессы, обменивающиеся информацией, должны одновременно существовать в вычислительной системе. Нельзя записать информацию в поток с помощью одного процесса, завершить его, а затем, через некоторое время, запустить другой процесс и прочитать записанную информацию.

2. Понятие о System V IPC

Указанные выше недостатки потоков данных привели к разработке других механизмов передачи информации между процессами. Часть этих механизмов, впервые появившихся в UNIX System V и впоследствии перекочевавших оттуда практически во все современные версии операционной системы UNIX, получила общее название System V IPC (IPC – сокращение от interprocess communications). В группу System V IPC входят: очереди сообщений, разделяемая память и семафоры. Эти средства организации взаимодействия процессов связаны не только общностью происхождения, но и обладают схожим интерфейсом для выполнения подобных операций, например, для выделения и освобождения соответствующего ресурса в системе. Мы будем рассматривать их в порядке от менее семантически нагруженных с точки зрения операционной системы к более семантически нагруженным. Иными словами, чем позже мы начнем заниматься каким-либо механизмом из System V IPC, тем больше действий по интерпретации передаваемой информации придется выполнять операционной системе при использовании этого механизма. Часть этого семинара мы посвятим изучению разделяемой памяти.

3. Пространство имен. Адресация в System V IPC. Функция ftok()

Все средства связи из System V IPC, как и уже рассмотренные нами pipe и FIFO, являются средствами связи с непрямой адресацией. Как мы установили на предыдущем семинаре, для организации взаимодействия неродственных процессов с помощью средства связи с непрямой адресацией необходимо, чтобы это средство связи имело имя. Отсутствие имен у pip'ов позволяет процессам получать информацию о расположении pip'а в системе и его состоянии только через родственные связи. Наличие ассоциированного имени у FIFO – имени специализированного файла в файловой системе – позволяет неродственным процессам получать эту информацию через интерфейс файловой системы.

Множество всех возможных имен для объектов какого-либо вида принято называть пространством имен соответствующего вида объектов. Для FIFO пространством имен является множество всех допустимых имен файлов в файловой системе. Для всех объектов из System V IPC таким пространством имен является множество значений некоторого целочисленного типа данных – `key_t` – ключа. Причем программисту не позволено напрямую присваивать значение ключа, это значение задается опосредованно: через комбинацию имени какого-либо файла, уже существующего в файловой системе, и небольшого целого числа – например, номера экземпляра средства связи.

Такой хитрый способ получения значения ключа связан с двумя соображениями:

- Если разрешить программистам самим присваивать значение ключа для идентификации средств связи, то не исключено, что два программиста случайно воспользуются одним и тем же значением, не подозревая об этом. Тогда их процессы будут несанкционированно взаимодействовать через одно и то же средство коммуникации, что может привести к нестандартному поведению этих процессов. Поэтому основным компонентом значения ключа является преобразованное в числовое значение полное имя некоторого файла, доступ к которому на чтение разрешен процессу. Каждый программист имеет возможность использовать для этой цели свой специфический файл, например исполняемый файл, связанный с одним из взаимодействующих процессов. Следует отметить, что преобразование из текстового имени файла в число основывается на расположении указанного файла на жестком диске или ином физическом носителе. Поэтому для образования ключа следует применять файлы, не меняющие своего положения в течение времени организации взаимодействия процессов;
- Второй компонент значения ключа используется для того, чтобы позволить программисту связать с одним и тем же именем файла более одного экземпляра каждого средства связи. В качестве такого компонента можно задавать порядковый номер соответствующего экземпляра.

Получение значения ключа из двух компонентов осуществляется функцией `ftok()`.

Прототип и описание функции для генерации ключа System V IPC

```
#include <sys/types.h>
#include <sys/ipc.h>
key_t ftok(char *pathname, char proj);
```

Функция `ftok` служит для преобразования имени существующего файла и небольшого целого числа, например, порядкового номера экземпляра средств связи, в ключ SV IPC.

Параметр `pathname` должен являться указателем на имя существующего файла, доступного для процесса, вызывающего функцию.

Параметр `proj` – это небольшое целое число, характеризующее экземпляр

средства связи.

В случае невозможности генерации ключа функция возвращает отрицательное значение, в противном случае она возвращает значение сгенерированного ключа. Тип данных `key_t` обычно представляет собой 32-битовое целое.

Еще раз подчеркнем три важных момента, связанных с использованием имени файла для получения ключа. Во-первых, необходимо указывать имя файла, который **уже существует** в файловой системе и для которого процесс **имеет право доступа на чтение** (не путать с заданием имени файла при создании FIFO, где указывалось имя **для вновь создаваемого** специального файла). Во-вторых, указанный файл должен **сохранять свое положение на диске** до тех пор, пока все процессы, участвующие во взаимодействии, не получат ключ SV IPC. В-третьих, задание имени файла, как одного из компонентов для получения ключа, не означает, что информация, передаваемая с помощью ассоциированного средства связи, будет располагаться в этом файле. Информация будет храниться **внутри адресного пространства ОС**, а заданное имя файла лишь позволяет различным процессам сгенерировать идентичные ключи.

4. Дескрипторы System V IPC

Мы говорили, что информацию о потоках ввода-вывода, с которыми имеет дело текущий процесс, в частности о pip'ах и FIFO, операционная система хранит в таблице открытых файлов процесса. Системные вызовы, осуществляющие операции над потоком, используют в качестве параметра индекс элемента таблицы открытых файлов, соответствующего потоку, – файловый дескриптор. Использование файловых дескрипторов для идентификации потоков внутри процесса позволяет применять к ним уже существующий интерфейс для работы с файлами, но в то же время приводит к автоматическому закрытию потоков при завершении процесса. Этим, в частности, объясняется один из перечисленных выше недостатков потоковой передачи информации.

При реализации компонентов SV IPC была принята другая концепция. Ядро операционной системы хранит информацию обо всех средствах System V IPC, используемых в системе, вне контекста пользовательских процессов. При создании нового средства связи или получении доступа к уже существующему процесс получает неотрицательное целое число – *дескриптор* (идентификатор) *этого средства связи*, которое однозначно идентифицирует его во всей вычислительной системе. Этот дескриптор должен передаваться в качестве параметра всем системным вызовам,

осуществляющим дальнейшие операции над соответствующим средством SV IPC.

Подобная концепция позволяет устранить один из самых существенных недостатков, присущих потоковым средствам связи – требование одновременного существования взаимодействующих процессов, но в то же время требует повышенной осторожности для того, чтобы процесс, получающий информацию, не принял взамен новых старые данные, случайно оставленные в механизме коммуникации.

5. Разделяемая память в UNIX. Системные вызовы shmget(), shmat(), shmdt()

С точки зрения ОС, наименее семантически нагруженным средством System V IPC является разделяемая память (shared memory). Мы уже упоминали об этой категории средств связи на лекции. Нам достаточно знать, что операционная система может позволить нескольким процессам совместно использовать некоторую область адресного пространства. Внутренние механизмы, позволяющие реализовать такое использование, будут подробно рассмотрены на лекции, посвященной сегментной, страничной и сегментно-страничной организации памяти.

Все средства связи SV IPC требуют предварительных инициализирующих действий (создания) для организации взаимодействия процессов.

Для создания области разделяемой памяти с определенным ключом или доступа по ключу к уже существующей области применяется системный вызов `shmget()`. Существует два варианта его использования для создания новой области разделяемой памяти.

- Стандартный способ. В качестве значения ключа системному вызову поставляется значение, сформированное функцией `ftok()` для некоторого имени файла и номера экземпляра области разделяемой памяти. В качестве флагов поставляется комбинация прав доступа к создаваемому сегменту и флага `IPC_CREAT`. Если сегмент для данного ключа еще не существует, то система будет пытаться создать его с указанными правами доступа. Если же вдруг он уже существовал, то мы просто получим его дескриптор. Возможно добавление к этой комбинации флагов флага `IPC_EXCL`. Этот флаг гарантирует нормальное завершение системного вызова только в том случае, если сегмент действительно был создан (т. е. ранее он не существовал), если же сегмент существовал, то системный вызов завершится с ошибкой, и значение системной переменной `errno`, описанной в файле `errno.h`, будет установлено в `EEXIST`.
- Нестандартный способ. В качестве значения ключа указывается специальное значение `IPC_PRIVATE`. Использование значения `IPC_PRIVATE` всегда приводит к попытке создания нового сегмента разделяемой памяти с заданными правами доступа и с ключом, который не совпадает со значением ключа ни одного из уже существующих сегментов и который не может быть получен с помощью функции `ftok()` ни при одной комбинации ее параметров. Наличие флагов `IPC_CREAT` и `IPC_EXCL` в этом случае игнорируется.

Прототип и описание системного вызова shmget()

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmget(key_t key, int size, int shmflg);
```

Системный вызов `shmget` предназначен для выполнения операции доступа к сегменту разделяемой памяти и, в случае его успешного завершения, возвращает дескриптор System V IPC для этого сегмента (целое неотрицательное число, однозначно характеризующее сегмент внутри вычислительной системы и использующееся в дальнейшем для других операций с ним).

Параметр `key` является ключом SV IPC для сегмента, т. е. фактически его именем из пространства имен SV IPC. В качестве значения этого параметра может использоваться значение ключа, полученное с помощью функции `ftok()`, или специальное значение `IPC_PRIVATE`. Использование значения `IPC_PRIVATE` всегда приводит к попытке создания нового сегмента разделяемой памяти с ключом, который не совпадает со значением ключа ни одного из уже существующих сегментов и который не может быть получен с помощью функции `ftok()` ни при одной комбинации ее параметров.

Параметр `size`. определяет размер создаваемого или уже существующего сегмента в байтах. Если сегмент с указанным ключом уже существует, но его размер не совпадает с указанным в параметре `size`, констатируется возникновение ошибки.

Параметр `shmflg` – флаги – играет роль только при создании нового сегмента разделяемой памяти и определяет права различных пользователей при доступе к сегменту, а также необходимость создания нового сегмента и поведение системного вызова при попытке создания. Он является некоторой комбинацией (с помощью операции побитовое или – «`|`») следующих предопределенных значений и восьмеричных прав доступа:

- `IPC_CREAT` – если сегмента для указанного ключа не существует, он должен быть создан;
- `IPC_EXCL` – применяется совместно с флагом `IPC_CREAT`. При совместном их использовании и существовании сегмента с указанным ключом, доступ к сегменту не производится и констатируется ошибочная ситуация, при этом переменная `errno`, описанная в файле `<errno.h>`, примет значение `EEXIST`;
- `0400` – разрешено чтение для пользователя, создавшего сегмент;
- `0200` – разрешена запись для пользователя, создавшего сегмент;
- `0040` – разрешено чтение для группы пользователя, создавшего сегмент;
- `0020` – разрешена запись для группы пользователя, создавшего сегмент;
- `0004` – разрешено чтение для всех остальных пользователей;
- `0002` – разрешена запись для всех остальных пользователей.

Возвращаемое значение

Системный вызов возвращает значение дескриптора SV IPC для сегмента разделяемой памяти при нормальном завершении и значение `-1` при возникновении ошибки.

Доступ к созданной области разделяемой памяти в дальнейшем обеспечивается ее дескриптором, который вернет системный вызов `shmget()`. Доступ к уже существующей области также может осуществляться двумя способами:

- Если мы знаем ее ключ, то, используя вызов `shmget()`,можем получить ее дескриптор. В этом случае нельзя указывать в качестве составной части флагов флаг `IPC_EXCL`, а значение ключа, естественно, не может быть `IPC_PRIVATE`. Права доступа игнорируются, а размер области должен совпадать с размером, указанным при ее создании.
- Либо мы можем воспользоваться тем, что дескриптор System V IPC действителен в рамках всей операционной системы, и передать его значение от процесса, создавшего разделяемую память, текущему процессу. Отметим, что при создании разделяемой памяти с помощью значения `IPC_PRIVATE` – это единственно возможный способ.

После получения дескриптора необходимо включить область разделяемой памяти в адресное пространство текущего процесса. Это осуществляется с помощью системного вызова `shmat()`. При нормальном завершении он вернет адрес разделяемой памяти в адресном пространстве текущего процесса. Дальнейший доступ к этой памяти осуществляется с помощью обычных средств языка программирования.

Прототип и описание системного вызова shmat()

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
char *shmat(int shmid, char *shmaddr, int shmflg);
```

Системный вызов `shmat` предназначен для включения области разделяемой памяти в адресное пространство текущего процесса.

Параметр `shmid` является дескриптором SV IPC для сегмента разделяемой памяти, т.е. значением, которое вернул системный вызов `shmget()` при создании сегмента или при его поиске по ключу.

В качестве параметра `shmaddr` мы всегда будем передавать значение `NULL`, позволяя ОС самой разместить разделяемую память в адресном пространстве нашего процесса.

Параметру `shmflg` мы будем задавать только два значения: `0` – для

осуществления операций чтения и записи над сегментом и SHM_RDONLY – если мы хотим только читать из него. При этом процесс должен иметь соответствующие права доступа к сегменту.

Возвращаемое значение

Системный вызов возвращает адрес сегмента разделяемой памяти в адресном пространстве процесса при нормальном завершении и значение -1 при возникновении ошибки.

После окончания использования разделяемой памяти процесс может уменьшить размер своего адресного пространства, исключив из него эту область с помощью системного вызова shmdt(). Отметим, что в качестве параметра системный вызов shmdt() требует адрес начала области разделяемой памяти в адресном пространстве процесса, т. е. значение, которое вернул системный вызов shmat(), поэтому данное значение следует сохранять на протяжении всего времени использования разделяемой памяти.

Прототип и описание системного вызова shmdt()

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int shmdt(char *shmaddr);
```

Системный вызов shmdt предназначен для исключения области разделяемой памяти из адресного пространства текущего процесса.

Параметр shmaddr является адресом сегмента разделяемой памяти, т.е. значением, которое вернул системный вызов shmat().

Возвращаемое значение

Системный вызов возвращает значение 0 при нормальном завершении и значение -1 при возникновении ошибки.

3-1. Прогон программ с использованием разделяемой памяти

Для иллюстрации использования разделяемой памяти давайте рассмотрим две взаимодействующие программы.

```
/* Программа 1 (04-1a.c) для иллюстрации работы с разделяемой памятью */
/* Мы организуем разделяемую память для массива из трех целых чисел.
Первый элемент массива является счетчиком числа запусков программы 1, т.е.
данной программы, второй элемент массива – счетчиком числа запусков программы
2, третий элемент массива – счетчиком числа запусков обеих программ */
```

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <errno.h>
int main()
{
    int *array;    /* Указатель на разделяемую память */
    int shmid;     /* IPC дескриптор для области разделяемой памяти */
    int new = 1;   /* Флаг необходимости инициализации элементов массива */
    char pathname[] = "04-1a.c"; /* Имя файла,
        используемое для генерации ключа. Файл с таким
        именем должен существовать в текущей директории */
    key_t key;     /* IPC ключ */
    /* Генерируем IPC ключ из имени файла 04-1a.c в текущей директории
        и номера экземпляра области разделяемой памяти 0 */
    if((key = ftok(pathname,0)) < 0){
        printf("Can\'t generate key\n");
        exit(-1);
    }
    /* Пытаемся эксклюзивно создать разделяемую память для сгенерированного
    ключа, т.е. если для этого ключа она уже существует, системный вызов
    вернет отрицательное значение. Размер памяти определяем как размер
    массива     из трех целых переменных, права доступа 0666 - чтение
    и запись разрешены для всех */
    if((shmid = shmget(key, 3*sizeof(int), 0666|IPC_CREAT|IPC_EXCL)) < 0){
    /* В случае ошибки пытаемся определить: возникла ли она из-за того,
    что сегмент разделяемой памяти уже существует или по другой причине */
        if(errno != EEXIST){
            /* Если по другой причине - прекращаем работу */
            printf("Can\'t create shared memory\n");
            exit(-1);
        } else {
            /* Если из-за того, что разделяемая память уже существует,
            то пытаемся получить ее IPC дескриптор и, в случае удачи,
            сбрасываем флаг необходимости инициализации элементов массива */
            if((shmid = shmget(key, 3*sizeof(int), 0)) < 0){
                printf("Can\'t find shared memory\n");
                exit(-1);
            }
            new = 0;
        }
    }
    /* Пытаемся отобразить разделяемую память в адресное пространство
    текущего процесса. Обратите внимание на то, что для правильного сравнения
    мы явно преобразовываем значение -1 к указателю на целое.*/
    if((array = (int *)shmat(shmid, NULL, 0)) == (int *)(-1)){
        printf("Can't attach shared memory\n");
        exit(-1);
    }
    /* В зависимости от значения флага new либо инициализируем массив,
    либо увеличиваем соответствующие счетчики */
    if(new){
        array[0] = 1;
        array[1] = 0;
        array[2] = 1;
    } else {
        array[0] += 1;
        array[2] += 1;
    }
    /* Печатаем новые значения счетчиков, удаляем разделяемую память
    из адресного пространства текущего процесса и завершаем работу */
    printf("Program 1 was spawn %d times, program 2 - %d times,
        total - %d times\n", array[0], array[1], array[2]);
```

```
    if(shmdt(array) < 0){
        printf("Can't detach shared memory\n");
        exit(-1);
    }
    return 0;
}
```

Программа 04-1a.c для иллюстрации работы с разделяемой памятью

```
/* Программа 2 (04-1b.c) для иллюстрации работы с разделяемой памятью */
/* Мы организуем разделяемую память для массива из трех целых чисел. Первый
элемент массива является счетчиком числа запусков программы 1, т.е. данной
программы, второй элемент массива – счетчиком числа запусков программы 2,
третий элемент массива – счетчиком числа запусков обеих программ */
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <errno.h>
int main()
{
    int *array;   /* Указатель на разделяемую память */
    int shmid;    /* IPC дескриптор для области разделяемой памяти */
    int new = 1;  /* Флаг необходимости инициализации элементов массива */
    char pathname[] = "04-1a.c"; /* Имя файла,
        используемое для генерации ключа. Файл с таким
        именем должен существовать в текущей директории */
    key_t key;    /* IPC ключ */
    /* Генерируем IPC ключ из имени файла 04-1a.c в
    текущей директории и номера экземпляра области
    разделяемой памяти 0 */
    if((key = ftok(pathname,0)) < 0){
        printf("Can\'t generate key\n");
        exit(-1);
    }
    /* Пытаемся эксклюзивно создать разделяемую память для сгенерированного
    ключа, т.е. если для этого ключа она уже существует, системный вызов
    вернет отрицательное значение. Размер памяти определяем как размер
    массива из трех целых переменных, права доступа 0666 – чтение
    и запись разрешены для всех */
    if((shmid = shmget(key, 3*sizeof(int), 0666|IPC_CREAT|IPC_EXCL)) < 0){
    /* В случае возникновения ошибки пытаемся определить:
    возникла ли она из-за того, что сегмент разделяемой
    памяти уже существует или по другой причине */
        if(errno != EEXIST){
            /* Если по другой причине – прекращаем работу */
            printf("Can\'t create shared memory\n");
            exit(-1);
        } else {
            /* Если из-за того, что разделяемая память уже существует,
            то пытаемся получить ее IPC дескриптор и, в случае удачи,
            сбрасываем флаг необходимости инициализации элементов массива */
            if((shmid = shmget(key, 3*sizeof(int), 0)) < 0){
                printf("Can\'t find shared memory\n");
                exit(-1);
            }
            new = 0;
        }
    }
    /* Пытаемся отобразить разделяемую память в адресное пространство
    текущего процесса. Обратите внимание на то, что для правильного
    сравнения мы явно преобразовываем значение -1 к указателю на целое.*/
    if((array = (int *)shmat(shmid, NULL, 0)) == (int *)(-1)){
        printf("Can't attach shared memory\n");
        exit(-1);
```

```
    }
    /* В зависимости от значения флага new либо инициализируем
    массив, либо увеличиваем соответствующие счетчики */
    if(new){
        array[0] = 0;
        array[1] = 1;
        array[2] = 1;
    } else {
        array[1] += 1;
        array[2] += 1;
    }
    /* Печатаем новые значения счетчиков, удаляем разделяемую память
    из адресного пространства текущего процесса и завершаем работу */
    printf("Program 1 was spawn %d times, program 2 - %d times,
        total - %d times\n", array[0], array[1], array[2]);
    if(shmdt(array) < 0){
        printf("Can't detach shared memory\n");
        exit(-1);
    }
    return 0;
}
```

Программа 2 (04-1b.c) для иллюстрации работы с разделяемой памятью

Эти программы очень похожи друг на друга и используют разделяемую память для хранения числа запусков каждой из программ и их суммы. В разделяемой памяти размещается массив из трех целых чисел. Первый элемент массива используется как счетчик для программы 1, второй элемент – для программы 2, третий элемент – для обеих программ суммарно. Дополнительный нюанс в программах возникает из-за необходимости инициализации элементов массива при создании разделяемой памяти. Для этого нам нужно, чтобы программы могли различать случай, когда они создали ее, и случай, когда она уже существовала. Мы добиваемся различия, используя вначале системный вызов `shmget()` с флагами `IPC_CREAT` и `IPC_EXCL`. Если вызов завершается нормально, то мы создали разделяемую память. Если вызов завершается с констатацией ошибки и значение переменной `errno` равняется `EEXIST`, то, значит, разделяемая память уже существует, и мы можем получить ее IPC дескриптор, применяя тот же самый вызов с нулевым значением флагов. Наберите программы, сохраните под именами `04-1a.c` и `04-1b.c` соответственно, откомпилируйте их и запустите несколько раз. Проанализируйте полученные результаты.

6. Команды ipcs и ipcrm

Как мы видели из предыдущего примера, созданная область разделяемой памяти сохраняется в операционной системе даже тогда, когда нет ни одного процесса, включающего ее в свое адресное пространство. С одной стороны, это имеет определенные преимущества, поскольку не требует одновременного существования взаимодействующих процессов, с другой стороны, может причинять существенные неудобства. Допустим, что предыдущие программы мы хотим использовать таким образом, чтобы подсчитывать количество запусков в течение одного, текущего, сеанса работы в системе. Однако в созданном сегменте разделяемой памяти остается

информация от предыдущего сеанса, и программы будут выдавать общее количество запусков за все время работы с момента загрузки операционной системы. Можно было бы создавать для нового сеанса новый сегмент разделяемой памяти, но количество ресурсов в системе не безгранично. Нас спасает то, что существуют способы удалять неиспользуемые ресурсы SV IPC как с помощью команд ОС, так и с помощью системных вызовов. Все средства SV IPC требуют определенных действий для освобождения занимаемых ресурсов после окончания взаимодействия процессов. Для того чтобы удалять ресурсы SV IPC из командной строки, нам понадобятся две команды, `ipcs` и `ipcrm`.

Команда `ipcs` выдает информацию обо всех средствах SV IPC, существующих в системе, для которых пользователь обладает правами на чтение: областях разделяемой памяти, семафорах и очередях сообщений.

Синтаксис и описание команды ipcs

```
ipcs [-asmq] [-tclup]
ipcs [-smq] -i id
ipcs -h
```

Команда `ipcs` предназначена для получения информации о средствах SV IPC, к которым пользователь имеет право доступа на чтение.

Опция `-i` позволяет указать идентификатор ресурсов. Будет выдаваться только информация для ресурсов, имеющих этот идентификатор.

Виды IPC ресурсов могут быть заданы с помощью следующих опций:

- `-s` для семафоров;
- `-m` для сегментов разделяемой памяти;
- `-q` для очередей сообщений;
- `-a` для всех ресурсов (по умолчанию).

Опции `[-tclup]` используются для изменения состава выходной информации. По умолчанию для каждого средства выводятся его ключ, идентификатор IPC, идентификатор владельца, права доступа и ряд других характеристик. Применение опций позволяет вывести:

- `-t` времена совершения последних операций над средствами IPC;
- `-p` идентификаторы процесса, создавшего ресурс, и процесса, совершившего над ним последнюю операцию;
- `-c` идентификаторы пользователя и группы для создателя ресурса и его собственника;
- `-l` системные ограничения для средств SV IPC;
- `-u` общее состояние IPC ресурсов в системе.

Опция `-h` используется для получения краткой справочной информации.

Из всего многообразия выводимой информации нас будут интересовать только IPC идентификаторы для средств, созданных вами. Эти идентификаторы будут использоваться в команде `ipcrm`, позволяющей удалить необходимый ресурс из системы. Для удаления сегмента разделяемой памяти эта команда имеет вид:

```
ipcrm shm <IPC идентификатор>
```

Удалите созданный вами сегмент разделяемой памяти из ОС, используя эти команды.

Синтаксис и описание команды ipcrm

```
ipcrm [shm | msg | sem] id
```

Команда `ipcrm` предназначена для удаления ресурса SV IPC из ОС. Параметр `id` задает IPC идентификатор для удаляемого ресурса, параметр `shm` используется для сегментов разделяемой памяти, параметр `msg` – для очередей сообщений, `sem` – для семафоров.

Если поведение программ, использующих средства SV IPC, базируется на предположении, что эти средства были созданы при их работе, не забывайте перед их запуском удалять уже существующие ресурсы.

7. Использование системного вызова shmctl() для освобождения ресурса

Для той же цели – удалить область разделяемой памяти из системы – можно воспользоваться и системным вызовом `shmctl()`. Этот системный вызов позволяет полностью ликвидировать область разделяемой памяти в ОС по заданному дескриптору средства IPC, если, конечно, у вас хватает для этого полномочий. Системный вызов `shmctl()` позволяет выполнять и другие действия над сегментом разделяемой памяти.

Прототип и описание системного вызова shmctl()

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

Системный вызов `shmctl` предназначен для получения информации об области разделяемой памяти, изменения ее атрибутов и удаления из системы.

Мы будем пользоваться системным вызовом `shmctl` только для удаления области разделяемой памяти из системы. Параметр `shmid` является

дескриптором SV IPC для сегмента разделяемой памяти, т. е. значением, которое вернул системный вызов `shmget()` при создании сегмента или при его поиске по ключу.

В качестве параметра `cmd` в рамках нашего курса мы всегда будем передавать значение `IPC_RMID` – команду для удаления сегмента разделяемой памяти с заданным идентификатором. Параметр `buf` для этой команды не используется, поэтому мы всегда будем подставлять туда значение `NULL`.

Возвращаемое значение

Системный вызов возвращает значение `0` при нормальном завершении и значение `-1` при возникновении ошибки.

8. Разделяемая память и системные вызовы fork(), exec() и функция exit()

Важным вопросом является поведение сегментов разделяемой памяти при выполнении процессом системных вызовов `fork()`, `exec()` и функции `exit()`.

При выполнении системного вызова `fork()` все области разделяемой памяти, размещенные в адресном пространстве процесса, наследуются порожденным процессом.

При выполнении системных вызовов `exec()` и функции `exit()` все области разделяемой памяти, размещенные в адресном пространстве процесса, исключаются из его адресного пространства, но продолжают существовать в операционной системе.

3-2. Самостоятельное написание, компиляция и запуск программы для организации связи двух процессов через разделяемую память.

Для закрепления полученных знаний напишите две программы, осуществляющие взаимодействие через разделяемую память. Первая программа должна создавать сегмент разделяемой памяти и копировать туда собственный исходный текст, вторая программа должна брать оттуда этот текст, печатать его на экране и удалять сегмент разделяемой памяти из системы.

9. Понятие о нити исполнения (thread) в UNIX. Идентификатор нити исполнения. Функция pthread_self()

Мы говорили, что во многих современных ОС существует расширенная реализация понятия процесс, когда процесс представляет собой совокупность выделенных ему ресурсов и набора нитей исполнения. Нити процесса разделяют его программный код, глобальные переменные и системные ресурсы, но каждая нить имеет собственный программный счетчик, свое содержимое регистров и свой стек. Поскольку глобальные переменные у нитей исполнения являются общими, они могут использовать их, как элементы разделяемой памяти, не прибегая к механизму, описанному выше.

В различных версиях ОС UNIX существуют различные интерфейсы, обеспечивающие работу с нитями исполнения. Мы кратко ознакомимся с некоторыми функциями, позволяющими разделить процесс на thread'ы и управлять их поведением, в соответствии со стандартом POSIX. Нити исполнения, удовлетворяющие стандарту POSIX, принято называть POSIX thread'ами или, кратко, pthread'ами.

К сожалению, ОС Linux не полностью поддерживает нити исполнения на уровне ядра системы. При создании нового thread'а запускается новый традиционный процесс, разделяющий с родительским традиционным процессом его ресурсы, программный код и данные, расположенные вне стека, т.е. фактически действительно создается новый thread, но ядро не умеет определять, что эти thread'ы являются составными частями одного целого. Это «знает» только специальный процесс-координатор, работающий на пользовательском уровне и стартующий при первом вызове функций, обеспечивающих POSIX интерфейс для нитей исполнения. Поэтому мы сможем наблюдать не все преимущества использования нитей исполнения (в частности, ускорить решение задачи на однопроцессорной машине с их помощью вряд ли получится), но даже в этом случае thread'ы можно задействовать как очень удобный способ для создания процессов с общими ресурсами, программным кодом и разделяемой памятью.

Каждая нить исполнения, как и процесс, имеет в системе уникальный номер – идентификатор thread'а. Поскольку традиционный процесс в концепции нитей исполнения трактуется как процесс, содержащий единственную нить исполнения, мы можем узнать идентификатор этой нити и для любого обычного процесса. Для этого используется функция `pthread_self()`. Нить исполнения, создаваемую при рождении нового процесса, принято называть **начальной** или **главной** нитью исполнения этого процесса.

Прототип и описание функции pthread_self()

```
#include <pthread.h>
pthread_t pthread_self(void);
```

Функция `pthread_self` возвращает идентификатор текущей нити исполнения.

Тип данных `pthread_t` является синонимом для одного из целочисленных типов языка C.

10. Создание и завершение thread'a. Функции pthread_create(), pthread_exit(), pthread_join()

Нити исполнения, как и традиционные процессы, могут порождать нити-потомки, правда, только внутри своего процесса. Каждый будущий thread внутри программы должен представлять собой функцию с прототипом

```
void *thread(void *arg);
```

Параметр `arg` передается этой функции при создании thread'a и может, до некоторой степени, рассматриваться как аналог параметров функции `main()`. Возвращаемое функцией значение может интерпретироваться как аналог информации, которую родительский процесс может получить после завершения процесса-ребенка. Для создания новой нити исполнения применяется функция `pthread_create()`.

Прототип и описание функции для создания нити исполнения

```
#include <pthread.h>
int pthread_create(pthread_t *thread, pthread_attr_t *attr,
    void * (*start_routine)(void *), void *arg);
```

Функция `pthread_create` служит для создания новой нити исполнения (thread'a) внутри текущего процесса.

Новый `thread` будет выполнять функцию `start_routine` с прототипом

```
void *start_routine(void *)
```

передавая ей в качестве аргумента параметр `arg`. Если требуется передать более одного параметра, они собираются в структуру, и передается адрес этой структуры. Значение, возвращаемое функцией `start_routine` не должно указывать на динамический объект данного thread'a.

Параметр `attr` служит для задания различных атрибутов создаваемого thread'a. Мы всегда будем считать их заданными по умолчанию, подставляя в качестве аргумента значение `NULL`.

Возвращаемые значения

При удачном завершении функция возвращает значение `0` и помещает идентификатор новой нити исполнения по адресу, на который указывает параметр `thread`. В случае ошибки возвращается **положительное значение** (а не отрицательное, как в большинстве системных вызовов и функций!), которое определяет код ошибки, описанный в файле `<errno.h>`. Значение системной переменной `errno` при этом не устанавливается.

Важным отличием этой функции от большинства других системных вызовов и функций является то, что в случае неудачного завершения она **возвращает не отрицательное, а положительное значение**, которое определяет код ошибки, описанный в файле `<errno.h>`. Значение системной переменной `errno` при этом не устанавливается. Результатом выполнения этой функции является появление в системе новой нити исполнения, которая будет выполнять функцию, ассоциированную со thread'ом, передав ей специфицированный параметр, параллельно с уже существовавшими нитями исполнения процесса.

Созданный thread может завершить свою деятельность тремя способами:

- С помощью выполнения функции `pthread_exit()`. Функция никогда не возвращается в вызвавшую ее нить исполнения. Объект, на который указывает параметр этой функции, может быть изучен в другой нити исполнения, например, в породившей завершившийся thread. Этот параметр, следовательно, должен указывать на объект, не являющийся локальным для завершившегося thread'а, например, на статическую переменную;
- С помощью возврата из функции, ассоциированной с нитью исполнения. Объект, на который указывает адрес, возвращаемый функцией, как и в предыдущем случае, может быть изучен в другой нити исполнения, например, в породившей завершившийся thread, и должен указывать на объект, не являющийся локальным для завершившегося thread'а;
- Если в процессе выполняется возврат из функции `main()` или где-либо в процессе (в любой нити исполнения) осуществляется вызов функции `exit()`, это приводит к завершению всех thread'ов процесса.

Прототип и описание функции для завершения нити исполнения

```
#include <pthread.h>
void pthread_exit(void *status);
```

Функция `pthread_exit` служит для завершения нити исполнения (`thread`) текущего процесса.

Функция никогда не возвращается в вызвавший ее `thread`. Объект, на который указывает параметр `status`, может быть впоследствии изучен в другой нити исполнения, например в нити, породившей завершившуюся нить. Поэтому он не должен указывать на динамический объект завершившегося thread'а.

Одним из вариантов получения адреса, возвращаемого завершившимся thread'ом, с одновременным ожиданием его завершения является использование функции `pthread_join()`. Нить исполнения, вызвавшая эту функцию, переходит в состояние **ожидание** до завершения заданного thread'a. Функция позволяет также получить указатель, который вернул завершившийся thread в ОС.

Прототип и описание функции pthread_join()

```
#include <pthread.h>
int pthread_join (pthread_t thread, void **status_addr);
```

Функция `pthread_join` блокирует работу вызвавшей ее нити исполнения до завершения thread'a с идентификатором `thread`. После разблокирования в указатель, расположенный по адресу `status_addr`, заносится адрес, который вернул завершившийся thread либо при выходе из ассоциированной с ним функции, либо при выполнении функции `pthread_exit()`. Если нас не интересует, что вернула нам нить исполнения, в качестве этого параметра можно использовать значение `NULL`.

Возвращаемые значения

Функция возвращает значение `0` при успешном завершении. В случае ошибки возвращается **положительное значение** (а не отрицательное, как в большинстве системных вызовов и функций!), которое определяет код ошибки, описанный в файле `<errno.h>`. Значение системной переменной `errno` при этом не устанавливается.

3-3. Прогон программы с использованием двух нитей исполнения

Для иллюстрации вышесказанного давайте рассмотрим программу, в которой работают две нити исполнения.

```
/* Программа 04-2.c для иллюстрации работы двух нитей исполнения.
Каждая нить исполнения просто увеличивает на 1 разделяемую переменную a. */
#include <pthread.h>
#include <stdio.h>
int a = 0;
/* Переменная a является глобальной статической для всей
программы, поэтому она будет разделяться обеими нитями исполнения.*/
/* Ниже следует текст функции, которая будет
ассоциирована со 2-м thread'ом */
void *mythread(void *dummy)
/* Параметр dummy в нашей функции не используется и присутствует только
для совместимости типов данных. По той же причине функция возвращает
значение void *, хотя это никак не используется в программе.*/
{
    pthread_t mythid; /* Для идентификатора нити исполнения */
    /* Заметим, что переменная mythid является динамической
    локальной переменной функции mythread(), т.е. помещается
    в стеке и, следовательно, не разделяется нитями исполнения. */
```

```
    /* Запрашиваем идентификатор thread'a */
    mythid = pthread_self();
    a = a+1;
    printf("Thread %d, Calculation result = %d\n", mythid, a);
    return NULL;
}
/* Функция main() – она же ассоциированная функция главного thread'a */
int main()
{
    pthread_t thid, mythid;
    int result;
    /* Пытаемся создать новую нить исполнения, ассоциированную с функцией
    mythread(). Передаем ей в качестве параметра значение NULL. В случае
    удачи в переменную thid занесется идентификатор нового thread'a.
    Если возникнет ошибка, то прекратим работу. */
    result = pthread_create( &thid, (pthread_attr_t *)NULL, mythread, NULL);
    if(result != 0){
       printf ("Error on thread create, return value = %d\n", result);
       exit(-1);
    }
    printf("Thread created, thid = %d\n", thid);
    /* Запрашиваем идентификатор главного thread'a */
    mythid = pthread_self();
    a = a+1;
    printf("Thread %d, Calculation result = %d\n", mythid, a);
    /* Ожидаем завершения порожденного thread'a, не интересуясь, какое
    значение он нам вернет. Если не выполнить вызов этой функции, то
    возможна ситуация, когда мы завершим функцию main() до того, как
    выполнится порожденный thread, что автоматически повлечет за
    собой его завершение, исказив результаты. */
    pthread_join(thid, (void **)NULL);
    return 0;
}
```

Программа 04-2.c для иллюстрации работы двух нитей исполнения.

Для сборки исполняемого файла при работе редактора связей необходимо явно подключить библиотеку функций для работы с pthread'ами, которая не подключается автоматически. Это делается с помощью добавления к команде компиляции и редактирования связей параметра `-lpthread` – подключить библиотеку `pthread`. Наберите текст, откомпилируйте эту программу и запустите на исполнение.

Обратите внимание на отличие результатов этой программы от похожей программы, иллюстрировавшей создание нового процесса (З-2 семинар 2). Программа, создававшая новый процесс, печатала дважды одинаковые значения для переменной `a`, так как адресные пространства различных процессов независимы, и каждый процесс прибавлял 1 к своей собственной переменной `a`. Рассматриваемая программа печатает два разных значения, так как переменная `a` является разделяемой, и каждый thread прибавляет 1 к одной и той же переменной.

3-4. Написание, компиляция и прогон программы с использованием трех нитей исполнения

Модифицируйте предыдущую программу, добавив к ней третью нить исполнения.

3-5. Необходимость синхронизации процессов и нитей исполнения, использующих общую память

Все рассмотренные на этом семинаре примеры являются не совсем корректными. В большинстве случаев они работают правильно, однако возможны ситуации, когда совместная деятельность этих процессов или нитей исполнения приводит к неверным и неожиданным результатам. Это связано с тем, что любые неатомарные операции, связанные с изменением содержимого разделяемой памяти, представляют собой критическую секцию процесса или нити исполнения. Вспомните рассмотрение критических секций на лекции.

Вернемся к рассмотрению программ 3-1. При одновременном существовании двух процессов в ОС может возникнуть следующая последовательность выполнения операций во времени:

```
...
Процесс 1: array[0] += 1;
Процесс 2: array[1] += 1;
Процесс 1: array[2] += 1;
Процесс 1: printf(
   "Program 1 was spawn %d times,
   program 2 - %d times, total - %d times\n",
   array[0], array[1], array[2]);
...
```

Тогда печать будет давать неправильные результаты. Естественно, что воспроизвести подобную последовательность действий практически нереально. Мы не сможем подобрать необходимое время старта процессов и степень загруженности вычислительной системы. Но мы можем смоделировать эту ситуацию, добавив в обе программы достаточно длительные пустые циклы перед оператором `array[2] += 1;` Это проделано в следующих программах.

```
/* Программа 1 (04-3a.c) для иллюстрации
некорректной работы с разделяемой памятью */
/* Мы организуем разделяемую память для массива из трех целых чисел. Первый
элемент массива является счетчиком числа запусков программы 1, т.е. данной
программы, второй элемент массива – счетчиком числа запусков программы 2,
третий элемент массива – счетчиком числа запусков обеих программ */
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <errno.h>
int main()
```

```
{
    int *array; /* Указатель на разделяемую память */
    int shmid; /* IPC дескриптор для области разделяемой памяти */
    int new = 1; /* Флаг необходимости инициализации элементов массива */
    char pathname[] = "04-3a.c"; /* Имя файла,
        использующееся для генерации ключа. Файл с таким
        именем должен существовать в текущей директории */
    key_t key; /* IPC ключ */
    long i;
    /* Генерируем IPC ключ из имени файла 04-3a.c в текущей директории
    и номера экземпляра области разделяемой памяти 0 */
    if((key = ftok(pathname,0)) < 0){
        printf("Can\'t generate key\n");
        exit(-1);
    }
    /* Пытаемся эксклюзивно создать разделяемую память для сгенерированного
    ключа, т.е. если для этого ключа она уже существует, системный вызов
    вернет отрицательное значение. Размер памяти определяем как размер
    массива из 3-х целых переменных, права доступа 0666 - чтение и
    запись разрешены для всех */
    if((shmid = shmget(key, 3*sizeof(int), 0666|IPC_CREAT|IPC_EXCL)) < 0){
    /* В случае возникновения ошибки пытаемся определить:
    возникла ли она из-за того, что сегмент разделяемой
    памяти уже существует или по другой причине */
        if(errno != EEXIST){
            /* Если по другой причине - прекращаем работу */
            printf("Can\'t create shared memory\n");
            exit(-1);
        } else {
            /* Если из-за того, что разделяемая память уже существует -
            пытаемся получить ее IPC дескриптор и, в случае удачи,
            сбрасываем флаг необходимости инициализации элементов массива */
            if((shmid = shmget(key, 3*sizeof(int), 0)) < 0){
                printf("Can\'t find shared memory\n");
                exit(-1);
            }
            new = 0;
        }
    }
    /* Пытаемся отобразить разделяемую память в адресное пространство
    текущего процесса. Обратите внимание на то, что для правильного сравнения
    мы явно преобразовываем  значение -1 к указателю на целое.*/
    if((array = (int *)shmat(shmid, NULL, 0)) == (int *)(-1)){
        printf("Can't attach shared memory\n");
        exit(-1);
    }
    /* В зависимости от значения флага new либо инициализируем
    массив, либо увеличиваем соответствующие счетчики */
    if(new){
        array[0] = 1;
        array[1] = 0;
        array[2] = 1;
    } else {
        array[0] += 1;
        for(i=0; i<1000000000L; i++);
        /* Предельное значение для i может меняться в зависимости
        от производительности компьютера */
        array[2] += 1;
    }
    /* Печатаем новые значения счетчиков, удаляем разделяемую память
    из адресного пространства текущего процесса и завершаем работу */
    printf("Program 1 was spawn %d times, program 2 - %d times,
        total - %d times\n", array[0], array[1], array[2]);
    if(shmdt(array) < 0){
```

```
        printf("Can't detach shared memory\n");
        exit(-1);
    }
    return 0;
}
```

Программа 1 (04-3a.c) для иллюстрации некорректной работы с разделяемой памятью.

```
/* Программа 2 (04-3b.c) для иллюстрации
некорректной работы с разделяемой памятью */
/* Мы организуем разделяемую память для массива из трех целых чисел. Первый
элемент массива является счетчиком числа запусков программы 1, т.е. данной
программы, второй элемент массива – счетчиком числа запусков программы 2,
третий элемент массива – счетчиком числа запусков обеих программ */
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <errno.h>
int main()
{
    int *array; /* Указатель на разделяемую память */
    int shmid; /* IPC дескриптор для области разделяемой памяти */
    int new = 1; /* Флаг необходимости инициализации
    элементов массива */
    char pathname[] = "04-3a.c"; /* Имя файла,
        использующееся для генерации ключа. Файл с таким
        именем должен существовать в текущей директории */
    key_t key; /* IPC ключ */
    long i;
    /* Генерируем IPC ключ из имени файла 04-3a.c в текущей
    директории и номера экземпляра области разделяемой памяти 0 */
    if((key = ftok(pathname,0)) < 0){
        printf("Can\'t generate key\n");
        exit(-1);
    }
    /* Пытаемся эксклюзивно создать разделяемую память для
    сгенерированного ключа, т.е. если для этого ключа она
    уже существует, системный вызов вернет отрицательное
    значение. Размер памяти определяем как размер массива
    из трех целых переменных, права доступа 0666 – чтение
    и запись разрешены для всех */
    if((shmid = shmget(key, 3*sizeof(int), 0666|IPC_CREAT|IPC_EXCL)) < 0){
    /* В случае ошибки пытаемся определить, возникла ли она из-за того,
    что сегмент разделяемой памяти уже существует или по другой причине */
        if(errno != EEXIST){
            /* Если по другой причине – прекращаем работу */
            printf("Can\'t create shared memory\n");
            exit(-1);
        } else {
            /* Если из-за того, что разделяемая память уже
            существует – пытаемся получить ее IPC дескриптор
            и, в случае удачи, сбрасываем флаг необходимости
            инициализации элементов массива */
            if((shmid = shmget(key, 3*sizeof(int), 0)) < 0){
                printf("Can\'t find shared memory\n");
                exit(-1);
            }
            new = 0;
        }
    }
    /* Пытаемся отобразить разделяемую память в адресное
    пространство текущего процесса. Обратите внимание на то,
    что  для правильного сравнения мы явно преобразовываем
    значение -1 к указателю на целое.*/
```

```
    if((array = (int *)shmat(shmid, NULL, 0)) == (int *)(-1)){
        printf("Can't attach shared memory\n");
        exit(-1);
    }
    /* В зависимости от значения флага new либо инициализируем
    массив, либо увеличиваем соответствующие счетчики */
    if(new){
        array[0] = 0;
        array[1] = 1;
        array[2] = 1;
    } else {
        array[1] += 1;
        for(i=0; i<1000000000L; i++);
        /* Предельное значение для i может меняться в зависимости
        от производительности компьютера */
        array[2] += 1;
    }
    /* Печатаем новые значения счетчиков, удаляем разделяемую память
    из адресного пространства текущего процесса и завершаем работу */
    printf("Program 1 was spawn %d times,
        program 2 - %d times, total - %d times\n",
        array[0], array[1], array[2]);
    if(shmdt(array) < 0){
        printf("Can't detach shared memory\n");
        exit(-1);
    }
    return 0;
}
```

Программа 2 (04-3b.c) для иллюстрации некорректной работы с разделяемой памятью.

Наберите программы, сохраните под именами `04-3a.c` и `04-3b.c` соответственно, откомпилируйте их и запустите любую из них один раз для создания и инициализации разделяемой памяти. Затем запустите другую и, пока она находится в цикле, запустите, например, с другого виртуального терминала, снова первую программу. Вы получите неожиданный результат: количество запусков по отдельности не будет соответствовать количеству запусков вместе.

Как мы видим, для написания корректно работающих программ необходимо обеспечивать взаимоисключение при работе с разделяемой памятью и, может быть, взаимную очередность доступа к ней. Это можно сделать с помощью рассмотренных в лекции алгоритмов синхронизации, например, алгоритма Петерсона или алгоритма булочной.

Задача: модифицируйте программы из этого раздела для корректной работы с помощью алгоритма Петерсона.

Семинар #5: Семафоры в UNIX как средство синхронизации процессов

Семафоры в UNIX. Отличие операций над UNIX-семафорами от классических операций. Создание массива семафоров или доступ к уже существующему массиву. Системный вызов semget(). Выполнение операций над семафорами. Системный вызов semop(). Удаление набора семафоров из системы с помощью команды ipcrm или системного вызова semctl(). Понятие о POSIX-семафорах.

1. Семафоры в UNIX. Отличие операций над UNIX-семафорами от классических операций

На семинаре 4 речь шла о необходимости синхронизации работы процессов для их корректного взаимодействия через разделяемую память. Одним из первых механизмов, предложенных для синхронизации поведения процессов, стали семафоры, концепцию которых описал Дейкстра (Dijkstra) в 1965 году. При разработке средств SV IPC семафоры вошли в их состав как неотъемлемая часть. Набор операций над семафорами SV IPC отличается от классического набора операций `{P, V}`, предложенного Дейкстрой. Он включает три операции:

- `A(S, n)` – увеличить значение семафора `S` на величину `n`;
- `D(S, n)` – пока значение семафора `S < n`, процесс блокируется. Далее `S = S-n;`
- `Z(S)` – процесс блокируется до тех пор, пока значение семафора `S` не станет равным `0`.

Изначально все IPC-семафоры инициируются нулевым значением.

Тогда классической операции `P(S)` соответствует операция `D(S,1)`, а классической операции `V(S)` соответствует операция `A(S,1)`. Аналогом ненулевой инициализации семафоров Дейкстры значением n может служить выполнение операции `A(S, n)` сразу после создания семафора `s`, с обеспечением атомарности создания семафора и ее выполнения посредством другого семафора. На лекции мы показали, что классические семафоры реализуются через семафоры SV IPC. Обратное не является верным. Используя операции `P(S)` и `V(S)`, мы не сумеем реализовать операцию `Z(S)`.

Поскольку IPC-семафоры являются составной частью средств System V IPC, то для них верно все, что говорилось об этих средствах. IPC-семафоры являются средством связи с непрямой адресацией, требуют инициализации для организации взаимодействия процессов и специальных действий для освобождения системных ресурсов по его окончании. Пространством имен IPC-семафоров является множество значений ключа, генерируемых с помощью функции `ftok()`. Для совершения операций над семафорами системным вызовам в качестве параметра передаются IPC-дескрипторы семафоров, однозначно идентифицирующих их во всей вычислительной системе, а вся информация о семафорах располагается в адресном пространстве ядра ОС. Это позволяет организовывать через семафоры взаимодействие процессов, даже не находящихся в системе одновременно.

2. Создание массива семафоров или доступ к уже существующему. Системный вызов semget()

В целях экономии системных ресурсов ОС UNIX позволяет создавать не по одному семафору для каждого конкретного значения ключа, а связывать с

ключом целый массив семафоров (в Linux – до 500 семафоров в массиве, хотя это количество может быть уменьшено). Для создания массива семафоров, ассоциированного с определенным ключом, или доступа по ключу к уже существующему массиву используется системный вызов `semget()`, являющийся аналогом системного вызова `shmget()` для разделяемой памяти, который возвращает значение IPC-дескриптора для этого массива. При этом применяются те же способы создания и доступа, что и для разделяемой памяти. Вновь созданные семафоры инициируются нулевым значением.

Прототип и описание системного вызова semget()

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semget(key_t key, int nsems, int semflg);
```

Системный вызов `semget` предназначен для выполнения операции доступа к массиву IPC-семафоров и, в случае ее успешного завершения, возвращает дескриптор SV IPC для этого массива (целое неотрицательное число, однозначно характеризующее массив семафоров внутри ВС и использующееся в дальнейшем для других операций с ним).

Параметр `key` является ключом SV IPC для массива семафоров, т. е. фактически его именем из пространства имен SV IPC. В качестве значения этого параметра может использоваться значение ключа, полученное с помощью функции `ftok()`, или специальное значение `IPC_PRIVATE`. Использование значения `IPC_PRIVATE` **всегда** приводит к попытке создания нового массива семафоров с ключом, который не совпадает со значением ключа ни одного из уже существующих массивов и не может быть получен с помощью функции `ftok()` ни при одной комбинации ее параметров.

Параметр `nsems` определяет количество семафоров в создаваемом или уже существующем массиве. В случае, если массив с указанным ключом уже имеется, но его размер не совпадает с указанным в параметре `nsems`, констатируется возникновение ошибки.

Параметр `semflg` – флаги, он играет роль только при создании нового массива семафоров и определяет права различных пользователей при доступе к массиву, а также необходимость создания нового массива и поведение системного вызова при попытке создания. Он является некоторой комбинацией (с помощью операции побитовое или – «`|`») следующих предопределенных значений и восьмеричных прав доступа:

- `IPC_CREAT` – если массива для указанного ключа не существует, он должен быть создан

- IPC_EXCL – применяется совместно с флагом IPC_CREAT. При совместном их использовании и существовании массива с указанным ключом, доступ к массиву не производится и констатируется ошибка, при этом переменная errno, описанная в файле <errno.h>, примет значение EEXIST
- 0400 — разрешено чтение для пользователя, создавшего массив
- 0200 — разрешена запись для пользователя, создавшего массив
- 0040 — разрешено чтение для группы пользователя, создавшего массив
- 0020 — разрешена запись для группы пользователя, создавшего массив
- 0004 — разрешено чтение для всех остальных пользователей
- 0002 — разрешена запись для всех остальных пользователей

Вновь созданные семафоры инициируются нулевым значением.

Возвращаемое значение

Системный вызов возвращает значение дескриптора SV IPC для массива семафоров при нормальном завершении и значение -1 при возникновении ошибки.

3. Выполнение операций над семафорами. Системный вызов semop()

Для выполнения операций A, D и Z над семафорами из массива используется системный вызов semop(), обладающий довольно сложной семантикой. Его можно применять не только для выполнения всех трех операций, но еще и для нескольких семафоров в массиве IPC-семафоров одновременно. Для правильного использования этого вызова необходимо выполнить следующие действия:

1. Выяснить, для каких семафоров из массива предстоит выполнить операции. Все операции реально совершаются только перед успешным возвращением из системного вызова, т.е. если вы хотите выполнить операции A(S1,5) и Z(S2) в одном вызове и оказалось, что S2 != 0, то значение семафора S1 не будет изменено до тех пор, пока значение S2 не станет равным 0. Порядок выполнения операций в случае, когда процесс не переходит в состояние **ожидание**, не определен. Например, при одновременном выполнении операций A(S1,1) и D(S2,1) в случае S2 > 1 неизвестно, что произойдет раньше – уменьшится значение семафора S2 или увеличится значение семафора S1. Если порядок очень важен, лучше применить несколько вызовов вместо одного.
2. После того как вы определились с количеством семафоров и совершаемыми операциями, необходимо завести в программе массив из элементов типа struct sembuf с размерностью, равной определенному количеству семафоров (если операция совершается только над одним семафором, можно конечно обойтись просто переменной). Каждый элемент этого массива будет соответствовать операции над одним семафором.
3. Заполнить элементы массива. В поле sem_flg каждого элемента нужно занести значение 0. В поля sem_num и sem_op следует занести номера семафоров в массиве IPC семафоров и соответствующие коды операций. Семафоры нумеруются, начиная с 0. Если у вас в массиве всего один семафор, то он будет иметь номер 0. Операции кодируются так:
    - для выполнения операции A(S, n) значение поля sem_op должно быть равно n;

- для выполнения операции `D(S, n)` значение поля `sem_op` должно быть равно `-n`;
- для выполнения операции `Z(S)` значение поля `sem_op` должно быть равно `0`.

4. В качестве второго параметра системного вызова `semop()` указать адрес заполненного массива, а в качестве третьего параметра – ранее определенное количество семафоров, над которыми совершаются операции.

Прототип и описание системного вызова semop()

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semop(int semid, struct sembuf *sops, int nsops);
```

Системный вызов semop предназначен для выполнения операций `A`, `D` и `Z` (см. п. 1).

Параметр `semid` является дескриптором SV IPC для набора семафоров, т. е. значением, которое вернул системный вызов `semget()` при создании набора семафоров или при его поиске по ключу.

Каждый из `nsops` элементов массива, на который указывает параметр sops, определяет операцию, которая должна быть совершена над каким-либо семафором из массива IPC семафоров, и имеет тип структуры `struct sembuf`, в которую входят следующие переменные:

- `short sem_num` — номер семафора в массиве IPC семафоров (нумеруются, начиная с `0`);
- `short sem_op` — выполняемая операция;
- `short sem_flg` — флаги для выполнения операции. Мы всегда будем считать эту переменную равной `0`.

Значение элемента структуры `sem_op` определяется следующим образом:

- для выполнения операции `A(S, n)` значение должно быть равно `n`;
- для выполнения операции `D(S, n)` значение должно быть равно `-n`;
- для выполнения операции `Z(S)` значение должно быть равно `0`.

Семантика системного вызова подразумевает, что все операции будут в реальности выполнены над семафорами только перед успешным возвращением из системного вызова. Если при выполнении операций `D` или `Z` процесс перешел в состояние ожидания, то он может быть выведен из этого состояния при возникновении следующих форс-мажорных ситуаций:

- массив семафоров был удален из системы;
- процесс получил сигнал, который должен быть обработан.

В этом случае происходит возврат из системного вызова с констатацией

ошибочной ситуации.

Возвращаемое значение

Системный вызов возвращает значение `0` при нормальном завершении и значение `-1` при возникновении ошибки.

3-1. Прогон примера с использованием семафора

Для иллюстрации сказанного рассмотрим простейшие программы, синхронизирующие свои действия с помощью семафоров

```
/* Программа s5-1a.c для иллюстрации работы с семафорами */
/* Эта программа получает доступ к одному системному семафору,
   ждет, пока его значение не станет больше или равным 1
   после запусков программы s5-1b.c,а затем уменьшает его на 1*/
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <stdio.h>
int main()
{
   int semid; /* IPC дескриптор для массива IPC семафоров */
   char pathname[] = "s5-1a.c"; /* Имя файла,
      использующееся для генерации ключа. Файл с таким
      именем должен существовать в текущей директории */
   key_t key; /* IPC ключ */
   struct sembuf mybuf; /* Структура для задания операции над семафором */
   /* Генерируем IPC-ключ из имени файла 05-1a.c в текущей
      директории и номера экземпляра массива семафоров 0 */
   if((key = ftok(pathname,0)) < 0){
      printf("Can\'t generate key\n");
      exit(-1);
   }
   /* Пытаемся получить доступ по ключу к массиву семафоров, если он
      существует, или создать его из одного семафора, если его еще не
      существует, с правами доступа  read & write для всех пользователей */
   if((semid = semget(key, 1, 0666 | IPC_CREAT)) < 0){
      printf("Can\'t get semid\n");
      exit(-1);
   }
   /* Выполним операцию D(semid1,1) для нашего массива семафоров.
      Для этого сначала заполним нашу структуру. Флаг полагаем
      равным 0. Наш массив семафоров состоит из одного
      семафора с номером 0. Код операции -1.*/
   mybuf.sem_op = -1;
   mybuf.sem_flg = 0;
   mybuf.sem_num = 0;
   if(semop(semid, &mybuf, 1) < 0){
      printf("Can\'t wait for condition\n");
      exit(-1);
   }
   printf("Condition is present\n");
   return 0;
}
```

Программа s5-1a.c для иллюстрации работы с семафорами

```
/* Программа s5-1b.c для иллюстрации работы с семафорами */
/* Эта программа получает доступ к одному системному семафору
   и увеличивает его на 1*/
```

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <stdio.h>
int main()
{
   int semid; /* IPC дескриптор для массива IPC семафоров */
   char pathname[] = "s5-1a.c"; /* Имя файла,
      использующееся для генерации ключа. Файл с таким
      именем должен существовать в текущей директории */
   key_t key; /* IPC ключ */
   struct sembuf mybuf; /* Структура для задания операции над семафором */
   /* Генерируем IPC ключ из имени файла 05-1a.c в текущей
      директории и номера экземпляра массива семафоров 0 */
   if((key = ftok(pathname,0)) < 0){
      printf("Can\'t generate key\n");
      exit(-1);
   }
   /* Пытаемся получить доступ по ключу к массиву семафоров, если он
      существует, или создать его из одного семафора, если его еще не
      существует, с правами доступа read & write для всех пользователей */
   if((semid = semget(key, 1, 0666 | IPC_CREAT)) < 0){
      printf("Can\'t get semid\n");
      exit(-1);
   }
   /* Выполним операцию A(semid1, 1) для нашего массива семафоров.
      Для этого сначала заполним нашу структуру. Флаг, как обычно,
      полагаем равным 0. Наш массив семафоров состоит из одного
      семафора с номером 0. Код операции 1.*/
   mybuf.sem_op = 1;
   mybuf.sem_flg = 0;
   mybuf.sem_num = 0;
   if(semop(semid, &mybuf, 1) < 0){
      printf("Can\'t wait for condition\n");
      exit(-1);
   }
   printf("Condition is set\n");
   return 0;
}
```

Программа s5-1b.c для иллюстрации работы с семафорами

Первая программа выполняет над семафором `s` операцию `D(S,1)`, вторая программа выполняет над тем же семафором операцию `A(S,1)`. Если семафора в системе не существует, любая программа создает его перед выполнением операции. Поскольку при создании семафор всегда инициируется `0`, то первая программа может работать без блокировки только после запуска второй программы. Откомпилируйте программы `s5-1a.c` и `s5-1b.c` и проверьте правильность их работы.

3-2. Изменение предыдущего примера

Измените программы `s5-1a.c` и `s5-1b.c` так, чтобы первая программа могла работать без блокировки после не менее 5 запусков второй программы.

4. Удаление набора семафоров из системы с помощью команды ipcrm или системного вызова semctl()

Как мы видели в примерах, массив семафоров может продолжать существовать в системе и после завершения использовавших его процессов, а семафоры будут сохранять свое значение. Это может привести к некорректному поведению программ, предполагающих, что семафоры были только что созданы и, следовательно, имеют нулевое значение. Необходимо удалять семафоры из системы перед запуском таких программ или перед их завершением. Для удаления семафоров можно воспользоваться командами `ipcs` и `ipcrm`. Команда `ipcrm` в этом случае должна иметь вид

```
ipcrm sem <IPC идентификатор>
```

Для этой же цели можно использовать системный вызов `semctl()`, который умеет выполнять и другие операции над массивом семафоров.

Прототип и описание системного вызова semctl()

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semctl(int semid, int semnum, int cmd, union semun arg);
```

Системный вызов `semctl` предназначен для получения информации о массиве IPC семафоров, изменения его атрибутов и удаления его из системы.

Мы будем применять системный вызов `semctl` только для удаления массива семафоров из системы. Параметр `semid` является дескриптором SV IPC для массива семафоров, т.е. значением, которое вернул системный вызов `semget()` при создании массива или при его поиске по ключу.

В качестве параметра `cmd` мы всегда будем передавать значение `IPC_RMID` – команду для удаления сегмента разделяемой памяти с заданным идентификатором. Параметры `semnum` и `arg` для этой команды не используются, поэтому мы всегда будем подставлять вместо них значение `0`.

Если какие-либо процессы находились в состоянии ожидание для семафоров из удаляемого массива при выполнении системного вызова `semop()`, то они будут разблокированы и вернутся из вызова `semop()` с индикацией ошибки.

Возвращаемое значение

Системный вызов возвращает значение `0` при нормальном завершении и значение `-1` при возникновении ошибки.

3-3. Написание, компиляция и прогон программы с организацией взаимоисключения с помощью семафоров для двух процессов, взаимодействующих через разделяемую память

На лекции было показано, что любые неатомарные операции, связанные с изменением содержимого разделяемой памяти, представляют собой критическую секцию процесса или нити исполнения. Модифицируйте программы 3-5 семинара 4, которые иллюстрировали некорректную работу через разделяемую память, обеспечив с помощью семафоров взаимоисключения для их правильной работы.

3-4. Написание, компиляция и прогон программы с организацией взаимной очередности с помощью семафоров для двух процессов, взаимодействующих через pipe

В материалах семинара 3, когда речь шла о связи родственных процессов через pipe, отмечалось, что pipe является однонаправленным каналом связи, и что для организации связи через один pipe в двух направлениях необходимо использовать механизмы взаимной синхронизации процессов. Организуйте двустороннюю поочередную связь процесса-родителя и процесса-ребенка через pipe, используя для синхронизации семафоры, модифицировав программу 3-4 семинара 3.

5. Понятие о POSIX-семафорах

В стандарте POSIX вводятся другие семафоры, полностью аналогичные семафорам Дейкстры. Для инициализации значения таких семафоров применяется функция `sem_init()`, аналогом операции `P` служит функция `sem_wait()`, а аналогом операции `V` – функция `sem_post()`. Так как в Linux такие семафоры реализованы только для нитей исполнения одного процесса, то поэтому на них мы останавливаться не будем.

Семинар #6: Очереди сообщений в UNIX

Сообщения как средства связи и средства синхронизации процессов. Очереди сообщений в UNIX как составная часть SV IPC. Создание очереди сообщений или доступ к уже существующей. Системный вызов msgget(). Реализация примитивов send и receive. Системные вызовы msgsnd() и msgrcv(). Удаление очереди сообщений из системы с помощью команды ipcrm или системного вызова msgctl(). Понятие мультиплексирования. Мультиплексирование сообщений. Модель взаимодействия процессов клиент-сервер. Неравноправность клиента и сервера. Использование очередей сообщений для синхронизации работы процессов.

1. Сообщения как средства связи и средства синхронизации процессов

Третьим и последним, наиболее семантически нагруженным средством, входящим в SV IPC, являются очереди сообщений. На лекции мы говорили о модели сообщений как о способе взаимодействия процессов через линии

связи, в котором на передаваемую информацию накладывается определенная структура, так что процесс, принимающий данные, может четко определить, где заканчивается одна порция информации и начинается другая. Такая модель позволяет задействовать одну и ту же линию связи для передачи данных в двух направлениях между несколькими процессами. Мы также рассматривали возможность использования сообщений с встроенными механизмами взаимоисключения и блокировки при чтении из пустого буфера и записи в переполненный буфер для организации синхронизации процессов.

Рассмотрим использовании очередей сообщений SV IPC для обеспечения обеих перечисленных функций.

2. Очереди сообщений в UNIX как составная часть SV IPC

Так как очереди сообщений входят в состав средств SV IPC, для них верно все, что говорилось ранее об этих средствах. Очереди сообщений, как и семафоры, и разделяемая память, являются средством связи с непрямой адресацией, требуют инициализации для организации взаимодействия процессов и специальных действий для освобождения системных ресурсов по окончании взаимодействия. Пространством имен очередей сообщений является то же самое множество значений ключа, генерируемых с помощью функции `ftok()`. Для выполнения примитивов `send` и `receive`, введенных в лекции, соответствующим системным вызовам в качестве параметра передаются IPC-дескрипторы очередей сообщений, однозначно идентифицирующих их во всей вычислительной системе.

Очереди сообщений располагаются в адресном пространстве ядра ОС в виде однонаправленных списков и имеют ограничение по объему информации, хранящейся в каждой очереди. Каждый элемент списка представляет собой отдельное сообщение. Сообщения имеют атрибут, называемый типом сообщения. Выборка сообщений из очереди (выполнение примитива `receive`) может осуществляться тремя способами:

1. В порядке FIFO, независимо от типа сообщения.
2. В порядке FIFO для сообщений конкретного типа.
3. Первым выбирается сообщение с минимальным типом, не превышающим некоторого заданного значения, пришедшее раньше других сообщений с тем же типом.

Реализация примитивов `send` и `receive` обеспечивает скрытое от пользователя взаимоисключение во время помещения сообщения в очередь или его получения из очереди. Также она обеспечивает блокировку процесса при попытке выполнить примитив `receive` над пустой очередью или очередью, в которой отсутствуют сообщения запрошенного типа, или при попытке выполнить примитив `send` для очереди, в которой нет свободного места.

Очереди сообщений, как и другие средства SV IPC, позволяют организовать взаимодействие процессов, не находящихся одновременно в вычислительной системе.

3. Создание очереди сообщений или доступ к уже существующей. Системный вызов msgget()

Для создания очереди сообщений, ассоциированной с определенным ключом, или доступа по ключу к уже существующей очереди используется системный вызов `msgget()`, являющийся аналогом системных вызовов `shmget()` для разделяемой памяти и `semget()` для массива семафоров, который возвращает значение IPC-дескриптора для этой очереди. При этом существуют те же способы создания и доступа, что и для разделяемой памяти или семафоров.

Прототип и описание системного вызова msgget()

```
#include <types.h>
#include <ipc.h>
#include <msg.h>

int msgget(key_t key, int msgflg);
```

Системный вызов `msgget` предназначен для выполнения операции доступа к очереди сообщений и, в случае ее успешного завершения, возвращает дескриптор SV IPC для этой очереди (целое неотрицательное число, однозначно характеризующее очередь сообщений внутри ВС и использующееся в дальнейшем для других операций с ней).

Параметр `key` является ключом SV IPC для очереди сообщений, т. е. фактически ее именем из пространства имен SV IPC. В качестве значения этого параметра может быть использовано значение ключа, полученное с помощью функции `ftok()`, или специальное значение `IPC_PRIVATE`. Использование значения `IPC_PRIVATE` всегда приводит к попытке создания новой очереди сообщений с ключом, который не совпадает со значением ключа ни одной из уже существующих очередей и не может быть получен с помощью функции `ftok()` ни при одной комбинации ее параметров.

Параметр `msgflg` – флаги, он играет роль только при создании новой очереди сообщений и определяет права различных пользователей при доступе к очереди, а также необходимость создания новой очереди и поведение системного вызова при попытке создания. Он является некоторой комбинацией (с помощью операции побитовое или – «|») следующих предопределенных значений и восьмеричных прав доступа:

- `IPC_CREAT` — если очереди для указанного ключа не существует, она должна быть создана;

- IPC_EXCL — применяется совместно с флагом IPC_CREAT. При совместном их использовании и существовании массива с указанным ключом доступ к очереди не производится и констатируется ошибочная ситуация, при этом переменная errno, описанная в файле <errno.h>, примет значение EEXIST;
- 0400 — разрешено чтение для пользователя, создавшего очередь;
- 0200 — разрешена запись для пользователя, создавшего очередь;
- 0040 — разрешено чтение для группы пользователя, создавшего очередь;
- 0020 — разрешена запись для группы пользователя, создавшего очередь;
- 0004 — разрешено чтение для всех остальных пользователей;
- 0002 — разрешена запись для всех остальных пользователей;

Очередь сообщений имеет ограничение по общему количеству хранимой информации, которое может быть изменено администратором системы. Текущее значение ограничения можно узнать с помощью команды

```
ipcs -l
```

Возвращаемое значение

Системный вызов возвращает значение дескриптора SV IPC для очереди сообщений при нормальном завершении и значение -1 при возникновении ошибки.

4. Реализация примитивов send и receive. Системные вызовы msgsnd() и msgrcv()

Для выполнения примитива send используется системный вызов msgsnd(), копирующий пользовательское сообщение в очередь сообщений, заданную IPC-дескриптором. При этом:

- Тип данных struct msgbuf не является типом данных для пользовательских сообщений, а представляет собой лишь шаблон для создания таких типов. Пользователь сам должен создать структуру для своих сообщений, в которой первым полем должна быть переменная типа long, содержащая положительное значение типа сообщения.
- В качестве третьего параметра – длины сообщения – указывается не вся длина структуры данных, соответствующей сообщению, а только длина полезной информации, т. е. информации, располагающейся в структуре данных после типа сообщения. Это значение может быть и равным 0 в случае, когда вся полезная информация заключается в самом факте прихода сообщения (сообщение используется как сигнальное средство связи).
- Как правило, мы будем использовать нулевое значение флага системного вызова, которое приводит к блокировке процесса при отсутствии свободного места в очереди сообщений.

Прототип и описание системного вызова msgsnd()

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgsnd(int msqid, struct msgbuf *ptr, int length, int flag);
```

Системный вызов `msgsnd` предназначен для помещения сообщения в очередь сообщений, т. е. является реализацией примитива send.

Параметр `msqid` является дескриптором SV IPC для очереди, в которую отправляется сообщение, т. е. значением, которое вернул системный вызов `msgget()` при создании очереди или при ее поиске по ключу.

Структура `struct msgbuf` описана в файле `<sys/msg.h>` как

```
struct msgbuf {
   long mtype;
   char mtext[1];
};
```

Она представляет собой некоторый шаблон структуры сообщения пользователя. Сообщение пользователя – это структура, первый элемент которой обязательно имеет тип `long` и содержит тип сообщения, а далее следует информативная часть теоретически произвольной длины (в Linux она ограничена размером 4080 байт и может быть еще уменьшена), содержащая собственно суть сообщения. Например:

```
struct mymsgbuf {
   long mtype;
   char mtext[1024];
} mybuf;
```

При этом информация вовсе не обязана быть текстовой, например:

```
struct mymsgbuf {
   long mtype;
   struct {
      int iinfo;
      float finfo;
   } info;
} mybuf;
```

Тип сообщения должен быть строго положительным числом. Действительная длина полезной части информации (т. е. информации, расположенной в структуре после типа сообщения) должна быть передана системному вызову в качестве параметра `length`. Этот параметр может быть равен и `0`, если вся полезная информация заключается в самом факте наличия сообщения. Системный вызов копирует сообщение, расположенное по адресу, на который указывает параметр `ptr`, в очередь сообщений, заданную дескриптором `msqid`.

Параметр `flag` может принимать два значения: `0` и `IPC_NOWAIT`. Если значение флага равно `0`, и в очереди не хватает места для того, чтобы поместить сообщение, то системный вызов блокируется до тех пор, пока не освободится место. При значении флага `IPC_NOWAIT` системный вызов в этой ситуации не

блокируется, а констатирует возникновение ошибки с установлением значения переменной `errno`, описанной в файле `<errno.h>`, равным `EAGAIN`.

Возвращаемое значение

Системный вызов возвращает значение `0` при нормальном завершении и значение `-1` при возникновении ошибки.

Примитив `receive` реализуется системным вызовом `msgrcv()`. При этом:

- Тип данных `struct msgbuf`, как и для вызова `msgsnd()`, является лишь шаблоном для пользовательского типа данных.
- Способ выбора сообщения (см. п. 2) задается нулевым, положительным или отрицательным значением параметра `type`. Точное значение типа выбранного сообщения можно определить из соответствующего поля структуры, в которую системный вызов скопирует сообщение.
- Системный вызов возвращает длину только полезной части скопированной информации, т. е. информации, расположенной в структуре после поля типа сообщения.
- Выбранное сообщение удаляется из очереди сообщений.
- В качестве параметра `length` указывается максимальная длина полезной части информации, которая может быть размещена в структуре, адресованной параметром `ptr`.
- Мы будем, как правило, пользоваться нулевым значением флагов для системного вызова, которое приводит к блокировке процесса в случае отсутствия в очереди сообщений с запрошенным типом и к ошибочной ситуации в случае, когда длина информативной части выбранного сообщения превышает длину, специфицированную в параметре `length`.

Прототип и описание системного вызова msgrcv()

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgrcv(int msqid, struct msgbuf *ptr, int length, long type, int flag);
```

Системный вызов `msgrcv` предназначен для получения сообщения из очереди сообщений, т. е. является реализацией примитива receive.

Параметр `msqid` является дескриптором SV IPC для очереди, из которой должно быть получено сообщение, т. е. значением, которое вернул системный вызов `msgget()` при создании очереди или при ее поиске по ключу.

Параметр `type` определяет способ выборки сообщения из очереди следующим образом

Способ выборки	Значение параметра type
В порядке FIFO, независимо от типа сообщения	0
В порядке FIFO для сообщений с типом n	n
Первым выбирается сообщение с минимальным типом, не превышающим значения n, пришедшее ранее всех других сообщений с тем же типом	-n

Структура struct msgbuf описана в файле <sys/msg.h> как

```
struct msgbuf {
   long mtype;
   char mtext[1];
};
```

Она представляет собой некоторый шаблон структуры сообщения пользователя. Сообщение пользователя – это структура, первый элемент которой обязательно имеет тип long и содержит тип сообщения, а далее следует информативная часть теоретически произвольной длины, содержащая собственно суть сообщения.

При этом информация вовсе не обязана быть текстовой, например:

```
struct mymsgbuf {
   long mtype;
   struct {
      int iinfo;
      float finfo;
   } info;
} mybuf;
```

Параметр length должен содержать максимальную длину полезной части информации (т. е. информации, расположенной в структуре после типа сообщения), которая может быть размещена в сообщении.

В случае удачи системный вызов копирует выбранное сообщение из очереди сообщений по адресу, указанному в параметре ptr, одновременно удаляя его из очереди сообщений.

Параметр flag может принимать значение 0 или быть какой-либо комбинацией флагов IPC_NOWAIT и MSG_NOERROR. Если флаг IPC_NOWAIT не установлен и очередь сообщений пуста или в ней нет сообщений с заказанным типом, то системный вызов блокируется до появления запрошенного сообщения. При установлении флага IPC_NOWAIT системный вызов в этой ситуации не блокируется, а констатирует возникновение ошибки с установлением значения переменной errno, описанной в файле <errno.h>, равным EAGAIN. Если действительная длина полезной части

информации в выбранном сообщении превышает значение, указанное в параметре `length` и флаг `MSG_NOERROR` не установлен, то выборка сообщения не производится, и фиксируется наличие ошибочной ситуации. Если флаг `MSG_NOERROR` установлен, то в этом случае ошибки не возникает, а сообщение копируется в сокращенном виде.

Возвращаемое значение

Системный вызов возвращает при нормальном завершении действительную длину полезной части информации (т. е. информации, расположенной в структуре после типа сообщения), скопированной из очереди сообщений, и значение `-1` при возникновении ошибки.

Максимально возможная длина информативной части сообщения в ОС Linux составляет 4080 байт и может быть уменьшена при генерации системы. Текущее значение максимальной длины можно определить с помощью команды

```
ipcs -l
```

5. Удаление очереди сообщений из системы с помощью команды ipcrm или системного вызова msgctl()

После завершения процессов, использовавших очередь сообщений, она не удаляется из системы автоматически, а продолжает сохраняться в системе вместе со всеми невостребованными сообщениями до тех пор, пока не будет выполнена специальная команда или специальный системный вызов. Для удаления очереди сообщений можно воспользоваться уже знакомой нам командой ipcrm, которая в этом случае примет вид:

```
ipcrm msg <IPC идентификатор>
```

Для получения IPC идентификатора очереди сообщений примените команду `ipcs`. Можно удалить очередь сообщений и с помощью системного вызова `msgctl()`. Этот вызов умеет выполнять и другие операции над очередью сообщений. Если какой-либо процесс находился в состоянии **ожидание** при выполнении системного вызова `msgrcv()` или `msgsnd()` для удаляемой очереди, то он будет разблокирован, и системный вызов констатирует наличие ошибочной ситуации.

Прототип и описание системного вызова msgctl()

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
```

```
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

Системный вызов `msgctl` предназначен для получения информации об очереди сообщений, изменения ее атрибутов и удаления из системы.

Мы будем пользоваться системным вызовом `msgctl` только для удаления очереди сообщений из системы. Параметр `msqid` является дескриптором SV IPC для очереди сообщений, т. е. значением, которое вернул системный вызов `msgget()` при создании очереди или при ее поиске по ключу.

В качестве параметра `cmd` мы всегда будем передавать значение `IPC_RMID` – команду для удаления очереди сообщений с заданным идентификатором. Параметр `buf` для этой команды не используется, поэтому мы всегда будем подставлять туда значение `NULL`.

Возвращаемое значение

Системный вызов возвращает значение `0` при нормальном завершении и значение `-1` при возникновении ошибки.

3-1. Прогон примера с однонаправленной передачей текстовой информации

Для иллюстрации сказанного рассмотрим две простые программы.

```
/* Программа s6-1a.c для иллюстрации работы с очередями сообщений */
/* Эта программа получает доступ к очереди сообщений,
   отправляет в нее 5 текстовых сообщений с типом 1 и одно пустое
   сообщение с типом 255, которое будет служить для программы
   s6-1b.c сигналом прекращения работы. */
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <string.h>
#include <stdio.h>
#define LAST_MESSAGE 255 /* Тип сообщения для
    прекращения работы программы s6-1b.c */
int main()
{
    int msqid; /* IPC дескриптор для очереди сообщений */
    char pathname[] = "s6-1a.c"; /* Имя файла,
        использующееся для генерации ключа. Файл с таким
        именем должен существовать в текущей директории */
    key_t key; /* IPC ключ */
    int i, len; /* Счетчик цикла и длина информативной части сообщения */
    /* Пользовательская структура для сообщения */
    struct mymsgbuf
    {
        long mtype;
        char mtext[81];
    } mybuf;
    /* Генерируем IPC ключ из имени файла s6-1a.c в текущей
       директории и номера экземпляра очереди сообщений 0. */
    if((key = ftok(pathname,0)) < 0){
        printf("Can\'t generate key\n");
        exit(-1);
```

```
    }
    /* Пытаемся получить доступ по ключу к очереди сообщений,
       если она существует, или создать ее, с правами доступа
       read & write для всех пользователей */
    if((msqid = msgget(key, 0666 | IPC_CREAT)) < 0){
        printf("Can\'t get msqid\n");
        exit(-1);
    }
    /* Посылаем в цикле 5 сообщений с типом 1 в
       очередь сообщений, идентифицируемую msqid.*/
    for (i=1; i<=5; i++){
        /* Сначала заполняем структуру для нашего сообщения
           и определяем длину информативной части */
        mybuf.mtype = 1;
        strcpy(mybuf.mtext, "This is text message");
        len = strlen(mybuf.mtext)+1;
        /* Отсылаем сообщение. В случае ошибки сообщаем об
           этом и удаляем очередь сообщений из системы. */
        if (msgsnd(msqid, (struct msgbuf *) &mybuf, len, 0) < 0){
            printf("Can\'t send message to queue\n");
            msgctl(msqid, IPC_RMID, (struct msqid_ds *) NULL);
            exit(-1);
        }
    }
    /* Отсылаем сообщение, которое заставит получающий процесс
       прекратить работу, с типом LAST_MESSAGE и длиной 0 */
    mybuf.mtype = LAST_MESSAGE;
    len = 0;
    if (msgsnd(msqid, (struct msgbuf *) &mybuf, len, 0) < 0){
        printf("Can\'t send message to queue\n");
        msgctl(msqid, IPC_RMID, (struct msqid_ds *) NULL);
        exit(-1);
    }
    return 0;
}
```

Программа s6-1a.c для иллюстрации работы с очередями сообщений.

```
/* Программа s6-1b.c для иллюстрации работы с очередями сообщений */
/* Эта программа получает доступ к очереди сообщений и читает из
   нее сообщения с любым типом в порядке FIFO до тех пор, пока не
   получит сообщение с типом 255, которое будет служить сигналом
   прекращения работы. */
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <string.h>
#include <stdio.h>
#define LAST_MESSAGE 255 /* Тип сообщения для прекращения работы */
int main()
{
    int msqid; /* IPC дескриптор для очереди сообщений */
    char pathname[] = "s6-1a.c"; /* Имя файла,
        использующееся для генерации ключа. Файл с таким
        именем должен существовать в текущей директории */
    key_t key; /* IPC ключ */
    int len, maxlen; /* Реальная длина и максимальная
        длина информативной части сообщения */
    /* Пользовательская структура для сообщения */
    struct mymsgbuf
    {
        long mtype;
        char mtext[81];
    } mybuf;
    /* Генерируем IPC ключ из имени файла s6-1a.c в текущей
       директории и номера экземпляра очереди сообщений 0 */
```

```
    if((key = ftok(pathname,0)) < 0){
        printf("Can\'t generate key\n");
        exit(-1);
    }
    /* Пытаемся получить доступ по ключу к очереди сообщений,
       если она существует, или создать ее, с правами доступа
       read & write для всех пользователей */
    if((msqid = msgget(key, 0666 | IPC_CREAT)) < 0){
        printf("Can\'t get msqid\n");
        exit(-1);
    }
    while(1){
        /* В бесконечном цикле принимаем сообщения любого типа в порядке
           FIFO с максимальной длиной информативной части 81 символ
           до тех пор, пока не поступит сообщение с типом LAST_MESSAGE */
        maxlen = 81;
        if((len = msgrcv(msqid, (struct msgbuf *) &mybuf, maxlen, 0, 0) < 0){
            printf("Can\'t receive message from queue\n");
            exit(-1);
        }
        /* Если принятое сообщение имеет тип LAST_MESSAGE, прекращаем
           работу и удаляем очередь сообщений из системы.
           В противном случае печатаем текст принятого сообщения. */
        if (mybuf.mtype == LAST_MESSAGE){
            msgctl(msqid, IPC_RMID, (struct msqid_ds *) NULL);
            exit(0);
        }
        printf("message type = %ld, info = %s\n", mybuf.mtype, mybuf.mtext);
    }
    return 0; /* Исключительно для отсутствия warning'ов при компиляции. */
}
```

Программа s6-1b.c для иллюстрации работы с очередями сообщений.

Первая из этих программ посылает пять текстовых сообщений с типом `1` и одно сообщение нулевой длины с типом `255` второй программе. Вторая программа в цикле принимает сообщения любого типа в порядке FIFO и печатает их содержимое до тех пор, пока не получит сообщение с типом `255`. Сообщение с типом `255` служит для нее сигналом к завершению работы и ликвидации очереди сообщений. Если перед запуском любой из программ очередь сообщений еще отсутствовала в системе, то программа создаст ее.

Обратите внимание на использование сообщения с типом `255` в качестве сигнала прекращения работы второго процесса. Это сообщение имеет нулевую длину, так как его информативность исчерпывается самим фактом наличия сообщения.

Откомпилируйте программы `s6-1a.c` и `s6-1b.c` и проверьте правильность их работы.

3-2. Модификация предыдущего примера для передачи числовой информации

В описании системных вызовов `msgsnd()` и `msgrcv()` говорится о том, что передаваемая информации не обязательно должна представлять собой текст.

Мы можем воспользоваться очередями сообщений для передачи данных любого вида. При передаче разнородной информации целесообразно информативную часть объединять внутри сообщения в отдельную структуру:

```
struct mymsgbuf {
    long mtype;
    struct {
        short sinfo;
        float finfo;
    } info;
} mybuf;
```

для правильного вычисления длины информативной части. В некоторых вычислительных системах числовые данные размещаются в памяти с выравниванием на определенные адреса (например, на адреса, кратные 4). Поэтому реальный размер памяти, необходимой для размещения нескольких числовых данных, может оказаться больше суммы длин этих данных, т. е. в нашем случае

```
sizeof(info)>=sizeof(short)+sizeof(float)
```

Для полной передачи информативной части сообщения в качестве длины нужно указывать не сумму длин полей, а полную длину структуры. Модифицируйте программы `s6-1a.c` и `s6-1b.c` для передачи нетекстовых сообщений.

3-3. Написание, компиляция и прогон программ для осуществления двусторонней связи через одну очередь сообщений

Наличие у сообщений типов позволяет организовать двустороннюю связь между процессами через одну и ту же очередь сообщений. Процесс 1 может посылать процессу 2 сообщения с типом 1, а получать от него сообщения с типом 2. При этом для выборки сообщений в обоих процессах следует пользоваться вторым способом выбора. Напишите, откомпилируйте и прогоните программы, осуществляющие двустороннюю связь через одну очередь сообщений.

6. Понятие мультиплексирования. Мультиплексирование сообщений. Модель взаимодействия процессов клиент-сервер. Неравноправность клиента и сервера

Используя технику из предыдущего примера, мы можем организовать получение сообщений одним процессом от множества других процессов через одну очередь сообщений и отправку им ответов через ту же очередь сообщений, т.е. осуществить мультиплексирование сообщений. Вообще под мультиплексированием информации понимают возможность

одновременного обмена информацией с несколькими партнерами. Метод мультиплексирования широко применяется в модели взаимодействия процессов клиент-сервер. В этой модели один из процессов является сервером. Сервер получает запросы от других процессов – клиентов – на выполнение некоторых действий и отправляет им результаты обработки запросов. Чаще всего модель клиент-сервер используется при разработке сетевых приложений. Она изначально предполагает, что взаимодействующие процессы неравноправны:

- Сервер, как правило, работает постоянно, на всем протяжении жизни приложения, а клиенты могут работать эпизодически.
- Сервер ждет запроса от клиентов, инициатором же взаимодействия является клиент.
- Как правило, клиент обращается к одному серверу за раз, в то время как к серверу могут одновременно поступать запросы от нескольких клиентов.
- Клиент должен знать, как обратиться к серверу (например, какого типа сообщения он воспринимает) перед началом организации запроса к серверу, в то время как сервер может получить недостающую информацию о клиенте из пришедшего запроса.

Рассмотрим следующую схему мультиплексирования сообщений через одну очередь сообщений для модели клиент-сервер. Пусть сервер получает из очереди сообщений только сообщения с типом `1`. В состав сообщений с типом `1`, посылаемых серверу, процессы-клиенты включают значения своих идентификаторов процесса. Приняв сообщение с типом `1`, сервер анализирует его содержание, выявляет идентификатор процесса, пославшего запрос, и отвечает клиенту, посылая сообщение с типом, равным идентификатору запрашивавшего процесса. Процесс-клиент после отправления запроса ожидает ответа в виде сообщения с типом, равным своему идентификатору. Поскольку идентификаторы процессов в системе различны, и ни один пользовательский процесс не может иметь `PID` равный `1`, все сообщения могут быть прочитаны только теми процессами, которым они адресованы. Если обработка запроса занимает продолжительное время, сервер может организовывать параллельную обработку запросов, порождая для каждого запроса новый процесс-ребенок или новую нить исполнения.

3-4. Написание, компиляция и прогон программ клиента и сервера

Напишите, откомпилируйте и прогоните программы сервера и клиентов для предложенной схемы мультиплексирования сообщений.

3-5. Использование очередей сообщений для синхронизации работы процессов

На лекции была показана эквивалентность очередей сообщений и семафоров в системах, где процессы могут использовать разделяемую память. В частности, было показано, как реализовать семафоры с помощью очередей

сообщений. Для этого вводился специальный синхронизирующий процесс-сервер, обслуживающий переменные-счетчики для каждого семафора. Процессы-клиенты для выполнения операции над семафором посылали процессу-серверу запросы на выполнение операции и ожидали ответа для продолжения работы. Теперь мы знаем, как это можно сделать в ОС UNIX и как, следовательно, можно использовать очереди сообщений для организации взаимоисключений и взаимной синхронизации процессов.

Задача: Реализовать семафоры через очереди сообщений.

Семинар #7: Организация файловой системы в UNIX. Работа с файлами и директориями. Понятие о memory mapped файлах

Разделы носителя информации (partitions) в UNIX. Логическая структура файловой системы и типы файлов в UNIX. Организация файла на диске в UNIX на примере файловой системы s5fs. Понятие индексного узла (inode). Организация директорий (каталогов) в UNIX. Понятие суперблока. Операции над файлами и директориями. Системные вызовы и команды для выполнения операций над файлами и директориями. Системный вызов open(). Системный вызов close(). Операция создания файла. Системный вызов creat(). Операция чтения атрибутов файла. Системные вызовы stat(), fstat() и lstat(). Операции изменения атрибутов файла. Операции чтения из файла и записи в файл. Операция изменения указателя текущей позиции. Системный вызов lseek(). Операция добавления информации в файл. Флаг O_APPEND. Операции создания связей. Команда ln, системные вызовы link() и symlink(). Операция удаления связей и файлов. Системный вызов unlink(). Специальные функции для работы с содержимым директорий. Понятие о файлах, отображаемых в память (memory mapped файлах). Системные вызовы mmap(), munmap().

1. Введение

В материалах первого и третьего семинаров уже затрагивались вопросы работы с файлами в UNIX. Теперь, зная понятие файловой системы, мы можем рассмотреть ее в целом для UNIX. Мы рассмотрим общие вопросами, связанными с организацией ФС, и системными вызовами.

2. Разделы носителя информации (partitions) в UNIX

Физические носители информации – магнитные или оптические диски, ленты и т.д., использующиеся как физическая основа для хранения файлов, в ОС принято логически делить на разделы (partitions) или логические диски. Это «деление» условно, в некоторых системах несколько физических дисков могут быть объединены в один раздел.

В ОС UNIX физический носитель информации обычно представляет собой один или несколько разделов. В большинстве случаев разбиение на разделы производится линейно, хотя некоторые варианты UNIX могут допускать некое подобие древовидного разбиения (Solaris). Количество разделов и их размеры определяются при форматировании диска.

Наличие нескольких разделов на диске может определяться требованиями ОС или пожеланиями пользователя. Если пользователь хочет разместить на одном жестком диске несколько ОС с возможностью попеременной работы в них, тогда он размещает каждую ОС в своем разделе. Или же необходимо

работы с несколькими видами ФС, тогда под каждый тип ФС выделяется отдельный логический диск. Третий случай – разбиение диска на разделы для размещения в разных разделах различных категорий файлов (например, в одном разделе помещаются все системные файлы, а в другом – все пользовательские файлы). Примером ОС, внутренние требования которой приводят к появлению нескольких разделов на диске, могут служить ранние версии MS-DOS, для которых максимальный размер логического диска не превышал 32 Мбайт.

Для простоты будем полагать, что у нас имеется только один раздел и, следовательно, одна ФС. Вопросы взаимного сосуществования нескольких ФС в рамках одной ОС мы рассмотрим позднее перед обсуждением реализации подсистемы ввода-вывода.

3. Логическая структура ФС, типы файлов в UNIX

В лекции было введено понятие о файлах, как об именованных абстрактных объектах, обладающих определенными свойствами. При этом в пространстве имен файлов одному файлу могут соответствовать несколько имен.

Как известно, файлы могут объединяться в директории, которые организуют древовидную структуру. В ОС UNIX существуют файлы нескольких типов, а именно:

- обычные или регулярные файлы;
- директории или каталоги;
- файлы типа FIFO или именованные pip'ы;
- специальные файлы устройств;
- сокеты (sockets);
- специальные файлы связи (link).

Файлы всех перечисленных типов логически объединены в ациклический граф с однонаправленными ребрами, получающийся из дерева в результате сращивания нескольких терминальных узлов дерева или нескольких его нетерминальных узлов таким образом, чтобы полученный граф не содержал циклов. В нетерминальных узлах такого ациклического графа (т.е. в узлах, из которых выходят ребра) могут располагаться только файлы типов «директория» и «связь». Причем из узла, в котором располагается файл типа «связь», может выходить только ровно одно ребро. В терминальных узлах (из которых не выходит ребер) этого ациклического графа могут располагаться файлы любых типов (рис. 7.1), хотя присутствие в них файла типа «связь» обычно говорит о некотором нарушении целостности файловой системы.

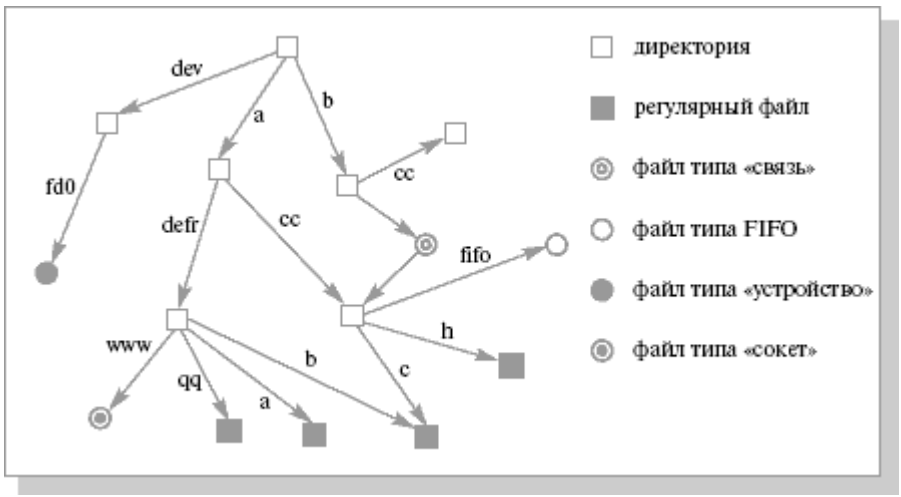

Рис. 7.1. Пример графа файловой системы

В отличие от древовидной структуры набора файлов, где имена файлов связывались с узлами дерева, в таком ациклическом графе имя файла связывается не с узлом, соответствующим файлу, а с входящим в него ребром. Ребра, выходящие из узлов, соответствующих файлам типа «связь», являются неименованными. Заметим, что практически во всех существующих реализациях UNIX-подобных систем в узел графа, соответствующий файлу типа «директория», не может входить более одного именованного ребра, хотя стандарт на ОС UNIX и не запрещает этого. В качестве полного имени файла может использоваться любое имя, получающееся при прохождении по ребрам от корневого узла графа до узла, соответствующего этому файлу, по любому пути с помощью следующего алгоритма:

1. Если интересующему нас файлу соответствует корневой узел, то файл имеет имя «/».
2. Берем первое именованное ребро в пути и записываем его имя, которому предваряем символ «/».
3. Для каждого очередного именованного ребра в пути приписываем к уже получившейся строке справа символ «/» и имя соответствующего ребра.

Полное имя является уникальным для всей файловой системы и однозначно определяет соответствующий ему файл.

4. Организация файла на диске в UNIX на примере файловой системы s5fs. Понятие индексного узла (inode)

Рассмотрим, как организуется на физическом носителе любой файл в UNIX на примере простой файловой системы, впервые появившейся в вариантах ОС SV и носящей поэтому название s5fs (System V file system).

Все дисковое пространство раздела в ФС s5fs логически разделяется на две части: заголовок раздела и **логические блоки данных**. Заголовок раздела

содержит служебную информацию, необходимую для работы ФС, и обычно располагается в самом начале раздела. Логические блоки хранят собственно содержательную информацию файлов и часть информации о размещении файлов на диске (т.е. какие логические блоки и в каком порядке содержат информацию, записанную в файл).

Для размещения любого файла на диске используется метод индексных узлов (inode – index node). Индексный узел содержит атрибуты файла и оставшуюся часть информации о его размещении на диске. Необходимо, однако, отметить, что такие типы файлов, как «связь», «сокет», «устройство», «FIFO» не занимают на диске никакого иного места, кроме индексного узла (им не выделяется логических блоков). Все необходимое для работы с этими типами файлов содержится в их атрибутах.

Перечислим часть атрибутов файлов, хранящихся в индексном узле и свойственных большинству типов файлов. К таким атрибутам относятся:

- Тип файла и права различных категорий пользователей для доступа к нему.
- Идентификаторы владельца-пользователя и владельца-группы.
- Размер файла в байтах (только для регулярных файлов, директорий и файлов типа «связь»).
- Время последнего доступа к файлу.
- Время последней модификации файла.
- Время последней модификации самого индексного узла.

Количество индексных узлов в разделе является постоянной величиной, определяемой на этапе генерации ФС. Все индексные узлы системы организованы в виде массива, хранящегося в заголовке раздела. Каждому файлу соответствует только один элемент этого массива и, наоборот, каждому непустому элементу этого массива соответствует только один файл. Таким образом, каждый файл на диске может быть однозначно идентифицирован номером своего индексного узла (его индексом в массиве).

На языке представления логической организации ФС в виде графа это означает, что каждому узлу графа соответствует только один номер индексного узла, и никакие два узла графа не могут иметь одинаковые номера.

Заметим, что свойством уникальности номеров индексных узлов, идентифицирующих файлы, мы уже неявно пользовались при работе с именованными pip'ами (семинар 3) и средствами SV IPC (семинар 4). Для именованного pip'а именно номер индексного узла, соответствующего файлу с типом FIFO, является той самой точкой привязки, пользуясь которой, неродственные процессы могут получить данные о расположении pip'а в адресном пространстве ядра и его состоянии и связаться друг с другом. Для средств SV IPC при генерации IPC-ключа с помощью функции `ftok()` в действительности используется не имя заданного файла, а номер

соответствующего ему индексного дескриптора, который по определенному алгоритму объединяется с номером экземпляра средства связи.

5. Организация директорий (каталогов) в UNIX

Содержимое регулярных файлов (информация, находящаяся в них, и способ ее организации) всецело определяется программистом, создающим файл. В отличие от регулярных, остальные типы файлов, содержащих данные, т. е. директории и связи, имеют жестко заданную структуру и содержание, определяемые типом используемой файловой системы.

Основным содержимым файлов типа «директория», если говорить на пользовательском языке, являются имена файлов, лежащих непосредственно в этих директориях, и соответствующие им номера индексных узлов. В терминах представления в виде графа содержимое директорий представляет собой имена ребер, выходящих из узлов, соответствующих директориям, вместе с индексными номерами узлов, к которым они ведут.

В ФС s5fs пространство имен файлов (ребер) содержит имена длиной не более 14 символов, а максимальное количество inode в одном разделе ФС не может превышать значения 65535. Эти ограничения не позволяют давать файлам осмысленные имена и приводят к необходимости разбиения больших жестких дисков на несколько разделов. Зато они помогают упростить структуру хранения информации в директории. Все содержимое директории представляет собой таблицу, в которой каждый элемент имеет фиксированный размер в 16 байт. Из них 14 байт отводится под имя соответствующего файла (ребра), а 2 байта – под номер его индексного узла. При этом первый элемент таблицы дополнительно содержит ссылку на саму данную директорию под именем «.», а второй элемент таблицы – ссылку на родительский каталог (если он существует), т.е. на узел графа, из которого выходит единственное именованное ребро, ведущее к текущему узлу, под именем «..».

В современной ФС **FFS** (**F**ast **F**ile **S**ystem) размерность пространства имен файлов (ребер) увеличена до 255 символов. Это позволило использовать практически любые мыслимые имена для файлов, но пришлось изменить структуру каталога (чтобы уменьшить его размеры и не хранить пустые байты). В системе FFS каталог представляет собой таблицу из записей переменной длины. В структуру каждой записи входят: номер индексного узла, длина этой записи, длина имени файла и собственно его имя. Две первых записи в каталоге, как и в s5fs, по-прежнему адресуют саму данную директорию и ее родительский каталог.

6. Понятие суперблока

Мы уже коснулись содержимого заголовка раздела, когда говорили о массиве индексных узлов ФС. Оставшуюся часть заголовка в s5fs принято называть суперблоком. Суперблок хранит информацию, необходимую для правильного функционирования ФС в целом. В нем содержатся, в частности, следующие данные.

- Тип файловой системы.
- Флаги состояния файловой системы.
- Размер логического блока в байтах (обычно кратен 512 байтам).
- Размер ФС в логических блоках (включая сам суперблок и массив inode).
- Размер массива индексных узлов (т.е. сколько файлов может быть размещено в файловой системе).
- Число свободных индексных узлов (сколько файлов еще можно создать).
- Число свободных блоков для размещения данных.
- Часть списка свободных индексных узлов.
- Часть списка свободных блоков для размещения данных.

В некоторых модификациях ФС s5fs последние два списка выносятся за пределы суперблока, но остаются в заголовке раздела. При первом же обращении к ФС суперблок обычно целиком считывается в адресное пространство ядра для ускорения последующих обращений. Поскольку количество логических блоков и индексных узлов в ФС может быть весьма большим, нецелесообразно хранить списки свободных блоков и узлов в суперблоке полностью. При работе с индексными узлами часть списка свободных узлов, находящаяся в суперблоке, постепенно убывает. Когда список почти исчерпан, ОС сканирует массив индексных узлов и заново заполняет список. Часть списка свободных логических блоков, лежащая в суперблоке, содержит ссылку на продолжение списка, расположенное где-либо в блоках данных. Когда эта часть оказывается использованной, ОС загружает на освободившееся место продолжение списка, а блок, применявшийся для его хранения, переводится в разряд свободных.

7. Операции над файлами и директориями

Хотя с точки зрения пользователя рассмотрение операций над файлами и директориями представляется достаточно простым и сводится к перечислению ряда системных вызовов и команд ОС, попытка систематического подхода к набору операций вызывает определенные затруднения. Далее речь пойдет в основном о регулярных файлах и файлах типа «директория».

Как известно, существует два основных вида файлов, различающихся по методу доступа: файлы последовательного доступа и файлы прямого доступа. Если рассматривать файлы прямого и последовательного доступа как абстрактные типы данных, то они представляются как нечто, содержащее информацию, над которой можно совершать следующие операции:

- Для последовательного доступа: чтение очередной порции данных (`read`), запись очередной порции данных (`write`) и позиционирование на начале файла (`rewind`).
- Для прямого доступа: чтение очередной порции данных (`read`), запись очередной порции данных (`write`) и позиционирование на требуемой части данных (`seek`).

Работа с объектами этих абстрактных типов подразумевает наличие еще двух необходимых операций: создание нового объекта (`new`) и уничтожение существующего объекта (`free`).

Расширение математической модели файла за счет добавления к хранимой информации атрибутов, присущих файлу (права доступа, учетные данные), влечет за собой появление еще двух операций: прочитать атрибуты (`get attribute`) и установить их значения (`set attribute`).

Наделение файлов какой-либо внутренней структурой (как у файла типа «директория») или наложение на набор файлов внешней логической структуры (объединение в ациклический направленный граф) приводит к появлению других наборов операций, составляющих интерфейс работы с файлами, которые, тем не менее, будут являться комбинациями перечисленных выше базовых операций.

Для директории, например, такой набор операций, определяемый ее внутренним строением, может выглядеть так: операции `new`, `free`, `set attribute` и `get attribute` остаются без изменений, а операции `read`, `write` и `rewind` (`seek`) заменяются более высокоуровневыми:

- прочитать запись, соответствующую имени файла, – `get record`;
- добавить новую запись – `add record`;
- удалить запись, соответствующую имени файла, – `delete record`.

Неполный набор операций над файлами, связанный с их логическим объединением в структуру директорий, будет выглядеть следующим образом:

- Операции для работы с атрибутами файлов – `get attribute`, `set attribute`.
- Операции для работы с содержимым файлов – `read`, `write`, `rewind(seek)` для регулярных файлов и `get record`, `add record`, `delete record` для директорий.
- Операция создания регулярного файла в некоторой директории (создание нового узла графа и добавление в граф нового именованного ребра, ведущего в этот узел из некоторого узла, соответствующего директории) – `create`. Эту операцию можно рассматривать как суперпозицию двух операций: базовой `new` для регулярного файла и `add record` для соответствующей директории.
- Операция создания поддиректории в некоторой директории – `make directory`. Эта операция отличается от предыдущей операции `create` занесением в файл новой директории информации о файлах с именами «`.`» и «`..`», т.е. по сути дела она есть суперпозиция операции `create` и двух операций `add record`.
- Операция создания файла типа «связь» – `symbolic link`.
- Операция создания файла типа «FIFO» – `make FIFO`.

- Операция добавления к графу нового именованного ребра, ведущего от узла, соответствующего директории, к узлу, соответствующему любому другому типу файла, – `link`. Это просто `add record` с некоторыми ограничениями.
- Операция удаления файла, не являющегося директорией или «связью» (удаление именованного ребра из графа, ведущего к терминальной вершине с одновременным удалением этой вершины, если к ней не ведут другие именованные ребра), – `unlink`.
- Операция удаления файла типа «связь» (удаление именованного ребра, ведущего к узлу, соответствующему файлу типа «связь», с одновременным удалением этого узла и выходящего из него неименованного ребра, если к этому узлу не ведут другие именованные ребра), – `unlink link`.
- Операция рекурсивного удаления директории со всеми входящими в нее файлами и поддиректориями – `remove directory`.
- Операция переименования файла (ребра графа) – `rename`.
- Операция перемещения файла из одной директории в другую (перемещается точка выхода именованного ребра, которое ведет к узлу, соответствующему данному файлу) – `move`.

Возможны и другие подобные операции.

Способ реализации ФС в реальной ОС также может добавлять новые операции. Если часть информации ФС или отдельного файла кэшируется в адресном пространстве ядра, то появляются операции синхронизации данных в кэше и на диске для всей системы в целом (`sync`) и для отдельного файла (`sync file`).

Все перечисленные операции могут быть выполнены процессом только при наличии у него определенных полномочий (прав доступа и т.д.). Для выполнения операций над файлами и директориями операционная система предоставляет процессам интерфейс в виде системных вызовов, библиотечных функций и команд операционной системы. Часть этих системных вызовов, функций и команд мы рассмотрим далее.

8. Системные вызовы и команды для выполнения операций над файлами и директориями

В материалах предыдущих семинаров уже говорилось о некоторых командах и системных вызовах, позволяющих выполнять операции над файлами в ОС UNIX.

На семинаре 1 рассматривался ряд команд, позволяющих изменять атрибуты файла – `chmod`, `chown`, `chgrp`, команду копирования файлов и директорий – `cp`, команду удаления файлов и директорий – `rm`, команду переименования и перемещения файлов и директорий – `mv`, команду просмотра содержимого директорий – `ls`.

На семинаре 3, рассказывалось о хранении информации о файлах внутри адресного пространства процесса с помощью таблицы открытых файлов, о понятии файлового дескриптора, о необходимости введения операций

открытия и закрытия файлов (системные вызовы `open()` и `close()`) и об операциях чтения и записи (системные вызовы `read()` и `write()`). Далее в этом пункте под словом «файл» будет подразумеваться регулярный файл.

Вся информация об атрибутах файла и его расположении на физическом носителе содержится в соответствующем файлу индексном узле и, возможно, в нескольких связанных с ним логических блоках. Для того чтобы при каждой операции над файлом не считывать эту информацию с физического носителя заново считав информацию один раз при первом обращении к файлу, хранить ее в адресном пространстве процесса или в части адресного пространства ядра, характеризующей данный процесс. Именно поэтому в лекции данные о файлах, используемых процессом, были отнесены к составу системного контекста процесса, содержащегося в его PCB.

С точки зрения пользовательского процесса каждый файл представляет собой линейный набор байт, снабженный указателем текущей позиции процесса в этом наборе. Все операции чтения из файла и записи в файл производятся в этом наборе с того места, на которое показывает указатель текущей позиции. По завершении операции чтения или записи указатель текущей позиции помещается после конца прочитанного или записанного участка файла. Значение этого указателя является динамической характеристикой файла для использующего его процесса и также должно храниться в PCB.

На самом деле организация информации, описывающей открытые файлы в адресном пространстве ядра ОС UNIX, является более сложной.

Некоторые файлы могут использоваться одновременно несколькими процессами независимо друг от друга или совместно. Для того чтобы не хранить дублирующуюся информацию об атрибутах файлов и их расположении на внешнем носителе для каждого процесса отдельно, такие данные обычно размещаются в адресном пространстве ядра ОС в единственном экземпляре, а доступ к ним процессы получают только при выполнении соответствующих системных вызовов для операций над файлами.

Независимое использование одного и того же файла несколькими процессами в ОС UNIX предполагает возможность для каждого процесса совершать операции чтения и записи в файл по своему усмотрению. При этом для корректной работы с информацией необходимо организовывать взаимоисключения для операций ввода-вывода. Совместное использование одного и того же файла в ОС UNIX возможно для близко родственных процессов, т.е. процессов, один из которых является потомком другого или которые имеют общего родителя. При совместном использовании файла процессы разделяют некоторые данные, необходимые для работы с файлом, в частности, указатель текущей позиции. Операции чтения или записи,

выполненные в одном процессе, изменяют значение указателя текущей позиции во всех близко родственных процессах, одновременно использующих этот файл.

Вся информация о файле, необходимая процессу для работы с ним, может быть разбита на три части:

- данные, специфичные для этого процесса;
- данные, общие для близко родственных процессов, совместно использующих файл, например, указатель текущей позиции;
- данные, являющиеся общими для всех процессов, использующих файл, – атрибуты и расположение файла.

Для хранения этой информации применяются три различные связанные структуры данных, лежащие, как правило, в адресном пространстве ядра ОС, – таблица открытых файлов процесса, системная таблица открытых файлов и таблица индексных узлов открытых файлов. Для доступа к этой информации в управляющем блоке процесса заводится таблица открытых файлов, каждый непустой элемент которой содержит ссылку на соответствующий элемент системной таблицы открытых файлов, содержащей данные, необходимые для совместного использования файла близко родственными процессами. Из системной таблицы открытых файлов мы, в свою очередь, можем по ссылке добраться до общих данных о файле, содержащихся в таблице индексных узлов открытых файлов (рис. 7.2). Только таблица открытых файлов процесса входит в состав его PCB и, соответственно, наследуется при рождении нового процесса. Индекс элемента в этой таблице (небольшое целое неотрицательное число) или файловый дескриптор является той величиной, характеризующей файл, которой может оперировать процесс при работе на уровне пользователя. В эту же таблицу открытых файлов помещаются и ссылки на данные, описывающие другие потоки ввода-вывода, такие как pipe и FIFO. Эта же таблица будет использоваться и для размещения ссылок на структуры данных, необходимых для передачи информации от процесса к процессу по сети.

Рис. 7.2. Взаимосвязи между таблицами, содержащими данные об открытых файлах в системе

1) Системный вызов `open()`. Для выполнения большинства операций над файлами через системные вызовы пользовательский процесс обычно должен указать в качестве одного из параметров системного вызова дескриптор файла, над которым нужно совершить операцию. Поэтому, прежде чем совершать операции, мы должны поместить информацию о файле в наши таблицы файлов и определить соответствующий файловый дескриптор. Для этого применяется процедура открытия файла, осуществляемая системным вызовом `open()`. При открытии файла ОС проверяет, соответствуют ли права, которые запросил процесс для операций над файлом, правам доступа, установленным для этого файла. В случае соответствия она помещает необходимую информацию в системную таблицу файлов и, если этот файл не был ранее открыт другим процессом, в таблицу индексных дескрипторов открытых файлов. Далее ОС находит пустой элемент в таблице открытых файлов процесса, устанавливает необходимую связь между всеми тремя таблицами и возвращает на пользовательский уровень дескриптор этого файла.

По сути дела, с помощью операции открытия файла ОС осуществляет отображение из пространства имен файлов в дисковое пространство ФС, подготавливая почву для выполнения других операций.

2) Системный вызов `close()`. Обратным системным вызовом по отношению к системному вызову `open()` является системный вызов `close()`. После завершения работы с файлом процесс освобождает выделенные ресурсы ОС и, возможно, синхронизирует информацию о файле, содержащуюся в таблице индексных узлов открытых файлов, с информацией на диске, используя этот системный вызов. Заметим, что место в таблице индексных узлов открытых файлов не освобождается по системному вызову `close()` до тех пор, пока в системе существует хотя бы один процесс, использующий этот файл. Для обеспечения такого поведения в ней для каждого индексного узла заводится счетчик числа открытий, увеличивающийся на 1 при каждом системном вызове `open()` для данного файла и уменьшающийся на 1 при каждом его закрытии. Очищение элемента таблицы индексных узлов открытых файлов с окончательной синхронизацией данных в памяти и на диске происходит только в том случае, если при очередном закрытии файла этот счетчик становится равным `0`.

3) Операция создания файла. **Системный вызов** `creat()`. При обсуждении системного вызова `open()` подробно рассказывалось о его использовании для создания нового файла. Для этих же целей можно использовать системный вызов `creat()`, являющийся, по существу, урезанным вариантом вызова `open()`.

Прототип и описание системного вызова creat()

```
#include <fcntl.h>
int creat(char *path, int mode);
```

Системный вызов `creat` эквивалентен системному вызову `open()` с параметром `flags`, установленным в значение `O_CREAT | O_WRONLY | O_TRUNC`.

Параметр `path` является указателем на строку, содержащую полное или относительное имя файла.

Если файла с указанным именем не существовало к моменту системного вызова, он будет создан и открыт только для выполнения операций записи. Если файл уже существовал, то он открывается также только для операции записи, при этом его длина уменьшается до 0 с одновременным сохранением всех других атрибутов файла.

Параметр `mode` устанавливает атрибуты прав доступа различных категорий пользователей к новому файлу при его создании. Этот параметр задается как сумма следующих восьмеричных значений:

- `0400` – разрешено чтение для пользователя, создавшего файл
- `0200` – разрешена запись для пользователя, создавшего файл
- `0100` – разрешено исполнение для пользователя, создавшего файл

- `0040` – разрешено чтение для группы пользователя, создавшего файл
- `0020` – разрешена запись для группы пользователя, создавшего файл
- `0010` – разрешено исполнение для группы пользователя, создавшего файл
- `0004` – разрешено чтение для всех остальных пользователей
- `0002` – разрешена запись для всех остальных пользователей
- `0001` – разрешено исполнение для всех остальных пользователей

При создании файла реально устанавливаемые права доступа получаются из стандартной комбинации параметра mode и маски создания файлов текущего процесса `umask`, а именно – они равны `mode & ~umask`.

Возвращаемое значение

Системный вызов возвращает значение файлового дескриптора для открытого файла при нормальном завершении и значение `-1` при возникновении ошибки.

4) Операция чтения атрибутов файла. **Системные вызовы** `stat()`, `fstat()` и `lstat()`. Для чтения всех атрибутов файла в специальную структуру могут применяться системные вызовы `stat()`, `fstat()` и `lstat()`.

Прототипы и описание системных вызовов для чтения атрибутов файла

```
#include <sys/stat.h>
#include <unistd.h>
int stat(char *filename, struct stat *buf);
int fstat(int fd, struct stat *buf);
int lstat(char *filename, struct stat *buf);
```

Системные вызовы `stat`, `fstat` и `lstat` служат для получения информации об атрибутах файла.

Системный вызов `stat` читает информацию об атрибутах файла, на имя которого указывает параметр `filename`, и заполняет ими структуру, расположенную по адресу `buf`. Заметим, что имя файла должно быть полным, либо должно строиться относительно той директории, которая является текущей для процесса, совершившего вызов. Если имя файла относится к файлу типа «связь», то читается информация (рекурсивно!) об атрибутах файла, на который указывает символическая связь.

Системный вызов `lstat` идентичен системному вызову `stat` за одним исключением: если имя файла относится к файлу типа «связь», то читается информация о самом файле типа «связь».

Системный вызов `fstat` идентичен системному вызову `stat`, только файл задается не именем, а своим файловым дескриптором (естественно, файл к

этому моменту должен быть открыт).

Для системных вызовов `stat` и `lstat` процессу не нужны никакие права доступа к указанному файлу, но могут понадобиться права для поиска во всех директориях, входящих в специфицированное имя файла.

Структура `stat` в различных версиях UNIX может быть описана по-разному. В Linux она содержит следующие поля:

```
struct stat {
  dev_t st_dev;       /* устройство, на котором расположен файл */
  ino_t st_ino;       /* номер индексного узла для файла */
  mode_t st_mode;     /* тип файла и права доступа к нему */
  nlink_t st_nlink;   /* счетчик числа жестких связей */
  uid_t st_uid;       /* идентификатор пользователя владельца */
  gid_t st_gid;       /* идентификатор группы владельца */
  dev_t st_rdev;      /*тип устройства для специальных файлов устройств*/
  off_t st_size;      /* размер файла в байтах (если определен
                         для данного типа файлов) */
  unsigned long st_blksize; /* размер блока для файловой системы */
  unsigned long st_blocks;  /* число выделенных блоков */
  time_t st_atime;          /* время последнего доступа к файлу */
  time_t st_mtime;          /* время последней модификации файла */
  time_t st_ctime;          /* время создания файла */
}
```

Для определения типа файла можно использовать следующие логические макросы, применяя их к значению поля `st_mode`:

- `S_ISLNK(m)` – файл типа «связь»?
- `S_ISREG(m)` – регулярный файл?
- `S_ISDIR(m)` – директория?
- `S_ISCHR(m)` – специальный файл символьного устройства?
- `S_ISBLK(m)` – специальный файл блочного устройства?
- `S_ISFIFO(m)` – файл типа FIFO?
- `S_ISSOCK(m)` – файл типа «socket»?

Младшие 9 бит поля `st_mode` определяют права доступа к файлу подобно тому, как это делается в маске создания файлов текущего процесса.

Возвращаемое значение

Системные вызовы возвращают значение `0` при нормальном завершении и значение `-1` при возникновении ошибки.

5) Операции изменения атрибутов файла. Большинство операций изменения атрибутов файла обычно выполняется пользователем в интерактивном режиме с помощью команд ОС. Заметим, что только операцию изменения размеров файла, а точнее операцию его обрезания, без изменения всех других атрибутов, кроме, быть может, времени последнего доступа к файлу и его последней модификации. Для того чтобы уменьшить

размеры существующего файла до 0, не затрагивая остальных его характеристик (прав доступа, даты создания, учетной информации и т.д.), можно при открытии файла использовать в комбинации флагов системного вызова `open()` флаг `O_TRUNC`. Для изменения размеров файла до любой желаемой величины (даже для его увеличения во многих вариантах UNIX, хотя изначально этого не предусматривалось!) может использоваться системный вызов `ftruncate()`. При этом, если размер файла мы уменьшаем, то вся информация в конце файла, не влезающая в новый размер, будет потеряна. Если же размер файла мы увеличиваем, то это будет выглядеть так, как будто мы дополнили его до недостающего размера нулевыми байтами.

Прототип и описание системного вызова ftruncate()

```
#include <sys/types.h>
#include <unistd.h>
int ftruncate(int fd, size_t length);
```

Системный вызов `ftruncate` предназначен для изменения длины открытого регулярного файла.

Параметр `fd` является дескриптором соответствующего файла, т. е. значением, которое вернул системный вызов `open()`.

Параметр `length` – значение новой длины для этого файла. Если параметр `length` меньше, чем текущая длина файла, то вся информация в конце файла, не влезающая в новый размер, будет потеряна. Если же он больше, чем текущая длина, то файл будет выглядеть так, как будто мы дополнили его до недостающего размера нулевыми байтами.

Возвращаемое значение

Системный вызов возвращает значение `0` при нормальном завершении и значение `-1` при возникновении ошибки.

6) Операции чтения из файла и записи в файл. Для операций чтения из файла и записи в файл применяются системные вызовы `read()` и `write()`, которые мы уже обсуждали ранее (семинар 3, п. 3).

Надо отметить, что их поведение при работе с файлами имеет определенные особенности, связанные с понятием указателя текущей позиции **в файле.**

При работе с файлами информация записывается в файл или читается из него, начиная с места, определяемого указателем текущей позиции в файле. Значение указателя увеличивается на количество реально прочитанных или

записанных байт. При чтении информации из файла она не пропадает из него. Если системный вызов `read` возвращает значение `0`, то это означает, что достигнут конец файла.

7) Операция изменения указателя текущей позиции. **Системный вызов** `lseek()`. С точки зрения процесса все регулярные файлы являются файлами прямого доступа. В любой момент процесс может изменить положение указателя текущей позиции в открытом файле с помощью системного вызова `lseek()`.

Особенностью этого системного вызова является возможность помещения указателя текущей позиции **в файле за конец файла (т.е. возможность установления значения указателя большего, чем длина файла).**

При любой последующей операции записи в таком положении указателя файл будет выглядеть так, как будто возникший промежуток от конца файла до текущей позиции, где начинается запись, был заполнен нулевыми байтами. Если операции записи в таком положении указателя не производится, то никакого изменения файла, связанного с необычным значением указателя, не произойдет (например, операция чтения будет возвращать нулевое значение для количества прочитанных байтов).

Прототип и описание системного вызова lseek()

```
#include <sys/types.h>
#include <unistd.h>
off_t lseek(int fd, off_t offset, int whence);
```

Системный вызов `lseek` предназначен для изменения положения указателя текущей позиции в открытом регулярном файле.

Параметр `fd` является дескриптором соответствующего файла, т. е. значением, которое вернул системный вызов `open()`.

Параметр `offset` совместно с параметром `whence` определяют новое положение указателя текущей позиции следующим образом:

- Если значение параметра `whence` равно `SEEK_SET`, то новое значение указателя будет составлять `offset` байт от начала файла. Естественно, что значение `offset` в этом случае должно быть не отрицательным.
- Значение параметра `whence` равно `SEEK_CUR`, то новое значение указателя будет составлять старое значение указателя `+ offset` байт. При этом новое значение указателя не должно стать отрицательным.
- Если значение параметра `whence` равно `SEEK_END`, то новое значение указателя будет составлять длина файла `+ offset` байт. При этом новое значение указателя не должно стать отрицательным.

Системный вызов `lseek` позволяет выставить текущее значение указателя за конец файла (т.е. сделать его превышающим размер файла). При любой последующей операции записи в этом положении указателя файл будет выглядеть так, как будто возникший промежуток был заполнен нулевыми битами.

Тип данных `off_t` обычно является синонимом типа `long`.

Возвращаемое значение

Системный вызов возвращает новое положение указателя текущей позиции в байтах от начала файла при нормальном завершении и значение `-1` при возникновении ошибки.

8) Операция добавления информации в файл. Флаг `O_APPEND`. Хотя эта операция по сути дела является комбинацией двух уже рассмотренных операций, мы считаем нужным упомянуть ее особо. Если открытие файла системным вызовом `open()` производилось с установленным флагом `O_APPEND`, то любая операция записи в файл **будет всегда добавлять новые данные в конец файла**, независимо от предыдущего положения указателя текущей позиции (как если бы непосредственно перед записью был выполнен вызов `lseek()` для установки указателя на конец файла).

9) Операции создания связей. **Команда** `ln`**, системные вызовы** `link()` **и** `symlink()`. С операциями, позволяющими изменять логическую структуру ФС, такими как создание файла, мы уже сталкивались в этом разделе. Однако операции создания связи служат для проведения новых именованных ребер в уже существующей структуре без добавления новых узлов или для опосредованного проведения именованного ребра к уже существующему узлу через файл типа «связь» и неименованное ребро. Такие операции мы до сих пор не рассматривали, поэтому давайте остановимся на них подробнее.

Допустим, что несколько программистов совместно ведут работу над одним и тем же проектом. Файлы, относящиеся к этому проекту, вполне естественно могут быть выделены в отдельную директорию так, чтобы не смешиваться с файлами других пользователей и другими файлами программистов, участвующих в проекте. Для удобства каждый из разработчиков, конечно, хотел бы, чтобы эти файлы находились в его собственной директории. Этого можно было бы добиться, копируя по мере изменения новые версии соответствующих файлов из директории одного исполнителя в директорию другого исполнителя. Однако тогда, во-первых, возникнет ненужное дублирование информации на диске. Во-вторых, появится необходимость решения тяжелой задачи: синхронизации обновления замены всех копий этих файлов новыми версиями.

Существует другое решение проблемы. Достаточно разрешить файлам иметь несколько имен. Тогда одному физическому экземпляру данных на диске могут соответствовать различные имена файла, находящиеся в одной или в разных директориях. Подобная операция присвоения нового имени файлу (без уничтожения ранее существовавшего имени) получила название операции создания связи.

В ОС UNIX связь может быть создана двумя различными способами.

Первый способ, наиболее точно следующий описанной выше процедуре, получил название способа создания жесткой связи (hard link). С точки зрения логической структуры ФС этому способу соответствует проведение нового именованного ребра из узла, соответствующего некоторой директории, к узлу, соответствующему файлу любого типа, получающему дополнительное имя. С точки зрения структур данных, описывающих строение ФС, в эту директорию добавляется запись, содержащая дополнительное имя файла и номер его индексного узла (уже существующий!). При таком подходе и новое имя файла, и его старое имя или имена абсолютно равноправны для ОС и могут взаимозаменяемо использоваться для осуществления всех операций.

Использование жестких связей приводит к возникновению двух проблем.

Первая проблема связана с операцией удаления файла. Если мы хотим удалить файл из некоторой директории, то после удаления из ее содержимого записи, соответствующей этому файлу, мы не можем освободить логические блоки, занимаемые файлом, и его индексный узел, не убедившись, что у файла нет дополнительных имен (к его индексному узлу не ведут ссылки из других директорий), иначе мы нарушим целостность ФС. Для решения этой проблемы файлы получают дополнительный атрибут – счетчик жестких связей (или именованных ребер), ведущих к ним, который, как и другие атрибуты, располагается в их индексных узлах. При создании файла этот счетчик получает значение `1`. При создании каждой новой жесткой связи, ведущей к файлу, он увеличивается на `1`. Когда мы удаляем файл из некоторой директории, то из ее содержимого удаляется запись об этом файле, и счетчик жестких связей уменьшается на `1`. Если его значение становится равным `0`, происходит освобождение логических блоков и индексного узла, выделенных этому файлу.

Вторая проблема связана с опасностью превращения логической структуры ФС из ациклического графа в циклический и с возможной неопределенностью толкования записи с именем «..» в содержимом директорий. Для их предотвращения во всех существующих вариантах ОС UNIX запрещено создание жестких связей, ведущих к уже существующим директориям (несмотря на то, что POSIX-стандарт для ОС UNIX разрешает

подобную операцию для пользователя root). Поэтому мы и говорили о том, что в узел, соответствующий файлу типа «директория», не может вести более одного именованного ребра.

Синтаксис и описание команды ln

```
ln [options] source [dest]
ln [options] source ... directory
```

Команда `ln` предназначена для реализации операции создания связи в ФС. Мы будем использовать две формы этой команды.

Первая форма команды, когда в качестве параметра `source` задается имя только одного файла, а параметр `dest` отсутствует, или когда в качестве параметра `dest` задается имя файла, не существующего в ФС, создает связь к файлу, указанному в качестве параметра `source`, в текущей директории с его именем (если параметр dest отсутствует) или с именем `dest` (полным или относительным) в случае наличия параметра `dest`.

Вторая форма команды, когда в качестве параметра `source` задаются имена одного или нескольких файлов, разделенные между собой пробелами, а в качестве параметра `directory` задается имя уже существующей в ФС директории, создает связи к каждому из файлов, перечисленных в параметре `source`, в директории `directory` с именами, совпадающими с именами перечисленных файлов.

Команда `ln` без опций служит для создания жестких связей (hard link), а команда `ln` с опцией `-s` – для создания мягких (soft link) или символических (symbolic) связей.

Примечание: во всех существующих версиях UNIX (несмотря на стандарт POSIX) запрещено создание жестких связей к директориям. Операционная система Linux запрещает также создание жестких связей к специальным файлам устройств.

Для создания жестких связей применяются команда операционной системы `ln` без опций и системный вызов `link()`.

Надо отметить, что системный вызов `link()` является одним из немногих системных вызовов, совершающих операции над файлами, которые не требуют предварительного открытия файла, поскольку он подразумевает выполнение единичного действия только над содержимым индексного узла, **выделенного связываемому файлу**.

Прототип и описание системного вызова link()

```
#include <unistd.h>
int link(char *pathname, char *linkpathname);
```

Системный вызов `link` служит для создания жесткой связи к файлу с именем, на которое указывает параметр `pathname`. Указатель на имя создаваемой связи задается параметром `linkpathname` (полное или относительное имя связи).

Возвращаемое значение

Системный вызов возвращает значение `0` при нормальном завершении и значение `-1` при возникновении ошибки.

Второй способ создания связи получил название способа создания мягкой (soft) или символической (symbolic) связи (link). В то время как жесткая связь файлов является аналогом использования прямых ссылок (указателей) в современных языках программирования, символическая связь, до некоторой степени, напоминает косвенные ссылки (указатель на указатель). При создании мягкой связи с именем `symlink` из некоторой директории к файлу, заданному полным или относительным именем `linkpath`, в этой директории действительно **создается новый** файл типа «связь» с именем `symlink` со своими собственными индексным узлом и логическими блоками. При тщательном рассмотрении можно обнаружить, что все его содержимое составляет только символьная запись имени `linkpath`. Операция открытия файла типа «связь» устроена таким образом, что в действительности открывается не сам этот файл, а тот файл, чье имя содержится в нем (при необходимости рекурсивно!). Поэтому операции над файлами, требующие предварительного открытия файла (как, впрочем, и большинство команд ОС, совершающих действия над файлами, где операция открытия файла присутствует, но скрыта от пользователя), в реальности будут совершаться не над файлом типа «связь», а над тем файлом, имя которого содержится в нем (или над тем файлом, который, в конце концов, откроется при рекурсивных ссылках). Отсюда, в частности, следует, что попытки прочитать **реальное** содержимое файлов типа «связь» с помощью системного вызова `read()` обречены на неудачу. Как видно, создание мягкой связи, с точки зрения изменения логической структуры ФС, эквивалентно опосредованному проведению именованного ребра к уже существующему узлу через файл типа «связь» и неименованное ребро.

Создание символической связи не приводит к проблеме, связанной с удалением файлов. Если файл, на который ссылается мягкая связь, удаляется с физического носителя, то попытка открытия файла мягкой связи (а, следовательно, и удаленного файла) приведет к ошибке «Файла с таким именем не существует», которая может быть аккуратно обработана приложением. Таким образом, удаление связанного объекта, как упоминалось ранее, лишь отчасти и не фатально нарушит целостность ФС.

Неаккуратное применение символических связей пользователями ОС может привести к превращению логической структуры ФС из ациклического графа в циклический граф. Это, конечно, нежелательно, но не носит столь разрушительного характера, как циклы, которые могли бы быть созданы жесткой связью, если бы не был введен запрет на организацию жестких связей к директориям. Поскольку мягкие связи принципиально отличается от жестких связей и связей, возникающих между директорией и файлом при его создании, мягкая связь легко может быть идентифицирована ОС или программой пользователя. Для предотвращения зацикливания программ, выполняющих операции над файлами, обычно ограничивается глубина рекурсии по прохождению мягких связей. Превышение этой глубины приводит к возникновению ошибки «Слишком много мягких связей», которая может быть легко обработана приложением. Поэтому ограничения на тип файлов, к которым может вести мягкая связь, в ОС UNIX не вводятся.

Для создания мягких связей **применяются уже знакомая нам команда ОС** `ln` **с опцией** `-s` **и системный вызов** `symlink()`. **Надо отметить, что системный вызов** `symlink()` **также не требует предварительного открытия связываемого файла, поскольку он вообще не рассматривает его содержимое.**

Прототип и описание системного вызова symlink()

```
#include <unistd.h>
int symlink(char *pathname, char *linkpathname);
```

Системный вызов `symlink` служит для создания символической (мягкой) связи к файлу с именем, на которое указывает параметр `pathname`. Указатель на имя создаваемой связи задается параметром `linkpathname` (полное или относительное имя связи).

Никакой проверки реального существования файла с именем `pathname` системный вызов не производит.

Возвращаемое значение

Системный вызов возвращает значение `0` при нормальном завершении и значение `-1` при возникновении ошибки.

10) Операция удаления связей и файлов. **Системный вызов** `unlink()`. При рассмотрении операции связывания файлов мы уже почти полностью рассмотрели, как производится операция удаления жестких связей и файлов. При удалении мягкой связи, т.е. фактически файла типа «связь», все происходит, как и для обычных файлов. Единственным изменением, с точки зрения логической структуры ФС, является то, что при действительном удалении узла, соответствующего файлу типа «связь», вместе с ним удаляется и выходящее из него неименованное ребро.

Дополнительно необходимо отметить, что условием реального удаления регулярного файла **с диска является не только равенство 0 значения его счетчика** жестких связей, **но и отсутствие процессов, которые держат этот файл открытым**. Если такие

процессы есть, то удаление регулярного файла будет выполнено при его полном закрытии последним использующим файл процессом.

Для осуществления операции удаления жестких связей и/или файлов можно задействовать уже известную из семинара 1 команду ОС `rm` или системный вызов `unlink()`.

Заметим, что системный вызов `unlink()` также не требует предварительного открытия удаляемого файла, поскольку после его удаления совершать над ним операции бессмысленно.

Прототип и описание системного вызова unlink()

```
#include <unistd.h>
int unlink(char *pathname);
```

Системный вызов `unlink` служит для удаления имени, на которое указывает параметр pathname, из файловой системы.

Если после удаления имени счетчик числа жестких связей у данного файла стал равным `0`, то возможны следующие ситуации.

- Если в ОС нет процессов, которые держат данный файл открытым, то файл полностью удаляется с физического носителя.
- Если удаляемое имя было последней жесткой связью для регулярного файла, но какой-либо процесс держит его открытым, то файл продолжает существовать до тех пор, пока не будет закрыт последний файловый дескриптор, ссылающийся на данный файл.
- Если имя относится к файлу типа socket, FIFO или к специальному файлу устройства, то файл удаляется независимо от наличия процессов, держащих его открытым, но процессы, открывшие данный объект, могут продолжать пользоваться им.
- Если имя относится к файлу типа «связь», то он удаляется, и мягкая связь оказывается разорванной.

Возвращаемое значение

Системный вызов возвращает значение `0` при нормальном завершении и значение `-1` при возникновении ошибки.

3-1. Практическое применение команд и системных вызовов для операций над файлами

Создайте жесткие и символические связи из вашей директории к другим файлам. Просмотрите содержимое директорий со связями с помощью команды `ls -al`. Обратите внимание на отличие мягких и жестких связей в листинге этой команды. Определите допустимую глубину рекурсии символических связей для вашей ОС.

9. Специальные функции для работы с содержимым директорий

Стандартные системные вызовы `open()`, `read()` и `close()` не могут помочь программисту изучить содержимое файла типа «директория». Для анализа содержимого директорий используется набор функций из стандартной библиотеки языка С.

Прототип и описание функции opendir()

```
#include <sys/types.h>
#include <dirent.h>
DIR *opendir(char *name);
```

Функция `opendir` служит для открытия потока информации для директории, имя которой расположено по указателю `name`. Тип данных `DIR` представляет собой некоторую структуру данных, описывающую такой поток. Функция `opendir` подготавливает почву для функционирования других функций, выполняющих операции над директорией, и позиционирует поток на первой записи директории.

Возвращаемое значение

При удачном завершении функция возвращает указатель на открытый поток директории, который будет в дальнейшем передаваться в качестве параметра всем другим функциям, работающим с этой директорией. При неудачном завершении возвращается значение `NULL`.

С точки зрения программиста в этом интерфейсе директория представляется как файл последовательного доступа, над которым можно совершать операции чтения очередной записи и позиционирования на начале файла. Перед выполнением этих операций директорию необходимо открыть, а после окончания – закрыть. Для открытия директории используется функция `opendir()`, которая подготавливает почву для совершения операций и позиционирует нас на начале файла. Чтение очередной записи из директории осуществляет функция `readdir()`, одновременно позиционируя нас на начале следующей записи (если она существует). Для операции нового позиционирования на начале директории (если вдруг понадобится) применяется функция `rewinddir()`. После окончания работы с директорией ее необходимо закрыть с помощью функции `closedir()`.

Прототип и описание функции readdir()

```
#include <sys/types.h>
#include <dirent.h>
struct dirent *readdir(DIR *dir);
```

Функция `readdir` служит для чтения очередной записи из потока информации для директории.

Параметр `dir` представляет собой указатель на структуру, описывающую

поток директории, который вернула функция `opendir()`.

Тип данных `struct dirent` представляет собой некоторую структуру данных, описывающую одну запись в директории. Поля этой записи сильно варьируются от одной ФС к другой, но одно из полей всегда присутствует в ней. Это поле `char d_name[ ]` неопределенной длины, не превышающей значения `NAME_MAX+1`, которое содержит символьное имя файла, завершающееся символом конца строки. Данные, возвращаемые функцией `readdir`, переписываются при очередном вызове этой функции для того же самого потока директории.

Возвращаемое значение

При удачном завершении функция возвращает указатель на структуру, содержащую очередную запись директории. При неудачном завершении или при достижении конца директории возвращается значение `NULL`.

Прототип и описание функции rewinddir()

```
#include <sys/types.h>
#include <dirent.h>
void rewinddir(DIR *dir);
```

Функция `rewinddir` служит для позиционирования потока информации для директории, ассоциированного с указателем `dir` (т.е. с тем, что вернула функция `opendir()`), на первой записи (или на начале) директории.

Прототип и описание функции closedir()

```
#include <sys/types.h>
#include <dirent.h>
int closedir(DIR *dir);
```

Функция `closedir` служит для закрытия потока информации для директории, ассоциированного с указателем `dir` (т.е. с тем, что вернула функция `opendir()`). После закрытия поток директории становится недоступным для дальнейшего использования.

Возвращаемое значение

При успешном завершении функция возвращает значение `0`, при неудачном завершении – значение `-1`.

3-2. Написание, прогон и компиляция программы, анализирующей содержимое директории

Напишите, откомпилируйте и прогоните программу, распечатывающую список файлов, входящих в директорию, с указанием их типов. Имя директории задается как параметр командной строки. Если оно отсутствует, то выбирается текущая директория.

Задача (+10): Напишите программу, распечатывающую содержимое заданной директории в формате, аналогичном формату выдачи команды `ls -al`. Для этого вам дополнительно понадобится функция `ctime(3)` и системные вызовы `time(2)`, `readlink(2)`. Цифры после имен функций и системных вызовов – это номера соответствующих разделов для UNIX Manual.

10. Понятие о файлах, отображаемых в память (memory mapped файлах). Системные вызовы mmap(), munmap()

Как уже говорилось, с помощью системного вызова `open()` ОС отображает файл из пространства имен в дисковое пространство ФС, подготавливая почву для осуществления других операций. С появлением концепции виртуальной памяти, когда физические размеры памяти перестали играть роль сдерживающего фактора в развитии ВС, стало возможным отображать файлы непосредственно в адресное пространство процессов. Иными словами, появилась возможность работать с файлами как с обычной памятью, заменив выполнение базовых операций над ними с помощью системных вызовов на использование операций обычных языков программирования. Файлы, чье содержимое отображается непосредственно в адресное пространство процессов, получили название файлов, отображаемых в память, или, по-английски, memory mapped файлов. Такое отображение может быть осуществлено не только для всего файла в целом, но и для его части.

С точки зрения программиста работа с такими файлами выглядит следующим образом:

- Отображение файла из пространства имен в адресное пространство процесса происходит в два этапа: сначала выполняется отображение в дисковое пространство, а уже затем из дискового пространства в адресное. Поэтому вначале файл необходимо открыть, используя обычный системный вызов `open()`.
- Вторым этапом является отображение файла целиком или частично из дискового пространства в адресное пространство процесса. Для этого используется системный вызов `mmap()`. Файл после этого можно и закрыть, выполнив системный вызов `close()`, так как необходимую информацию о расположении файла на диске мы уже сохранили в других структурах данных при вызове `mmap()`.

Прототип и описание системного вызова mmap()

```
#include <sys/types.h>
#include <unistd.h>
#include <sys/mman.h>
void *mmap (void *start, size_t length,
            int prot, int flags, int fd, off_t offset);
```

Системный вызов `mmap` служит для отображения предварительно открытого файла (например, с помощью системного вызова `open()`) в адресное пространство ВС. После его выполнения файл может быть закрыт (например, системным вызовом `close()`), что никак не повлияет на дальнейшую работу с отображенным файлом.

Параметр `fd` является файловым дескриптором для файла, который мы хотим отобразить в адресное пространство (т.е. значением, которое вернул системный вызов `open()`).

Ненулевое значение параметра `addr` может использоваться только очень квалифицированными системными программистами, поэтому мы всегда будем полагать его равным значению `NULL`, позволяя ОС самой выбрать начало области адресного пространства, в которую будет отображен файл.

В память будет отображаться часть файла, начиная с позиции внутри его, заданной значением параметра `offset` – смещение от начала файла в байтах, и длиной, равной значению параметра `length` (в байтах). Значение параметра `length` может и превышать реальную длину от позиции `offset` до конца существующего файла. На поведении системного вызова это никак не отразится, но в дальнейшем при попытке доступа к ячейкам памяти, лежащим вне границ реального файла, возникнет сигнал `SIGBUS` (реакция на него по умолчанию – прекращение процесса с образованием core файла).

Параметр `flags` определяет способ отображения файла в адресное пространство. Мы будем использовать только два его возможных значения: `MAP_SHARED` и `MAP_PRIVATE`. Если в качестве его значения выбрано `MAP_SHARED`, то полученное отображение файла впоследствии будет использоваться и другими процессами, вызвавшими mmap для этого файла с аналогичными значениями параметров, а все изменения, сделанные в отображенном файле, будут сохранены во вторичной памяти. Если в качестве значения параметра `flags` указано `MAP_PRIVATE`, то процесс получает отображение файла в свое монопольное распоряжение, но все изменения в нем не могут быть занесены во вторичную память (т.е., проще говоря, не сохранятся).

Параметр `prot` определяет разрешенные операции над областью памяти, в которую будет отображен файл. В качестве его значения мы будем использовать значения `PROT_READ` (разрешено чтение), `PROT_WRITE` (разрешена запись) или их комбинацию через операцию «побитовое или» – «`|`». Необходимо отметить две существенные особенности системного вызова, связанные с этим параметром:

1. Значение параметра `prot` не может быть шире, чем операции над файлом, заявленные при его открытии в параметре `flags` системного вызова `open()`. Например, нельзя открыть файл только для чтения, а при его отображении в память использовать значение `prot = PROT_READ | PROT_WRITE`.
2. В результате ошибки в ОС Linux при работе на 486-х и 586-х процессорах попытка записать в отображение файла, открытое только для записи, более 32-х байт одновременно приводит к ошибке (возникает сигнал о нарушении защиты памяти).

Возвращаемое значение

При нормальном завершении системный вызов возвращает начальный адрес области памяти, в которую отображен файл (или его часть), при возникновении ошибки – специальное значение `MAP_FAILED`.

- После этого с содержимым файла можно работать, как с содержимым обычной области памяти.
- По окончании работы с содержимым файла, необходимо освободить дополнительно выделенную процессу область памяти, предварительно синхронизировав содержимое файла на диске с содержимым этой области (если, конечно, необходимо). Эти действия выполняет системный вызов `munmap()`.

Прототип и описание системного вызова munmap

```
#include <sys/types.h>
#include <unistd.h>
#include <sys/mman.h>
int munmap (void *start, size_t length);
```

Системный вызов `munmap` служит для прекращения отображения memory mapped файла в адресное пространство ВС. Если при системном вызове `mmap()` было задано значение параметра `flags`, равное `MAP_SHARED`, и в отображении файла была разрешена операция записи (в параметре `prot` использовалось значение `PROT_WRITE`), то `munmap` синхронизирует содержимое отображения с содержимым файла во вторичной памяти. После его выполнения области памяти, использовавшиеся для отображения файла, становятся недоступны текущему процессу.

Параметр `addr` является адресом начала области памяти, выделенной для отображения файла, т.е. значением, которое вернул системный вызов `mmap()`.

Параметр `length` определяет ее длину, и его значение должно совпадать со значением соответствующего параметра в системном вызове `mmap()`.

Возвращаемое значение

При нормальном завершении системный вызов возвращает значение `0`, при возникновении ошибки – значение `-1`.

3-3. Анализ, компиляция и прогон программы для создания memory mapped файла и записи его содержимого

Для закрепления материала, изложенного в предыдущем разделе, рассмотрим пример программы.

```
/* Программа s7-2.c для иллюстрации работы с memory mapped файлом */
```

```
int main(void)
{
    int fd; /* Файловый дескриптор для файла, в котором
               будет храниться наша информация */
    size_t length; /* Длина отображаемой части файла */
    int i;
    /* Ниже следует описание типа структуры, которым мы забьем
       файл, и двух указателей на подобный тип. Указатель ptr
       будет использоваться в качестве начального адреса
       выделенной области памяти, а указатель tmpptr – для
       перемещения внутри этой области. */
    struct A {
        double f;
        double f2;
    } *ptr, tmpptr;
    /* Открываем файл или сначала создаем его (если такого файла не было).
       Права доступа к файлу при создании определяем как read-write для всех
       категорий пользователей (0666). Из-за ошибки в Linux мы будем
       вынуждены ниже в системном вызове mmap() разрешить в отображении
       файла и  чтение, и запись, хотя реально нам нужна только запись.
       Поэтому и при открытии файла мы вынуждены задавать O_RDWR. */
    fd = open("mapped.dat", O_RDWR | O_CREAT, 0666);
    if( fd == -1){
        /* Если файл открыть не удалось, выдаем
           сообщение об ошибке и завершаем работу */
        printf("File open failed!\n");
        exit(1);
    }
    /* Вычисляем будущую длину файла (мы собираемся записать
       в него 100000 структур) */
    length = 100000*sizeof(struct A);
    /* Вновь созданный файл имеет длину 0. Если мы его
       отобразим в память с такой длиной, то любая попытка
       записи в выделенную память приведет к ошибке. Увеличиваем
       длину файла с помощью вызова ftruncate(). */
    ftruncate(fd, length);
    /* Отображаем файл в память. Разрешенные операции над отображением
       указываем как PROT_WRITE | PROT_READ по уже названным причинам.
       Значение флагов ставим в MAP_SHARED, так как мы хотим сохранить
       информацию, которую занесем в отображение, на диске.
       Файл отображаем с его начала (offset = 0) и до конца
       (length = длине файла). */
    ptr = (struct A )mmap(NULL, length, PROT_WRITE |
             PROT_READ, MAP_SHARED, fd, 0);
    /* Файловый дескриптор нам более не нужен, и мы его закрываем */
    close(fd);
    if( ptr == MAP_FAILED ){
        /* Если отобразить файл не удалось, сообщаем об
           ошибке и завершаем работу */
        printf("Mapping failed!\n");
        exit(2);
    }
    /* В цикле заполняем образ файла числами от 1 до 100000
       и их квадратами. Для перемещения по области памяти
       используем указатель tmpptr, так как указатель ptr на
       начало образа файла нам понадобится для прекращения
       и отображения вызовом munmap(). */
    tmpptr = ptr;
    for(i = 1; i <=100000; i++){
        tmpptr->f = i;
        tmpptr->f2 = tmpptr->f*tmpptr->f;
        tmpptr++;
    }
    /* Прекращаем отображать файл в память, записываем
```

```
       содержимое отображения на диск и освобождаем память. */
    munmap((void *)ptr, length);
    return 0;
}
```

Программа s7-2.c для иллюстрации работы с memory mapped файлом.

Эта программа создает файл, отображает его в адресное пространство процесса и заносит в него информацию с помощью обычных операций языка С.

Обратите внимание на необходимость увеличения размера файла перед его отображением. Созданный файл имеет нулевой размер, и если его с этим размером отобразить в память, то мы сможем записать в него или прочитать из него не более 0 байт, т.е. ничего. Для увеличения размера файла использован системный вызов `ftruncate()`, **хотя это можно было бы сделать и любым другим способом.**

При отображении файла мы вынуждены разрешить в нем и запись, и чтение, хотя реально совершаем только запись. Это сделано для того, чтобы избежать ошибки в ОС Linux, связанной с использованием 486-х и 586-х процессоров. Такой список разрешенных операций однозначно требует, чтобы при открытии файла системным вызовом `open()` файл открывался и на запись, и на чтение. Поскольку информацию мы желаем сохранить на диске, при отображении использовано значение флагов `MAP_SHARED`. Откомпилируйте эту программу и запустите ее.

3-4. Изменение предыдущей программы для чтения из файла, используя его отображение в память

Модифицируйте программу из предыдущего раздела так, чтобы она отображала файл, записанный программой s7-2.c, в память и считала сумму квадратов чисел от 1 до 100000, которые уже находятся в этом файле.

Задача (+10): Напишите две программы, использующие memory mapped файл для обмена информацией при одновременной работе, подобно тому, как они могли бы использовать разделяемую память.

VII. ПЕРЕЧЕНЬ ПРОГРАММНЫХ ПРОДУКТОВ, РЕАЛЬНО ИСПОЛЬЗУЕМЫХ В ПРАКТИКЕ ДЕЯТЕЛЬНОСТИ ВЫПУСКНИКОВ И СООТВЕТСТВУЮЩЕЕ УЧЕБНО-МЕТОДИЧЕСКОЕ ПОСОБИЕ, РАСКРЫВАЮЩЕЕ ОСОБЕННОСТИ И ПЕРСПЕКТИВЫ ИСПОЛЬЗОВАНИЯ ДАННЫХ ПРОГРАММНЫХ ПРОДУКТОВ

Студенты могут создавать приложения в области своей профессиональной деятельности на основе операционных систем Windows, Linux, UNIX, OS/2, NETWARE. Преполагается использования языка высокого уровня, например Си.

VIII. МЕТОДИЧЕСКИЕ УКАЗАНИЯ ПО ПРИМЕНЕНИЮ СОВРЕМЕННЫХ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ ДЛЯ ПРЕПОДАВАНИЯ УЧЕБНОЙ ДИСЦИПЛИНЫ (В Т. Ч. РАЗРАБОТАННЫЕ ВЕДУЩИМИ ПРЕПОДАВАТЕЛЯМИ ФИЛИАЛА)

При преподавании данной дисциплины можно использовать электронные тестирующие и учебные материалы. Методические указания прилагаются к этим материалам.

IX. МЕТОДИЧЕСКИЕ УКАЗАНИЯ ПРОФЕССОРСКО-ПРЕПОДАВАТЕЛЬСКОМУ СОСТАВУ ПО ОРГАНИЗАЦИИ МЕЖСЕССИОННОГО И ЭКЗАМЕНАЦИОННОГО КОНТРОЛЯ ЗНАНИЙ СТУДЕНТОВ (МАТЕРИАЛЫ ПО КОНТРОЛЮ КАЧЕСТВА ОБРАЗОВАНИЯ)

Преподаватель готовит контролирующие материалы в виде тестов, задач и в другой форме. Во время проведения контроля знаний студентов преподаватель объясняет студентам правила работы с контролирующими материалами и выдаёт эти материалы студентам. После истечения установленного времени контролирующие материалы собираются и обрабатываются.

X. КОМПЛЕКТЫ ЗАДАНИЙ ДЛЯ СЕМИНАРСКИХ ЗАНЯТИЙ, КОНТРОЛЬНЫХ РАБОТ, ДОМАШНИХ ЗАДАНИЙ

Задания для практических семинаров, контрольных работ и домашних заданий берутся из книг, реквизиты которых приведены в рабочей программе.

XI. ФОНД ТЕСТОВЫХ И КОНТРОЛЬНЫХ ЗАДАНИЙ ДЛЯ ОЦЕНКИ КАЧЕСТВА ЗНАНИЙ ПО ДИСЦИПЛИНЕ

Фонд тестовых и контрольных заданий для оценки качества знаний по дисциплине приведен в приложении А.

XII. КОМПЛЕКТЫ ЭКЗАМЕНАЦИОННЫХ БИЛЕТОВ ПО ДИСЦИПЛИНЕ

Комплекты экзаменационных билетов составляются на основе перечня вопросов, приведенного в рабочей программе, по следующей форме.

ГОУВПО «Амурский государственный университет»

Утверждено на заседании кафедры
«___» _____ 200 г.
Заведующий кафедрой

Утверждаю: _____

Кафедра *математического анализа и моделирования*
Факультет *математики и информатики*
Курс 3
Дисциплина "*Операционные системы и сетевые технологии"*

Экзаменационный билет 1

1. Основные понятия компьютерных систем. Эволюция вычислительных систем. Предпосылки создания компьютерных сетей.
2. Протокол. Стандартные стеки коммуникационных протоколов.
3. Глобальные сети с удаленным доступом.

XIII. КАРТА ОБЕСПЕЧЕННОСТИ ДИСЦИПЛИНЫ КАДРАМИ ПРОФЕССОРСКО-ПРЕПОДАВАТЕЛЬСКОГО СОСТАВА

Дисциплину в полном объёме ведёт:

1. Фамилия, имя, отчество: Рыженко Андрей Викторович
2. Учёное звание: –
3. Учёная степень: канд. техн. наук

ПРИЛОЖЕНИЕ А

1. Какие из перечисленных алгоритмов представляют собой частные случаи планирования с использованием приоритетов?
    1. FCFS
    2. RR
    3. SJF
    4. гарантированное планирование
2. Какой из вариантов адресации может использоваться для организации передачи информации через pipe?
    1. симметричная прямая адресация
    2. асимметричная прямая адресация
    3. непрямая адресация
3. Набор из двух активностей, P и Q является

P:	Q:
y=x+2	z=x-3
f=y-4	f=z+1

    1. детерминированным
    2. недетерминированным
    3. детерминированность зависит от значения x
4. Если для некоторого набора активностей условия Бернстайна не выполняются, то набор активностей является
    1. детерминированным
    2. недетерминированным
    3. может быть как недетерминированным, так и детерминированным
5. Внутренняя фрагментация - это
    1. потеря части памяти, выделенной процессу, но не используемой им
    2. разбиение адресного пространства процесса на фрагменты
    3. потеря части памяти в схеме с фиксированными разделами
6. Таблица страниц процесса - это
    1. структура, используемая для отображения логического адресного пространства в физическое при страничной организации памяти
    2. структура, организованная для учета свободных и занятых страничных блоков
    3. структура, организованная для контроля доступа с страницам процесса
7. В вычислительной системе со страничной организацией памяти и 32 битовым адресом размер страницы составляет 8 Мбайт. Для некоторого процесса таблица страниц в этой системе имеет вид

№ страницы	Адрес начала страницы
1	0x00000000
2	0x02000000
5	0x06000000
6	0x10000000

Какому физическому адресу соответствует виртуальный адрес 0x00827432?

1. 0x27432
2. 0x02027432
3. 0x10027432

8. Инвертированная таблица страниц дает возможность
   1. получить номер страничного кадра по номеру виртуальной страницы
   2. ускорить процесс трансляции адреса
   3. уменьшить объем памяти, расходуемой на отображение виртуального адресного пространства в физическое

9. Для некоторого процесса известна следующая строка запросов страниц памяти: 7, 1, 2, 3, 2, 4, 2, 1, 0, 3, 7, 2, 1, 2, 7, 1, 7, 2, 3. Сколько ситуаций отказа страницы (page fault) возникает для данного процесса при использовании алгоритма замещения страниц FIFO (Fist Input Fist Output)?
   1. 13
   2. 12
   3. 11

10. Схема выделения дискового пространства связным списком блоков не нашла широкого применения, так как
   1. неэффективно использует дисковое пространство
   2. требует большого количества обращений к диску при работе с файлами
   3. страдает от внутренней фрагментации

11. Пусть у нас есть локальная вычислительная сеть, достаточно долгое время работающая с неизменной топологией и без сбоев. Какие алгоритмы маршрутизации гарантируют доставку пакетов данных от отправителя к получателю по кратчайшему пути?
   1. алгоритмы лавинной маршрутизации
   2. алгоритмы состояния связей
   3. маршрутизация от источника данных

12. Для чего может использоваться функция MD4?
   1. для шифрования с симметричным ключом
   2. для шифрования с асимметричным ключом
   3. для шифрования паролей в качестве односторонней функции

13. Из какого системного вызова при нормальной работе пользователь может наблюдать два возвращения
   1. exit()
   2. exec()
   3. fork()

14. Какой тип связи обеспечивает FIFO()?
   1. симплексную связь
   2. полудуплексную связь
   3. дуплексную связь

15. Какая из операций над семафорами SYSTEM V IPC является аналогом операции P(S) над семафорами Дейкстры:
   1. A(S,n)
   2. D(S,n)
   3. Z(S)

4. не имеет аналогов

16. Сколько различных типов файлов существует в операционной системе UNIX?
    1. 4
    2. 6
    3. 8

17. Что такое топология пассивная звезда?
    1. звезда, в центре которой расположен не компьютер, а концентратор
    2. звезда, центральный компьютер которой пассивно ждет обращений к нему
    3. звезда, в которой к центральному компьютеру подключаются только пассивные устройства
    4. звезда, центральный компьютер которой занимается только обслуживанием обмена в сети
    5. звезда, которая нечувствительна к обрывам кабеля сети

18. Каков размер MAC-адреса абонентов в сети Ethernet?
    1. 8 бит
    2. 8 байт
    3. 10 бит
    4. 6 байт
    5. 12 байт

19. На каком уровне модели OSI работает коммутатор?
    1. на физическом уровне
    2. на сеансовом уровне
    3. на канальном уровне
    4. на транспортном уровне
    5. на сетевом уровне

20. Функции каких уровней модели OSI выполняет драйвер сети?
    1. физического и сетевого
    2. сетевого и транспортного
    3. физического и канального
    4. физического, канального и сетевого
    5. канального и сетевого

СОДЕРЖАНИЕ