

Федеральное агентство по образованию

Томский государственный университет  
систем управления и радиоэлектроники

**В.В. Одинок**  
**В.П. Коцубинский**

# **ОПЕРАЦИОННЫЕ СИСТЕМЫ И СЕТИ**

Издание второе, дополненное

Учебное пособие

Рекомендовано учебно-методическим объединением вузов  
Российской Федерации по образованию в области радиотехники,  
электроники, биомедицинской техники и автоматизации  
в качестве учебного пособия для студентов высших  
учебных заведений, обучающихся по специальности 220201  
«Управление и информатика в технических системах»

Томск  
ТУСУР  
2008

УДК 681.3.066.+681.324](075.8)  
ББК 32.973.2-018я73+32.988-5я73  
О-42

Рецензенты:

**В.А. Силич** — д-р техн. наук,  
профессор Томск. политехн. ун-та;  
**Е.А. Вейсов** — канд. техн. наук,  
доцент Краснояр. гос. техн. ун-та

**Одинокое В.В.**

О-42      Операционные системы и сети : учеб. пособие /  
В.В. Одинокое, В.П. Коцубинский. – 2-е изд., доп. – Томск  
: Томск. гос. ун-т систем упр. и радиоэлектроники, 2007. –  
391 с.

ISBN 978-5-86889-374-2

Рассматриваются вопросы организации и использования операционной системы UNIX: интерфейсы пользователя, принципы обеспечения мульти-программирования и многопользовательской работы, организация основных подсистем UNIX (подсистем управления процессами, оперативной памятью и файлами). С целью практического изучения рассматриваемых вопросов приводится курс лабораторных работ.

Предназначено для специалистов по разработке и применению ЭВМ, программистов различной квалификации, а также студентов вузов соответствующих специальностей.

УДК 681.3.066.+681.324](075.8)  
ББК 32.973.2-018я73+32.988-5я73

© Одинокое В. В.,  
Коцубинский В. П., 2006

© Одинокое В. В.,  
Коцубинский В. П., 2008,  
с изменениями

© Том. гос. ун-т систем упр.  
и радиоэлектроники, 2008

ISBN 978-5-86889-374-2

# Оглавление

Введение .....	8
1. ИНТЕРФЕЙСЫ ПОЛЬЗОВАТЕЛЯ СИСТЕМЫ	
1.1. Функции системных программ.....	11
1.2. Файлы.....	16
1.3. Утилиты .....	22
1.4. Трансляторы .....	30
1.5. Язык управления операционной системой.....	37
1.5.1. Типы языков управления.....	37
1.5.2. Простые команды.....	39
1.5.3. Составные команды .....	43
1.5.4. Переменные и выражения .....	46
1.5.5. Управляющие операторы .....	52
1.5.6. Командные файлы.....	60
2. СИСТЕМНАЯ ПОДДЕРЖКА МУЛЬТИПРОГРАММИРОВАНИЯ	
2.1. Мультипрограммирование.....	65
2.2. Процессы .....	66
2.3. Ресурсы .....	70
2.4. Синхронизация параллельных процессов .....	73
2.4.1. Синхронизация с помощью сигналов.....	73
2.4.2. Терминальное управление процессами.....	76
2.4.3. Синхронизация конкурирующих процессов .....	80
2.4.4. Синхронизация кооперирующихся процессов.....	85
2.5. Информационные взаимодействия между процессами.....	86
2.5.1. Понятие информационного канала.....	86
2.5.2. Обыкновенные программные каналы .....	90
2.5.3. Именованные программные каналы.....	92
2.6. Тупики .....	92
2.6.1. Причины появления тупиков .....	92
2.6.2. Методы предотвращения тупиков .....	94
2.6.3. Методы ликвидации тупиков.....	99
3. ПОДДЕРЖКА МНОГОПОЛЬЗОВАТЕЛЬСКОЙ РАБОТЫ И СТРУКТУРА СИСТЕМЫ	
3.1. Управление доступом пользователя в систему.....	103
3.2. Защита файлов .....	108
3.3. Укрупненная структура операционной системы .....	113
3.4. Структура сетевой операционной системы.....	116
4. ПОДСИСТЕМА УПРАВЛЕНИЯ ПРОЦЕССАМИ	
4.1. Состояния процесса.....	124
4.2. Создание процесса .....	128
4.3. Обработка сигналов.....	131
4.4. Диспетчеризация процессов .....	133

4.5. Использование таймера для управления процессами.....	141
4.6. Информационные взаимодействия между процессами.....	143
4.6.1. Разделяемая память.....	143
4.6.2. Очереди сообщений.....	146
4.6.3. Сокеты .....	149
5. УПРАВЛЕНИЕ ОПЕРАТИВНОЙ ПАМЯТЬЮ И ПРОЦЕССОРОМ	
5.1. Задачи управления памятью .....	158
5.2. Сегментная виртуальная память.....	162
5.2.1. Преобразование адресов.....	162
5.2.2. Распределение памяти .....	168
5.2.3. Защита информации в оперативной памяти .....	171
5.3. Линейная виртуальная память .....	176
5.3.1. Преобразование адресов.....	176
5.3.2. Распределение памяти .....	178
5.4. Переключение процессора.....	181
5.4.1. Межсегментные переходы внутри процесса .....	181
5.4.2. Аппаратное переключение процессов.....	185
5.4.3. Обработка прерываний.....	188
6. УПРАВЛЕНИЕ ФАЙЛАМИ	
6.1. Виртуальная файловая система .....	192
6.1.1. Логические файлы.....	192
6.1.2. Открытие файла .....	194
6.1.3. Другие операции с файлами.....	199
6.2. Реальные файловые системы.....	204
6.2.1. Критерии оценки файловых систем .....	204
6.2.2. Физическое размещение информации на носителе .....	207
6.2.3. Каталоги.....	213
6.2.4. Управляющие структуры данных.....	215
6.3. Объединение реальных файловых систем.....	219
6.4. Кэширование блоков данных.....	225
6.4.1. Традиционный подход к реализации дискового КЭШа ..	225
6.4.2. Использование подсистемы управления памятью .....	229
7. ПОДСИСТЕМА ВВОДА-ВЫВОДА	
7.1. Предоставляемый интерфейс.....	232
7.2. Классификация драйверов .....	236
7.3. Аппаратный интерфейс .....	239
7.4. Одноуровневые драйверы.....	240
7.5. Двухуровневые драйверы .....	245
7.5.1. Двухуровневый драйвер с опросом .....	247
7.5.2. Двухуровневый драйвер с прерыванием на байт .....	251
7.5.3. Блочные драйверы с прямым доступом в память .....	255

## 8. ПРИКЛАДНОЙ УРОВЕНЬ СЕТИ

8.1. Общая структура сетевого приложения.....	259
8.2. Выбор транспортного информационного канала.....	261
8.3. Транспортные интерфейсы .....	267
8.3.1. Транспортные порты.....	267
8.3.2. Интерфейс сокетов <i>UDP</i> -канала .....	269
8.3.3. Интерфейс сокетов <i>TCP</i> -канала .....	270
8.4. Трансляция имен хостов .....	271
8.5. Приложения для передачи файлов .....	276
8.5.1. Приложения на основе протокола <i>FTP</i> .....	276
8.5.2. Электронная почта .....	282
8.5.3. <i>WEB</i> -приложения .....	290

## 9. ЛАБОРАТОРНЫЙ КУРС

### 9.1 Лабораторная работа № 1.

Первоначальное знакомство с <i>UNIX</i> .....	300
9.1.1. Подготовка к выполнению работы .....	300
9.1.2. Вход в систему и выход из нее .....	301
9.1.3. Текстовый редактор <i>ed</i> .....	304
9.1.4. Работа в среде <i>Midnight Commander</i> .....	307
9.1.5. Задание .....	312

### 9.2. Лабораторная работа № 2.

Дальнейшее знакомство с командами <i>UNIX</i> .....	313
9.2.1. Подготовка к выполнению работы .....	313
9.2.2. Задание .....	313

### 9.3. Лабораторная работа № 3. Управляющие операторы командного языка.....

9.3.1. Подготовка к выполнению работы .....	314
9.3.2. Задание .....	315

### 9.4. Лабораторная работа № 4. Процессы в *UNIX*.....

9.4.1. Подготовка к выполнению работы .....	315
9.4.2. Сообщения другому пользователю .....	316
9.4.3. Задание .....	317

### 9.5. Лабораторная работа № 5. Операции с файлами

в программе на языке СИ .....	318
9.5.1. Подготовка к выполнению работы .....	318
9.5.2. Характеристика языка СИ .....	318
9.5.3. Открытие существующего файла .....	319
9.5.4. Создание файла .....	322
9.5.5. Указатель файла .....	324
9.5.6. Чтение из файла.....	326
9.5.7. Запись в файл.....	328
9.5.8. Заккрытие и уничтожение файла .....	329

9.5.9. Задание .....	330
9.6. Лабораторная работа № 6. Системные вызовы	
для управления процессами .....	330
9.6.1. Подготовка к выполнению работы .....	331
9.6.2. Создание процесса .....	331
9.6.3. Ожидание завершения потомка .....	332
9.6.4. Загрузка новой программы .....	334
9.6.5. Совместное применение вызовов <i>fork</i> и <i>exec</i> .....	335
9.6.6. Наследование данных при создании процесса и при загрузке программы .....	337
9.6.7. Задание .....	339
9.7. Лабораторная работа № 7. Обработка сигналов .....	339
9.7.1. Подготовка к выполнению работы .....	339
9.7.2. Изменение диспозиции сигналов .....	339
9.7.3. Наборы сигналов .....	342
9.7.4. Сеансы и группы процессов .....	344
9.7.5. Посылка сигналов другим процессам .....	346
9.7.6. Сигнал таймера .....	348
9.7.7. Задание .....	349
9.8. Лабораторная работа № 8. Управление терминалом .....	350
9.8.1. Функции терминальной линии .....	350
9.8.2. Специальные символы редактирования и выдачи сигналов .....	353
9.8.3. Управляющая структура <i>termios</i> .....	355
9.8.4. Задание .....	358
9.9. Лабораторная работа № 9. Датаграммные локальные каналы .....	359
9.9.1. Подготовка к выполнению работы .....	359
9.9.2. Порядок выдачи системных вызовов .....	359
9.9.3. Создание сокета .....	360
9.9.4. Связывание сокета со своим адресом .....	361
9.9.5. Прием и передача датаграмм .....	363
9.9.6. Программы взаимодействующих процессов .....	366
9.9.7. Задание .....	369
9.10. Лабораторная работа № 10. Сетевые датаграммные каналы .....	370
9.10.1. Состав и порядок выдачи системных вызовов .....	370
9.10.2. Создание сокета .....	370
9.10.3. Связывание сокета со своим адресом .....	371
9.10.4. Прием и передача датаграмм .....	373
9.10.5. Задание .....	375

9.11. Лабораторная работа № 11. Локальные виртуальные соединения.....	376
9.11.1. Порядок выдачи системных вызовов .....	376
9.11.2. Создание очереди запросов на соединение и работа с ней.....	377
9.11.3. Работа с виртуальным соединением .....	379
9.11.4. Закрытие виртуального соединения.....	380
9.11.5. Программы взаимодействующих процессов.....	380
9.11.6. Задание.....	384
9.12. Лабораторная работа № 12. Сетевые виртуальные соединения.....	385
9.12.1. Состав и порядок выдачи системных вызовов.....	385
9.12.2. Задание адреса главного сокета.....	386
9.12.3. Использование адреса главного сокета.....	387
9.12.4. Задание.....	388
Литература.....	390

## Введение

Данное пособие предназначено для обучения студентов специальности 220300 «Системы автоматизированного проектирования» по односеместровой дисциплине «Операционные системы», изучаемой во втором семестре обучения. Кроме того, оно может использоваться в качестве основного или дополнительного пособия по курсам:

- 1) «Организация ЭВМ и систем» (специальность 220300);
- 2) «Сети ЭВМ и телекоммуникации» (специальность 220300);
- 3) «Системное программное обеспечение» (специальность 220201);
- 4) «Вычислительные машины, системы и сети» (специальность 220201);
- 5) «Информационные сети и коммуникации» (специальность 220201).

Основной целью дисциплины «Операционные системы» является изучение принципов организации операционных систем на примере операционной системы *UNIX*. Задачи изучения дисциплины:

- 1) изучение основных принципов реализации пользовательских интерфейсов операционной системы и получение практических навыков по программированию на командном языке операционной системы *UNIX*;
- 2) изучение основных принципов реализации мультипрограммирования и многопользовательской работы системы;
- 3) ознакомление с программными и аппаратными средствами вычислительной системы, предназначенными для реализации мультипрограммирования;
- 4) изучение принципов организации информации на периферийных устройствах, а также принципов управления этими устройствами;
- 5) ознакомление с принципами построения сетей передачи данных.

Изучение данной дисциплины предполагает предварительное освоение студентами следующих дисциплин: 1) «Информатика»; 2) «Алгоритмические языки и программирование». Изучение дисциплины «Операционные системы» заканчивается получением зачета и сдачей экзамена.

Данное пособие состоит из девяти глав, которые можно разбить на четыре группы:



1) пользовательские интерфейсы и основные принципы построения операционных систем — главы 1–3;

2) реализация основных программных интерфейсов — главы 4, 6 и 8;

3) управление аппаратными средствами вычислительной системы — главы 5 и 7;

4) лабораторные работы — глава 9.

В соответствии с данным разбиением пособия на группы глав могут быть реализованы различные учебные курсы по дисциплине «Операционные системы»:

1) вводный курс — главы 1–3, разделы 9.1–9.4;

2) базовый курс — главы 1–4, 6, разделы 9.1–9.8;

3) подробный курс — главы 1–9.

Так как для специальности 220300 требуется углубленная подготовка в области программирования, то используется третий вариант, представленный в данном пособии.

Существующие подходы к изложению дисциплины «Операционные системы» можно разделить на два: 1) односистемный; 2) многосистемный. В первом из этих подходов изложение ведется на примере одной, а во втором — на примере нескольких операционных систем. Несмотря на то что многосистемный подход преследует цель обеспечения системности в восприятии материала, практическое достижение этой цели весьма затруднено, во-первых, из-за чрезмерного объема информации (попытка объять необъятное). Во-вторых, абстрактное изложение материала достаточно скучно для читателя, привыкшего к работе в среде одной-двух конкретных операционных систем.

В данном учебном пособии используется односистемный подход, предполагающий рассмотрение операционной системы *UNIX*. Применительно к этой системе термин «односистемный подход» достаточно условный, так как в настоящее время «*UNIX*» является собирательным названием, обозначающим достаточно большую группу реальных ОС, имеющих схожие пользовательские и программные интерфейсы. Существенными достоинствами любой *UNIX*-системы являются, во-первых, универсальность — пригодность для решения практически любой задачи по переработке информации, независимо от особенностей алгоритма этой задачи и от числа пользователей, участвующих в ее решении; во-вторых, *UNIX* неприсотлива к используемой аппаратной базе и может выполняться на различных конфигурациях аппаратуры

и на процессорах различных моделей. С учетом того что практически все читатели имеют опыт работы в среде операционной системы *Windows*, а многие — и в среде *MS-DOS*, в первых главах пособия наряду с *UNIX* приводятся сведения и об этих системах.

Следует отметить, что при описании системных программных вызовов в пособии приводятся не реальные системные вызовы *UNIX*, записанные на языке программирования этой системы СИ, а упрощенные их варианты, записанные на псевдоязыке. Его применение преследует цель сделать материал пособия доступным для читателя, не знакомого с СИ, а также позволяет избавиться от деталей, связанных с применением данного языка и не относящихся к ОС. Запись любого системного вызова на этом псевдоязыке представляет собой русское название требуемой операции, за которым в круглых скобках приведен список параметров вызова, причем входные и выходные параметры разделяются символами «||». При этом с целью упрощения изложения материала некоторые второстепенные параметры системных вызовов *UNIX* опущены.

# 1. ИНТЕРФЕЙСЫ ПОЛЬЗОВАТЕЛЯ СИСТЕМЫ

## 1.1. Функции системных программ

Любая **вычислительная система (ВС)** предназначена для выполнения некоторого множества задач по переработке информации. Сущность подобной задачи состоит в том, что имеется некоторая исходная информация, на основе которой требуется получить другую — результирующую информацию.

Каждая задача, решаемая ВС, имеет алгоритм решения. **Алгоритм** — правило, определяющее последовательность действий над исходными данными, приводящую к получению искомых результатов. Форма представления алгоритма решения задачи, ориентированная на машинную реализацию, называется **прикладной программой**. Совокупность аппаратных средств ВС, предназначенных для выполнения машинных программ, часто называют просто **аппаратурой**.

На рис. 1 приведена структура аппаратных средств однопроцессорной ЭВМ с общей шиной. В данной структуре центральным связывающим звеном между основными блоками является **общая шина (ОШ)**. При этом под термином **шина** понимается группа параллельных проводов. ОШ в общем случае есть объединение трех шин, таких как: 1) шина управления; 2) шина адреса; 3) шина данных. Рассмотрим кратко назначение других аппаратных блоков.

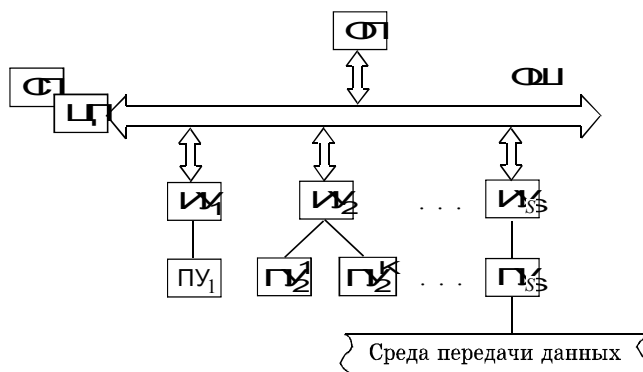


Рис. 1. Однопроцессорная ЭВМ с общей шиной:  
ЦП — центральный процессор; СП — сопроцессор; ОШ — общая шина;  
ОП — оперативная память; ПУ — периферийное устройство;  
ИУ — интерфейсное устройство

Главным аппаратным блоком любой ЭВМ является **центральный процессор (ЦП)**. Это «мозг» ЭВМ, который непосредственно выполняет все программы, в том числе и прикладные. Любая программа представляет собой последовательность машинных команд, каждая из которых требует для своего размещения один, два или большее число байтов. На рис. 2 приведена структура наиболее типичной машинной команды. Здесь **КОП** — код операции. Это комбинация битов, кодирующая тип операции, которую следует выполнить над операндами (например, суммирование). Операнд 1, операнд 2 — это или сами данные, над которыми выполняется машинная команда, или адреса в памяти (ОП или регистры), где эти данные находятся.

КОП	Операнд 1	Операнд 2
-----	-----------	-----------

Рис. 2. Структура машинной команды

Термин «однопроцессорная ЭВМ» означает, что в рассматриваемой ЭВМ используется единственный центральный процессор. Кроме него практически любая ВС имеет специализированные процессоры. Примером специализированного процессора является **сопроцессор** — процессор, расположенный на той же плате, что и ЦП, и имеющий такой же доступ к ОП. В отличие от ЦП сопроцессор предназначен для выполнения не всей прикладной программы, а лишь отдельных ее команд (чаще всего команд обработки данных с плавающей точкой). Каждая такая команда выполняется сопроцессором как отдельная программа, записанная на его специализированном машинном языке.

**Оперативная память (ОП)** предназначена для кратковременного хранения программ и обрабатываемых ими данных. Название обусловлено тем, что операции чтения содержимого ячеек памяти и записи в них нового содержимого производятся достаточно быстро. Иногда используют другое название — **операционная память**. Это название обусловлено тем, что ЦП может достаточно просто считывать машинные команды из ОП и исполнять их. Логическая структура любой ОП представляет собой линейную последовательность ячеек, которые пронумерованы (рис. 3). В зависимости от ЭВМ ячейкой является или байт или машинное слово. Номер ячейки называется **физическим** или **реальным адресом** этой ячейки. В простейших ЭВМ поле операнда в машинной команде содержит этот номер.

**Периферийные устройства (ПУ)** — устройства ввода-вывода, устройства внешней памяти, а также устройства сопряжения со средой передачи данных. Посредством **устройств ввода-вывода** ЭВМ «разговаривает» с человеком-пользователем. Сюда относятся: клавиатура, экран (дисплей), мышь, принтер и т.д.

**Устройство внешней памяти** предназначено для работы с **носителем внешней памяти**. Примером такого устройства является дисковод. Он работает с носителем внешней памяти — магнитным диском. **Внешняя память (ВП)** имеет следующие отличия от ОП:

1) обмен информацией между ЦП и ВП выполняется во много раз медленнее, чем между ЦП и ОП;

2) ЦП не может выполнять команды, записанные в ВП. Для выполнения этих команд их необходимо предварительно переписать в ОП;

3) информация на носителе ВП сохраняется и после выключения питания.

В прошлом емкость носителей ВП многократно превосходила емкость ОП. В настоящее время это выполняется не для всех типов носителей.

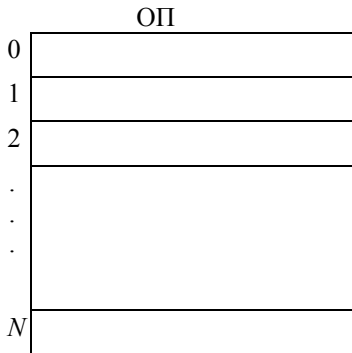


Рис. 3. Логическая структура ОП

**Устройство сопряжения со средой передачи** данных используется для подсоединения ЭВМ к сети. Например, если в качестве среды передачи данных используется пара проводов, то в качестве устройства сопряжения может быть применен модем.

**Интерфейсное устройство (ИУ)** предназначено для того, чтобы согласовать стандартную для данной ЭВМ структуру ОШ с конкретным типом ПУ, которых существует очень много.

Поясним смысл слова: ***интерфейс*** — граница между двумя взаимодействующими системами. В данном случае речь идет о взаимодействии ОШ (с одной стороны) и ПУ (с другой).

Несмотря на то что наличие аппаратуры и прикладных программ является минимально достаточным условием для решения задач по переработке информации, ВС, состоящие только из этих двух подсистем, на практике не используются. Обязательной подсистемой любой ВС являются также системные программы. Благодаря им и пользователи ВС, и их прикладные программы имеют дело не с реальной («голой») аппаратурой, а взаимодействуют с ***виртуальной*** (кажущейся) ***ЭВМ***.

На рис. 4 показано наиболее укрупненное представление ВС, в том числе наиболее важные системные интерфейсы. Например, интерфейс пользователя ВС с аппаратурой представляет собой клавиши мыши и клавиатуры, поверхность экрана, а также различные кнопки на системном блоке и на периферийных устройствах. Интерфейс между аппаратурой и программами представляет собой язык машинных команд. Совокупность интерфейсов, используемых пользователем или прикладной программой, и представляет собой соответствующую виртуальную машину.

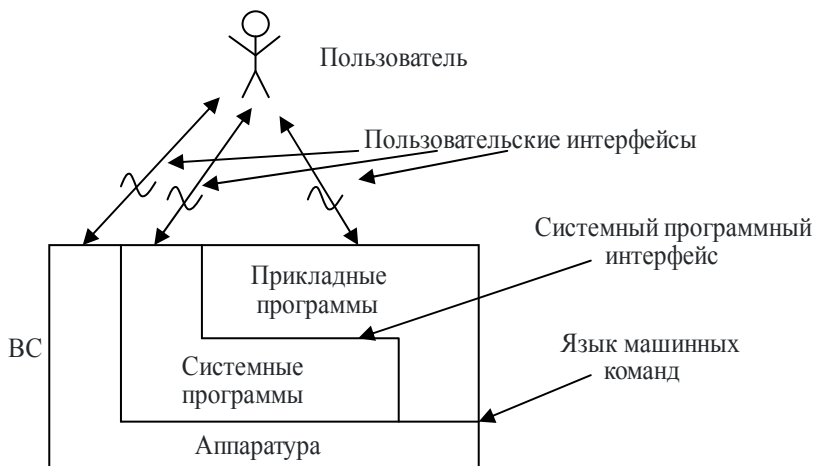


Рис. 4. Укрупненное представление вычислительной системы

На рис. 5 приведена классификация системных программ.

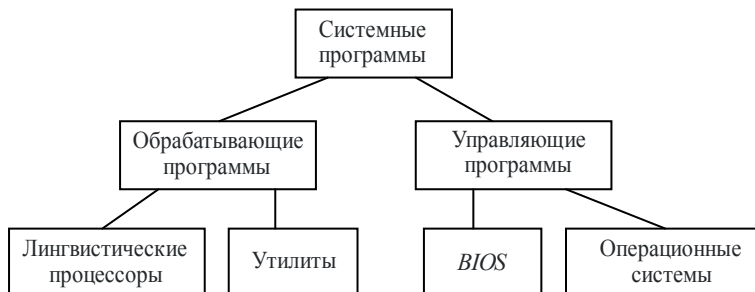


Рис. 5. Классификация системных программ

**Обрабатывающие системные программы** отличаются от управляющих программ как по своим функциям, так и по способу их инициирования (запуска). Основные функции обрабатывающих программ:

1) перенос информации. Перенос может выполняться между различными устройствами или в пределах одного устройства. При этом под устройствами понимаются: ОП, устройства ВП, устройства ввода-вывода;

2) преобразование информации. То есть после считывания информации с устройства обрабатывающая программа преобразует эту информацию, а уж затем записывает ее на это же или на другое устройство.

В зависимости от того, какая из этих двух функций является основной, обрабатывающие системные программы делятся на утилиты и лингвистические процессоры. Основной функцией **утилиты** является перенос информации, а основная функция **лингвистического процессора** — преобразование информации.

Запуск обрабатывающей системной программы аналогичен запуску прикладной программы.

Основные функции **управляющих программ**:

1) оказание помощи прикладным и системным обрабатывающим программам в использовании ими ресурсов ВС. При этом различают информационные, программные и аппаратные ресурсы. Данная функция реализуется во всех ВС;

2) обеспечение **однопользовательской мультипрограммности** — одновременное выполнение нескольких прикладных и (или) системных обрабатывающих программ в интересах одного пользователя. Эта функция реализуется лишь в мультипрограммных ВС.

В однопользовательских однопрограммных ВС эта функция отсутствует;

3) обеспечение *многопользовательской мультипрограммности* — одновременное выполнение нескольких обрабатывающих программ (прикладных и системных) в интересах нескольких пользователей. Данная функция реализуется лишь в многопользовательских ВС.

Управляющие системные программы делятся на две группы: программы *BIOS* и программы операционной системы. ***BIOS*** — базовая система ввода-вывода. Сюда относятся системные программы, находящиеся в ПЗУ (постоянное запоминающее устройство). Эти программы выполняют многие функции обмена с периферийными устройствами, участвуя таким образом в выполнении первой из перечисленных выше функций управляющих программ.

***Операционная система (ОС)*** — множество управляющих программ, предназначенных для выполнения всех трех перечисленных выше функций. Примером однопользовательской однопрограммной ОС является *MS-DOS*. Примерами однопользовательских мультипрограммных ОС являются различные *Windows*. Операционная система *UNIX* является примером многопользовательской системы.

При рассмотрении пользовательских интерфейсов невозможно обойтись без понятия файла. В отличие от многих других объектов, управляемых ОС, файлы «видимы» для пользователя и используются им при формировании своих команд для ОС.

## 1.2. Файлы

Вся информация, обрабатываемая ВС, содержится в ней на устройствах: ОП, ВП, устройствах ввода-вывода. При этом на любом из устройств информация хранится в виде длинной битовой строки, т.е. в виде последовательности нулей и единиц. Длина одной такой битовой строки может составлять многие сотни гигабит ( $1 \text{ Гбит} = (1024)^3$ ). Так как работать с такой длинной строкой чрезвычайно неудобно, то она разделяется на поименованные части разной длины, называемые файлами. Более точное определение: ***файл*** — часть пространства носителя ВП (разрывная или непрерывная), которой присвоено имя, уникальное для данной ВС.

Информация на носителе делится на части-файлы по смысловому принципу. Например, один файл может содержать текст



исходной программы, второй — ее объектный модуль, а третий — загрузочный модуль. Кроме того, в большинстве современных ОС каждое устройство ввода-вывода также считается файлом. Что касается ОП, то информация на ней делится не на файлы, а на сегменты. Так как это разбиение на сегменты скрыто от пользователя ВС, оно будет рассмотрено нами позже.

Прежде чем обсуждать имена файлов, рассмотрим небольшую классификацию имен объектов, находящихся под управлением ОС (рис. 6). В качестве первого признака классификации использовано назначение имени. **Пользовательские имена объектов** используются для общения между пользователем и ВС, то есть для реализации пользовательских интерфейсов. **Программные имена объектов** используются при реализации системных программных интерфейсов. **Системные имена объектов** используются самой ОС для своих внутренних потребностей. Существуют также смешанные имена, которые могут быть одновременно и пользовательскими и программными, или и программными и системными. В качестве второго признака классификации имен объектов используется форма представления имени — символьная или численная. Пользовательские имена бывают только символьными, а системные — только численными. Программные имена бывают и символьными и численными. В дальнейшем мы будем часто использовать данную классификацию имен.

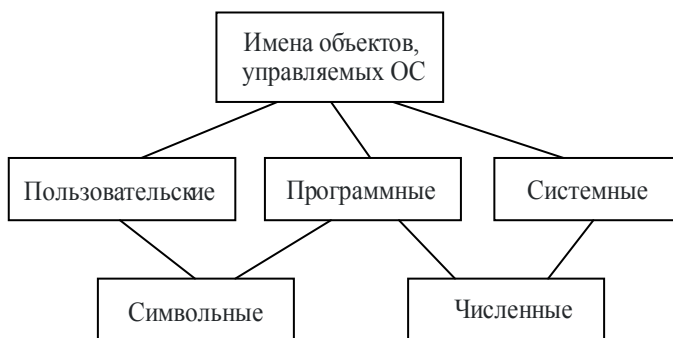


Рис. 6. Классификация имен объектов, управляемых ОС

Применительно к файлу существуют три пользовательских имени. Наиболее короткое из них — **простое имя файла**. Это имя дает файлу пользователь при его создании. Ограничения на выбор простого имени файла определяются типом используемой ОС.

В достаточно старых операционных системах используется короткое простое имя файла. Например, в *MS-DOS* (за исключением версии 7.0) длина простого имени не может превышать 12 символов по схеме «8.3». При этом до восьми символов имеет **собственно имя файла**, до трех символов — **расширение имени файла**, один символ — разделительная точка. Часть имени левее точки называют также **префиксом имени**, а правее точки — **суффиксом имени**. В старых версиях *UNIX* максимальная длина простого имени файла составляет 14 символов. При этом обычно никакой символ не играет особой роли в имени файла (исключением являются некоторые системные обрабатывающие программы, учитывающие точку). В современных реализациях *UNIX*, а также в *MS-DOS* версии 7.0 и в различных *Windows* длина простого имени файла может достигать 255 символов.

Файл является достаточно крупной единицей информации. Для удобства работы с ним битовая строка, образующая файл, делится на части, называемые **записями**. Так как при взаимодействии с ВС пользователь «не видит» записи файла, мы вернемся к ним позже, когда будем рассматривать не пользовательские, а программные интерфейсы.

В реальной ВС одновременно существуют многие тысячи файлов. Для того чтобы пользователь мог ориентироваться в этом «море», используется **файловая структура системы**. Для построения такой структуры применяются специальные файлы, называемые **каталогами** (в *Windows* каталоги называются **папками**). Каждая запись файла-каталога содержит сведения об одном файле, «зарегистрированном» в данном каталоге. Применительно к *UNIX* это: 1) простое имя файла; 2) численный номер файла, используемый ОС для получения уникального имени файла. Если файл «зарегистрирован» в каталоге, то говорят, что между каталогом и файлом существует **жесткая связь**. При этом каталог является по отношению к файлу **родительским каталогом**.

Так как в каталоге могут быть «зарегистрированы» не только файлы данных, но и каталоги, то появляется возможность связать все файлы системы в единую **иерархическую (древовидную) структуру**. При этом корнем дерева является **корневой каталог**. На следующем уровне дерева находятся те файлы и каталоги, сведения о которых содержатся в корневом каталоге. Аналогично каталоги первого уровня дерева «порождают» файлы и каталоги второго уровня и т.д.

В *MS-DOS* файловая структура системы представляет собой не одно, а несколько деревьев по числу логических дисков. При этом корень каждого дерева имеет имя «\» (обратный слеш). Простое имя любого другого каталога отличается от имени обычного файла данных тем, что оно не может иметь расширения имени.

Пользователь *Windows* имеет дело не с реальной файловой структурой системы, аналогичной файловой структуре *MS-DOS*, а с виртуальной структурой, представляющей собой единое дерево. Корнем этого дерева является виртуальный каталог «Рабочий стол». Его подкаталогами являются другие виртуальные каталоги, один из которых («Мой компьютер») является «родителем» для корневых каталогов логических дисков.

В *UNIX* файловая структура системы представляет собой единое дерево. На рис. 7 приведен фрагмент этого дерева. Корневой каталог имеет имя «/» (прямой слеш). Простое имя любого другого каталога ничем не отличается от имени файла данных. В отличие от *MS-DOS* и *Windows* файловая структура в *UNIX* представляет собой дерево с пересечениями. Наличие таких пересечений обусловлено тем, что один и тот же файл может быть одновременно «зарегистрирован» не в одном, а в нескольких каталогах. При этом все жесткие связи файла полностью равноправны. Следствием этого является то, что для уничтожения файла требуется уничтожение записей о нем во всех его родительских каталогах.

Большим достоинством любой древовидной файловой структуры является то, что она позволяет пользователю не заботиться об уникальности простых имен файлов. Это объясняется тем, что ОС работает не с этими пользовательскими именами файлов, а с путями. **Имя-путь файла**, называемое также **абсолютным именем файла**, представляет собой последовательность всех имен, начиная с корневого каталога и кончая простым именем файла. При этом имя каждого промежуточного каталога в имени-пути завершается символом «/» для *UNIX* или «\» для *MS-DOS*. Например, на рис. 7 два файла имеют одинаковое простое имя *a.txt*, но абсолютные имена у них разные: */home/vlad/a.txt* и */home/andrei/a.txt*.

В любой конкретный момент времени один каталог в файловой структуре однопользовательской системы является особенным. Это **текущий каталог**. В многопользовательской системе, например, в *UNIX*, у каждого пользователя есть свой текущий каталог. Если искомый файл «зарегистрирован» в текущем каталоге, то его можно задать для ОС не с помощью имени-пути, а пользуясь его

простым именем. ОС сама получит имя-путь файла, соединив имя-путь каталога с простым именем файла.

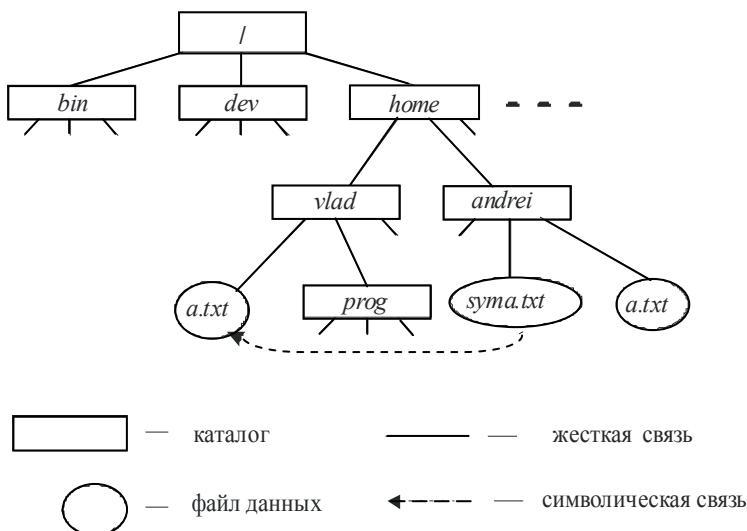


Рис. 7. Фрагмент файловой структуры UNIX

Если адресуемый файл является «потомком» текущего каталога, то в качестве имени этого файла можно использовать «смещение» относительно текущего каталога. Такое пользовательское имя файла называется **относительным именем**. Например, если текущим каталогом является `/home`, то записанное выше имя-путь файла `/home/vlad/a.txt` может быть заменено более коротким именем `vlad/a.txt`. Обратите внимание на отсутствие в начале этого имени символа «/». Его наличие всегда говорит о том, что записано полное имя-путь.

Общепринято использовать для обозначения текущего каталога символ «.», а для обозначения родительского каталога (по отношению к текущему каталогу) — «..». Например, если текущим каталогом является `/home/vlad`, то для задания файла `/home/andrei/a.txt` пользователь может использовать относительное имя `../andrei/a.txt`.

Особенностью файловой структуры в мультипрограммной системе является наличие символических связей. **Символическая связь** (в Windows — **ярлык**) — специальный файл, содержащий пользовательское имя другого — целевого файла. Например, на

рис. 7 файл *syml.txt* является символической связью с файлом *a.txt*. То есть содержимым файла *syml.txt* является */home/vlad/a.txt*. Символическая связь используется для косвенной адресации целевого файла. При этом, задав в команде пользовательского или программного интерфейса имя символической связи, мы заставим ОС выполнить требуемое действие над целевым файлом, например, выполнить создание этого файла. Некоторые из команд «не следуют символической связи» и выполняют заданное действие не над целевым файлом, а над самой символической связью. Примером является команда удаления файла.

Кроме перечисленных выше типов файлов (файлы данных, каталоги, символические связи) любая современная ОС имеет еще один тип файлов — файлы-устройства, называемые также **специальными файлами**. Наличие таких файлов позволяет и пользователю и программам работать с периферийными устройствами почти так же, как и с файлами данных. Если в однопрограммной системе, например в *MS-DOS*, прикладная программа может взаимодействовать с ПУ не только через файловую систему, но и через драйверы *MS-DOS*, *BIOS*, а также напрямую через порты, то в мультипрограммной системе такая «роскошь» недопустима. В *UNIX* специальные файлы находятся или в каталоге */dev*, или в его дочерних каталогах. Заметим, что в качестве «устройств» могут выступать не только настоящие ПУ (устройства ввода-вывода и ВП), но и области ОП и ВП. Примеры специальных файлов:

*/dev/fd0* — дисковод гибких дисков;  
*/dev/lp0* — параллельный порт 0;  
*/dev/tty00* — последовательный порт *COM1*;  
*/dev/rz0a* — первый (*a*) раздел первого (0) жесткого диска;  
*/dev/kmem* — линейная виртуальная ОП ядра ОС.

В каталоге */bin* находятся наиболее часто используемые утилиты *UNIX*. Несмотря на то что утилиты не являются частью ОС и рассматриваются ею как обычные прикладные программы, их рассмотрение будет полезно нам по следующим соображениям:

- 1) с точки зрения рядового пользователя (не программиста) утилиты являются неотъемлемой частью ВС;
- 2) рассмотрение утилит позволяет лучше выделить саму ОС;
- 3) практическое применение некоторых утилит позволяет выявить многие свойства используемой ОС.

### 1.3. Утилиты

Как отмечалось ранее, основной функцией утилиты является перенос информации в пределах ВС. При рассмотрении каждой конкретной утилиты пользователя системы интересуют функции этой утилиты, а также ее имя, используемое для передачи в систему через пользовательский интерфейс в качестве команды для ОС. При работе с системой *UNIX* общий формат такой пользовательской команды следующий:

*имя [флаги] [файлы],*

где: 1) квадратные скобки заключают необязательную часть команды;

2) *имя* — пользовательское имя исполняемого файла, содержащего загрузочный модуль (машинный код) утилиты;

3) *файлы* — имена файлов, над которыми утилита выполняет свои действия. Различают **входные файлы**, информация из которых (или информация о которых) используется утилитой в качестве ее исходных данных, а также **выходные файлы**, в которые утилита помещает результаты своей работы. По умолчанию большинство системных утилит использует в качестве входного файла клавиатуру, а в качестве выходного файла — экран. Эти устройства (и соответствующие им файлы) часто называют соответственно **стандартным вводом** и **стандартным выводом**;

4) *флаги* — двоичные параметры команды, уточняющие действие, которое должна выполнить запускаемая утилита. Флаг задается своим именем из одной буквы, которой предшествует символ «-». Некоторые флаги уточняются своими параметрами, которые отделяются от имени флага пробелами.

Ниже приводится краткое описание утилит, используемых пользователями операционной системы *UNIX* для работы с файлами. После имени каждой утилиты в скобках приводится название аналогичной или близкой команды в *MS-DOS*. Рассматриваемые утилиты можно разбить на группы: 1) идентификация и установка текущего каталога; 2) создание каталогов и анализ их содержимого; 3) копирование, переименование и перенос файлов; 4) уничтожение файлов и каталогов; 5) работа с текстовой информацией; 6) поиск информации; 7) выдача справочной информации; 8) упрощение пользовательского интерфейса. Рассмотрение утилит, участвующих в обеспечении многопользовательской работы ВС, будет выполнено в других разделах.

## 1. Идентификация и установка текущего каталога

1. Вывод абсолютного имени текущего каталога (в *MS-DOS* отсутствует, так как это имя является частью приглашения к вводу команды):

***pwd***

Это наиболее простая команда *UNIX*, которая не имеет ни одного параметра.

2. Смена текущего каталога (в *MS-DOS* — *cd*):

***cd*** [*каталог*]

Если каталог опущен, то текущим каталогом станет корневой каталог поддерева каталогов данного пользователя (например каталог *vlad* на рис. 7).

Имя каталога может быть как абсолютным, так и относительным. Задание абсолютного имени позволяет сделать текущим любой каталог, а задание относительного имени — только каталог-потомок действующего текущего каталога. Если в начале относительного имени каталога записать символы «~/», то смещение нового текущего каталога вычисляется относительно корневого каталога данного пользователя. Если в качестве имени каталога задать символы «..», то новым текущим каталогом станет «родитель» действующего текущего каталога.

Данная утилита не имеет флагов. К этому добавим, что *cd* не является утилитой в полном смысле этого слова, так как она существует не в виде отдельного исполняемого файла, а в виде подпрограммы ОС (точнее — ее интерпретатора команд). Подобное свойство обусловлено небольшими размерами данной подпрограммы и для пользователя ВС не заметно.

## 2. Создание каталогов и анализ их содержимого

1. Создание нового каталога (каталогов) (в *MS-DOS* — *mkdir*):

***mkdir*** [*каталоги*]

Имена создаваемых каталогов могут быть заданы в любом виде: простые, относительные, абсолютные.

Единственный флаг данной утилиты:

***-m*** — создать каталог с заданным режимом доступа. Режимы доступа будут рассмотрены в подразд. 3.2.

2. Вывод содержимого каталога на экран (в *MS-DOS* — *dir*):

***ls*** [*каталог или файлы*]

Если параметр опущен, то на экран выводится содержимое текущего каталога, иначе — содержимое заданного каталога. Если заданы имена файлов, то на экран выводятся сведения об этих файлах, при условии что их имена присутствуют в текущем каталоге.

Данная утилита имеет 23 флага. Приведем только некоторые из них:

- 1) **-R** — рекурсивный вывод подкаталогов заданного каталога;
- 2) **-F** — пометить исполняемые файлы символом «\*», каталоги — символом «/», а символические связи — «@»;
- 3) **-l** — вывод наиболее подробной информации о файлах;
- 4) **-a** — вывод списка всех файлов и подкаталогов заданного каталога (по умолчанию не выводятся имена, начинающиеся с символа «.»).

### 3. Копирование, переименование и перенос файлов

#### 1. Копирование файла (в *MS-DOS* — *copy*):

**ср** *исходный\_файл (или каталог) конечный\_файл (или каталог)*

Первый параметр команды задает источник копирования, а второй параметр — место размещения копии. При этом копирование может производиться из файла в файл, из файла в каталог, а также из каталога в каталог. В любом из этих случаев создается не новая жесткая связь (связи), а новый файл (файлы).

При копировании из файла в каталог в последнем создается новая запись, состоящая из простого имени исходного файла и из системного номера нового файла. При копировании из каталога в каталог копируются все файлы (в том числе и подкаталоги) из исходного каталога в конечный каталог. При этом для каждого копируемого файла создается новый файл с точно таким же содержанием, после чего новый файл регистрируется в конечном каталоге. Для копирования из каталога в каталог требуется, чтобы был записан флаг **-r**.

#### 2. Переименование файлов и их перемещение (в *MS-DOS* — *rename, move*):

**mv** *исходный\_файл (или каталог) конечный\_файл (или каталог)*

Если исходный и конечный файлы находятся в одном и том же каталоге, то данная утилита заменяет имя исходного файла на имя конечного файла. Если же эти файлы находятся в разных каталогах,



то производится «перемещение» файла по файловой структуре системы. При этом запись файла в исходном каталоге уничтожается, а точно такая же запись в конечном каталоге, наоборот, создается. Если в качестве первого операнда задан файл, а в качестве второго — каталог, то также производится перемещение файла в заданный каталог. Если в качестве обоих операндов заданы имена каталогов, то производится переименование каталога, соответствующего первому операнду.

3. Создание жестких и символических связей (в *MS-DOS* отсутствует):

*ln* *исходный\_файл* *файл\_ссылка* (или *каталог*)

Эта команда создает новую связь с исходным файлом. При отсутствии флага *-s* создается жесткая связь с этим файлом. В этом случае файл-ссылка представляет собой новое имя уже существующего файла. Если в качестве второго параметра команды задано не имя файла, а имя каталога, то в этом каталоге исходный файл будет зарегистрирован под своим простым прежним именем. При наличии флага *-s* создаваемый файл-ссылка представляет собой символическую связь с исходным файлом.

## 4. Уничтожение файлов и каталогов

1. Удаление файлов и каталогов (в *MS-DOS* — *del*):

*rm* *файлы* (или *каталог*)

Эта утилита удаляет не сами файлы, а записи о них в родительских каталогах. Само удаление файла происходит только в том случае, если число жестких связей для этого файла станет равным 0.

Если задать флаг *-r*, то данная команда выполнит удаление заданного каталога и всех содержащихся в нем файлов и подкаталогов. Другие флаги:

*-f* — удаление файлов без запроса подтверждения;

*-i* — обязательный запрос подтверждения при удалении каждого файла.

2. Удаление каталогов (в *MS-DOS* — *rmdir*):

*rmdir* *каталоги*

Данная команда может уничтожить каталог только в том случае, если он не содержит файлов и подкаталогов.

## 5. Работа с текстовой информацией

1. Создание новых текстовых файлов и корректировка существующих. Данную функцию выполняют утилиты, называемые **текстовыми редакторами**. Примеры текстовых редакторов: *ed*, *sed*, *vi*. (Текстовый редактор в *MS-DOS* — *edit*.) В качестве примера приведем вызов редактора *sed*:

***sed*** [*файлы*]

Данный редактор редактирует заданные в команде файлы построчно, от меньших номеров строк к большим, без возврата к ранее пройденным строкам. Редактирование строк производится согласно командам редактирования, заданным одним из двух способов:

- 1) в качестве параметров флага **-e**;
- 2) команды редактирования содержатся в файле, имя которого задано в качестве параметра флага **-f**.

Если ни одно имя файла в команде не задано, то по умолчанию входным файлом считается клавиатура. Набираемые на ней строки и будут подвергаться редактированию. В этом случае произойдет создание нового текстового файла, который с помощью интерпретатора команд ОС может быть записан на диск.

2. Вывод текстового файла на экран (в *MS-DOS* — *type*):

***cat*** [*файлы*]

Данная утилита выводит на экран содержимое всех текстовых файлов, заданных в качестве ее параметров. При этом содержимое выводимых файлов на экране никак не разделяется. Если ни один из файлов не задан, то на экран выводится последовательность символов, введенная с клавиатуры (напомним, что клавиатура — тоже файл). Ввод с клавиатуры будет выполняться также в том случае, если вместо любого имени файла записан символ «-». Для завершения ввода символов с клавиатуры следует одновременно нажать две клавиши: **<Ctrl>&<D>** («конец файла»).

3. Сортировка и слияние файлов (в *MS-DOS* — *sort*):

***sort*** *файлы*

Если флагов нет, то данная команда выполняет слияние перечисленных файлов в единый файл. Причем строки этого файла сортируются в лексикографическом порядке. По умолчанию результат выводится на экран.

Два флага этой команды:

**-и** — при наличии нескольких одинаковых строк результат содержит только одну строку;

**-о файл** — вывод результата делается не на экран, а в заданный файл.

4. Вывод текста, вводимого с клавиатуры, на экран и одновременное копирование этого текста в заданный файл (файлы):

**tee файлы**

Один из флагов этой команды:

**-а** — запись текста не в начало файла (при этом файл создается заново), а добавление текста в конец существующего файла (файлов).

5. Вывод строки символов на экран (в *MS-DOS* — *echo*):

**echo строка**

Как и команда *cd*, данная команда выполняется не отдельной утилитой, а подпрограммой интерпретатора команд ОС.

## 6. Поиск информации

1. Поиск файлов (в *MS-DOS* — *find*):

**find каталог [флаги]**

Данная утилита осуществляет поиск файлов в поддереве файловой структуры, корнем которого является заданный каталог. Условия поиска задаются с помощью флагов. В отличие от ранее перечисленных утилит, флаги задаются в конце команды. Из всех многочисленных флагов обратим внимание на два:

1) **-type тип** — поиск файлов указанного типа. Аргумент *тип* может принимать следующие значения: **b** (файл — блочное устройство), **c** (файл — символьное устройство), **d** (файл — каталог), **f** (обычный файл), **l** (файл — символическая связь), **p** (файл — именований канал);

2) **-name имя** — поиск файлов с указанным именем.

В отличие от ранее рассмотренных команд, утилита *find* имеет собственные метасимволы. **Метасимвол** — символ, имеющий для рассматриваемой программы специальное значение. Метасимволы утилиты *find* позволяют задавать простые имена сразу нескольких искомых файлов в виде всего одного имени. Перечислим эти метасимволы: «\*», «?», « [...] ». Назначение каждого из этих метасимволов аналогично назначению одноименного метасимвола *shell* и будет рассмотрено нами позднее.

В одной команде *find* можно задать несколько условий поиска, соединив их при помощи следующих логических операторов:

- a — логическое И;
- o — логическое ИЛИ;
- ! — логическое НЕ.

2. Поиск строк в текстовых файлах (в *MS-DOS* отсутствует):

***fgrep*** *подстрока* [*файлы*]

Данная утилита осуществляет поиск в перечисленных файлах строк, имеющих в своем составе шаблон — заданную подстроку. Найденные строки выводятся на экран. Если имена файлов опущены, то поиск осуществляется в тексте, вводимом с клавиатуры. При вводе с клавиатуры каждая строка, содержащая требуемую подстроку, повторяется дважды: первый раз она содержит «эхо» вводимых с клавиатуры символов, а второй раз выводится командой *fgrep*.

Некоторые флаги этой команды:

-x — выводятся только строки, полностью совпадающие с шаблоном;

-c — выводится только количество строк, содержащих шаблон;

-i — при поиске не различаются строчные и прописные буквы;

-l — выводятся только имена файлов, содержащих требуемые подстроки;

-n — перед каждой выводимой строкой записывается ее относительный номер в файле.

Если задан поиск в нескольких файлах, то перед выводом каждой строки выводится имя соответствующего файла.

## 7. Выдача справочной информации

1. Выдача статистики о текстовых файлах (в *MS-DOS* отсутствует):

***wc*** [*файлы*]

Данная утилита выдает статистику о своих входных файлах. Если эти файлы не заданы, выдается статистика о тексте, введенном с клавиатуры.

Флаги этой команды:

-l — вывод числа строк;

-w — вывод числа слов;

-c — вывод числа символов.

По умолчанию все три флага установлены (*-lwc*). Поэтому флаги записываются в этой команде только тогда, когда требуется ограничить выходную статистику.

2. Вывод и установка даты и времени (в *MS-DOS* – *date, time*):

**date** [*mmddhhnn*[*yy*]]

Если параметр команды не задан, то на экран выводятся текущие дата и время. Это день недели, месяц, число, время (час, минуты, секунды), год.

Если параметр команды задан, то она выполняет установку текущей даты и времени. При этом параметр команды *date* включает:

*mm* — номер месяца;

*dd* — число;

*hh* — час (в 24-часовой системе);

*nn* — минуту;

*yy* — последние две цифры года (необязательная часть параметра команды).

Следует отметить, что выполнять установку даты может только суперпользователь (администратор).

3. Следующая утилита выводит краткую информацию о системе (в *MS-DOS* – *ver*):

**uname** флаги

Значения флагов:

**-a** — вывод всей доступной информации (объединение всех остальных флагов);

**-m** — вывод информации об аппаратуре ВС;

**-n** — вывод имени узла сети;

**-p** — вывод типа процессора;

**-r** — вывод главного номера версии ОС;

**-s** — вывод названия ОС;

**-v** — вывод дополнительного номера версии ОС.

4. Выдача справочной информации о пользовательском и программном интерфейсах:

**man** имя

где *имя* — имя одной из системных программ или подпрограмм, используемое в пользовательских и программных интерфейсах. Сюда относятся имена системных обрабатывающих программ (утилит и лингвистических процессоров), имена системных программных вызовов, а также имена библиотечных функций. Задав имя интересующей вас системной программы, вы можете получить подробные сведения об ее использовании (правда, на английском языке). Например, можно спросить утилиту *man* о ней самой.

## 8. Упрощение пользовательского интерфейса

Эту функцию выполняют достаточно сложные утилиты, в названии которых часто присутствует слово **commander**. Примером такой утилиты для *MS-DOS* является *Norton Commander*. Аналогичная утилита для *UNIX* называется *Midnight Commander*. (Для того чтобы запустить *Midnight Commander*, достаточно набрать команду *UNIX — mc*.)

Любая подобная программа предназначена для того, чтобы предоставить пользователю ВС удобный интерфейс для общения с этой системой. Это обеспечивается, во-первых, наглядным выводом на экран информации о файловой структуре системы. Для этого по запросу пользователя утилита переносит с диска на экран информацию, содержащуюся в любом каталоге файловой структуры системы. Во-вторых, любой *commander* существенно упрощает для пользователя ввод команд ОС за счет того, что он переносит имя исполняемого файла программы из позиции экрана, отмеченной пользователем с помощью псевдокурсора (**псевдо-курсор** — светящийся прямоугольник, получаемый, в отличие от обычного курсора, не аппаратно, а программно), в то место памяти, откуда это имя может взять интерпретатор команд ОС.

В отличие от лингвистических процессоров, утилиты используются не только программистами, но и **пользователями-непрограммистами**. Эта наиболее многочисленная категория пользователей ВС работает на виртуальных машинах, предоставляемых готовыми прикладными программами, а также утилитами. Что касается программистов, то они просто вынуждены использовать наряду с утилитами еще и лингвистические процессоры. Напомним, что целью применения любой ВС является выполнение прикладных машинных программ. В следующем разделе рассмотрим применение лингвистических процессоров для получения таких программ.

## 1.4. Трансляторы

**Программисты** — не самая многочисленная, но очень важная часть пользователей ВС. Конечной задачей любого программирования является получение реальной программы, записанной на машинном языке. Только такая программа может быть понята и выполнена центральным процессором. К сожалению, трудоемкость программирования на машинном языке очень велика и

не позволяет записывать на нем сколько-нибудь сложные (по решаемым задачам) программы. Решением данной проблемы является предоставление программисту возможности разрабатывать не реальную, а виртуальную прикладную программу.

**Виртуальная прикладная программа** записывается на языке программирования, отличном от языка машинных команд. Преобразование этой программы в реальную программу выполняет системная обрабатывающая программа, называемая **лингвистическим процессором**. Эта программа (не путать с аппаратным процессором) выполняет перевод описания алгоритма с одного языка на другой. Сущность алгоритма при этом сохраняется, но форма его представления, ориентированная на программиста, преобразуется в форму, ориентированную на ЦП. Лингвистические процессоры делятся на трансляторы и интерпретаторы. В результате работы **транслятора** алгоритм, записанный на языке программирования (исходная виртуальная программа), преобразуется в алгоритм, записанный на машинном языке. (На самом деле, как будет показано позже, машинная программа является результатом совместной работы нескольких лингвистических процессоров.)

**Интерпретатор**, в отличие от транслятора, не выдает машинную программу целиком. Выполнив перевод очередного оператора исходной программы в соответствующую совокупность машинных команд, интерпретатор обеспечивает их выполнение. Затем преобразуется тот исходный оператор, который должен выполняться следующим по логике алгоритма и т.д. Интерпретаторы будут рассматриваться нами в следующем разделе, а сейчас обратимся к трансляторам.

В качестве примера рассмотрим преобразование виртуальной программы на языке *СИ* в исполняемый файл для *UNIX*-системы. Общая схема такого преобразования приведена на рис. 8. На этой схеме указанное преобразование выполняет цепочка из пяти трансляторов: препроцессор, компилятор, оптимизатор, ассемблер и редактор связей. Цепочка из этих последовательно выполняемых трансляторов также является транслятором, выполняющим преобразование совокупности исходных модулей программы в соответствующий загрузочный модуль.

**Исходный модуль программы** — текстовый файл, содержащий всю виртуальную программу или ее часть. Если речь идет о программе на *СИ*, то данный файл имеет имя с суффиксом «.с». Любой

исходный модуль состоит из операторов двух типов — псевдооператоров и исполнительных операторов.

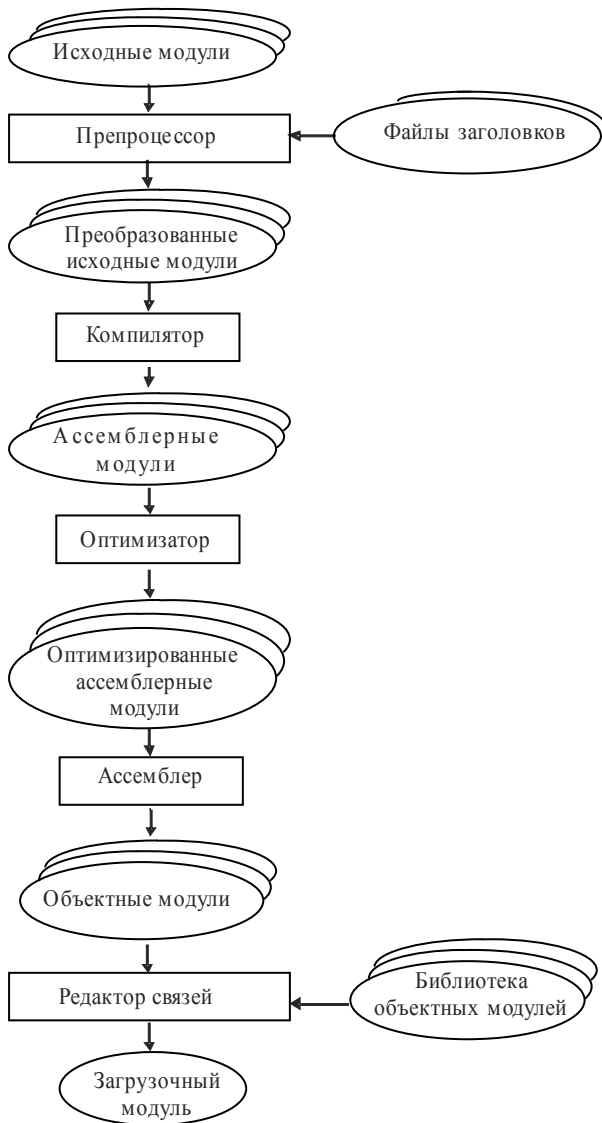


Рис. 8. Преобразование исходной программы в загрузочный модуль



**Исполнительный оператор** — оператор исходной программы, преобразуемый в результате трансляции в машинные команды. При этом исполнительный оператор языка высокого уровня, например языка *СИ*, преобразуется в несколько машинных команд. **Псевдооператор** — оператор исходной программы, представляющий собой указание транслятору. В машинные команды этот оператор не транслируется.

**Препроцессор** — транслятор, который выполняет обработку исходных модулей программы, подсоединяя к ним содержимое файлов заголовков и выполняя подстановки, заданные в этих файлах. **Файл заголовков** — текстовый файл с суффиксом «*.h*», заданный в исходном модуле программы в качестве параметра псевдооператора **#include**. Если этот файл находится в одном из специально предназначенных для этого каталогов */usr/include* или */usr/include/sys*, то имя файла заключается в угловые скобки. Иначе — абсолютное или относительное имя файла помещается в кавычки. Каждая строка файла заголовков содержит или прототип системной функции, или определение константы, или определение структуры данных. В то время как определения констант и структур данных используются препроцессором для замены символьных имен констант и структур данных их значениями, прототипы системных функций используются для проверки правильности вызова этих функций в программе.

**Функцией** при программировании на *СИ* называется любая подпрограмма, в том числе и системная. **Системная функция** — подпрограмма, объектный модуль которой находится в одной из системных библиотек. **Прототип функции** — правильно записанный ее виртуальный интерфейс, с указанием типов всех ее параметров, включая значение, возвращаемое функцией. Выполняя сравнение прототипов системных функций с их фактическими вызовами в исходных модулях программы, транслятор может оказать помощь программисту в обнаружении некоторых типов ошибок: неправильно заданное число параметров или неправильно определенные их типы.

**Компилятор** — транслятор, выполняющий преобразование текста программы на языке высокого уровня в программу на языке низкого уровня. Один оператор языка низкого уровня соответствует одной машинной команде. На рассматриваемой схеме (см. рис. 8) языком низкого уровня является ассемблер.

**Ассемблер** — язык низкого уровня, пригодный для написания виртуальных программ. Трудоемкость разработки таких программ весьма велика. Поэтому на ассемблере пишутся исходные тексты программ только в двух случаях: 1) программа выполняет непосредственное управление аппаратурой ВС; 2) если предъявляются повышенные требования к эффективности программы. Поэтому программированием на ассемблере для мультипрограммных систем, например для *Windows* или *UNIX*, занимается лишь небольшая часть системных программистов. Это не умаляет роль ассемблера для программирования в однопрограммных системах, а также в качестве учебного языка.

На рис. 8 ассемблер используется не в качестве языка программирования, а в качестве промежуточного языка. Получая программу на ассемблере, компилятор заменяет псевдооператоры и исполнительные операторы *СИ*, соответственно, на псевдооператоры и исполнительные операторы ассемблера. При этом многократно сокращается число типов данных и широко используется внутрипроцессорная память — регистры.

В отличие от программиста, компилятор «программирует» на ассемблере не очень эффективно. При этом под **эффективностью программы** понимаются два критерия: затраты времени ЦП на выполнение программы и затраты ОП для ее размещения. Для улучшения оценок программы по этим двум критериям ее ассемблерные модули, выданные компилятором, подаются на вход следующего транслятора — **оптимизатора**.

Оптимизированные ассемблерные модули являются входными данными для **транслятора-ассемблера**. В результате одного выполнения этого транслятора ассемблерный модуль преобразуется в **объектный программный модуль**, записываемый в файл с суффиксом «*.о*». При этом частично решается задача получения последовательности машинных команд, отображающих алгоритм модуля. Транслятор окончательно записывает коды операций машинных команд, а также проставляет в эти команды номера используемых регистров. Что касается символьных имен, то они обрабатываются транслятором-ассемблером по-разному.

Рассмотрим преобразование адресов транслятором. **Адрес** — место в ОП, которое займет соответствующий программный объект — команда или данное. Любой язык программирования (включая *СИ* и ассемблер) позволяет программисту использовать в программе не адреса, а их заменители — **символьные имена (метки)**.

Основные типы меток: 1) метки операторов; 2) имена переменных; 3) имена подпрограмм (имя подпрограммы заменяет для программиста адрес, по которому первая команда подпрограммы находится в ОП).

Все символьные имена в исходном (в том числе ассемблерном) модуле делятся на внешние и внутренние. Символьное имя называется **внутренним**, если выполняются два условия: 1) соответствующий программный объект (оператор, данное или процедура) находится в этом же модуле; 2) данный программный объект используется (вызывается) только внутри данного исходного модуля.

Все внешние метки делятся на входные и выходные. **Внешние выходные метки** определены внутри данного исходного модуля, а используются вне него. Это: 1) имена процедур, входящих в состав данного исходного модуля, но которые могут вызываться в других исходных модулях; 2) имена переменных, которые определены в данном исходном модуле, а используются вне него. **Внешние входные метки** определены вне данного исходного модуля, а используются в нем. Это: 1) имена процедур, не входящих в данный исходный модуль, но используемых в нем; 2) имена переменных, которые используются в исходном модуле, но определены вне него.

При получении объектного модуля транслятор-ассемблер представляет в машинные команды вместо внутренних меток и внешних выходных меток или смещение относительно текущего содержимого указателя команд (это специальный регистр ЦП), или смещение относительно начала сегмента ОП, в котором находится соответствующий программный объект. Что касается внешних входных меток, то обработать их транслятор-ассемблер не может. Он ничего не знает о размещении соответствующих программных объектов в памяти, так как имеет в своем распоряжении единственный ассемблерный модуль, в котором этих объектов нет. Дальнейшее преобразование программы выполняет системная программа, называемая редактором связей.

**Редактор связей (компоновщик)** связывает («сшивает») все объектные модули программы в единый **загрузочный модуль**. Кроме того, редактор связей объединяет с программой системные объектные модули, находящиеся в библиотечных файлах с суффиксом **«.a»**.

При получении загрузочного модуля редактор связей записывает в ОП один за другим объектные модули. Поэтому он «знает», где расположен в памяти каждый программный объект. Следовательно, он может заменить все оставленные транслятором-ассемблером внешние метки на соответствующие численные адреса. В конце своей работы редактор связей записывает загрузочный модуль в файл, называемый **исполняемым файлом**. По умолчанию этот файл помещается в текущий каталог и имеет простое имя *a.out*.

Рассмотренные выше трансляторы обычно реализуются в качестве подпрограмм более крупной программы, называемой **системой программирования**. Кроме них в систему программирования входит также подпрограмма-оболочка, выполняющая диалог с пользователем-программистом, а также утилиты: 1) текстовый редактор, предназначенный для набора текстов исходных модулей; 2) отладчик, позволяющий выполнять пошаговое выполнение программы с целью обнаружения в ней ошибок.

Например, существует большое количество систем программирования, предназначенных для поддержки программирования на языках *СИ* и *СИ++* в *UNIX*-системах. Некоторые из них: *сс*, *сpp*, *gсс*, *с++*, *g++*. Команды для запуска этих систем программирования похожи на команды для запуска утилит (см. подразд. 1.3). Точно так же в качестве параметров команды задаются обрабатываемые файлы, а ее функции уточняются с помощью флагов. Вот некоторые флаги для программы *сс*:

1) *-o* — требуется дать исполняемому файлу программы имя, отличное от *a.out*;

2) *-с* — требуется получить не загрузочный, а объектный модуль;

3) *-I имя* — при получении загрузочного модуля использовать требуемую библиотеку объектных модулей.

Благодаря наличию системы программирования программист работает на **виртуальной машине пользователя системы программирования** (рис. 9). Эта ВМ «понимает» операторы используемого языка программирования, а также команды управления работой системы программирования. Предоставляя программисту возможность работать с виртуальной машиной, сама система программирования «выполняется» на ВМ, аналогичной той, на которой выполняется прикладная программа (*ВМ\_ПП*). Это обусловлено тем, что и прикладная программа, и система про-

граммирования являются машинными программами. Более того, с точки зрения самой ВС, между ними нет принципиальной разницы, так как и та, и другая программа относятся к классу обрабатываемых программ.

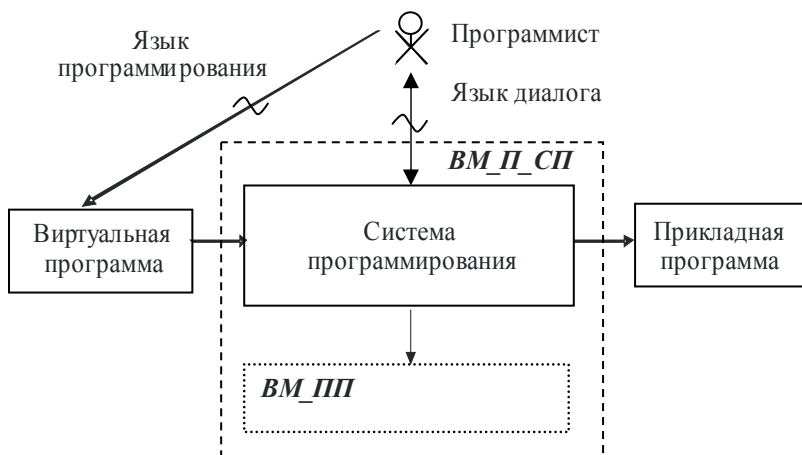


Рис. 9. Виртуальная машина пользователя системы программирования:  
*ВМ\_П\_СП* — виртуальная машина пользователя системы программирования;  
*ВМ\_ПП* — виртуальная машина прикладной программы

Строго говоря, исполняемый файл (загрузочный модуль) и машинная программа — не одно и то же. Для того чтобы загрузочный модуль стал машинной программой, необходимо выполнить операции загрузки и динамического связывания. Так как эти операции обычно скрыты от пользователя, мы их рассмотрим позже. А сейчас перейдем к рассмотрению программы, позволяющей пользователю ВС запускать на выполнение прикладные и системные обрабатываемые программы, зная лишь имя соответствующего исполняемого файла. Речь идет об интерпретаторе команд ОС.

## 1.5. Язык управления операционной системой

### 1.5.1. Типы языков управления

Любая операционная система предоставляет своему пользователю (пользователям) возможность управлять своей работой. Поэтому язык управления ОС является обязательной частью интерфейса между пользователем и ВС. Существуют два основных типа таких языков.

Первый тип языка управления ОС ориентирован на работу системы с неподготовленным пользователем и заключается в использовании **меню**: в любой момент времени пользователь видит на экране набор доступных команд, из которых он должен сделать выбор. Такой подход реализован в различных *WINDOWS*. В этих системах используется графическое меню: на экране представлены значки, соответствующие исполняемым файлам, файлам данных, а также каталогам (папкам). Пользователь сообщает о своем выборе в «меню», наведя курсор мыши, а затем нажав на ее клавишу. При выборе исполняемого файла (расширение имени файла — *com*, *exe* или *bat*) ОС запускает на выполнение соответствующую программу или программы (для *bat*-файла). Выбор файла данных означает, что на исполнение должна быть запущена системная утилита, выполняющая обработку данного файла. Выбор каталога приводит к выводу на экран меню, состоящего из файлов и подкаталогов этого каталога.

Второй тип языков управления ОС — **языки команд**. Каждый такой язык ориентирован на подготовленного пользователя, знакомого с языком команд. Набрав на клавиатуре свою команду, пользователь нажимает клавишу *<Enter>*, сообщая тем самым системе, что она может приступить к выполнению команды. Такой подход используется в операционных системах *MS-DOS* и *UNIX*.

Любой из подходов к организации пользовательского интерфейса предполагает, что обработку команд управления ОС выполняет ее модуль, называемый **интерпретатором команд ОС** (сокращенно **ИК**). Как и любой интерпретатор, данная программа выполняет обработку поступающих на ее вход команд по одной, запуская на выполнение требуемую машинную программу или подпрограмму. Являясь для пользователя частью ОС, ИК рассматривается основной частью этой системы (ядром ОС) как обычная обрабатываемая программа. Следствием этого является то, что ИК размещается в отдельном исполняемом файле. Для *MS-DOS* это *command.com*, а в любой *UNIX*-системе существует несколько взаимозаменяемых ИК. Наиболее известные из них: *Bourne shell* — файл */bin/sh*, *C shell* — */bin/csh*, *Korn shell* — */bin/ksh*, *Bourne-Again*

*shell* — */bin/bash*. Все эти ИК имеют общее название — *shell*. В качестве примера далее рассматривается язык команд для наиболее типичного *shell* — *Bourne shell*.

После входа пользователя в систему и запуска первоначального *shell* (эти операции будут рассмотрены в подразд. 3.1) на экран выводится приглашение ввести следующую команду. Часто в качестве такого приглашения используется символ «\$». В ответ пользователь набирает команду одного из следующих типов:

- 1) простая команда;
- 2) составная команда;
- 3) вызов подпрограммы на языке *shell*;
- 4) управляющий оператор;
- 4) командный файл.

Пользователи-непрограммисты обычно ограничиваются первыми двумя типами команд, так как применение остальных типов команд фактически означает программирование на языке команд *shell*.

### 1.5.2. Простые команды

Простые команды *shell* делятся на: а) команды запуска программ; б) команды вызова функций *shell*; в) вспомогательные команды. Последняя из перечисленных групп команд будет рассмотрена нами в п. 1.5.4 при описании переменных *shell*.

Первые две группы команд, которые мы будем сейчас рассматривать, различаются по реализации: команда запуска программы требует от *shell* обеспечить выполнение какой-то обрабатывающей программы (прикладной, утилиты или лингвистического процессора), а команда вызова функции *shell* запускает внутреннюю подпрограмму самого *shell*. С точки зрения пользователя ВС эти два типа команд почти не различимы. Единственное различие: команда запуска программы имеет, а команда вызова функции *shell* не имеет кода завершения. **Код завершения** — целое неотрицательное число: 0 — запущенная программа завершилась успешно; > 0 — программа завершилась с ошибкой.

Так как число исполняемых файлов практически не ограничено, то и не ограничено число простых команд запуска программ. По своей форме наиболее распространенная команда *shell* представляет собой имя исполняемого файла прикладной или системной обрабатывающей программы, за которым через разделитель (символ пробела) записаны параметры команды. Пример команды:

### **\$ cat abc.txt**

Эта команда выведет на экран содержимое файла *abc.txt*. Это наиболее простое задание исполняемого файла, но любой ИК, в том числе и *shell*, позволяет задавать дополнительные условия выполнения запускаемой программы, существенно помогающие пользователю ВС в изложении требуемой ему задачи. Рассмотрим способы указания таких условий.

**Использование метасимволов.** Оно позволяет пользователю существенно сократить число набираемых имен файлов. Основные метасимволы:

1) \* — соответствует любой последовательности символов, в том числе и пустой, кроме последовательностей, начинающихся с символа «.»;

2) ? — соответствует любому одиночному символу;

3) [. . .] — соответствует любому одиночному символу из тех, что перечислены без разделяющих символов в квадратных скобках. Пара символов, разделенных символом «-», соответствует одиночному символу, код которого попадает в диапазон между кодами указанных символов, включая их самих.

Для самих запускаемых программ метасимволы *shell* «не заметны», так как *shell* подставляет вместо имен, использующих их, обычные имена файлов. Например, пусть пользователь набрал последовательность команд (напомним, что утилита *ls* без параметров выводит на экран содержимое текущего каталога):

```
$ ls
client client.c server.c
$ cc *.c
```

Тогда действительный вызов транслятора *cc*, выполняемый *shell*, имеет вид *cc client.c server.c*.

**Перенаправление ввода-вывода.** Оно позволяет пользователю в удобной форме выполнить замену файлов, используемых в качестве стандартного ввода и стандартного вывода запускаемой программы. Напомним, что по умолчанию стандартным вводом является клавиатура, а стандартным выводом — экран. По умолчанию экран используется также в качестве второго выходного файла, в который выводятся сообщения об ошибках. Перечислим операции перенаправления ввода-вывода:

1) > *файл* — программа выполняет вывод данных не на экран, а в заданный файл, начиная с его начала. Если файл с таким именем



уже существует, то его прежнее содержимое будет уничтожено. Если файл не существует, то он будет создан;

2) *>> файл* — программа добавляет свои выходные данные в конец существующего файла. Если файла не было, то он создается;

3) *< файл* — программа выполняет ввод данных не с клавиатуры, а из заданного файла;

4) *<< слово* — программа выполняет ввод данных с клавиатуры до тех пор, пока в этих данных не встретится заданное слово или не будет введен символ конца файла (*<Ctrl>&<D>*).

Подобно использованию метасимволов, сама запускаемая программа ничего «не знает» об используемых в команде операциях перенаправления ввода-вывода. Дело в том, что программа обращается к экрану и клавиатуре не по их пользовательским именам (именам соответствующих файлов), а использует для этого программные имена файлов: клавиатура — 0; экран — 1; экран для вывода ошибок — 2. Поэтому обработка операции перенаправления ввода-вывода в ИК заключается в том, что прежде чем будет запущена требуемая программа, ИК откроет под номерами 0, 1, 2 не клавиатуру и экран, а файлы, указанные в пользовательской команде.

В следующих примерах операции перенаправления ввода-вывода демонстрируются на примере утилиты *cat*:

1) *\$ cat > abc.txt*

В этом примере *cat* используется в качестве простейшего текстового редактора, который позволяет вводить текст, строка за строкой, с клавиатуры в файл *abc.txt*, начиная с его начала. Каждая введенная строка может быть сразу же отредактирована. Ввод символов заканчивается символом конца файла (*<Ctrl>&<D>*);

2) *\$ cat >> abc.txt*

Отличие этого примера от предыдущего в том, что вводимые с клавиатуры символы добавляются в конец файла *abc.txt*;

3) *\$ cat < abc.txt*

Эта команда выводит на экран содержимое файла *abc.txt*. Точно такого же эффекта можно достичь и командой *\$ cat abc.txt*. Разница в том, что в первом случае запускаемая утилита *cat* не получает никаких параметров, а во втором случае таким параметром является имя файла *abc.txt*;

4) *\$ cat <<! > abc.txt*

Ввод с клавиатуры помещается в файл *abc.txt* до тех пор, пока не будет введен символ «!». В этом примере и в следующем используются сразу две операции перенаправления ввода-вывода;

5) `$ cat <xy.txt > abc.txt`

Эта команда выполняет копирование файла *xy.txt* в файл *abc.txt*. То есть эта команда является некоторым аналогом команды *cp*.

Так как системные утилиты выводят свои сообщения об ошибках на экран, то эти сообщения иногда мешают восприятию с экрана другой информации и тем самым раздражают пользователя. Для подавления таких сообщений их перенаправляют с экрана в другой файл, например в файл с именем */dev/null*. Этот файл соответствует псевдоустройству, вывод в который означает уничтожение выводимой информации. Сама операция перенаправления сообщений об ошибках аналогична перенаправлению стандартного вывода с тем лишь отличием, что слева от операции «>» или «>>>» записывается цифра «2» — программное имя файла, предназначенного для вывода ошибок. Например, следующие две команды выводят на экран содержимое всех текстовых файлов, содержащихся в текущем каталоге:

а) `cat ./*`

б) `cat ./* 2>/dev/null`

Программа *cat*, запущенная первой из этих команд, выводит на экран свое «ругательство» по поводу каждого нетекстового файла или подкаталога. Второй запуск этой программы выводит на экран лишь содержимое текстовых файлов.

**Запуск исполняемого файла в фоновом режиме.** Если запускаемая программа использует клавиатуру и (или) экран, то она относится к запускающему ее ИК логически так же, как относится подпрограмма к запускающей ее программе. То есть так как ИК не может выполняться без экрана и клавиатуры, которые существуют для конкретного пользователя в единственном экземпляре, то до завершения запущенной программы ИК будет «без движения». При этом говорят, что программа запускается в *оперативном режиме*.

Если программе не нужны ни клавиатура, ни экран, то ее можно запустить в фоновом режиме. Это означает, что после запуска программы она и ИК выполняются асинхронно (независимо). (На самом деле, как будет показано в следующих разделах, ИК может выполнять некоторые действия по управлению запу-

щенной программой, но эти действия не являются обязательными и зависят от желания пользователя.)

Для запуска исполняемого файла в фоновом режиме достаточно в конце команды записать символ «&». Например, следующая команда выполняет в фоновом режиме копирование подде-рева файловой структуры с корнем *dir1* в поддерево с корнем *dir2*:

```
$ cp -r dir1 dir2 &
```

### 1.5.3. Составные команды

В отличие от простой команды, **составная команда** позволяет пользователю запустить не один, а несколько исполняемых файлов. Такая команда представляет собой или конвейер, или командный список, или многоуровневую команду.

**Конвейер программ.** Имена исполняемых файлов, образующих конвейер, разделяются символом «|». Стандартный вывод программы, стоящей слева от этого символа, одновременно является стандартным вводом для программы, записанной справа. Пример конвейера:

```
$ find dir1 -name a1 | cat > file
```

В этом примере утилита *find* выводит список файлов с простым именем *a1*, находящихся в поддереве файловой структуры с корнем *dir1* (это подкаталог текущего каталога). Причем вывод осуществляется не на экран, а в файл на диске, из которого утилита *cat* переписывает имена файлов в файл *file*.

Программы, образующие конвейер, не конкурируют между собой из-за экрана и клавиатуры, так как клавиатура может быть нужна только первой, а экран — только последней программе конвейера. Поэтому данные программы запускаются *shell* одновременно (асинхронно). После своего запуска программы, расположенные по соседству в конвейере, взаимодействуют между собой через промежуточный файл следующим образом. Программа, для которой этот файл является выходным, помещает в него свои данные построчно (в приведенном примере каждая строка содержит имя файла). Другая программа считывает эти данные также построчно, не дожидаясь завершения работы первой программы. Заметим, что несмотря на то что промежуточный файл реально

существует на диске, его имя неизвестно для пользователя, которому, впрочем, это имя и не нужно.

Для того чтобы сохранить промежуточный файл, скопировав его в другой файл, используется команда *tee*, помещаемая в то место конвейера, где находится промежуточный файл. Если сравнить конвейер с водопроводной трубой, то эта команда играет роль «тройника» (рис. 10). Переделаем предыдущий пример так, чтобы, по-прежнему сохраняя результаты поиска в файле *file*, обеспечить их вывод на экран:

```
$ find dir1 -name a1 | tee file | cat
```

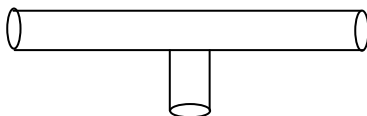


Рис. 10. Наглядное изображение команды *tee*

Если первой и последней командам конвейера не нужны, соответственно, клавиатура и экран (например, благодаря перенаправлению ввода-вывода), то все программы конвейера могут быть запущены в фоновом режиме записью символа «&» в конце конвейера.

Подобно тому как команда запуска программы имеет код завершения, подобный код имеет и конвейер. При этом **код завершения конвейера** определяется кодом завершения программы, записанной в конвейере последней.

**Командные списки.** Такой список образуют конвейеры, разделенные символами «;», «&», «&&», «||». При этом в качестве конвейеров могут выступать и отдельные исполняемые файлы. Рассмотрим назначение перечисленных символов:

;  
— элементы списка, соединяемые этим символом, запускаются последовательно. То есть программа (конвейер) справа от символа «;» начинает выполняться только после завершения программы (конвейера) слева. При этом программа слева выполняется в оперативном режиме и поэтому может использовать экран и клавиатуру;

&  
— элементы списка, соединяемые этим символом, запускаются асинхронно. При этом программа или конвейер слева от этого символа запускается в фоновом режиме;

**&&** — элементы списка, соединяемые этим символом, запускаются последовательно. При этом конвейер справа будет запущен только в том случае, если конвейер слева завершился успешно — с нулевым кодом завершения;

**||** — в отличие от предыдущего случая для запуска конвейера справа требуется неудачное завершение конвейера слева (завершение с ненулевым кодом завершения).

Примеры командных списков:

- 1) `$ mkdir dir1; cd dir1; pwd`  
`/home/vlad/dir1`
- 2) `$ cp -r dir2 dir3 & cat > file1`
- 3) `$ mkdir dir1 && cd dir1`
- 4) `$ mkdir dir1 || echo "Ошибка создания каталога dir1"`

**Многоуровневая команда.** Такая команда представляет собой текст одной команды, в которую должны быть подставлены результаты выполнения другой команды или команд. При этом результат вкладываемой команды представляет собой одну или несколько текстовых строк, отображаемых в стандартный вывод. Примеры таких команд: *pwd*, *wc*, *ls*, *find*. Каждая вкладываемая команда должна быть выделена одним из двух способов: а) заключена в обратные апострофы «```»; б) заключена в круглые скобки, которым предшествует символ «`$`». Первый из этих способов применяется, в основном, для двухуровневых команд, а второй — для любого числа уровней вложенности.

*Shell* обрабатывает многоуровневую команду (как и любую другую) слева направо. При этом, встретив очередную вложенную команду, *shell* обеспечивает ее выполнение, а затем подставляет текст, полученный в результате этого выполнения, в командную строку. Если данный текст состоит из нескольких строк, то *shell* заменяет каждую пару символов «возврат каретки» и «перевод строки», разделяющих соседние строки, на символ пробела. В результате этого вставляемый текст представляет собой одну большую строку.

### Пример

Следующая двухуровневая команда выполняет уничтожение всех файлов и каталогов, простые имена которых начинаются с буквы *d* и которые расположены в поддереве файловой структуры, принадлежащем данному пользователю:

```
$ rm -r `find ~/ -name 'd*'`
```

Обратите внимание, что при задании в команде *find* имени файла (или каталога) с помощью метасимволов, это имя обязательно должно быть заключено в кавычки (одиночные или двойные). Это объясняется тем, что *shell* имеет свои метасимволы, одноименные метасимволам утилиты *find*. Кавычки играют роль «экранирующих» символов, сообщая *shell* о том, что все символы, заключенные между ними, являются обычными символами, которые должны быть переданы без изменений в запускаемую программу (в данном случае в программу *find*).

Полезно сравнить действие записанной двухуровневой команды с командой `$ rm -r ~/d*`. Последняя команда выполнит уничтожение не всех файлов и каталогов с заданным именем в поддереве пользователя, а лишь тех, для которых родительским каталогом является корень этого поддерева.

### Пример

Следующая команда имеет 4-уровневую структуру:

```
$ echo 111$(echo 222$(echo 333$(echo 444)))  
111222333444
```

## 1.5.4. Переменные и выражения

Как и любой язык программирования, входной язык ИК позволяет задавать переменные. При этом под **переменной** понимается небольшая область ОП, содержащая данное, которое может быть использовано при выполнении команд *shell*. В отличие от других языков программирования переменные *shell* не нуждаются в объявлении типа, так как все они содержат данные одного типа — символьные строки.

Имя переменной может включать символы: латинские буквы, цифры, «\_». При этом имя не может начинаться с цифры. Как и всегда, для задания значения переменной используется оператор присваивания. В *shell* это символ «=». Имя присваиваемой переменной помещается слева от этого символа, а справа — новое значение переменной. Для задания этого значения могут быть использованы следующие способы.

**Непосредственное задание строки символов.** Если эта строка состоит из одного слова, то ее можно не выделять. Для задания строки из нескольких слов обязательны двойные кавычки.

### Примеры

а) `$ var1=/home/vlad/a.txt`

б) `$ var2="/home/vlad/a.txt"`

В результате выполнения этих операторов переменные *var1* и *var2* имеют одинаковое значение.

**Задание значения другой переменной.** Для этого перед именем переменной в правой части оператора присваивания должен быть помещен символ «\$».

**Пример**

```
$ var1=/home/vlad/a.txt
```

```
$ var2=$var1
```

```
$ echo $var2
```

```
/home/vlad/a.txt
```

В этом примере значение переменной *var1* используется для задания значения переменной *var2*, а значение *var2* используется в качестве параметра команды *echo*.

**Использование выходных данных команды *shell*.** Оно выполняется точно так же, как и в многоуровневой команде (см. п. 1.5.3). При этом имя любой программы, которая выдает свой результат в виде одной или нескольких строк символов, может быть записано справа от оператора присваивания, заключенным в обратные апострофы «'», или может быть заключено в круглые скобки, которым предшествует символ «\$». *shell* сначала запускает указанную программу на выполнение, а затем подставляет ее выходные данные в качестве значения заданной переменной.

**Пример**

```
$ var=`pwd`
```

В данном примере *shell* сначала запустит на выполнение утилиту *pwd*, которая выдаст имя текущего каталога, затем подставит это имя в качестве значения переменной *var*.

**Использование команды ввода *read*.** Оно позволяет ввести значения переменных со стандартного ввода (с клавиатуры), не используя оператор присваивания. Для этого имена определяемых переменных должны быть перечислены в качестве параметров этой команды. Вводимые далее с клавиатуры (до нажатия <Enter>) слова распределяются между переменными так, что в одну переменную записывается одно слово. Если число переменных меньше числа слов во введенной строке, то все оставшиеся слова записываются в последнюю переменную. Если, наоборот, число переменных больше, то последние переменные получают пустое значение. Если с клавиатуры вместо обычных символов будет введен символ кон-

ца файла (`<Ctrl>&<D>`), то данная команда завершится с ненулевым кодом завершения.

Команду `read` удобно использовать для того, чтобы присваивать переменным значения слов из некоторого текстового файла. Для этого достаточно перенаправить стандартный ввод с клавиатуры на ввод из требуемого файла. Например, следующая команда присваивает переменным `x`, `y`, `z` значения слов из первой строки файла `file`:

```
$ read x y z <file
```

Таким образом, для того чтобы определить обыкновенную переменную, достаточно хотя бы раз записать ее имя слева от оператора присваивания или записать его в качестве параметра команды `read`. После этого до конца вашей работы с данным `shell` значение данной переменной может использоваться в любом месте любой команды. Для этого имени переменной должен предшествовать символ «\$», смысл которого в данном случае есть «значение переменной». Выше приведены примеры использования значения переменной в правой части оператора присваивания, а также в качестве параметра команды `echo`.

Если имя переменной следует отделить от символов, записанных сразу за ним, то это имя следует заключить в символы «{» и «}».

### Пример

```
$ var = abc
$ echo $varxy
$ echo ${var}xy
abcxy
```

В данном примере в качестве параметра первого оператора `echo` записано значение неопределенной ранее переменной `varxy`. В подобных случаях `shell` подставляет вместо неопределенной переменной пустое место. Вторым оператором `echo` получает в качестве своего параметра значение переменной `var` (символы `abc`), к которому «подсоединены» символы `xy`.

Используя утилиту `set` (без параметров), можно вывести на экран перечень переменных, определенных в данном сеансе работы с `shell`, а также их значения. В состав данных переменных входят не только обыкновенные переменные, заданные операторами присваивания, но и **переменные окружения** — переменные, которые `shell` «наследует» от программы, запустившей его. (В гл. 2 будет



показано, что любая обрабатывающая программа имеет «родительскую программу».) Рассмотрим некоторые переменные окружения:

1) **HOME** — содержит полное имя корневого каталога пользователя;

2) **PATH** — содержит перечень абсолютных имен каталогов, в которых *shell* выполняет поиск исполняемого файла в том случае, если для его задания в своей команде пользователь использовал простое имя файла. Имена-пути каталогов, записанные в данной переменной, разделяются символом «:». Следует отметить, что *shell* не производит поиск исполняемого файла в текущем каталоге по умолчанию, и для этого требуется явное задание текущего каталога в переменной **PATH** (для этого используются символы «./»). Кроме того, заметим, что *shell* не использует данную переменную для поиска неисполняемых, например текстовых, файлов. Поэтому для задания таких файлов в команде пользователя требуется или использовать их абсолютные имена, или обеспечить перед выполнением такой команды переход в родительский каталог файла;

3) **PS1** — приглашение *shell* в командной строке (обычно — \$);

4) **PS2** — вторичное приглашение *shell* в командной строке (обычно — >). Выводится на экран в том случае, когда вводимая команда пользователя занимает более одной строки.

Используя оператор присваивания, можно заменить содержимое переменной окружения подобно тому, как это делается для обыкновенных переменных. Например, следующая команда добавляет в переменную **PATH** имя текущего каталога:

```
$ PATH=${PATH}":./"
```

Изменение переменной окружения приведет к тому, что не только текущая программа *shell*, но и все запущенные ею программы будут использовать новое значение данной переменной. В то же время программы, являющиеся «предками» данного *shell*, по-прежнему будут работать с ее прежним значением.

Обыкновенные переменные можно добавить к переменным окружения, сделав их «наследуемыми». Для этого достаточно перечислить имена этих переменных в качестве параметров команды **export**.

Подобно другим языкам программирования, входной язык *shell* позволяет записывать выражения. **Выражение** — совокупность нескольких переменных и (или) констант, соединенных знаками операций. Различают арифметические и логические выражения.

**Арифметические выражения** имеют при программировании для *shell* весьма ограниченное значение. Вспомним, что данный язык даже не имеет арифметических типов данных. Если все-таки требуется выполнить над переменными *shell* арифметические действия, то для этого следует использовать команду (функцию *shell*) *expr*.

### Примеры

а) `$ expr 5 + 3`

8

б) `$ expr 5 "*" 3`

15

в) `$ x=10`

`$ expr $x + 1`

11

г) `$ x=20`

`$ x=`expr $x + 2``

`$ echo $x`

22

Заключение в кавычки знака умножения обусловлено тем, что данный символ является для *shell* метасимволом, и поэтому для устранения его специальных свойств он должен быть «экранирован». Обратите внимание на то, что знаки арифметических операций должны быть окружены пробелами.

**Логическое выражение** — выражение, имеющее только два значения — 0 (истина) и 1 (ложь). Операндами логического выражения являются **операции отношения**, каждая из которых или проверяет отношение файла (или строки символов) к заданному свойству, или сравнивает между собой два заданных числа (или две строки символов). Если проверяемое отношение выполняется, то результатом операции отношения является 0, иначе — 1. Перечислим некоторые из операций отношения:

<b>-s file</b>	размер файла <i>file</i> больше 0;
<b>-r file</b>	для файла <i>file</i> разрешен доступ на чтение;
<b>-w file</b>	для файла <i>file</i> разрешен доступ на запись;
<b>-x file</b>	для файла <i>file</i> разрешено выполнение;
<b>-f file</b>	файл <i>file</i> существует и является обычным файлом;
<b>-d file</b>	файл <i>file</i> существует и является каталогом;
<b>-z string</b>	строка <i>string</i> имеет нулевую длину;

<b>-n</b> <i>string</i>	строка <i>string</i> имеет ненулевую длину;
<i>string1</i> = <i>string2</i>	две строки идентичны;
<i>string1</i> != <i>string2</i>	две строки различны;
<i>i1</i> <b>-eq</b> <i>i2</i>	число <i>i1</i> равно числу <i>i2</i> ;
<i>i1</i> <b>-ne</b> <i>i2</i>	число <i>i1</i> не равно числу <i>i2</i> ;
<i>i1</i> <b>-lt</b> <i>i2</i>	число <i>i1</i> строго меньше числа <i>i2</i> ;
<i>i1</i> <b>-le</b> <i>i2</i>	число <i>i1</i> меньше или равно числу <i>i2</i> ;
<i>i1</i> <b>-gt</b> <i>i2</i>	число <i>i1</i> строго больше числа <i>i2</i> ;
<i>i1</i> <b>-ge</b> <i>i2</i>	число <i>i1</i> больше или равно числу <i>i2</i> .

При записи логического выражения отдельные операции отношения могут соединяться друг с другом с помощью логических операций:

**!** — логическое отрицание;

**-a** — логическое И;

**-o** — логическое ИЛИ.

При этом наибольший приоритет имеет операция «!», а наименьший — «-o». Приоритеты операций определяют порядок их выполнения. Порядок выполнения логических операций можно изменить, используя круглые скобки.

Для *shell* характерно то, что запись логического выражения еще не означает его автоматического вычисления. Для этого требуется, чтобы элементы логического выражения были записаны в качестве параметров команды **test**. Результатом выполнения этой команды является код завершения: 0 — логическое выражение истинно, 1 — ложно. Команду *test* можно задать одним из двух способов: 1) обычным способом; 2) заключив логическое выражение в квадратные скобки.

### Примеры

```
$ var1=10; var2=20; var3=30
```

```
1) $ test $var1 -gt $var2 && echo "var1 > var2"
```

```
2) $ [ $var1 -gt $var2 ] || echo "var1 <= var2"
```

```
var1 <= var2
```

```
3) $ test $var1 = $var2 && echo "var1 = var2"
```

```
4) $ [ $var1 = $var2 -o $var2 != $var3 ] && echo "===="
```

```
====
```

```
5) $ test \( $var1 -eq $var2 \) && echo "var1 = var2"
```

```
6) $ [ ! \( $var1 -eq 0 \) ] && echo "var1 не равно 0"
```

```
var1 не равно 0
```

```
7) $ test \( $var1 != $var2 \) -a \( $var1 -eq $var2 \) || echo "????"
```

```
????
```

```
8) $ [ abc ] && echo true
```

*true*

В примерах 1 и 2 операции отношения сравнивают численные значения переменных. В зависимости от этих значений выводятся соответствующие сообщения на экран. В примере 2 команда *test* задается с помощью квадратных скобок, которые обязательно отделяются от логического выражения пробелами.

В примерах 3 и 4 содержимое каждой переменной рассматривается не как число, а как строка символов. Обратите внимание, что операция проверки идентичности строк (=) в отличие от операции присваивания выделяется с обеих сторон пробелами.

В примерах 5, 6 и 7 для выделения логического выражения или его частей используются круглые скобки. При записи каждой круглой скобки должны быть выполнены два требования: 1) непосредственно перед скобкой должен быть помещен символ «\»; 2) перед символом «\» и после скобки обязательно должны быть записаны пробелы. Наличие первого требования обусловлено тем, что круглые скобки рассматриваются интерпретатором *shell* как служебные символы. Для того чтобы эти скобки не выполняли свои служебные функции, а были переданы без изменений в подпрограмму, выполняющую команду *test*, они должны быть «экранированы». Такое «экранирование» и выполняет символ «\». Интересно отметить, что в примере 7 операции отношения (записаны в круглых скобках) всегда дают противоположный результат. При этом в первой из них переменные сравниваются как строки символов, а во второй рассматриваются их численные значения.

Пример 8 иллюстрирует тот факт, что команда *test* выдает значение 0 (истина), если вместо логического выражения записано любое непустое слово.

Во всех приведенных выше примерах команда *test*, выполняющая вычисление логического выражения, записывается в качестве первой команды командного списка, управляя выполнением второй команды этого списка. Другим применением этой команды являются управляющие операторы *shell*.

### 1.5.5. Управляющие операторы

Как и любой алгоритмический язык программирования, входной язык *shell* имеет управляющие операторы. Данные операторы предназначены для того, чтобы задавать порядок выполнения простых и составных команд. Рассмотрим эти управляющие операторы.

**1. Условный оператор *if*** позволяет выполнить одну из нескольких взаимоисключающих последовательностей команд *shell*. Данный оператор имеет несколько форм записи, наиболее простая из которых следующая:

```
if команда-условие
then
    последовательность команд
fi
```

Работа данного оператора начинается с выполнения команды-условия. Это может быть любая простая или составная команда, имеющая код завершения. Но чаще всего в качестве этой команды используют команду *test*, вычисляющую логическое выражение. Если при выполнении данной команды получен нулевой код завершения (напомним, что такой код соответствует успешному завершению программы или значению «истина» логического выражения), то далее выполняется последовательность команд, записанная после ключевого слова *then*. При получении ненулевого кода завершения команды-условия выполнение условного оператора завершается без выполнения каких-либо действий.

Форма оператора *if*, предполагающая выполнение одной из двух последовательностей команд:

```
if команда-условие
then
    последовательность команд 1
else
    последовательность команд 2
fi
```

### Пример

Допустим, что переменная *dir* содержит простое имя каталога. Тогда следующая совокупность команд выполняет уничтожение каталога в том случае, если он пуст; в противном случае выполняется его переименование добавлением к простому прежнему имени символа «a»:

```
$ ls $dir >fil1
$ if [-s fil1 ]
> then
>   mv $dir a$dir
> else
>   rmdir $dir
> fi
```

В данном примере содержимое заданного каталога помещается во вспомогательный файл *fill*. Если длина этого файла ненулевая, то каталог переименовывается, иначе — уничтожается.

Наиболее общая структура условного оператора:

```

if команда-условие 1
then
    последовательность команд 1
elif команда-условие 2
    последовательность команд 2
elif
    .....
else
    последовательность команд N
fi

```

Если был получен ненулевой код завершения команды-условия 1, то далее выполняется команда-условие 2. В случае успешного ее завершения выполняется последовательность команд 2. Выполнение команд-условий продолжается до тех пор, пока очередная такая команда не даст нулевой код завершения. В случае выполнения с ненулевым кодом завершения последней команды-условия выполняется последовательность команд, расположенная после ключевого слова *else*.

В отличие от распространенных языков программирования, условный оператор *if* для *shell* обеспечивает не двух-, а многоальтернативный выбор. Это приближает выразительные возможности данного управляющего оператора к возможностям оператора *case*.

**2. Оператор варианта *case*** позволяет выбрать для выполнения одну из нескольких последовательностей команд. Структура оператора:

```

case слово in
    шаблон1)    последовательность команд 1;;
    шаблон2)    последовательность команд 2;;
    . . . . .
    *)          последовательность команд N;;
esac

```

Здесь *слово* — набор символов без пробелов или с пробелами. Наличие пробелов делает необходимым заключение слова в кавычки. При выполнении оператора *case* слово последовательно сравнивается с шаблонами. **Шаблон** — слово, которое может иметь

наряду с обычными символами метасимволы (*?*, *\**, *[ ]*). Если входное слово удовлетворяет первому шаблону, то выполняется первая последовательность команд, после чего делается выход из оператора *case*. Иначе входное слово сравнивается со вторым шаблоном и так далее. Если при этом обнаружится, что входное слово не отвечает ни одному из шаблонов, то выполняется последовательность команд, которой предшествует шаблон «*\**». Обратите внимание, что любой шаблон отделяется от соответствующей последовательности команд символом «*)*», а каждая последовательность команд заканчивается двумя символами «*;*».

### Пример

В данном примере в качестве входного слова используется имя файла — содержимое переменной *name*. В зависимости от суффикса имени файла этот файл обрабатывается или утилитой *cat* (вывод содержимого файла), или интерпретатором *shell* (выполнение командного файла), или утилитой *wc* (вывод статистики о файле):

```
$ read name
$ case $name in
>     *.txt) cat $name ;;
>     *.sh) sh $name ;;
>     *)    wc $name ;;
> esac
```

### 3. Оператор цикла с перечислением *for*. Его структура:

```
for переменная in список слов
do
    последовательность команд
done
```

Заданная последовательность команд выполняется столько раз, сколько слов в списке. При этом указанная переменная последовательно принимает значения, равные словам из списка. Никакого дополнительного определения переменной, выполняемого перед ее использованием в операторе *for*, не требуется.

### Пример

В данном примере все файлы с суффиксом *txt*, расположенные в поддереве данного пользователя, копируются в каталог *k2*:

```
$ for var in `find ~/ -name "*.txt"`
> do
>     cp $var ~/k2
> done
```

#### 4. Оператор цикла с условием *while*. Его структура:

*while* команда-условие  
*do*  
    последовательность команд  
*done*

Аналогично оператору *if*, условие задается одной из команд *shell*. Пока эта команда возвращает код возврата, равный 0, повторяется последовательность команд, заключенная между словами *do* и *done*. При этом возможна ситуация, когда тело цикла не будет выполнено ни разу.

##### Пример

```
$ while read var1 var2
> do
>     case $var1 in
>         1) echo $var2 >>file1 ;;
>         2) echo $var2 >>file2 ;;
>         *) echo $var2 >>file3 ;;
>     esac
> done
```

Особенностью приведенного примера является использование вложенных управляющих структур: оператор выбора *case* вложен в оператор цикла *while*. Выполнение цикла начинается с выполнения оператора *read*, который вводит строку с клавиатуры, записывая ее первое слово в качестве содержимого переменной *var1*, а все последующие слова — в качестве содержимого *var2*. Допустим, что ввод строки символов завершился успешно и команда *read* выдала код возврата 0. В этом случае в зависимости от значения переменной *var1* (1, 2 или любое другое значение) содержимое введенной строки (за исключением первого слова) записывается в один из трех файлов.

Выполнение данного цикла продолжается до тех пор, пока вместо набора очередной строки вы не наберете комбинацию клавиш *<Ctrl>&<D>*, что означает для файла-клавиатуры «конец файла». В этом случае команда *read* возвратит ненулевой код возврата и выполнение цикла завершится.

Переделаем записанную выше совокупность команд для обработки строк файла *file*. Простое перенаправление стандартного ввода для команды *read* в этом случае не помогает, так как на каждой итерации цикла будет производиться чтение одной и той же первой строки файла. Поэтому запишем конвейер, первая команда которого *cat* будет выполнять чтение строк файла:



```
$ cat file|
> while read var1 var2
> do
> case $var1 in
> 1) echo $var2 >>file1 ;;
> 2) echo $var2 >>file2 ;;
> *) echo $var2 >>file3 ;;
> esac
> done
```

**5. Оператор цикла с инверсным условием *until*.** Его структура:

```
until   команда-условие
do
        последовательность команд
done
```

Команды, заключенные между **do** и **done**, повторяются до тех пор, пока команда-условие не выполнится с кодом завершения 0. Первое же выполнение условия означает выход из цикла. При этом возможна ситуация, когда тело цикла не будет выполнено ни разу. Нетрудно заметить, что операторы *while* и *until* будут выполнять одно и то же, если условие одного из них противоположно условию другого.

### Пример 1

Приведенная ниже последовательность команд выполняет то же самое, что и пример использования *while*, с тем отличием, что завершение ввода определяется не нажатием клавиш <Ctrl>&<D>, а вводом какого-то слова, например слова «!/>

Обратите внимание, что в качестве условия записана команда *test*.

### Пример 2

Запустив в первой половине дня следующую последовательность команд, мы получим на экране напоминание о начале время обеда:

```
$ until date | fgrep 13:30:  
> do  
> sleep 60  
> done && echo "Пора идти обедать" &
```

В данном примере используется командный список, состоящий из операторов *until* и *echo*, соединенных символами «&&». Напомним, что такое соединение обеспечивает запуск второй части командного списка только в случае успешного завершения его первой части. Запись в конце командного списка символа «&» обеспечивает запуск обеих его частей в фоновом режиме.

В качестве условия завершения цикла в операторе *until* записан конвейер команд *date* и *fgrep*. Первая из этих команд передает в свой стандартный вывод текущую дату и время, а команда *fgrep* ищет в этих данных, получаемых в своем стандартном вводе, заданное время (час и минуту). Для того чтобы во время ожидания не занимать бесполезно ЦП, в качестве оператора, повторяемого циклически, записан *sleep*. Этот оператор приостанавливает выполнение программы на указанное в нем число секунд (60), после чего опять проверяется условие завершения цикла. Так как программы, соответствующие перечисленным командам, выполняются в фоновом режиме, то вывод на экран результирующего сообщения «Пора идти обедать» может привести к некоторому искажению выходных данных программ, выполняемых в оперативном режиме.

**6. Операторы завершения цикла *break* и продолжения цикла *continue*.** Общая структура оператора *break*:

***break* число**

Данный оператор завершает выполнение того цикла, в котором он записан. Если в операторе задано число, то делается выход из соответствующего количества циклов, охватывающих оператор *break*. Отсутствие числа в операторе означает завершение одного цикла.

Общая структура оператора *continue*:

***continue* число**

Данный оператор вызывает переход к следующей итерации того цикла, в котором он стоит. Если в операторе задано число, то оно задает относительный номер того цикла, который охватывает

оператор *continue* и который должен продолжаться на своей следующей итерации. Отсутствие числа эквивалентно 1.

Обычное интерактивное взаимодействие пользователя с *shell*, как правило, не требует применения рассмотренных выше управляющих операторов. В самом деле, зачем применять автоматический выбор последовательности выполняемых команд, если пользователь вынужден сам задавать с помощью клавиатуры все возможные варианты выполнения таких команд. Для пользователя намного проще дожидаться завершения предыдущей команды, а затем в зависимости от ее результатов выполнить набор следующей. Областью применения управляющих операторов являются командные файлы.

### 1.5.6. Командные файлы

**Командный файл** — файл, содержащий список команд интерпретатора команд ОС. Применение командных файлов позволяет избежать повторения набора часто используемых команд, и фактически каждый такой файл представляет собой виртуальную программу, записанную на входном языке ИК.

В операционной системе *MS-DOS* командный файл имеет обязательное расширение имени файла — *bat*. В *UNIX* командные файлы называются **скриптами**, и к их имени не предъявляются столь жесткие требования. Заметим лишь, что имя скрипта часто начинается с символа «.», что позволяет не выводить на экран это имя при выполнении утилиты *ls* без записи специального ключа *-a*.

Так как по своей форме командный файл представляет собой обыкновенный текстовый файл, то для его получения и редактирования может быть использован любой текстовый редактор, например *edit* в *MS-DOS* или *sed* в *UNIX*. Для записи скриптов можно использовать и утилиту *cat*. Например, получим скрипт *file*, выполняющий задачу из п. 1.5.5, которая состоит в копировании всех файлов в поддереве данного пользователя, имеющих суффикс *txt*, в каталог *k2*:

```
$ cat >file
# Копирование всех файлов пользователя с суффиксом txt
# в каталог k2
for var in `find $HOME -name "*.txt"`
do
    cp $var ${HOME}/k2
done
<Ctrl>&<D>
```

Обратите внимание, что для задания корневого каталога поддерева пользователя используется не символ «~», а переменная окружения *HOME*. Это позволяет существенно увеличить число способов запуска данного скрипта. Допустим, что командный файл *file* находится в текущем каталоге, тогда он может быть запущен на выполнение следующими способами:

- 1) \$ *file*
- 2) \$ *./file*
- 3) \$ *sh file*
- 4) \$ *.file*
- 5) \$ *..*file**

В первых трех перечисленных способах запуска скрипта *file* для его выполнения создается новый экземпляр интерпретатора *shell*. При этом в третьей команде этот новый *shell* задается явно, а в двух предыдущих командах — неявно. При явном задании *shell* имя скрипта записывается в качестве параметра команды, следствием чего являются пониженные требования к правам доступа пользователя к файлу-скрипту: достаточно иметь лишь право на чтение этого файла. Задание имени скрипта в качестве самой команды (способы 1 и 2) требует наличия права пользователя на выполнение файла-скрипта. (Вопрос о правах доступа к файлу будет рассмотрен нами в гл. 3.)

Отличием первой команды от второй является использование в ней простого имени файла-скрипта. В связи с этим напомним, что *shell* не производит поиск исполняемого файла в текущем каталоге по умолчанию, и для этого требуется явное задание текущего каталога в переменной *PATH* с помощью команды *PATH=* $\{PATH\}:/$ . При способе 2 такого определения текущего каталога в переменной *PATH* не требуется.

При способах 4 и 5 имя скрипта задается в качестве параметра команды «.» *shell*. Наличие данной команды означает, что текущий *shell* должен выполнить заданный скрипт сам, а не породить для этого новый экземпляр *shell*. Одним из следствий этого является то, что при выполнении скрипта могут использоваться любые переменные текущего *shell*, а не только переменные окружения. Что касается отличий между командами 4 и 5, то они аналогичны различиям между командами 1 и 2.

Запуск скрипта из его родительского каталога имеет ограниченное применение и используется в основном при отладке скрипта. Большой интерес представляет запуск скрипта из любого текущего каталога. Для этого достаточно добавить абсолютное имя родительского каталога скрипта в переменную *PATH*, а затем использовать один из следующих способов:

1) *\$ file*

2) *\$. file*

Первая из этих команд запускает для выполнения скрипта новый экземпляр *shell*, а вторая — нет.

Скрипт может быть запущен на выполнение не только из командной строки, но и из другого командного файла аналогично обычной команде. В этом случае запускаемый скрипт называется **вложенным скриптом**, а запускающий — **главным скриптом**.

Как и любая виртуальная программа, командный файл может иметь **комментарии** — любой текст, предваряемый особым символом. Для скриптов *UNIX* таким символом является «#». Комментарии различаются: а) **вводные комментарии** поясняют назначение и запуск командного файла; б) **текущие комментарии** используются для пояснения внутреннего содержимого командного файла. Напомним, что, как и для любого исходного текста программы, командный файл без комментариев — черновик его автора.

Обычно *shell*, как и другие лингвистические процессоры, игнорирует комментарии. Исключением является комментарий, записанный в начале скрипта: если этот комментарий начинается с символа «!», то сразу за этим символом в комментарии записано абсолютное имя исполняемого файла, содержащего тот *shell*, который должен быть запущен текущим *shell* для выполнения скрипта. (Напомним, что в *UNIX*-системах существуют различные варианты *shell*.)

### Примеры

- 1) `#!/bin/sh` запускается Bourne shell
- 2) `#!/bin/csh` запускается C shell
- 3) `#!/bin/ksh` запускается Korn shell

Подобно обычным программам, командный файл может запускаться из командной строки *shell* (или из главного скрипта) с параметрами, которые, как обычно, отделяются друг от друга, а также от имени команды пробелами. Благодаря параметрам пользователь влияет на выполнение командного файла, задавая для него исходную информацию. Так как порядок записи параметров для каждого командного файла фиксирован, то такие параметры называются **позиционными параметрами**.

Например, скорректируем рассмотренный ранее пример скрипта так, чтобы скрипт имел два позиционных параметра: 1) требуемое окончание имени файла; 2) имя каталога, в который следует копировать найденные файлы. В результате вызов скрипта может выглядеть, например, следующим образом:

```
$ file .txt k2
```

Для того чтобы при выполнении командного файла *shell* мог использовать значения позиционных параметров, полученные от пользователя, каждый из этих параметров имеет свое имя, в качестве которого используется порядковый номер той позиции, которую занимает параметр в командной строке. При этом в качестве

параметра 0 рассматривается имя скрипта. Например, в рассматриваемом примере позиционные параметры имеют следующие значения: параметр 0 — *file*; параметр 1 — *.txt*; параметр 2 — *k2*.

Как и для переменной, значение позиционного параметра обозначается его именем, которому предшествует символ «\$». При выполнении скрипта каждое значение его позиционного параметра заменяется его значением, полученным из командной строки. Само это значение в командном файле может быть изменено только с помощью команды *set*, и поэтому позиционный параметр никогда не записывается в левой части операции присваивания и, как следствие, его имя всегда предворяется символом «\$».

С учетом сделанных замечаний выполним запись рассматриваемого скрипта:

```
$ cat >file
# Копирование всех файлов с заданным окончанием
# имени, принадлежащих данному пользователю,
# в заданный каталог
# параметр 1 – окончание имени файла
# параметр 2 – имя каталога
for var in `find $HOME -name \"*$1`
do
    cp $var ${HOME}/${2}
done
<Ctrl>&<D>
```

Обратим внимание, что для «экранирования» символа «\*» используется символ «\», а не кавычки. Это вызвано тем, что экранирующее действие символа «\» распространяется только на соседний справа символ и поэтому не действует на символ «\$», который в данном примере должен оставаться специальным символом *shell*, обозначая значение позиционного параметра.

Команда *set*, вводимая со своими параметрами, обеспечивает присваивание значений этих параметров позиционным параметрам скрипта.

### Пример

Создадим скрипт *k1*, который сначала выводит на экран значения своих позиционных параметров, а затем изменяет эти значения с помощью команды *set*.

```
$ cat >k1
# Меняет значения своих параметров на a и b
# параметры 1 и 2 — любые слова
echo $1; echo $2
```

```
set a b
echo $1; echo $2
<Ctrl>&<D>
```

Допустим, что скрипт *k1* запущен с параметрами 7 и 8:

```
$ ./k1 7 8
7
8
a
b
```

Кроме позиционных параметров, передаваемых в *shell* при вызове скрипта, *shell* автоматически устанавливает значения следующих **специальных параметров**:

? — код завершения последней выполненной команды;  
\$ — системный номер процесса, выполняющего *shell*;  
! — системный номер фонового процесса, запущенного последним;

# — число позиционных параметров, переданных в *shell*. Имя скрипта (параметр 0) в это число не входит;

\* — перечень позиционных параметров, переданных в *shell*. Этот перечень представляет собой строку, словами которой являются позиционные параметры.

Значения перечисленных специальных параметров могут использоваться не только в скриптах, но и в командных строках. При этом для записи значения параметра, как всегда, используется \$.

### Пример

Выполним запись скрипта, который добавляет к содержимому одного файла содержимое других файлов. Имя первого файла задается первым параметром скрипта, а имена других файлов — последующими параметрами.

```
$ cat >k2
# Добавление к содержимому файла содержимого других
# файлов
# параметр 1 — имя исходного файла
# параметр 2, 3,... — имена добавляемых файлов
x=1
for i in $*
do
    if [ $x -gt 1 ]
    then
        cat <$i >>$1
```



```
fi
x=`expr $x + 1`
done
<Ctrl>&<D>
```

В отличие от предыдущего скрипта, число параметров при вызове данного скрипта может быть задано любое. Если это число меньше двух, то выполнение скрипта не приводит к изменению содержимого какого-либо файла. Заметим также, что оператор *for i in \$\** во второй строке скрипта может быть заменен на оператор *for i*. При этом используется следующее свойство оператора *for*: при отсутствии части этого оператора, начинающейся со слова *in*, в качестве перечня значений заданной переменной (*i*) используется перечень позиционных параметров, заданный при вызове скрипта.

Особую роль играют **инициализационные командные файлы**. Они содержат команды ОС, выполняемые в самом начале сеанса работы пользователя. В любой однопользовательской системе *MS-DOS* всего один такой файл — *autoexec.bat*. В системе *UNIX* для любого пользователя первоначально выполняется инициализационный скрипт */etc/profile* (или другой подобный файл). Кроме того, каждого пользователя обслуживает свой инициализационный скрипт *.profile*, записанный самим пользователем в свой корневой каталог. Этот файл может содержать, например, приглашение к последующей работе, задание путей поиска исполняемых файлов, «переделку» приглашений *shell*. Первое из этих действий реализуется командой *echo*, а два последних — операциями присваивания, выполненных для переменных окружения.

После того как мы рассмотрели работу ВС с точки зрения конкретного пользователя, перейдем к рассмотрению способов реализации такой работы в системе. При этом в качестве первого вопроса рассмотрим реализацию в системе мультипрограммирования.



## 2. СИСТЕМНАЯ ПОДДЕРЖКА МУЛЬТИПРОГРАММИРОВАНИЯ

### 2.1. Мультипрограммирование

Любая мультипрограммная система, независимо от того, является ли она однопользовательской или многопользовательской, обеспечивает одновременное выполнение нескольких последовательных обрабатывающих программ, прикладных и (или) системных. Термин **последовательная программа** означает, что даже при наличии в системе нескольких ЦП в любой момент времени не может выполняться более одной команды этой программы.

В недалеком прошлом все обрабатывающие программы относились к классу последовательных программ. В настоящее время значительная часть обрабатывающих программ относится к классу параллельных программ. Для **параллельной программы** характерно то, что несколько ее команд могут выполняться одновременно (параллельно). Наличие нескольких ЦП делает при этом возможным физическую параллельность. Если же в системе всего один ЦП, то применительно к параллельной программе имеет место логическая (виртуальная) параллельность. Как правило, параллельную программу можно представить в виде совокупности нескольких последовательных программ, каждая из которых выполняется в значительной степени асинхронно (независимо). Нетрудно предположить, что одновременное выполнение даже одной параллельной программы возможно только в мультипрограммной системе.

Однопрограммная ОС, например *MS-DOS*, также позволяет нескольким выполняющимся последовательным программам одновременно находиться в ОП. Наличие таких программ обусловлено тем, что одна обрабатывающая программа может выполнить запуск другой программы. После выполнения такого запуска родительская программа переходит в состояние бездействия до тех пор, пока дочерняя программа не завершится и не возвратит управление в ту точку родительской программы, из которой она была запущена. Следовательно, программы, находящиеся одновременно в ОП, связаны друг с другом по управлению аналогично процедурам. При этом какая-либо асинхронность (параллельность) между программами отсутствует, а реализация в системе управляющих взаимодействий между программами не вызывает каких-либо трудностей.

Следует заметить, что полностью избавиться от асинхронного выполнения программ в однопрограммных системах не удастся. Это объясняется принципиальной асинхронностью событий, происходящих на ПУ, по отношению к программе, выполняемой в данный момент времени на ЦП. Заметим, что в данном случае речь идет об асинхронности между обрабатывающей программой, с одной стороны, и управляющими подпрограммами (обработчиками аппаратных прерываний) — с другой. При этом асинхронность обеспечивается, в основном, не программно (то есть операционной системой), а аппаратно.

В последующих разделах данной главы рассматриваются постановки задач, решение которых обеспечивает наличие мультипрограммирования в системе, а также рассматриваются методы решения этих задач. Реализация данных методов в системе *UNIX* будет рассмотрена в гл. 4 и 5.

## 2.2. Процессы

Важнейшим понятием любой мультипрограммной системы является понятие процесса. В данном разделе мы будем использовать наиболее простое определение: **процесс** — одно выполнение последовательной программы. Так как параллельную программу можно представить в виде совокупности нескольких последовательных программ, то выполнение каждой из этих последовательных программ есть отдельный процесс. Характерной особенностью процесса является то, что он никак не связан по управлению с огромным большинством других процессов. Следствием этого является **параллельность процессов**: этап выполнения одного процесса никак не связан с этапом выполнения другого процесса.

Так как процесс есть выполнение программы, то кто-то должен начать (инициировать) это выполнение. Это делает другой процесс, являющийся по отношению к первому процессу «процессом-отцом». Общим предком всех (или почти всех) процессов в системе является процесс, созданный сразу же после выполнения начальной загрузки ОС в оперативную память. Допустим, что этот процесс есть выполнение программы с именем *init*. Тогда «дочерними» процессами процесса *init* являются системные процессы, выполняющие служебные функции по поддержанию работоспособности системы, а также интерпретатор команд ОС, например *shell*.

После того как ИК (*shell*) будет создан и инициирован процессом *init*, он перейдет к ожиданию команды пользователя,

набираемой на клавиатуре или вводимой с помощью мыши в качестве выбранного варианта из меню, предлагаемого пользователю. В любом случае ИК получает имя программы, подлежащей выполнению путем создания и инициирования соответствующего программного процесса. Принципиальной особенностью мультипрограммной системы является то, что запуск новой «дочерней» программы может быть выполнен до завершения предыдущей «дочерней» программы. Поэтому в отличие от однопрограммной системы количество одновременно существующих дочерних процессов может быть более одного. Например, одновременно запускаются программы, образующие конвейер (см. п. 1.5.3). Параллельно с программой в оперативном режиме могут выполняться программы в фоновом режиме.

Например, пусть пользователь ввел следующую команду:

```
$ find ~/ -name 'f*' | tee file1 | fgrep f1 >file2 & script7
```

где *script7* — командный файл следующего содержания:

```
$ cat script7
cat file3
echo +++++
cat file4
```

Тогда вначале обработки данной команды *shell* дерево процессов принимает вид, приведенный на рис. 11. Обратим внимание, что выполнение команды *echo* не приводит к появлению нового процесса, так как эту команду выполняет не отдельная утилита, а подпрограмма самого *shell*. Кроме того, заметим, что не могут существовать одновременно два процесса *cat*, так как каждый из них должен выполняться в оперативном режиме (требуется экран), и поэтому один из процессов изображен пунктиром.

Заметим, что символьное имя программы (имя исполняемого файла) используется нами в качестве имени процесса только с целью наглядности. В действительности оно не может использоваться в качестве имени процесса, так как применительно к процессу не обладает свойством уникальности. Подобным свойством обладает **номер процесса**, используемый в качестве системного, программного, а также пользовательского имени процесса.

Находясь в командной строке *shell*, пользователь может получить информацию о своих программных процессах, используя команду *ps*. Ее применение без флагов позволяет вывести на экран минимум информации о процессах.

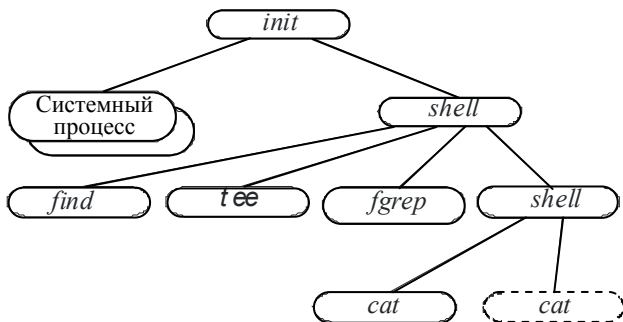


Рис. 11. Пример дерева процессов

### Пример

```

$ ps
  PID  TTY    TIME  CMD
  145  tty4    0:01   /user/bin/sh
  313  tty4    0:03   /user/bin/ed
  431  tty4    0:01   /user/bin/ps
$
  
```

В выдаче команды:

- 1) *PID* — номер процесса;
- 2) *TTY* — имя управляющего терминала процесса (*co* — операторская консоль; ? — процесс не управляется терминалом). Терминальное управление процессами рассматривается в п. 2.4.2;
- 3) *TIME* — затраты времени ЦП на выполнение процесса;
- 4) *CMD* — имя команды *shell*, выполнение которой привело к созданию процесса.

Некоторые флаги этой команды:

- e** — вывод информации обо всех без исключения процессах;
  - f** — вывод достаточно подробной информации о процессах: системное имя пользователя, номер процесса-отца, время создания процесса;
  - l** — вывод наиболее подробной информации о процессах.
- Содержание этой информации будет рассмотрено нами в последующих разделах.

В рассматриваемом примере два процесса *shell* и два процесса *cat*. Использование несколькими процессами одинаковых программ приводит к мысли о возможности использования ими одного и того же экземпляра программы. Такая идея реализуема в том случае, если программа процесса является реентерабельной. Программа

называется *реентерабельной*, если содержимое занимаемой ею области памяти не изменяется при выполнении программы.

Так как выполнение любой сколько-нибудь полезной программы требует выполнения операций записи в ОП, необходимо отделить все изменяемые данные в отдельную область. Неизменяемую область памяти программы будем называть *сегментом кода*, а изменяемую — *сегментом данных*. При этом сегмент кода может содержать не только команды (код) программы, но и ее неизменяемые данные (константы). При совместном использовании несколькими процессами одной и той же программы каждый из них использует единственный общий экземпляр сегмента кода, но свой сегмент данных. Так как сегмент кода неизменен, то выполнение одного процесса никак не влияет на выполнение других процессов.

В мультипрограммных системах широко используются не только реентерабельные программы, но и реентерабельные подпрограммы (процедуры). Обычные нереентерабельные подпрограммы «встраиваются» в загрузочный модуль (исполняемый файл) программы при его получении редактором связей (см. подразд. 1.4). Так как подобное связывание программы производится до начала ее выполнения, то оно называется *статическим связыванием*. При этом предполагается, что если несколько программ вызывают одну и ту же подпрограмму (процедуру), то каждая из них будет обладать своим отдельным экземпляром этой процедуры. Следствием такого «тиражирования» являются повышенные затраты памяти. Использование реентерабельных подпрограмм позволяет эти затраты существенно сократить.

Реентерабельные подпрограммы собраны в *динамически компоновемые библиотеки* — *DLL* (*dynamic link library*). Связывание этих подпрограмм с программой осуществляется *динамическим редактором связей* во время загрузки программы в ОП. Загрузочный модуль программы, выполняемой в среде *UNIX*, содержит не только перечень требуемых *DLL*, но и имя файла, содержащего динамический редактор связей. Получив управление при загрузке программы, этот редактор связей обеспечивает загрузку в память системы недостающих *DLL*, а затем помещает численные адреса требуемых подпрограмм в команды программы, выполняющие вызов подпрограмм. Закончив свою работу, динамический редактор связей передает управление в точку входа загруженной программы. Параллельные процессы, использующие *DLL*, работают

с одним и тем же экземпляром сегмента кода этой библиотеки, но используют ее различные сегменты данных.

После того как процесс создан, он может вступать с другими процессами в управляющие и информационные взаимодействия. Примерами управляющих взаимодействий являются операции создания и уничтожения процессов. Примером информационного взаимодействия является обмен информацией между процессами, образующими конвейер.

Для реализации управляющих и информационных взаимодействий между процессами им требуется помощь со стороны ОС.

## 2.3. Ресурсы

В отличие от однопрограммной ОС, выполняющей распределение ресурсов системы между программами, являющимися «близкими родственниками», мультипрограммная ОС должна заниматься их распределением между параллельными процессами, в общем случае «чужими» по отношению друг к другу. Следствием этого является то, что основные решения по распределению ресурсов между процессами теперь должен принимать не прикладной программист, а сама ОС. Поэтому наряду с программными процессами ресурсы являются важнейшими объектами, подлежащими управлению со стороны мультипрограммной ОС.

Определение: **логическим ресурсом** или просто **ресурсом** называется объект, нехватка которого приводит к блокированию процесса и переводу его в состояние «Сон». Подробная речь о состояниях процесса будет идти в подразд. 4.1, а пока лишь заметим, что в данном состоянии процесс не может выполняться на ЦП до тех пор, пока причина блокирования не будет устранена.

На рис. 12 приведена классификация ресурсов. К **аппаратным ресурсам** относятся ЦП, ОП, устройства ввода-вывода, устройства ВП, носители ВП. Все аппаратные ресурсы являются **повторно используемыми**. То есть после того как данный ресурс стал не нужен тому процессу, которому он был выделен, он может быть распределен какому-то другому процессу.

Под **информационными ресурсами** понимаются какие-то данные, не получив доступ к которым конкретный программный процесс блокируется. Фактически информационные ресурсы представляют собой области памяти, заполненные какой-то полезной информацией. В отличие от них пустые области ОП и ВП являются аппаратными ресурсами.



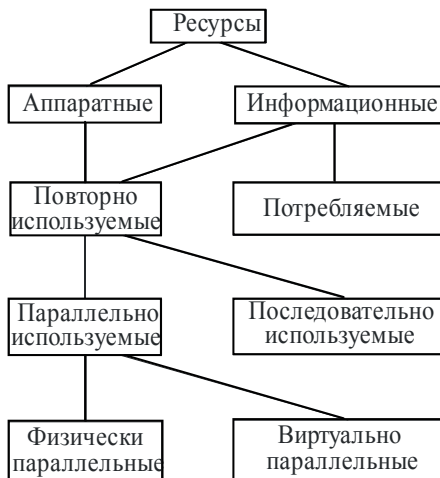


Рис. 12. Классификация ресурсов

Информационные ресурсы делятся на повторно используемые и потребляемые. **Потребляемым ресурсом** является сообщение, которое один процесс выдает другому процессу. После того как сообщение обработано процессом-потребителем, оно больше не нужно и может быть уничтожено. **Повторно используемыми** информационными ресурсами являются данные, совместно используемые несколькими процессами. К таким ресурсам относятся файлы (в том числе библиотеки), базы данных, совместно используемые программы, а также некоторые структуры данных в ОП. Повторно используемые ресурсы (как аппаратные, так и информационные) делятся на параллельно и последовательно используемые.

**Последовательно используемый ресурс** выделяется некоторому процессу и не может быть перераспределен до тех пор, пока первому процессу этот ресурс не будет больше нужен. К таким ресурсам относятся многие устройства ввода-вывода и ВП (терминал, сканер, стример и т.д.), а также последовательно используемые программы — программы, которые не являются реентерабельными, но которые должны обслуживать запросы нескольких параллельных процессов. Примером последовательно используемой программы является ядро *UNIX* (рассматривается в подразд. 3.3).

**Параллельно используемые ресурсы** могут действительно использоваться одновременно несколькими процессами (**физическая**

*параллельность*), или они используются параллельно лишь виртуально (*виртуальная параллельность*). Примеры первого типа: ОП, ВП прямого доступа (например магнитный диск), реентерабельные программы и *DLL* (при наличии в ВС нескольких ЦП). Примеры второго типа: ЦП, доступ к параллельно используемому устройству (например к дисководу), реентерабельные программы и *DLL* (при одном ЦП).

Покажем разницу между физически параллельными и виртуально параллельными ресурсами на примере использования двух ресурсов — пространства памяти на магнитном диске и доступа к дисководу, на котором этот диск установлен. В то время как свободное пространство диска реально делится на области между процессами, выполняющими операции записи в файлы, доступ к дисководу в каждый конкретный момент времени имеет лишь один прикладной процесс. В следующий момент доступ к устройству ВП может быть передан другому процессу, затем опять возвращен первому процессу и т.д.

Наличие в системе разнообразных ресурсов, а также наличие параллельных процессов, которым эти ресурсы требуются, приводит к необходимости решения ОС следующих задач:

1) распределение повторно используемых ресурсов между процессами, учитывающее: а) свойства распределяемого ресурса; б) потребности тех процессов, которым ресурс распределяется. Решение данной задачи рассматривается в гл. 4, 5 на примере наиболее важных ресурсов ВС — оперативной памяти и времени ЦП;

2) синхронизация параллельных процессов, совместно использующих информационные ресурсы, потребляемые или повторно используемые. Методы решения этой задачи рассматриваются в пп. 2.4.3, 2.4.4 и 2.5;

3) оказание помощи процессам (при совместном использовании ими повторно используемых ресурсов) по устранению такого неприятного явления, как тупики. Решение этой задачи приведено в подразд. 2.6;

4) защита информации, принадлежащей какому-то процессу от воздействия других процессов. При этом наиболее важная задача, решаемая с целью поддержки мультипрограммирования — защита информации в ОП. Эта задача решается не столько программными, сколько аппаратными средствами. Ее решение рассматривается в п. 5.2.3.

## 2.4. Синхронизация параллельных процессов

Параллельные процессы, существующие в ВС, нуждаются в синхронизации. **Синхронизация** — согласование этапов выполнения двух или более параллельных процессов путем обмена ими инициирующими (командными) воздействиями. Перечислим некоторые задачи, решаемые с помощью синхронизации процессов:

- 1) обработка программой процесса некоторых аппаратных прерываний;

- 2) терминальное управление процессами;

- 3) синхронизация параллельных процессов, выполняющих действия с общей областью ОП;

- 4) синхронизация параллельных процессов, выполняющих информационный обмен, используя общую область ОП.

Для решения первых двух задач в *UNIX*-системах используются сигналы, а для решения двух последних задач — семафоры.

### 2.4.1. Синхронизация с помощью сигналов

**Сигнал** — команда, которую один процесс посылает другому процессу (процессам) с целью оказания влияния на ход выполнения этого процесса (процессов).

В отличие от других известных нам команд, к которым относятся машинные команды и команды *shell*, команды-сигналы не имеют операндов (параметров) и представляют собой только коды операций. Другой особенностью сигнала является то, что моменты выдачи и обработки сигнала могут быть довольно существенно разнесены во времени. Причиной этого является то, что в момент выдачи сигнала процесс-отправитель, а в момент обработки сигнала процесс-получатель обязательно должны выполняться на ЦП. Еще одной особенностью сигналов является их ненакапливаемость. То есть если в момент поступления сигнала в процесс еще не начата обработка предыдущего однотипного сигнала, то вновь пришедший сигнал теряется.

Вспомним, что в однопрограммной системе (например в *MS-DOS*) возникающие в системе прерывания обрабатываются или подпрограммами ядра, или подпрограммами прикладной программы. При этом прикладная программа может перехватывать и обрабатывать любые прерывания. Подобный подход совершенно неприемлем для мультипрограммной системы, в которой обработка прерываний должна находиться под управлением ОС. С другой

стороны, полное отстранение процессов от обработки прерываний делает их выполнение негибким, лишая их многих возможностей. Следствием этого является предоставление процессам возможности участвовать в обработке некоторых типов прерываний. При этом процесс может обрабатывать не сам аппаратный сигнал прерывания, а его преобразованную форму — сигнал, выдаваемый обработчиком прерываний.

К прерываниям, которые не преобразуются в сигналы, а обрабатываются полностью обработчиками ядра ОС, относятся программные прерывания и большинство внешних аппаратных прерываний. Преобразуются в сигналы лишь прерывания-исключения, а также некоторые внешние аппаратные прерывания. С другой стороны, не только обработчики перечисленных прерываний, но и сами процессы могут быть источниками сигналов, используя для этого специально предназначенные системные вызовы.

Современная *UNIX*-система различает примерно 30 типов сигналов, каждый из которых имеет свое системное имя — номер сигнала, а также программное (символьное) имя. Перечислим символьные имена некоторых из сигналов, источниками которых являются обработчики прерываний:

1) *SIGFPE* — сигнал возникновения переполнения результата, например, из-за деления на 0, во время выполнения текущей машинной команды процесса. Обработка по умолчанию — завершение процесса и создание файла *core* в текущем каталоге. Этот файл может использоваться при запуске утилиты-отладчика с целью обнаружения ошибки в программе процесса;

2) *SIGILL* — сигнал выполнения программой процесса недопустимой машинной команды. Обработка по умолчанию — завершение процесса и создание файла *core* в текущем каталоге;

3) *SIGEGV* — попытка программы процесса обратиться к ячейке ОП, которая или не существует, или для доступа к которой у процесса нет прав. Обработка по умолчанию — завершение процесса и создание файла *core* в текущем каталоге;

4) *SIGTRAP* — сигнал о возникновении исключения «трасировка». Такой сигнал используется отладчиками программ;

5) *SIGPWR* — сигнал угрозы потери питания;

6) *SIGALRM* — сигнал таймера. Передается обработчиком прерываний таймера при завершении интервала времени, заданного ранее этому обработчику с помощью специально предназначенного для этого системного вызова.

Сигналы, выдаваемые процессами:

1) *SIGKILL* — сигнал уничтожения процесса. Единственно возможная реакция процесса на этот сигнал — немедленное завершение;

2) *SIGSTOP* — сигнал останова процесса. Единственно возможная реакция процесса на этот сигнал — переход в состояние «Останов»;

3) *SIGCONT* — продолжение работы остановленного процесса. Если процесс не был ранее остановлен, то он игнорирует данный сигнал;

4) *SIGTERM* — сигнал «добровольного» завершения процесса. Получив данный сигнал, процесс может подготовиться к своему уничтожению, выполнив самые неотложные действия. Часто выдача данного сигнала предшествует выдаче сигнала *SIGKILL*;

5) *SIGCHLD* — сигнал, посылаемый процессу-отцу при останове или при завершении дочернего процесса. Источник сигнала — дочерний процесс;

6) *SIGUSR1*, *SIGUSR2* — пользовательские сигналы. Они предназначены для взаимодействия между прикладными процессами, характер которого определяется разработчиком прикладных программ.

Для того чтобы послать сигнал другому процессу (процессам) программа данного процесса должна обратиться за помощью к ОС. Для получения любой помощи со стороны системы обрабатывающая программа должна содержать специальную команду — **системный вызов**.

При записи системных вызовов нами будет использоваться следующее правило. Во-первых, с целью пояснения смысла вызова его имя будет записываться прописными русскими буквами курсивом. Во-вторых, параметры вызова перечисляются через запятую в круглых скобках после имени вызова. В-третьих, входные параметры вызова отделяются от его выходных параметров символами «||». Причем входные параметры располагаются слева, а выходные — справа от этих символов. В-четвертых, справа от системного вызова в круглых скобках помещается аналогичный системный вызов *UNIX*, записанный на языке СИ.

Системный вызов для отправки сигнала:

*ПОСЛАТЬ\_СИГНАЛ* ( $i_p$ ,  $S$  || ) (на СИ — *kill*),

где  $i_p$  — номер процесса, которому посылается сигнал;  $S$  — имя сигнала.

Если  $i_p = 0$ , то сигнал посылается всем процессам из той же группы процессов, что и отправитель сигнала. При  $i_p = -1$  сигнал посылается всем процессам данного пользователя. Понятие группы процессов рассматривается в следующем подразделе.

## 2.4.2. Терминальное управление процессами

Любая интерактивная ВС имеет среди своих ПУ по крайней мере один *терминал* — совокупность устройства ввода и устройства вывода. Терминал позволяет пользователю выполнять запуск программ. Кроме того, пользователь системы имеет возможность влиять на выполнение любого своего программного процесса путем нажатия специальных комбинаций клавиш на клавиатуре. Поэтому терминал пользователя является для всех его программных процессов *управляющим терминалом*.

Вспомним, что корнем поддерева процессов, принадлежащих конкретному пользователю, является процесс *shell* (интерпретатор команд ОС). Этот процесс осуществляет «открытие» управляющего терминала, делая его доступным не только для себя, но и для своих будущих «потомков». Такое множество потомков процесса *shell* называется *сеансом*, а процесс *shell* — *лидером сеанса*. Каждый сеанс в системе имеет уникальное системное имя — *номер сеанса*, совпадающий с номером процесса, являющегося лидером сеанса.

Управляющее воздействие передается процессу от своего управляющего терминала в виде сигнала. Примером является сигнал освобождения линии *SIGHUP*, выдаваемый одновременно всем процессам сеанса при завершении работы пользователя и отключении им терминала. Стандартная реакция процесса на этот сигнал — завершение.

Отношение процессов сеанса к управляющему терминалу неодинаково. Среди них есть процессы-изгои, выполняемые в фоновом режиме. (Напомним, что в фоновом режиме программа процесса не выполняет операций ввода-вывода с терминалом. Для запуска из командной строки процесса или конвейера процессов в фоновом режиме необходимо в конце командной строки набрать символ «&».)

При этом процессы, запущенные из одной командной строки, обычно информационно связаны друг с другом. Так как пользователю удобно выполнять с терминала совместное управление этими процессами, то процесс-*shell* объявляет свои совместно запускаемые дочерние программные процессы единой *группой*

**процессов.** Каждая группа процессов имеет уникальный для всей системы **номер группы процессов**, в качестве которого *shell* назначает номер одного из процессов-членов группы. Такой процесс называется **лидером группы процессов**. Для создания новой группы или для включения процесса в уже существующую группу *shell* обращается к ядру, используя системный вызов

`УСТАНОВИТЬ_ГРУППУ_ПРОЦЕССОВ( ig, ip || )` (на СИ — `setpgid`),

где *ig* — номер группы процессов; *ip* — номер процесса.

Если  $i_g = i_p$ , то в результате данного системного вызова создается новая группа, лидером которой является процесс  $i_p$ . Иначе процесс  $i_p$  включается в уже существующую группу с номером  $i_g$ .

Вообще говоря, сразу же после своего создания процесс уже принадлежит к той же группе, что и процесс-отец. Смену группы у дочерних процессов *shell* производит с целью «отмежеваться» от них при получении ими различных сигналов. В любом случае каждый процесс сеанса в конкретный момент времени принадлежит одной (и только одной) группе процессов.

Среди всех групп процессов, на которые разбито множество процессов одного сеанса, одна группа имеет особое значение. Это **оперативная группа** — совокупность процессов, выполняемых в оперативном режиме, в котором процесс может выполнять информационный обмен с управляющим терминалом. Например, *shell* выполняет диалог с пользователем только в оперативном режиме. Введя команду или группу команд без завершающего символа «&», *shell* запускает соответствующую группу процессов в оперативном режиме, не забыв при этом установить себе другую группу процессов (фоновую). В результате *shell* становится недостижим для сигналов, воздействующих одновременно на все члены оперативной группы. Перечислим эти сигналы:

1) *SIGINT* — сигнал прерывания программы. Выдается одновременно всем процессам оперативной группы вследствие нажатия пользователем клавиши <Del> или <Ctrl>&<C>. Обработка по умолчанию — завершение процесса;

2) *SIGQUIT* — сигнал о выходе. Выдается одновременно всем процессам оперативной группы вследствие нажатия пользователем клавиш <Ctrl>&<\>. Отличается от сигнала *SIGINT* тем, что кроме завершения процесса, на диске в текущем каталоге создается дамп памяти процесса — файл *core*;

3) *SIGTSTP* — терминальный сигнал останова. Выдается одновременно всем процессам оперативной группы вследствие нажатия пользователем клавиш *<Ctrl>&<Z>*. Стандартная реакция процесса на этот сигнал — переход процесса в состояние «Останов».

Следующие два сигнала выдаются подпрограммами управления терминалом в том случае, если фоновый процесс сделает попытку выполнить операцию ввода-вывода с управляющим терминалом:

1) *SIGTTIN* — сигнал о попытке ввода с терминала фоновым процессом. Обработка по умолчанию — перевод процесса в состояние «Останов»;

2) *SIGTTOU* — сигнал о попытке вывода на терминал фоновым процессом. Обработка по умолчанию — перевод процесса в состояние «Останов».

Существуют системные вызовы, позволяющие процессу делать оперативной любую фоновую группу своего сеанса, и наоборот — делать фоновой оперативную группу. Несмотря на то что любой процесс может воспользоваться этими вызовами, на практике это делает только *shell*. Кроме того, любой процесс, не являющийся лидером сеанса, может покинуть свой прежний сеанс и стать лидером нового сеанса, воспользовавшись системным вызовом

*УСТАНОВИТЬ\_СЕАНС* (||) (на СИ — *setsid*)

Данный вызов не имеет параметров, так как номер нового сеанса будет совпадать с номером процесса, сделавшего вызов. В случае успешного завершения вызова появится новый сеанс и новая группа, единственным членом и лидером которых будет процесс, сделавший вызов. Отличительной чертой нового сеанса является то, что он не имеет управляющего терминала.

Новый сеанс может или вообще не иметь управляющего терминала, или же его лидер должен открыть новый управляющий терминал. Так как реальный терминал уже занят прежним сеансом, то в качестве нового терминала обычно открывается **псевдотерминал**. Подробно псевдотерминалы рассматриваются в подразд. 3.1, а пока лишь заметим, что они используются для доступа к системе удаленных пользователей.

Лидер нового сеанса вообще не открывает управляющий терминал в том случае, если он хочет оградить себя и своих потомков от воздействия сигналов с управляющего терминала. Именно с этой целью в данном случае и создается новый сеанс. Подобный процесс, не связанный ни с каким управляющим терминалом,



называется *демоном*. Демоны широко используются ядром ОС для выполнения общесистемных функций, поддерживающих работоспособность системы. Например, демоном является процесс *init*.

Пользователь системы является первичным источником сигналов не только при нажатии им одной из специальных комбинаций клавиш, которые были рассмотрены нами выше. Он может выдать требуемый сигнал нужному процессу, используя команду *shell – kill*:

**\$ kill** – сигнал процесс,

где *сигнал* — имя сигнала (номер или символьное имя), предваряемое символом «–». Этот параметр необязателен. Если он опущен, то по умолчанию посылается сигнал *SIGTERM* (просьба о добровольном завершении процесса);

процесс — номер того процесса, которому направляется сигнал.

Администратор системы может послать сигнал любому процессу, а обычный пользователь — только своему. Для определения номера требуемого процесса используется команда *ps* (рассматривается в подразд. 2.2). Кроме того, для задания номера процесса иногда оказывается полезной внутренняя переменная *shell* с именем «!». Эта переменная содержит номер того процесса, который был запущен последним в фоновом режиме. Как и другие внутренние переменные *shell*, значение переменной «!» задается самим *shell*, и поэтому ни в командных строках, ни в скриптах имя данной переменной никогда не встречается без предварительной записи символа «\$», наличие которого означает «значение переменной».

### Пример

Следующая команда посылает в процесс, запущенный последним в фоновом режиме, сигнал *SIGKILL*, выдача которого приводит к жесткому прекращению процесса без сохранения какой-либо информации о его завершении:

**\$ kill -SIGKILL \$!**

Для выполнения команды *kill shell* использует свою внутреннюю подпрограмму. Применение данной команды не ограничивается управлением готовыми процессами, что позволяет, например, уничтожать процессы, находящиеся в тупике. Ее применение позволяет также имитировать выдачу любого сигнала с целью проверки правильности его обработки программой процесса, что весьма полезно при разработке этой программы.

### 2.4.3. Синхронизация конкурирующих процессов

В однопрограммной ВС единственным способом реализации информационного обмена между программными модулями (программами и подпрограммами) является использование общей памяти, доступной для взаимодействующих модулей. В качестве такой памяти могут использоваться рабочие регистры ЦП, стек, другие области ОП. В мультипрограммной системе для информационного взаимодействия между процессами не могут использоваться ни регистры ЦП, ни программный стек, так как каждый из процессов пользуется своим набором этих модулей (регистры ЦП разделяются процессами виртуально, а стеки изолированы между собой физически).

Единственный тип области ОП, пригодный для непосредственного информационного взаимодействия между процессами — разделяемый сегмент данных. Принципиальное отличие такого сегмента от разделяемого сегмента кода, содержащего реентерабельную программу или *DLL*, состоит в том, что программы процессов могут не только читать из этого сегмента, но и выполнять в него запись. Назначение процессам разделяемого сегмента данных выполняется ОС и будет рассмотрено нами позднее (в подразд. 5.2). Сейчас нам важно уяснить, что без дополнительной синхронизации процессов наличие разделяемого сегмента явно недостаточно для их информационного взаимодействия. Для того чтобы показать это, рассмотрим следующую задачу.

Допустим, что процесс-сервер выполняет запросы процесс-клиентов по распечатке текстовых файлов на принтере. При этом информационное взаимодействие клиентов и сервера осуществляется через разделяемый сегмент памяти (рис. 13). В этом сегменте расположена структура данных — связанная очередь, в которую процессы-клиенты помещают свои запросы на обслуживание (имена текстовых файлов), а процесс-сервер извлекает эти запросы из очереди по одному и выполняет.

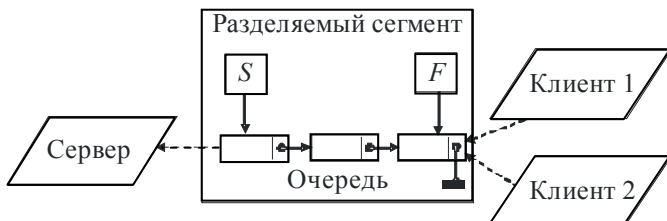


Рис. 13. Три процесса разделяют сегмент памяти

Обычно в разделяемой области памяти находится не одна, а несколько переменных. Если два процесса могут работать с одной и той же переменной  $x$ , причем, по крайней мере, один из них может выполнять запись в  $x$ , то говорят, что эти два процесса являются **конкурирующими** из-за  $x$ .

В рассматриваемом примере разделяемый сегмент содержит три переменные: очередь (массив), а также указатели  $S$  и  $F$ , являющиеся указателями на начало и конец очереди. Процессы-клиенты конкурируют из-за очереди, а также из-за  $F$ . Любой клиент конкурирует с сервером из-за очереди. Покажем, что конкурирующие процессы обязательно должны быть синхронизированы.

Допустим, что в какой-то момент времени очередь имеет состояние, приведенное на рис. 14,а. Пусть клиент 1 хочет поместить в очередь элемент  $y$ , а клиент 2 — элемент  $z$ . Очередь, измененная в результате правильного включения, приведена на рис. 14,б.

Но возможен другой исход, если учесть, что операция включения элемента в очередь состоит из трех более мелких операций:

- 1) чтение переменной  $F$ , указывающей на последний элемент в списке;
- 2) корректировка указателя в элементе  $n$  на новый элемент;
- 3) запись в  $F$  указателя на новый элемент.

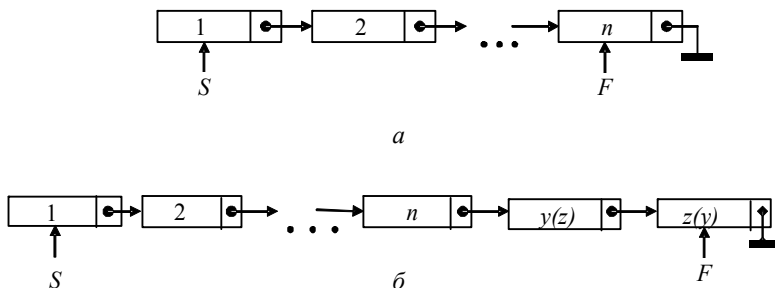


Рис. 14. Состояния очереди:

$a$  — исходное;  $б$  — правильное включение элементов  $y$  и  $z$

Допустим, что клиент 1 успел выполнить операции 1 и 2. Потом он был прерван на некоторое время, в течение которо-

го клиент 2 выполнил все три операции. Потом клиент 1 смог выполнить операцию 3. Соответствующие изменения очереди приведены на рис. 15. В результате единая очередь оказалась разорванной на две несвязанные части, что не допустимо. Рассмотрим, как этого можно избежать.

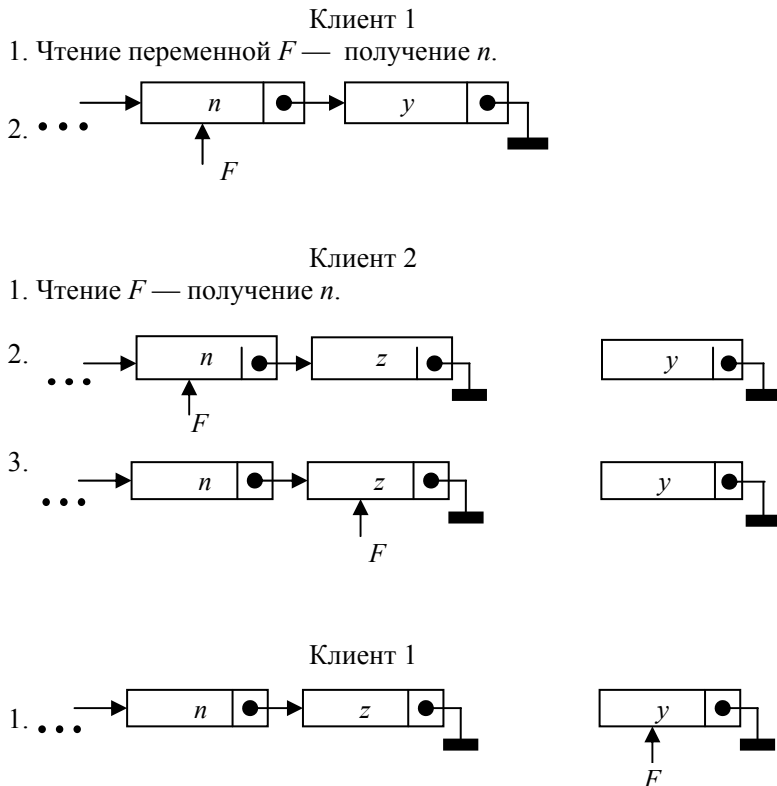


Рис. 15. Возможное преобразование очереди при отсутствии синхронизации

Пусть два процесса  $p_1$  и  $p_2$  конкурируют из-за переменной  $x$ . При выполнении действия конкурирующего процесса с переменной, как правило, выполняется не один, а целая группа операторов соответствующей программы. Каждая такая группа называется **критической секцией**. В принципе никаких ошибочных взаимодействий между процессами не будет, если им запретить одно-

временно выполнять свои критические секции. Рассмотрим подходы для обеспечения этого.

Первый, наиболее очевидный подход заключается в том, чтобы на время выполнения процессом своей критической секции вообще запретить выполнение других процессов на ЦП. Такой подход используется довольно широко в обработчиках аппаратных прерываний, которые на время выполнения своих критических секций производят запрет внешних (маскируемых) аппаратных прерываний. Недосток такого подхода: допускается делать запрет внешних прерываний лишь на очень короткое время. (Обработчики внешних аппаратных прерываний могут рассматриваться как процессы реального времени, для каждого из которых есть предельное время реакции.) Другой недостаток: данный метод применим лишь в однопроцессорных ВС. В системе с несколькими ЦП запрет прерываний в одном ЦП не влияет на выполнение программ на других ЦП.

Другое применение этого же подхода: в системе *UNIX* выполнение процесса не может быть прервано в состоянии «Ядро» другими процессами. Это обеспечивает целостность системных данных, так как работа с этими данными производится процессом только в состоянии «Ядро».

Второй подход предполагает запрещение выполнения на время выполнения процессом своей критической секции, не всех процессов, а лишь тех, которые конкурируют с ним из-за этой же переменной. Для этого перед входом в свою критическую секцию процесс посылает сигнал запрета своим конкурентам. После выхода процесса из критической секции он посылает процессам-конкурентам сигнал разрешения. С другой стороны, каждый процесс перед входом в свою критическую секцию проверяет, какой из двух сигналов он получил последним. Если это сигнал разрешения, то процесс сам посылает сигнал запрета. Недостатком метода является то, что процессы должны находиться не просто в дружеских отношениях, при отсутствии которых сигналы могут вообще игнорироваться процессом, а в родственных отношениях, требуемых для определения номера процесса. Другой недостаток: лишние затраты времени ЦП при ожидании процессом доступа в свою критическую секцию.

Третий подход заключается в использовании для управления доступом к разделяемым переменным двоичных переменных (флагов). Эти флаги находятся в разделяемой области, и каждый из них

управляет доступом к одной переменной: если флаг установлен, то работать с переменной можно, а если сброшен, то нельзя. Перед входом в критическую секцию процесс проверяет значение флага. Если он сброшен, то процесс в цикле ожидает его установки. Очевидно, что в течение текущего кванта времени ЦП процесс этого не дожидается, и время ЦП будет потрачено зря. Другой недостаток: проверку установки флага и переход в случае успеха должна выполнять одна неделимая машинная команда. Если это не так, то возможна ситуация, когда после выполнения команды проверки флага процесс будет прерван своим конкурентом, что может привести к непредсказуемому результату.

Четвертый подход является развитием предыдущего, позволяя избавиться от бесполезных затрат времени ЦП на ожидание освобождения разделяемой переменной. Это использование семафоров Дейкстры.

**Семафором Дейкстры  $S$**  называется целая неотрицательная переменная  $S = 0; 1; 2; 3; 4 \dots$ , над которой допустимы две операции:

- 1)  $v(S)$  — переменная  $S$  увеличивается на 1;
- 2)  $p(S)$  — переменная  $S$  уменьшается на 1, если это возможно.

Если  $S = 0$ , то ее уменьшить нельзя и процесс, содержащий  $p(S)$ , будет ждать (в состоянии «Сон») до тех пор, пока  $S$  не станет  $> 0$  и операция  $p(S)$  не станет возможной.

Операции  $v(S)$  и  $p(S)$  реализуются или аппаратно (в виде одной машинной команды), или программно. Алгоритмы этих операций:

```

 $p(S)$ : ЕСЛИ  $S > 0$ 
      ТО  $S \leftarrow S - 1$ ; продолжение выполнения процесса
      ИНАЧЕ блокирование процесса по  $S$ 
              вызов диспетчера ЦП
 $v(S)$ : ЕСЛИ (очередь процессов, ожидающих  $S$ , непустая)
      ТО деблокирование процесса, заблокированного по  $S$ 
      ИНАЧЕ  $S \leftarrow S + 1$ ; продолжение выполнения процесса
  
```

Здесь действие «блокирование процесса по  $S$ » означает, что, во-первых, процесс переводится в состояние «Сон», а во-вторых, процесс (а точнее его номер) помещается в очередь процессов, каждый из которых ожидает завершения своей операции  $p(S)$ . Очередь процессов — структура данных, обязательно присущая любому семафору. В ней находятся процессы, ожидающие получения того ресурса, который контролируется данным семафором  $S$ . (В нашем случае таким ресурсом является разделяемая перемен-

ная.) В случае блокирования процесса вызывается диспетчер ЦП с целью установки на ЦП какого-то процесса, ожидающего доступа к ЦП.

Если семафор  $S$  может принимать только два значения (0 и 1), то он называется **двоичным семафором**. Для синхронизации любого числа процессов, конкурирующих между собой из-за некоторой переменной  $x$ , достаточно:

- 1) в программе каждого процесса выделить критические секции, каждая из которых выполняет некоторое действие над  $x$ ;
- 2) задать начальное значение двоичного семафора  $S$ , равное 1;
- 3) перед каждой критической секцией записать оператор  $p(S)$ , а после нее —  $v(S)$ .

#### 2.4.4. Синхронизация кооперирующихся процессов

Два процесса, совместно использующие общую переменную, могут не только конкурировать из-за нее, но и кооперироваться. **Кооперирующиеся процессы** — процессы, выполняющие общую работу, которая заключается в том, что один процесс («писатель») выполняет запись в общую структуру данных (буфер), а другой процесс («читатель») выполняет считывание данных оттуда. Например, в приведенном ранее примере со связанной очередью каждый процесс-клиент является «писателем», а процесс-сервер — «читателем». Поэтому процесс-сервер образует с каждым процессом-клиентом пару кооперирующихся процессов.

Кооперирующиеся процессы нуждаются в синхронизации, во-первых, для того, чтобы процесс-«писатель» не записывал в буфер тогда, когда он уже полон. Во-вторых, процесс-«читатель» не должен читать из буфера тогда, когда он пуст. Для подобной синхронизации могут быть использованы сигналы, но гораздо удобнее использовать для этого семафоры Дейкстры.

Допустим, что несколько процессов-«писателей» выполняют запись элементов данных в буфер, а несколько других процессов-«читателей» выполняют чтение элементов данных из этого же буфера. Пусть емкость буфера составляет  $n$  элементов. Тогда для синхронизации их деятельности достаточно ввести два семафора —  $S_1$  и  $S_2$ . Семафор  $S_1$  управляет работой «писателей». Его значение есть число пустых позиций в буфере. Семафор  $S_2$  управляет работой «читателей». Его значение есть число занятых позиций в буфере. Первоначальные значения:  $S_1 = n$ ,  $S_2 = 0$ . Так как при

работе с буфером кооперирующиеся процессы одновременно являются и конкурирующими, то для исключения одновременной работы с буфером введем двоичный семафор  $S3$ . Возможные программы процессов:

#### ПРОЦЕСС-ПИСАТЕЛЬ:

$M1$ :  
 .....  
*подготовка элемента данных*  
 $p(S1)$   
 $p(S3)$   
*запись элемента данных в буфер*  
 $v(S3)$   
 $v(S2)$   
*переход  $M1$*   
 .....

#### ПРОЦЕСС-ЧИТАТЕЛЬ:

$M2$ :  
 .....  
 $p(S2)$   
 $p(S3)$   
*чтение элемента данных из буфера*  
 $v(S3)$   
 $v(S1)$   
*обработка элемента данных*  
*переход на  $M2$*   
 .....

## 2.5. Информационные взаимодействия между процессами

### 2.5.1. Понятие информационного канала

Разделяемая память является наиболее быстрым средством межпроцессного информационного взаимодействия, так как при ее применении, во-первых, не требуется применение ВП, а во-вторых, не требуются какие-либо переносы данных внутри ОП. Но возникающие при ее использовании проблемы синхронизации делают необходимым применение семафоров Дейкстры, что существенно ограничивает возможности применения данного средства информационного взаимодействия по следующим причинам. Во-первых, программы всех процессов, разделяющих какие-то области ОП,



должны содержать правильно записанные системные вызовы  $p(S)$  и  $v(S)$ . При этом пропуск хотя бы одного такого вызова или нарушение порядка их записи приведет к ошибкам синхронизации. Во-вторых, каждый взаимодействующий процесс должен сам знать, где находится разделяемая переменная, так как используемый метод синхронизации об этом ничего не говорит. Подобным требованиям могут отвечать только процессы, находящиеся в «дружественных отношениях». Такие отношения обычно имеются у процессов, совместно выполняющих единую параллельную программу. Другие параллельные процессы не находятся в таких отношениях ни между собой, ни по отношению к разделяемой переменной. В этом случае все операции по синхронизации доступа к разделяемой переменной должны быть выведены из прикладной части процесса.

Другие средства информационного обмена, предоставляемые процессам со стороны ОС, не требуют от этих процессов какой-либо синхронизации. В отличие от разделяемой памяти эти методы предполагают перенос данных из области памяти, доступной процессу-источнику в область памяти, доступную процессу-потребителю. Такой перенос выполняется всегда через области памяти (ОП и ВП), находящиеся в ведении ОС и непосредственно не доступные процессам. Иными словами, ОС предоставляет в распоряжение процессов **информационные каналы** (рис. 16).

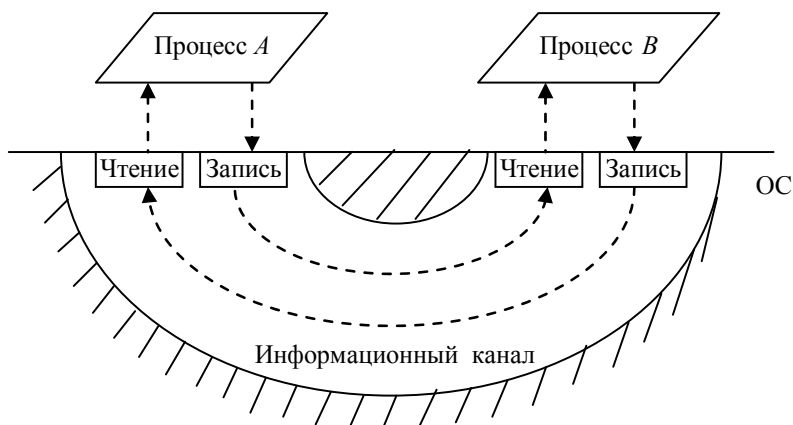


Рис. 16. Взаимодействие процессов через информационный канал

Порция информации, передаваемая или принимаемая процессом за одно обращение к информационному каналу, называется

**сообщением.** Любой информационный канал позволяет подключенным к нему процессам использовать для приема и передачи сообщений два системных вызова:

*ЧТЕНИЕ\_ИНФ.КАНАЛА (...||...),*

*ЗАПИСЬ\_ИНФ.КАНАЛ (...||...).*

При этом каждый информационный канал обладает характеристиками, отражающими его пригодность для передачи сообщений между процессами. Некоторые из этих характеристик:

1) *направление передачи данных.* Информационные каналы делятся на симплексные, полудуплексные и дуплексные. **Симплексный канал** позволяет передавать сообщения только в одном направлении, **полудуплексные** — в обоих направлениях одновременно, а **дуплексный** — в обоих направлениях одновременно;

2) *структурированность передаваемых сообщений.* Будем различать информационные каналы, передающие неструктурированные сообщения и передающие пакеты. **Неструктурированное сообщение** — сообщение, в состав которого не входят какие-то особые байты, используемые информационным каналом. **Пакет** — последовательность байтов, предваряемая специальными служебными байтами, используемыми информационным каналом. Заметим, что данная характеристика относится лишь к интерфейсу между процессом и информационным каналом, так как внутри канала поступающие от процесса неструктурированные сообщения обычно преобразуются в пакеты. Но для процесса подобное преобразование «не заметно»;

3) *устойчивость информационного канала.* Будем различать **виртуальные соединения** — устойчивые каналы, предназначенные для передачи логически связанных последовательностей сообщений, а также **датаграммные каналы**, предназначенные для передачи отдельных, не связанных между собой сообщений, называемых **датаграммами**;

4) *параллельность передаваемых данных.* Будем различать моноканалы и мультиплексные каналы. В любой момент времени **моноканал** содержит по каждому из двух направлений передачи только однотипные передаваемые данные. **Мультиплексный канал** выполняет на своем передающем конце (концах) **мультиплексирование** — объединение данных, передаваемых от нескольких источников. На приемном конце (концах) такой канал выполняет **демультиплексирование** — разделение переданных данных между несколькими приемниками.



### 2.5.2. Обыкновенные программные каналы

В качестве примера рассмотрим одно из средств межпроцессного информационного взаимодействия — *программные каналы*. Наглядно такой канал можно представить в виде трубы, в которую с одной стороны один или несколько процессов направляют (записывают) последовательности байтов, а с другой — один или несколько процессов выбирают (считывают) последовательности байтов. Основные свойства канала:

1) последовательность байтов, помещаемых в канал одним процессом за одну операцию записи, не может быть перемешана с какой-то последовательностью байтов от другого процесса;

2) длины считываемых из канала последовательностей байтов никак не зависят от длин записываемых последовательностей байтов.

Более точное определение: *канал* — специальный файл, запись в который возможна только с одного, а чтение — с другого конца. Причем операции различных процессов с этим файлом синхронизированы. Существуют два типа каналов — обыкновенные и именованные.

*Обыкновенный канал* создается в результате системного вызова:

*СОЗДАТЬ\_КАНАЛ* ( $\parallel i_1, i_2$ ) (на СИ — *pipe*),

где  $i_1$  — программное имя (номер) файла, открытого для записи в канал;  $i_2$  — номер файла, открытого для чтения из канала.

Таким образом, один и тот же файл-канал оказывается открытым дважды и поэтому имеет два номера. Теперь процесс, выдавший данный вызов, может записывать в канал и считывать из него, пользуясь обычными системными вызовами для работы с файлами:

*ЧТЕНИЕ\_ФАЙЛА* ( $i_2, A_b, n_b \parallel$ ) (на СИ — *read*),

*ЗАПИСЬ\_ФАЙЛ* ( $i_1, A_b, n_b \parallel$ ) (на СИ — *write*),

где  $A_b$  — начальный адрес области ОП, в которую производится чтение из файла-канала или из которой производится запись в канал;  $n_b$  — число читаемых (записываемых) байтов.

Конечно, процесс создает канал не для того, чтобы посылать данные самому себе. Напомним, что в однопрограммной системе номера открытых файлов наследуются дочерними программами. В мультипрограммной системе это полезное свойство полностью сохраняется. Поэтому процесс, создавший канал, может использовать его для информационного обмена, как с дочерними процессами, так и с более «младшими» потомками. Во избежание

путаницы рекомендуется, чтобы в каждой программе, использующей канал, этот канал был бы открыт только один раз — или на чтение, или на запись. Поэтому второй файл следует закрыть. В результате канал становится симплексным. Другие свойства этого канала следуют из записанных ранее системных вызовов чтения и записи: устойчивый моноканал для передачи неструктурированных сообщений.

Операция чтения из канала может завершиться одним из следующих вариантов:

1) при чтении меньшего числа байтов, чем находится в канале, операция завершается успешно;

2) при чтении большего числа байтов, чем находится в канале, считывается имеющееся число байтов. Процесс сам должен обрабатывать ситуацию, когда получено меньше, чем заказано;

3) если канал пуст и не открыт на запись ни в одном из процессов, то в результате чтения из канала выдается 0 байтов. Если же канал открыт на запись хотя бы в одном процессе, системный вызов чтения будет заблокирован до появления в канале данных.

Операция записи в канал может завершиться одним из следующих вариантов:

1) при записи меньшего числа байтов, чем имеется свободного места в канале, операция завершается успешно;

2) при записи в канал большего числа байтов, чем имеется в нем свободного места, вызов *ЗАПИСЬ\_ФАЙЛ* блокируется до освобождения требуемого места;

3) если процесс пытается выполнить запись в канал, не открытый ни одним процессом на чтение, то в этот процесс посылается сигнал *SIGPIPE*. По умолчанию сигнал *SIGPIPE* завершает процесс.

Существенным недостатком обыкновенных каналов является то, что они могут использоваться для информационного взаимодействия только между родственными процессами. Для неродственных процессов номера открытых файлов не могут использоваться в качестве идентификаторов совместно используемых каналов. Другим существенным недостатком обыкновенных каналов является то, что они существуют недолго, лишь во время существования использующих их процессов.

Несмотря на перечисленные недостатки, обыкновенные каналы относятся к самым используемым средствам информационного взаимодействия между процессами. В частности такими каналами

соединяются все процессы, команды запуска которых образуют конвейер, набираемый пользователем в командной строке *shell* (см. п. 1.5.3.). В данном случае каналы создает *shell*, дочерними процессами которого и являются процессы, образующие конвейер.

### 2.5.3. Именованные программные каналы

Если требуется обеспечить обмен информацией между «не родственными» процессами, то для этого можно использовать **именованные каналы**, которые реализованы не во всех *UNIX*-системах. Именованный канал имеет имя, аналогичное имени обычного файла. Это имя канал получает при своем создании в результате выполнения процессом системного вызова:

*СОЗДАТЬ\_ИМЕННОЙ\_КАНАЛ* (*I, m*||) (на СИ — *mkfifo*),

где *I* — символьное имя файла-канала; *m* — права доступа к каналу (рассматриваются в подразд. 3.2).

После создания канала он может быть открыт на чтение или на запись в любом числе процессов с помощью вызова:

*ОТКРЫТЬ\_ФАЙЛ*(*I, r*|| *i*) (на СИ — *open*),

где *r* — режим работы процесса с файлом-каналом (чтение и (или) запись); *i* — программное имя (номер) файла, уникальное для данного процесса.

Процесс, открывший канал, может использовать далее вызовы чтения и записи точно так же, как и при работе с обыкновенным каналом.

В отличие от обыкновенных каналов, именованные каналы «видны» для пользователя. С помощью команды *shell* — *mknod* он может создавать новые именованные каналы, чтобы затем использовать их для связи между процессами. Для того чтобы затем подсоединить канал к процессу, достаточно перенаправить стандартный ввод (вывод) процесса на этот файл-канал.

Другие методы информационного взаимодействия между процессами будут рассмотрены нами в подразд. 4.6.

## 2.6. Тупики

### 2.6.1. Причины появления тупиков

Сущность тупика заключается в том, что два или более параллельных процесса могут взаимно заблокироваться из-за своих

динамических запросов на ресурсы так, что без осуществления специальных мероприятий эти процессы не смогут впоследствии перейти в состояние исполнения на ЦП. Следующий пример показывает возможность возникновения тупика.

Пусть каждый из параллельных процессов  $P_1$  и  $P_2$  осуществляет операции записи в файл  $D$ , размещенный на магнитной ленте. В ВС имеется единственный стример (лентопротяжное устройство)  $T$ . Известно, что перед работой с файлом  $D$   $P_2$  использует  $T$  для работы с другим файлом. Выполнение  $P_1$  и  $P_2$  во времени может иметь вид

$P_1$  ..... ВЫДАТЬ_РЕСУРС ( $D, \dots    \dots$ ) M1: ВЫДАТЬ_РЕСУРС ( $T, \dots    \dots$ )	$P_2$  ВЫДАТЬ_РЕСУРС ( $T, \dots    \dots$ ) ..... M2: ВЫДАТЬ_РЕСУРС ( $D, \dots    \dots$ )
--	--

В данном примере  $P_2$  сначала получил  $T$ , а затем  $P_1$  получил  $D$ . Далее  $P_1$  дошел до метки M1, а  $P_2$  достиг метки M2. Сразу же после достижения процессом  $P_2$  метки M2 оба процесса оказываются в тупике. При этом  $P_1$  будет заблокирован по  $T$ , сохраняя за собой  $D$ , тогда как  $P_2$  заблокирован по  $D$ , сохраняя  $T$ .  $P_1$  не может продолжаться, если не продолжается  $P_2$ , и наоборот. Таким образом, процессы сомкнулись «в смертельном объятии». Эту тупиковую ситуацию можно представить в виде графа (рис. 17).

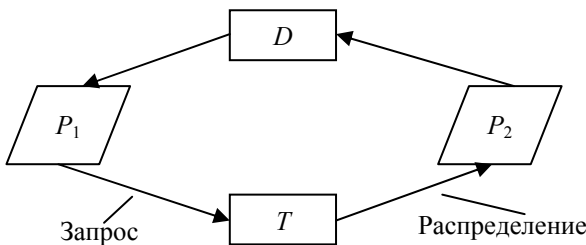


Рис. 17. Процессы  $P_1$  и  $P_2$  находятся в тупике из-за ресурсов  $T$  и  $D$

Следующий пример показывает возможность возникновения тупика из-за одного ресурса. Пусть повторно используемый ресурс  $R$  (например ОП или ВП) содержит  $m$  единиц и распределяется среди  $n$  процессов  $P_1, \dots, P_n$  ( $2 \leq m \leq n$ ). Причем каждый процесс использует элементы ресурса  $R$  в последовательности

$\text{ВЫДАТЬ\_РЕСУРС } (R, \dots || \dots)$

$ВЫДАТЬ\_РЕСУРС(R, \dots || \dots)$   
 $ОСВОБОДИТЬ\_РЕСУРС(R, \dots || \dots)$   
 $ОСВОБОДИТЬ\_РЕСУРС(R, \dots || \dots),$

где каждая из записанных операций действует на единицу  $R$ .

Когда распределены все единицы ресурса, легко может возникнуть тупик, так как процессы, держащие по единице  $R$ , могут навсегда заблокироваться на второй операции  $ВЫДАТЬ\_РЕСУРС$ . Некоторые процессы могут навсегда заблокироваться уже на первой единице. Возможный тупик при  $n = 3, m = 2$  приведен на рис. 18.

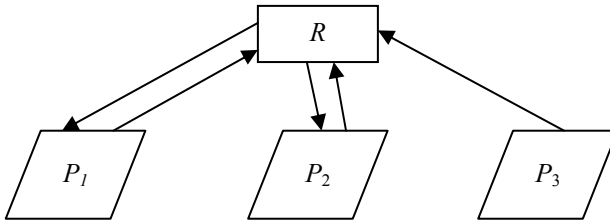


Рис. 18. Процессы  $P_1, P_2, P_3$  находятся в тупике из-за ресурса  $R$

## 2.6.2. Методы предотвращения тупиков

Существуют два основных способа борьбы с тупиками: предотвращение тупиков и ликвидация тупиков. **Методы предотвращения тупиков** исключают возможность возникновения тупика в будущем за счет контроля или на этапе запуска процесса, или на этапе выделения процессу ресурсов. В качестве основного критерия сравнения методов борьбы с тупиками будем использовать **коэффициент эффективности распределения ресурса** — отношение объема полезноиспользуемого ресурса к общему объему распределенного ресурса.

Рассмотрим простой метод предотвращения тупика на этапе запуска процесса. Данный метод, как и многие другие, использует количественную информацию о потребностях процессов в ресурсах, представленную в виде векторов ресурсов и матриц распределения ресурсов среди процессов. Каждый **вектор ресурсов  $R$**  имеет размерность  $m$  — число ресурсов, распределение которых может привести к возникновению тупиков. Сами эти ресурсы обозначим как  $r_1, \dots, r_m$ . Размер **матрицы распределения  $T$**  —  $n \times m$ , где  $n$  — число процессов, существующих в системе. Со



временем число  $n$  может изменяться. В данном методе используются один вектор ресурсов и одна матрица распределения:

1)  $R_{\text{общ}}$  — вектор общего наличия ресурсов. Компонента вектора  $R_{\text{общ } i}$  показывает, сколько всего единиц  $i$ -го ресурса распределяются в системе среди процессов;

2)  $T_{\text{наиб}}$  — матрица наибольших требований процессов к ресурсам. Элемент матрицы  $T_{\text{наиб } ij}$  показывает, какое наибольшее количество  $j$ -го ресурса может потребоваться  $i$ -му процессу. Строка  $T_{\text{наиб } i}$  содержит максимальные потребности  $i$ -го процесса во всех рассматриваемых ресурсах.

Сущность данного метода заключается в том, что новый  $(n+1)$ -й процесс запускается только в том случае, если выполняется условие

$$R_{\text{общ}} \geq T_{\text{наиб}(n+1)} + \sum_{i=1}^n T_{\text{наиб } i} \quad (*)$$

### Пример

Исходное состояние системы имеет вид

$$T_{\text{наиб}} = P_1 \begin{array}{|c|c|c|c|} \hline r_1 & r_2 & r_3 & r_4 \\ \hline 7 & 2 & 4 & 3 \\ \hline \end{array} \quad R_{\text{общ}} = \begin{array}{|c|c|c|c|} \hline 13 & 6 & 8 & 6 \\ \hline \end{array}$$

Допустим, что готовы к запуску процессы  $P_2$  и  $P_3$ , имеющие следующие максимальные потребности в ресурсах:

$$T_{\text{наиб } 2} = P_2 \begin{array}{|c|c|c|c|} \hline r_1 & r_2 & r_3 & r_4 \\ \hline 1 & 3 & 6 & 3 \\ \hline \end{array} \quad T_{\text{наиб } 3} = P_3 \begin{array}{|c|c|c|c|} \hline r_1 & r_2 & r_3 & r_4 \\ \hline 4 & 2 & 1 & 3 \\ \hline \end{array}$$

Для проверки возможности запуска  $P_2$  найдем сумму

$$T_{\text{наиб}} + T_{\text{наиб } 2} = \begin{array}{|c|c|c|c|} \hline 8 & 5 & 10 & 6 \\ \hline \end{array}$$

Так как найденная сумма не отвечает условию (\*), то процесс  $P_2$  запущен быть не может. Проверим теперь возможность запуска процесса  $P_3$ :

$$T_{\text{наиб}} + T_{\text{наиб } 3} = \begin{array}{|c|c|c|c|} \hline 11 & 4 & 5 & 6 \\ \hline \end{array}$$

Найденная сумма отвечает условию (\*), и поэтому процесс  $P_3$  запускается. В результате этого состояние системы принимает вид

$$\begin{array}{l} T_{\text{наиб}} = P_1 \\ P_3 \end{array} \begin{array}{|c|c|c|c|} \hline r_1 & r_2 & r_3 & r_4 \\ \hline 7 & 2 & 4 & 3 \\ 4 & 2 & 1 & 3 \\ \hline \end{array} \quad R_{\text{общ}} = \begin{array}{|c|c|c|c|} \hline 13 & 6 & 8 & 6 \\ \hline \end{array}$$

В рассмотренном методе учитываются максимальные потребности процессов в ресурсах. Независимо от того, когда процесс делает системные вызовы с целью получения ресурсов, фактическое резервирование ресурсов для процесса производится перед его запуском, исходя из его максимальных потребностей. Не имеет значения и время освобождения ресурсов процессом. Следствием этого являются очень низкие значения коэффициентов эффективности распределения ресурсов.

Простейший *метод предотвращения тупиков на этапе выделения ресурсов* заключается в том, что все требуемые ресурсы запрашиваются процессом сразу, в начале своего выполнения. Для того чтобы тупик не возник при этом в начале процесса, на множестве ресурсов ВС должен быть задан порядок, согласно которому ресурсы могут запрашиваться в процессах. Что касается дальнейшего хода процесса, то тупик в принципе невозможен, так как процесс больше не будет обращаться с просьбами выделения ресурсов. Освобождать свои ресурсы процесс может в произвольные моменты времени. Например, перепишем согласно этому методу программы процессов  $P_1$ ,  $P_2$  в примере, изображенном на рис. 17. В результате тупик больше не угрожает выполнению этих процессов:

$P_1$	$P_2$
ВЫДАТЬ_РЕСУРС (T,...  ...)	
ВЫДАТЬ_РЕСУРС (D,...  ...)	
.....	
ОСВОБОДИТЬ_РЕСУРС (T,...  ...)	
ОСВОБОДИТЬ_РЕСУРС (D,...  ...)	
.....	.....
	ВЫДАТЬ_РЕСУРС (T,...  ...)
	ВЫДАТЬ_РЕСУРС (D,...  ...)
	.....

Недостаток данного метода — ресурсы могут быть запрошены задолго до того, как они действительно будут использоваться. Более того, некоторые из запрошенных ресурсов могут и не потребоваться при выполнении процесса. Коэффициенты эффективности распределения ресурсов в данном методе несколько выше, чем у ранее рассмотренного метода контроля запуска процессов. Это обусловлено тем, что процессы, ожидающие получение ресурсов, получают эти ресурсы по мере их освобождения выполняющимися процессами, а не только в моменты завершения этих процессов.

Второй метод безтупикового распределения ресурсов использует следующую количественную информацию:

- 1)  $R_{\text{общ}}$  — вектор общего наличия ресурсов;
- 2)  $R_{\text{своб}}$  — вектор свободных ресурсов. Компонент вектора  $R_{\text{своб } i}$  показывает, сколько единиц  $i$ -го ресурса в данный момент времени свободны;
- 3)  $T_{\text{наиб}}$  — матрица наибольших требований процессов к ресурсам;
- 4)  $T_{\text{рас}}$  — матрица распределения ресурсов между процессами. Элемент матрицы  $T_{\text{рас } ij}$  показывает, какое количество  $j$ -го ресурса на данный момент времени распределено  $i$ -му процессу. Строка  $T_{\text{рас } i}$  содержит объемы всех рассматриваемых ресурсов, полученные  $i$ -м процессом.

Совокупность перечисленных векторов и матриц образует состояние системы. Данное состояние называется **безопасным состоянием**, если все находящиеся в системе процессы могут быть успешно завершены. При этом считается, что для завершения процесса ему необходимо предоставить те объемы ресурсов, которые указаны в матрице  $T_{\text{наиб}}$ . Порядок завершения процессов значения не имеет. Если хотя бы один процесс не может быть завершён, то состояние системы является **опасным состоянием**, из которого, возможно, в будущем система перейдет в состояние тупика.

Сущность данного метода состоит в том, что каждый раз, когда какой-то процесс обращается с просьбой предоставить ему какие-то ресурсы, делается оценка безопасности того состояния, в которое перешла бы система в случае удовлетворения запросов этого процесса. Если новое состояние будет безопасным, то ресурсы процессу выделяются, иначе — процесс блокируется, ожидая дополнительного освобождения требуемых ресурсов.

### Пример

Пусть система находится в исходном состоянии:

	$r_1$	$r_2$	$r_3$	$r_4$					
$T_{\text{наиб}} = P_1$	7	2	4	3	$R_{\text{общ}} =$ <table><tr><td>13</td><td>6</td><td>8</td><td>6</td></tr></table>	13	6	8	6
13	6	8	6						
$P_2$	1	3	6	3					
$P_3$	4	2	1	3					
$P_4$	7	3	0	2					

	$r_1$	$r_2$	$r_3$	$r_4$					
$T_{\text{рас}} = P_1$	3	1	2	1	$R_{\text{своб}} =$ <table><tr><td>2</td><td>1</td><td>2</td><td>3</td></tr></table>	2	1	2	3
2	1	2	3						

$P_2$	1	1	4	1
$P_3$	2	1	0	1
$P_4$	5	2	0	0

Пусть процесс  $P_2$  сделал дополнительный запрос на 1 единицу ресурса  $r_4$ . Тогда новое состояние системы будет следующим:

	$r_1$	$r_2$	$r_3$	$r_4$					
$T_{\text{наиб}} = P_1$	7	2	4	3	$R_{\text{общ}} =$ <table><tr><td>13</td><td>6</td><td>8</td><td>6</td></tr></table>	13	6	8	6
13	6	8	6						
$P_2$	1	3	6	3					
$P_3$	4	2	1	3					
$P_4$	7	3	0	2					

	$r_1$	$r_2$	$r_3$	$r_4$					
$T_{\text{pac}} = P_1$	3	1	2	1	$R_{\text{своб}} =$ <table><tr><td>2</td><td>1</td><td>2</td><td>2</td></tr></table>	2	1	2	2
2	1	2	2						
$P_2$	1	1	4	2					
$P_3$	2	1	0	1					
$P_4$	5	2	0	0					

Для проверки безопасности этого состояния попробуем завершить какой-нибудь процесс, исходя из его максимальных потребностей в ресурсах. Это можно сделать для  $P_3$ , так как выполняется условие  $T_{\text{наиб } 3} - T_{\text{рас } 3} \leq R_{\text{своб}}$ . В результате завершения  $P_3$  система окажется в новом состоянии:

$T_{\text{наиб}} = P_1$	7	2	4	3	$R_{\text{общ}} =$	13	6	8	6
$P_2$	1	3	6	3					
$P_4$	7	3	0	2					

	$r_1$	$r_2$	$r_3$	$r_4$					
$T_{\text{pac}}=P_1$	3	1	2	1	$R_{\text{своб}}=$ <table><tr><td>4</td><td>2</td><td>2</td><td>3</td></tr></table>	4	2	2	3
4	2	2	3						
$P_2$	1	1	4	2					
$P_4$	5	2	0	0					

Нетрудно убедиться, что далее процессы  $P_1$ ,  $P_2$ ,  $P_4$  могут быть завершены в любой последовательности и, следовательно, исходное состояние системы является безопасным.

Если видоизменить задачу так, что  $P_2$  запрашивает не одну, а две единицы ресурса  $r_4$ , то, повторив изложенные рассуждения, нетрудно убедиться, что новое состояние системы будет опасным, и поэтому процесс  $P_2$  должен быть заблокирован.

Так как при оценке состояний системы в данном методе используются не суммарные максимальные потребности процессов в ресурсах, а текущее распределение ресурсов между процессами, то этот метод имеет гораздо лучшие оценки по критериям эффективности использования ресурсов по сравнению с предыдущими методами.

### 2.6.3. Методы ликвидации тупиков

Данные методы не предотвращают возникновение тупика, а уничтожают его тогда, когда он случится. Так как внешне тупик никак себя не проявляет, то прежде чем применить метод ликвидации тупика, этот тупик необходимо обнаружить. Рассмотрим один из методов обнаружения тупиков.

Данный метод использует следующую информацию о системе:

- 1)  $R_{\text{своб}}$  — вектор свободных ресурсов;
- 2)  $T_{\text{рас}}$  — матрица распределения ресурсов между процессами;
- 3)  $T_{\text{тек}}$  — матрица текущих требований процессов на получение дополнительных ресурсов. Элемент матрицы  $T_{\text{тек } ij}$  показывает, какое количество  $j$ -го ресурса на данный момент времени дополнительно требуется  $i$ -му процессу. До тех пор пока процесс не получит этот ресурс, он будет заблокирован.

Сущность рассматриваемого метода состоит в том, что ищется любая допустимая последовательность завершения процессов, существующих в данный момент в системе. При этом принимается допущение, что для завершения процесса ему достаточно выделить те ресурсы, которые он запрашивает дополнительно. После того как процесс завершится, распределенные ему ранее ресурсы добавляются к свободным ресурсам. Если какие-то процессы завершить не удастся, то делается вывод о том, что эти процессы находятся в тупике.

Приведем алгоритм метода.

**Шаг 1.** В матрице  $T_{\text{рас}}$  исключаем все строки, состоящие из одних нулей. Строки с этими же номерами убираем и из  $T_{\text{тек}}$ .

**Шаг 2.** Находим номер строки  $i$  такой, что  $T_{\text{тек } i} \leq R_{\text{своб}}$ . Если такого  $i$  нет, то алгоритм завершился.

**Шаг 3.** Выполняем сложение:  $R_{\text{своб}} := R_{\text{своб}} + T_{\text{рас } i}$ . Удаляем строку  $i$  в  $T_{\text{рас}}$  и  $T_{\text{тек}}$ . Переход на шаг 2.

Тупик имеется тогда и только тогда, когда после завершения алгоритма в матрицах  $T_{\text{рас}}$  и  $T_{\text{тек}}$  остались процессы. Эти процессы и находятся в тупике.

**Пример**

Требуется определить наличие тупика в следующем состоянии системы:

$$T_{\text{рас}} = \begin{matrix} & r_1 & r_2 & r_3 & r_4 \\ P_1 & \begin{matrix} 3 \\ 1 \\ 2 \\ 1 \end{matrix} & \begin{matrix} 1 \\ 1 \\ 1 \\ 2 \end{matrix} & \begin{matrix} 2 \\ 4 \\ 0 \\ 0 \end{matrix} & \begin{matrix} 1 \\ 1 \\ 4 \\ 0 \end{matrix} \end{matrix}$$

$$R_{\text{своб}} = \begin{bmatrix} 1 & 1 & 1 & 2 \end{bmatrix}$$

$$T_{\text{тек}} = \begin{matrix} & r_1 & r_2 & r_3 & r_4 \\ P_1 & \begin{matrix} 0 \\ 1 \\ 0 \\ 2 \end{matrix} & \begin{matrix} 3 \\ 0 \\ 2 \\ 0 \end{matrix} & \begin{matrix} 0 \\ 0 \\ 2 \\ 1 \end{matrix} & \begin{matrix} 0 \\ 1 \\ 0 \\ 0 \end{matrix} \end{matrix}$$

Для поиска тупика применим изложенный выше алгоритм.

**Шаг 1.** В матрице  $T_{\text{рас}}$  нулевых строк нет.

*Итерация 1*

**Шаг 2.** Для  $i = 2$  выполняется  $T_{\text{тек } 2} \leq R_{\text{своб}}$ .

**Шаг 3.** Выполняем сложение:  $R_{\text{своб}} := R_{\text{своб}} + T_{\text{рас } 2}$ . Корректируем матрицы  $T_{\text{рас}}$  и  $T_{\text{тек}}$ . Переход на шаг 2.

*Итерация 2*

На начало данной итерации имеем состояние системы:

$$T_{\text{рас}} = \begin{matrix} & r_1 & r_2 & r_3 & r_4 \\ P_1 & \begin{matrix} 3 \\ 2 \\ 1 \end{matrix} & \begin{matrix} 1 \\ 1 \\ 2 \end{matrix} & \begin{matrix} 2 \\ 0 \\ 0 \end{matrix} & \begin{matrix} 1 \\ 4 \\ 0 \end{matrix} \\ P_3 & & & & \\ P_4 & & & & \end{matrix}$$

$$R_{\text{своб}} = \begin{bmatrix} 2 & 2 & 5 & 3 \end{bmatrix}$$

$$T_{\text{тек}} = \begin{matrix} & r_1 & r_2 & r_3 & r_4 \\ P_1 & \begin{matrix} 0 \\ 0 \\ 2 \end{matrix} & \begin{matrix} 3 \\ 2 \\ 0 \end{matrix} & \begin{matrix} 0 \\ 2 \\ 1 \end{matrix} & \begin{matrix} 0 \\ 0 \\ 0 \end{matrix} \\ P_3 & & & & \\ P_4 & & & & \end{matrix}$$

**Шаг 2.** Для  $i = 3$  выполняется:  $T_{\text{тек } 3} \leq R_{\text{своб}}$ .

**Шаг 3.** Выполняем сложение:  $R_{\text{своб}} := R_{\text{своб}} + T_{\text{рас } 3}$ . Корректируем матрицы  $T_{\text{рас}}$  и  $T_{\text{тек}}$ . Переход на шаг 2.

*Итерация 3*

На начало данной итерации имеем состояние системы:

$$r_1 \quad r_2 \quad r_3 \quad r_4$$

$$T_{\text{рас}} = \begin{matrix} P_1 \\ P_4 \end{matrix} \begin{array}{|c|c|c|c|} \hline 3 & 1 & 2 & 1 \\ \hline 1 & 2 & 0 & 0 \\ \hline \end{array} \quad R_{\text{своб}} = \begin{array}{|c|c|c|c|} \hline 4 & 3 & 5 & 7 \\ \hline \end{array}$$

$$T_{\text{тек}} = \begin{matrix} P_1 \\ P_4 \end{matrix} \begin{array}{|c|c|c|c|} \hline r_1 & r_2 & r_3 & r_4 \\ \hline 0 & 3 & 0 & 0 \\ \hline 2 & 0 & 1 & 0 \\ \hline \end{array}$$

**Шаг 2.** Для  $i = 1$  выполняется:  $T_{\text{тек } 1} \leq R_{\text{своб}}$ .

**Шаг 3.** Выполняем сложение:  $R_{\text{своб}} := R_{\text{своб}} + T_{\text{рас } 1}$ . Корректируем матрицы  $T_{\text{рас}}$  и  $T_{\text{тек}}$ . Переход на шаг 2.

*Итерация 4*

На начало данной итерации имеем состояние системы:

$$T_{\text{рас}} = \begin{matrix} P_4 \end{matrix} \begin{array}{|c|c|c|c|} \hline r_1 & r_2 & r_3 & r_4 \\ \hline 1 & 2 & 0 & 0 \\ \hline \end{array} \quad R_{\text{своб}} = \begin{array}{|c|c|c|c|} \hline 7 & 4 & 7 & 8 \\ \hline \end{array}$$

$$T_{\text{тек}} = \begin{matrix} P_4 \end{matrix} \begin{array}{|c|c|c|c|} \hline r_1 & r_2 & r_3 & r_4 \\ \hline 2 & 0 & 1 & 0 \\ \hline \end{array}$$

**Шаг 2.** Для  $i = 4$  выполняется:  $T_{\text{тек } 4} \leq R_{\text{своб}}$ .

**Шаг 3.** Выполняем сложение:  $R_{\text{своб}} := R_{\text{своб}} + T_{\text{рас } 4}$ . Корректируем матрицы  $T_{\text{рас}}$  и  $T_{\text{тек}}$ . Переход на шаг 2.

*Итерация 4*

Так как в матрице  $T_{\text{рас}}$  не осталось строк, алгоритм завершился. Тупик отсутствует.

Нетрудно убедиться, что если взять начальный вектор свободных ресурсов

$$R_{\text{своб}} = \begin{array}{|c|c|c|c|} \hline 1 & 1 & 1 & 0 \\ \hline \end{array},$$

то применение данного алгоритма завершится на первой же итерации. Это означает, что все четыре процесса  $P_1, P_2, P_3, P_4$  находятся в тупике.

Положительной особенностью данного метода является то, что для его применения не требуется дополнительная информация от процессов в виде матрицы  $T_{\text{наиб}}$ . Вся требуемая информация может быть получена от подпрограмм ОС, занимающихся распределением ресурсов. Кроме того, данный метод не ухудшает оценки эффективности использования ресурсов, так как никак не влияет ни на получение процессами ресурсов, ни на освобождение этих ресурсов. Впрочем, сравнение по данному критерию с предыдущими методами бессмысленно, так как данный метод не предотвращает возникновение тупика, а лишь сообщает о его наличии.

После того как выявлено наличие тупика, необходимо его устранить. Основные подходы к решению этой задачи:

1) уничтожение всех процессов, находящихся в тупике. Несмотря на «жестокость», это наиболее распространенный подход;

2) последовательное уничтожение процессов, оказавшихся в тупике. После уничтожения очередного процесса применяется алгоритм определения наличия тупика. При отсутствии тупика уничтожение процессов прекращается;

3) «откат» каждого из процессов, оказавшихся в тупике, в какое-то прежнее состояние. Для применения этого метода необходимо время от времени сохранять в ВП текущие состояния процессов;

4) перераспределение тех ресурсов, дефицит которых привел к тупику. Ресурсы отнимаются у одних процессов и отдаются другим, находящимся в тупике. Процессы, у которых отняли соответствующие ресурсы, остаются заблокированными по этим ресурсам.

Часто методы ликвидации тупиков реализуются не в виде алгоритмов соответствующих подпрограмм ОС, а в виде последовательности действий пользователя ВС, который может прекращать выполнение процессов в тупике с помощью языка управления ОС.

В заключение отметим, что в настоящее время не существует универсального метода борьбы с тупиками.



### 3. ПОДДЕРЖКА МНОГОПОЛЬЗОВАТЕЛЬСКОЙ РАБОТЫ И СТРУКТУРА СИСТЕМЫ

#### 3.1. Управление доступом пользователя в систему

Поддержка (то есть обеспечение) многопользовательской работы системы выполняется в основном на программном уровне, то есть самой ОС. Для этого в рамках мультипрограммной системы выполняются дополнительные системные программы, обеспечивающие такую поддержку. Наиболее общая задача по обеспечению многопользовательской работы заключается в том, чтобы оградить работу в системе каждого конкретного пользователя от воздействий со стороны других пользователей.

Многие однопользовательские системы, например современные *Windows*, предусматривают работу многих пользователей, но не в параллельном, а в последовательном режиме. Поддержка последовательной работы пользователей предусматривает реализацию некоторых проектных решений, полученных при разработке многопользовательских систем, наиболее ярким представителем которых является система *UNIX*.

Изучим, как рассматривает своего пользователя ОС. Будучи «бюрократам», любая ОС рассматривает пользователя не по его многочисленным достоинствам и недостаткам, а исходя из нескольких формальных показателей (атрибутов), зарегистрированных в системе. Первым из этих признаков, как всегда, является имя. Любой пользователь имеет два имени: символьное имя и номер, причем каждое из этих имен уникально в пределах всей системы. **Символьное имя пользователя** предназначено для использования самими пользователями (при общении с ОС), а **номер пользователя**, как всегда, используется самой системой. Например, обязательным пользователем в любой системе является **суперпользователь** или **администратор системы**, имеющий в системе неограниченные права. Имена этого пользователя: *root* и 0.

Все пользователи системы делятся ее администратором на пересекающиеся **группы пользователей** (не путать с группами процессов из п. 2.4.2). За счет включения пользователя в группу он наделяется дополнительными правами, которыми обладают все члены группы. Каждая группа имеет два имени, уникальных по

всей системе, — *символьное имя группы* и *номер группы*. Например, административная группа может иметь имена: *root* и 0. Имеющееся здесь совпадение имен группы и пользователя вполне допустимо, так как каждое из имен должно быть уникально только в одной группе объектов (пользователи или группы). Каждый конкретный пользователь может быть одновременно членом одной, двух или большего числа групп. Группа пользователей, первая для данного пользователя, является для него *первичной группой*.

Еще один атрибут пользователя — пароль. *Пароль* представляет собой символьную строку (без пробелов) и используется ОС для идентификации пользователя в самом начале его сеанса с системой. Подобная идентификация пользователя называется *аутентификацией*. Пароль выбирается самим пользователем с учетом требований, существующих в конкретной системе. Хороший пароль должен отвечать двум требованиям: 1) легко вспоминаться; 2) трудно подбираться. Рассмотрим теперь вопрос о том, где находится в системе пароль пользователя и другие его атрибуты.

Любой модуль системы, аппаратный или программный, достойный внимания со стороны ОС, обязательно имеет блок управления. *Блок управления* или *дескриптор* — «паспорт» модуля, содержащий его атрибуты. При этом каждый блок управления может рассматриваться как структура данных, полями которой являются атрибуты модуля. Если в системе несколько однотипных модулей, то их блоки управления объединяются в единую таблицу в качестве ее строк. Некоторые из блоков управления реализуются в виде не одной, а нескольких структур данных, каждая из которых может являться строкой в одной из таблиц ОС. На протяжении данного пособия нам встретится много различных блоков управления.

Пользователи ВС конечно не являются ее модулями, но, представляя для ОС значительный интерес, имеют свои блоки управления. Каждый такой блок представляет собой логическую запись в файле */etc/passwd*, а весь этот файл может рассматриваться как таблица, содержащая сведения обо всех пользователях системы. Полями указанной логической записи являются:

1) символьное имя пользователя;

2) пароль пользователя в закодированном виде. Алгоритм кодирования пароля известен, но он не позволяет сделать обратный переход для получения самого пароля. В современных *UNIX*-системах с целью повышения безопасности данное поле содержит

не пароль, который хранится в другом файле, а лишь символ «x» или «!»;

3) номер пользователя;

4) номер первичной группы пользователя;

5) комментарии, содержащие настоящее имя пользователя и, возможно, некоторые другие сведения о нем. Данное поле используется некоторыми утилитами;

6) начальный каталог пользователя. Этот каталог является корнем поддерева, принадлежащего данному пользователю в файловой структуре системы;

7) имя исполняемого файла программы, которую ОС запустит на выполнение в качестве первого ИК, обслуживающего данного пользователя. Обычно это один из *shell*, но можно указать любую другую обрабатывающую программу, не забывая о том, что завершение данной программы означает завершение текущего сеанса работы пользователя в системе.

Файл */etc/passwd* по своей форме является обыкновенным текстовым файлом, доступным для чтения всем пользователям системы. Но вносить изменения в этот файл может только администратор. Только он может вносить изменения и в текстовый файл */etc/group*, содержащий перечень всех групп пользователей с указанием членов каждой группы.

Теперь рассмотрим порядок записи атрибутов пользователя во времени. Во-первых, прежде чем пользователь начнет свой первый сеанс работы в системе, администратор запишет его атрибуты, включая какой-то первоначальный пароль, в файл */etc/passwd*.

Во-вторых, сеанс работы пользователя в системе начинается с включения терминала. Согласно подразд. 2.2, после включения терминала системный процесс *init* порождает процесс *shell*, обслуживающий данного пользователя. Но это упрощенная схема. На самом деле *init* сначала порождает процесс *getty*, ожидающий включения терминала. После этого включения программа процесса заменяется: вместо *getty* загружается программа *login*, которая запрашивает у пользователя имя и пароль. Если имя зарегистрировано в файле */etc/passwd* и пароль назван правильно, то *login* загружает вместо себя тот *shell* (или его заменитель), который указан в последнем поле записи файла */etc/passwd*, которая соответствует данному пользователю.

Таким образом, программа одного и того же процесса заменяется дважды на другую программу: *getty* → *login* → *shell*. Это

не трудно обеспечить, так как любая программа может перекрыть себя другой программой, используя соответствующий системный вызов (рассматривается в подразд. 4.2). Кроме того, отметим, что *login* сравнивает не сами пароли, а их закодированные варианты. В ходе своей работы в системе пользователь может заменить пароль, используя утилиту *passwd*.

Рассмотрим теперь, как осуществляет доступ в систему удаленный пользователь, соединенный с *UNIX*-системой не с помощью локального терминала, а с помощью линии связи или даже целой сети передачи данных. Для обеспечения такого доступа изложенная выше схема должна быть скорректирована (рис. 19).

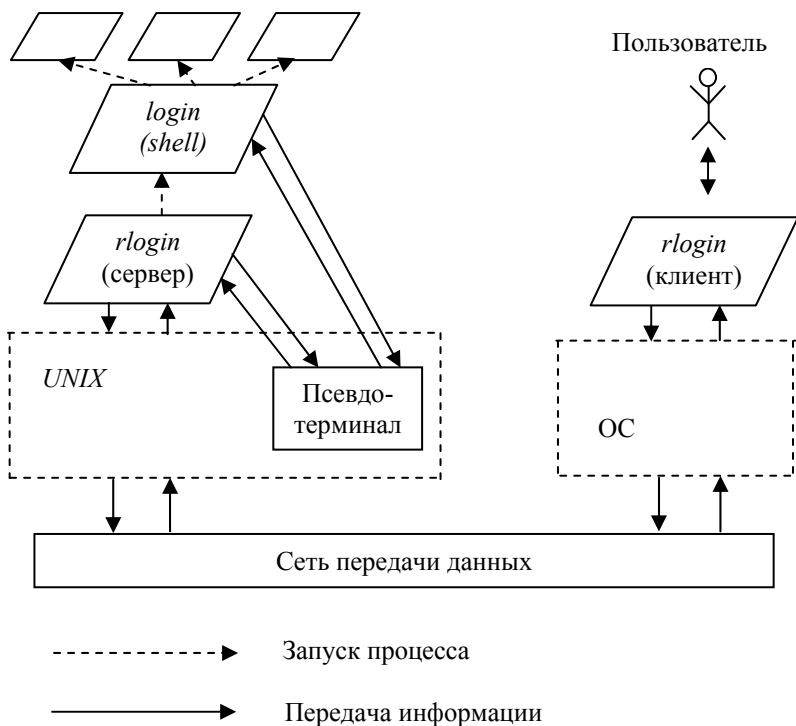


Рис. 19. Доступ удаленного пользователя в *UNIX*-систему

Удаленный пользователь запускает на своей локальной ЭВМ программу *rlogin* (клиент), которая с помощью своей ОС (не обязательно *UNIX*) посылает по сети в программу *rlogin* (сервер) запрос на вход в систему *UNIX*. Процесс *rlogin* (сервер) является

дочерним процессом процесса *init*, порождаемым им аналогично процессу *getty*, но предназначен для обслуживания не локальных, а удаленных запросов на вход в систему. Так как процессу *rlogin* (сервер) терминал не нужен, то он создается процессом *init* как процесс-демон. После поступления каждого удаленного вызова *rlogin* (сервер) создает очередной экземпляр заменителя терминала — псевдотерминал и порождает дочерний процесс *login*.

**Псевдотерминал** — программный модуль ОС, предназначенный для передачи информации между двумя процессами, а также для некоторого преобразования этой информации. Это преобразование нужно для того, чтобы один из процессов, соединяемых псевдотерминалом, выполнялся бы так, как будто он взаимодействует с настоящим терминалом. В нашем случае таким «обманываемым» процессом является *login*. Перед его созданием процесс-отец *rlogin* (сервер) производит перенаправление стандартных ввода и вывода на псевдотерминал. Такое перенаправление выполняется очень просто: достаточно открыть псевдоустройство на ввод под программным именем 0, а на вывод — под именем 1. Так как *login* является дочерним процессом *rlogin* (сервер), то он наследует эти имена псевдоустройства и будет пользоваться ими так, как будто речь идет о настоящем терминале.

После того как *login* начнет выполняться, он будет действовать как обычно, то есть проверит имя и пароль пользователя, а затем загрузит вместо себя *shell*. При этом весь вывод *login* (а потом *shell*) будет поступать в псевдотерминал, из него в *rlogin* (сервер), а затем по сети в *rlogin* (клиент). Ввод *login* выполняется по этому же пути, но в обратную сторону. Так как все потомки процесса *shell* наследуют открытые для него файлы (и устройства), то запущенные в оперативном режиме, они пользуются псевдотерминалом аналогично *shell*.

Поясним теперь, как псевдотерминал выполняет функции настоящего терминала. Допустим, что удаленный пользователь захотел оказать воздействие на свои процессы, выполняемые под управлением *UNIX*. Для этого он нажимает одну из комбинаций клавиш, рассмотренных в п. 2.4.2. Получив от *rlogin* (сервер) код этой комбинации, псевдотерминал распознает ее и выдает тот сигнал, который соответствует этой комбинации клавиш. Таким образом, удаленный терминал действует на *UNIX*-систему точно так же, как и локальный терминал.

После того как пользователь вошел в систему, ОС должна обеспечивать защиту результатов его работы от воздействия других пользователей. Так как эти результаты есть информация, то речь идет о защите информационных ресурсов пользователя. При этом защита информации в ОП выполняется аппаратными и программными средствами поддержки мультипрограммирования. Что касается защиты информации на устройствах ВП, то эта важнейшая функция поддержки многопользовательской работы выполняется исключительно самой ОС.

### 3.2. Защита файлов

Каждый пользователь, зарегистрированный в системе, имеет в ней «информационный след» — совокупность информации, связанной с данным пользователем. Этот информационный след схематично показан на рис. 20. На этом рисунке пунктиром показана краткосрочная информация, связанная с выполнением процессов. Долгосрочная информация хранится на устройствах ВП в виде файлов.

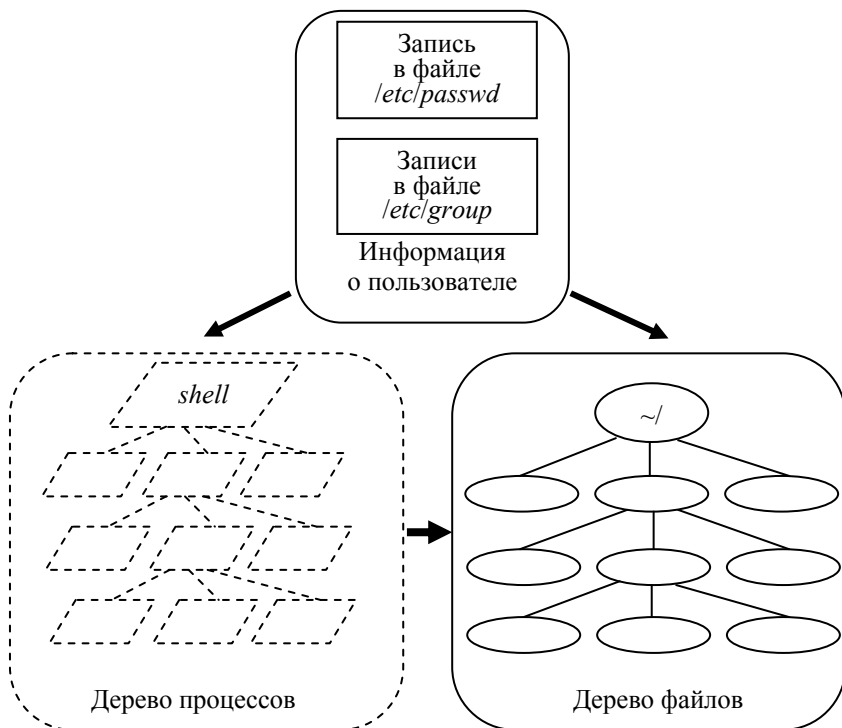


Рис. 20. Информационный след пользователя в системе

ОС имеет дело не с самими процессами и файлами, а с их блоками управления. Структура этих дескрипторов будет рассмотрена нами в других разделах, а пока лишь заметим, что кроме прочей информации они обязательно содержат имя пользователя-владельца, а каждый дескриптор файла содержит еще и имя пользователя-группы. Логика появления этих имен в дескрипторах следующая. Во-первых, дескриптор корня дерева процессов (процесса *shell*) получает имя пользователя сразу же после завершения входа пользователя в систему. Все порожденные им процессы наследуют это имя и передают его своим дочерним процессам. Каждый файл получает имя пользователя-владельца от процесса, создавшего этот файл. Кроме того, при создании файла из записи файла */etc/passwd*, соответствующей пользователю, переписывается имя его первичной группы.

Пользователь-владелец и пользователь-группа могут быть изменены с помощью команд *shell* — ***chown*** и ***chgrp***. В качестве первого параметра этих команд записывается соответственно символьное имя нового пользователя или имя новой группы. В качестве других параметров записываются имена тех файлов, для которых производится замена. В некоторых UNIX-системах команды ***chown*** и ***chgrp*** могут выдаваться прежним владельцем файла, а в других — только суперпользователем.

При создании файла в его дескриптор записывается не только имя пользователя-владельца и его первичной группы, но и права доступа к файлу трех разновидностей его пользователей: 1) пользователь-владелец; 2) пользователь-группа; 3) остальные пользователи системы. ***Правами доступа к файлу*** называются формы доступа к файлу, разрешенные для этого пользователя. Суперпользователь всегда имеет неограниченные права доступа. Права доступа остальных пользователей могут быть ограничены.

Различают основные и дополнительные формы доступа. Перечислим основные формы доступа к файлу:

- 1) чтение файла — ***r***;
- 2) запись в файл — ***w***;

3) выполнение файла — ***x***. Выполнение файла означает, что этот файл является или исполняемым файлом программы и может быть выполнен после загрузки в ОП как машинная программа, или файл является командным (скриптом).

Например, суперпользователь имеет по отношению к любому файлу неограниченные права доступа — *гix*.

Применительно к каталогу перечисленные формы доступа имеют особенный смысл. Операция «чтение каталога» означает получение перечня имен файлов, записанных в данном каталоге. Получение никакой другой информации о файлах эта форма доступа не предполагает. Например, форма доступа *r* достаточна для того, чтобы выполнить команду *shell* — *ls* без флагов, но не достаточна для выполнения этой же команды с флагом *-l*, так как наличие этого флага предполагает использование информации, хранящейся не в каталоге, а в блоках управления файлами.

Для получения дополнительной информации о дочерних файлах каталога необходимо иметь к нему форму доступа *x* (выполнение). Эта же форма необходима для того, чтобы сделать каталог текущим (с помощью команды *cd*). Более того, чтобы сделать каталог текущим, необходимо иметь форму доступа *x* ко всем каталогам в имени-пути данного каталога, иначе в данный каталог мы не попадем.

Очевидно, что для создания файла (в том числе и каталога) мы должны иметь форму доступа *w* (запись) к родительскому каталогу. Эта же форма доступа требуется и при удалении файла. Может показаться странным, но для удаления файла какие-то формы доступа к нему самому не требуются. При этом если форма доступа *w* к файлу есть, команда удаления файла *rm* не выдает на экран дополнительного запроса подтвердить удаление файла. Иначе — такой запрос выдается. Применительно к каталогу сочетание прав *r* и *x* позволяет получить интересный эффект, называемый «темным каталогом». Он заключается в том, что для всех пользователей, кроме владельца, запрещено чтение каталога, но разрешено его выполнение. Поэтому только «посвященные» пользователи, знающие о наличии файла в каталоге, имеют возможность работать с ним.

Для того чтобы узнать права доступа к файлам текущего каталога, воспользуемся командой *ls* с флагом *-l*.

### Пример

```
$ ls -l
$ -rw-r--r-- 1 vlad gro 1120 Dec15 15:03 abc.txt
  d-rwxr-xr-- 2 vlad gro 11500 Aug28 17:38 mac
$
```

Информация о файле, выданная *UNIX*, состоит из 8 колонок. В первой позиции первой колонки записан тип файла (*d* — каталог;



«-» — прочий файл). В остальных девяти позициях первой колонки записаны права доступа к файлу: первые три символа — права доступа пользователя-владельца; вторые три символа — права доступа пользователя-группы; последние три символа — права доступа прочим пользователям.

В приведенном примере для первого файла владелец имеет права доступа *rw* (понятно, что «выполнять» текстовый файл бессмысленно), а все остальные пользователи, включая и членов группы, могут только читать информацию из файла. Что касается второго файла (это каталог), то относительно его владелец обладает всеми правами доступа; член группы — всеми правами, за исключением записи, а остальные пользователи могут только читать файл.

Что касается остальных колонок информации о файлах, то в них содержится:

1) число жестких связей — количество каталогов, в которых «зарегистрирован» данный файл;

2) имя пользователя-владельца файла;

3) имя пользователя-группы;

4) длина файла в байтах;

5) дата последнего изменения файла;

6) время последнего изменения;

7) имя файла.

Владелец файла может не только вывести права доступа на экран, но и внести в них изменения. Для этого используется команда *shell* — ***chmod***. Ее общий формат:

```

chmod  | u | | + | | r |
      | g | | - | | w | <имя файла> [<имя файла...>]
      | o | | = | | x |
      | a |

```

где «*u*» — владелец; «*g*» — группа; «*o*» — остальные пользователи; «*a*» — все пользователи; «+» — добавить; «-» — удалить; «=» — присвоить.

### Пример

Следующая команда лишает членов пользователя-группы файла *abcfile* права на запись и выполнение этого файла:

```
chmod g-wx abcfile
```

Проверка прав доступа к файлу выполняется в момент открытия файла программой процесса. Если пользователь-владелец процесса одновременно является и владельцем файла, то принадлеж-

ность пользователя к группе не проверяется. Если пользователь процесса не является владельцем файла, то проверяется его принадлежность к пользователю-группе. В случае принадлежности к группе пользователь процесса наделяется правами группы, иначе — правами доступа к файлу для остальных пользователей.

Кроме основных форм доступа к файлам (и соответствующих прав) существуют дополнительные формы доступа, каждая из которых может рассматриваться как модификатор формы доступа  $x$  (выполнение):

1)  $s$  (для пользователя-владельца файла) — динамически задать имя пользователя процесса таким, каково оно у владельца файла. Данная форма доступа, как и следующая, имеет смысл только для выполнимых файлов. Такая форма доступа, например, позволяет пользователю производить замену своего пароля, запуская утилиту *passwd*. Владелец исполняемого файла этой утилиты является суперпользователем, который одновременно является и владельцем файла паролей. Так как этот исполняемый файл предоставляет своему владельцу (суперпользователю) права доступа  $r$ ,  $x$  и  $s$ , а всем остальным пользователям — права доступа  $r$  и  $x$ , то при его запуске процесс пользователя получает временно в качестве имени владельца имя суперпользователя и поэтому может выполнять запись в файл паролей. Естественно, что программа *passwd* написана так, что пользователь может заменить только свой пароль, не влияя на пароли других пользователей;

2)  $s$  (для пользователя-группы файла) — динамически задать имя пользователя-группы процесса таким, каково оно у пользователя-группы файла. Данная форма доступа похожа на предыдущую, но используется реже;

3)  $t$  — в настоящее время используется только для каталогов, разрешая пользователю удалять только те файлы, для которых он или является владельцем, или ему разрешена в них запись. Например, такая форма доступа назначается каталогу */tmp*, предназначенному для хранения временных файлов всех пользователей. Имея форму доступа  $w$  (запись) к этому каталогу, пользователь тем не менее не может удалить файлы других пользователей.

Для просмотра дополнительных прав доступа по-прежнему используется команда *ls* с флагом *-l*. При наличии дополнительного права доступа для какого-то пользователя (владелец, группа или остальные пользователи) вместо права  $x$ , которое в этом случае обязательно, записывается его модификатор  $s$  или  $t$ .

После того как мы рассмотрели основные методы поддержки мультипрограммирования и многопользовательской работы, перейдем к рассмотрению укрупненной структуры ОС.

### 3.3. Укрупненная структура операционной системы

На рис. 21 приведена укрупненная структура операционной системы *UNIX*.

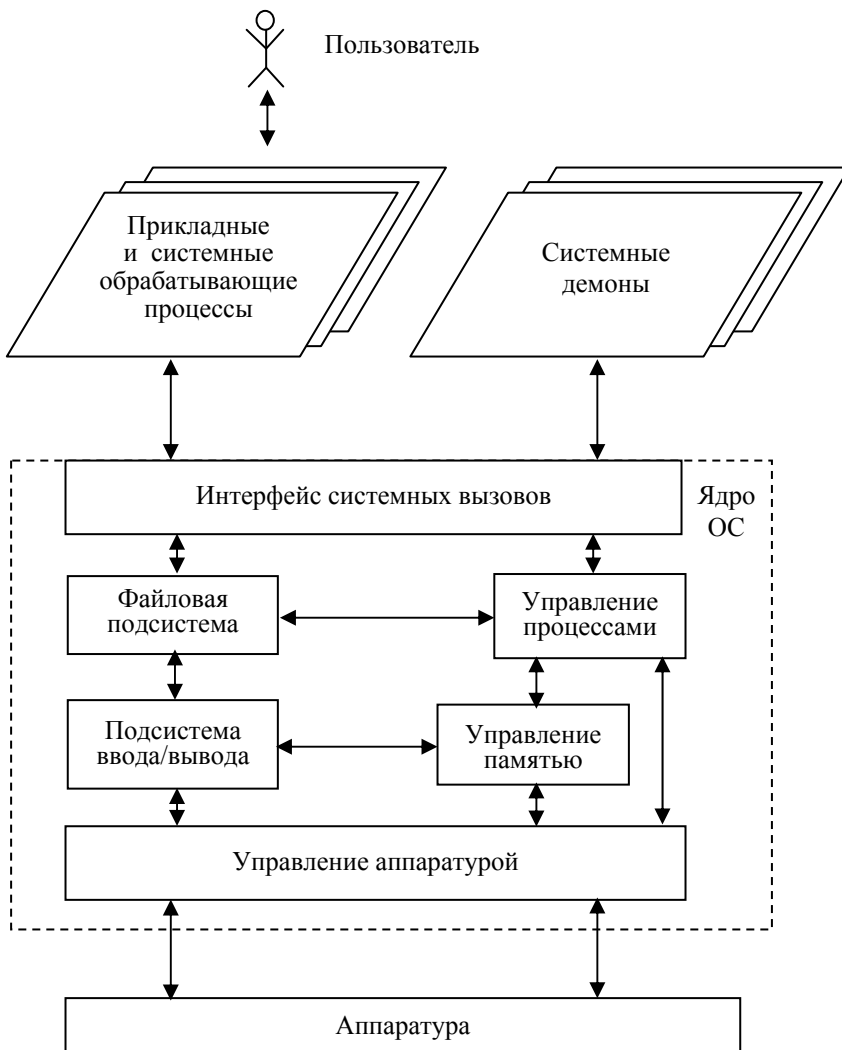


Рис. 21. Укрупненная структура ОС

Основная часть ОС, выполняемая в привилегированном режиме ЦП, называется **ядром**. Подсистемы ядра выполняют управление основными объектами ВС: процессами, файлами, периферийными устройствами, центральным процессором и оперативной памятью. Часть программ ОС реализована вне ядра. Сюда относятся интерпретаторы команд ОС (*shell*), а также системные демоны (*init*, *getty*, *rlogin* и т.д.).

Напомним, что *shell*, как и другие обрабатывающие системные и прикладные процессы, имеет управляющий терминал, позволяющий пользователю влиять на выполнение своих программ. Демоны, наоборот, не имеют управляющего терминала, что позволяет им жить «вечно» (до перезагрузки системы).

Все обрабатывающие процессы, а также демоны пользуются услугами ядра, используя для обращения к нему **системные вызовы**. По своей форме каждый системный вызов представляет собой машинную команду передачи управления — *call*, *int* или *jmp*. В однопрограммной системе (например в *MS-DOS*) все системные вызовы выполняются командой *int* (команда прерывания). В мультипрограммных системах роль этой команды скрыта от прикладных программ благодаря интерфейсу системных вызовов.

**Интерфейс системных вызовов** содержит процедуры, доступные для вызова программами, расположенными вне ядра, а также подпрограммы, расположенные внутри ядра и не доступные для такого вызова. Первые процедуры принято называть «заглушками», и обычно эти процедуры объединены в *DLL*. Для вызова «заглушки» прикладная программа использует обычную команду *call* (вызов процедуры). После того как требуемая «заглушка» вызвана, она сама переключает ЦП в привилегированный режим, используя команду *int* или *jmp*, и одновременно запускает скрытую подпрограмму ядра, выполняющую координационные функции. Суть этих функций сводится к тому, что в соответствии с принятыми параметрами вызова (как и любая другая, процедура «заглушка» получает параметры) определяется последовательность внутренних процедур ядра, подлежащих исполнению.

После того как последовательность внутренних процедур ядра завершена, «заглушка», возможно, возвратит управление в прикладную программу. Такой возврат выполняется в том случае, если выполняемый системный вызов является **синхронным**. При **асин-**

**хронном системном вызове** прикладная программа не должна ждать завершения запрошенной работы ядра, и поэтому сразу же после запуска скрытой подпрограммы ядра «заглушка» возвращает управление в вызвавшую ее программу.

Обратим внимание, что функции подсистемы ввода/вывода, а также функции управления аппаратурой не доступны для непосредственного использования в прикладных программах. Это принципиально отличает мультипрограммную систему от однопрограммной, в которой все эти функции доступны в любой программе.

Рассмотренная укрупненная схема присуща не только *UNIX*, но с некоторыми отличиями и другим мультипрограммным системам. Основное различие здесь заключается в распределении функций между ядром и внешними процессами. В другой модели операционных систем, называемой моделью «клиент-сервер», значительная часть функций ядра передается внешним процессам, то есть демонам. Теперь эти демоны становятся серверами, предназначенными для обслуживания прикладных процессов (клиентов). Для получения системного обслуживания процесс-клиент посылает сообщение серверу, используя ядро как «почту». После того как требуемая системная функция выполнена, сервер посылает ответное сообщение, также пользуясь «почтовыми» услугами ядра. При использовании данной модели ядро выполняет лишь наиболее часто используемые функции, к которым относятся, например, передача сообщений между процессами или обработка прерываний. Такое ядро принято называть **микроядром**. Заметим, что реализация данной модели ОС возможна без изменения программного интерфейса системных вызовов. Для этого достаточно поручить посылку сообщения для сервера ранее упомянутой процедуре-«заглушке». Эта же процедура может выполнять прием ответных сообщений сервера к клиенту, который таким образом не заметит замену модели ОС.

Другое принципиальное отличие ядра *UNIX* от некоторых других ОС состоит в том, что ядро разделяется внешними параллельными процессами последовательно. То есть до тех пор пока ядро занято обслуживанием какого-то конкретного процесса и это обслуживание не завершится, оно не начнет обслуживание никакого другого процесса. При этом на время системного обслуживания программа ядра «подсоединяется» к прикладной программе процесса, не образуя никакого нового процесса ядра. Реализация такого подхода обеспечивает целостность системных данных ядра.

Заметим, что ядро *UNIX* имеет несколько своих внутренних процессов ядра, но эти процессы выполняют общесистемные функции, не принимая непосредственного участия в выполнении системных вызовов. Для сравнения: в *Windows-NT* основное системное обслуживание прикладных процессов выполняют процессы ядра, которые, таким образом, являются серверами. Подобная видоизмененная модель «клиент-сервер» требует меньше переключений ЦП из одного режима в другой по сравнению с классической моделью «клиент-сервер».

Модель «клиент-сервер» применяется не только для организации операционных систем, но и для реализации распределенных прикладных программ. Примером такой распределенной программы является программа *rlogin* (см. подразд. 3.1). Для выполнения распределенных программ требуется помощь со стороны ОС, которая для этого должна быть сетевой. Рассмотренная выше общая структура ОС может быть использована и для реализации сетевой операционной системы.

### 3.4. Структура сетевой операционной системы

Краткое определение: **сеть передачи данных** — совокупность ЭВМ, связанных каналами передачи данных. В общем случае такая сеть нужна для того, чтобы прикладная программа, выполняющаяся на одной ЭВМ, могла использовать ресурсы (аппаратные, программные и информационные) другой ЭВМ. При этом под прикладными программами будем понимать не только собственно прикладные программы, но и системные обрабатывающие программы, например утилиты (вспомним, что ОС не отличает системные обрабатывающие программы от обычных прикладных программ).

Любое сетевое взаимодействие предполагает участие двух удаленных друг от друга программ: клиентской и серверной. **Клиентская программа** делает запрос на получение удаленного ресурса, а **серверная программа** выполняет получение и использование запрашиваемого ресурса. Далее будем называть ЭВМ, содержащую клиентскую программу, **узлом-клиентом**, а ЭВМ, содержащую серверную программу, **узлом-сервером**. В состав любого из этих узлов входит **сетевая операционная система**, которая может рассматриваться как расширение обычной локальной ОС.

Прикладные программы, выполняемые в сети, можно разделить на локальные и распределенные. Распределенные программы разрабатываются специально для выполнения в сети. Примером такой программы является рассмотренная в подразд. 3.1 программа *rlogin*. Другими примерами являются распределенные СУБД, а также сетевые игры. Каждая такая программа содержит серверную часть и одну или несколько клиентских частей, расположенных в различных узлах сети. В отличие от локальной реализации модели «клиент-сервер», удаленные модули распределенной программы находятся на разных ЭВМ, и, следовательно, для передачи информации между ними не могут использоваться общие области ОП. Единственным способом информационного обмена между ними является использование информационного канала, выполняющего передачу сообщений (см. подразд. 2.5).

Так как клиентская и серверная части распределенной программы выполняют общую работу, то эти два модуля должны «общаться» на понятном им обоим языке. Иными словами, они должны выполняться согласно общему алгоритму взаимодействия. Вспомним, что алгоритм взаимодействия двух соседних модулей называется *интерфейсом*. Так как удаленные клиент и сервер не имеют общей границы, то понятие интерфейса между ними не имеет смысла. Это понятие заменяется понятием протокола. **Протокол** — алгоритм взаимодействия модулей, удаленных друг от друга. Протокол взаимодействия клиентской и серверной частей распределенной программы будем называть далее **прикладным протоколом**. Подобно интерфейсу, протокол имеет статическую и динамическую части. **Статика протокола** описывает состав и структуру информационных конструкций, которыми могут обмениваться партнеры по диалогу. **Динамика протокола** регламентирует порядок выдачи этих конструкций относительно друг друга и, возможно, во времени. В зависимости от реализации прикладного протокола в распределенной программе, существуют два основных подхода к разработке таких программ.

В первом из подходов статика и динамика прикладного протокола распределенной программы реализуются ее разработчиками без использования какой-то особой помощи со стороны системных программ. Единственная дополнительная помощь, которая требуется при выполнении распределенной программы от сетевых ОС — предоставление ей информационного канала, выполняющего передачу удаленных сообщений. Допустим, что такой канал предоставляет модуль ОС — **транспорт сообщений** (рис. 22). Реализация

данного модуля будет рассмотрена значительно позже, а пока для нас важен лишь тот факт, что модуль транспорта обеспечивает передачу сообщения требуемому удаленному программному процессу.

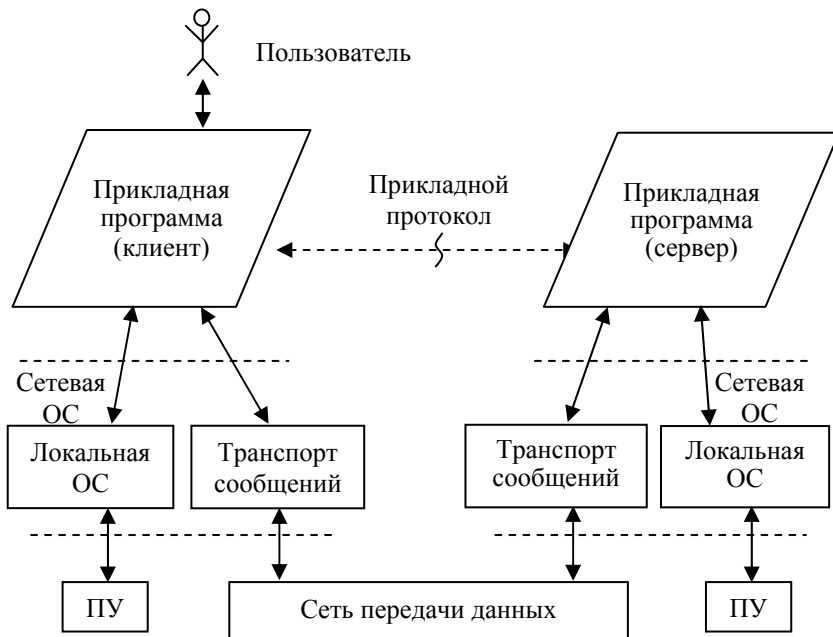


Рис. 22. Выполнение распределенной программы в сети

Второй подход к реализации распределенной программы называется **удаленным вызовом процедур** — **RPC** (*remote procedure call*). В этом подходе прикладная программа освобождена от выполнения прикладного протокола за счет того, что клиентская часть программы выполняет нужные ей подпрограммы серверной части, используя обычные локальные вызовы процедур с помощью команды *call*. Это не означает, что прикладной протокол теперь вовсе не нужен. Просто этот протокол выносится за пределы прикладной программы и выполняется вызываемыми ею системными подпрограммами (рис. 23).

При проектировании распределенной программы производится распределение ее подпрограмм между клиентом и сервером. Все процедуры сервера, которые могут потребоваться клиенту, нумеруются, и для каждой из них создаются две процедуры-заглушки,



одна из которых подсоединяется к программе-клиенту, а другая — к серверу. Такое связывание выполняется или статически, или динамически. В первом случае заглушка является модулем обычной библиотеки, а во втором — модулем *DLL*.

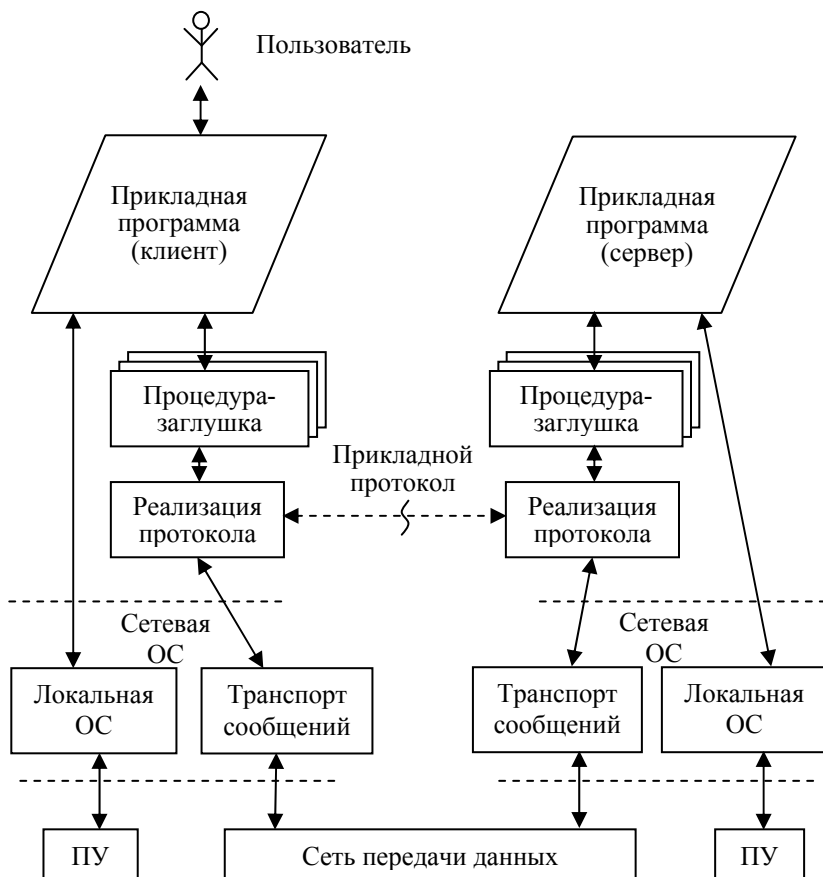


Рис. 23. Распределенная программа с удаленным вызовом процедур

Допустим, что в программе-клиенте требуется вызвать какую-то удаленную процедуру, принадлежащую программе-серверу. Тогда клиент вызывает свою локальную процедуру-заглушку, имеющую точно такой же интерфейс (имя процедуры и состав параметров), как и настоящая удаленная процедура. Далее заглушка представляет полученные ею параметры в стандартном виде

и передает их, а также свой номер в системную процедуру, ответственную за реализацию прикладного протокола. Эта процедура определяет сетевой адрес серверной части прикладной программы и отправляет по этому адресу сообщение (с помощью модуля транспорта), содержащее номер требуемой процедуры сервера и значения ее входных параметров.

В узле-сервере модуль транспорта передает полученное сообщение процедуре, ответственной за реализацию прикладного протокола, которая по номеру процедуры сервера определяет требуемую заглушку и иницирует ее. Далее заглушка преобразует полученные параметры в обычный формат и вызывает по имени (с помощью команды *call*) настоящую процедуру сервера. Когда эта процедура завершится, заглушка получит ее выходные параметры и передаст их в процедуру, ответственную за прикладной протокол, для передачи в узел клиента.

В узле клиента полученные выходные параметры поступают в заглушку, которая, возвращая управление в программу-клиент (с помощью машинной команды *ret*), одновременно передает в нее и эти параметры. В результате программа-клиент имеет дело только со своей локальной заглушкой, не замечая все действия по взаимодействию с сервером. Несмотря на то что процедуры, ответственные за прикладной протокол, являются системными, они не являются частью ОС, а подсоединяются к частям прикладной программы (клиенту и серверу) как модули *DLL*.

Новые модули должны быть включены в сетевую ОС в том случае, когда она обслуживает локальные (нераспределенные) прикладные программы. Рассмотрим несколько примеров таких прикладных программ и используемых ими удаленных ресурсов.

1. Допустим, что пользователя интересует содержимое магнитного диска, принадлежащего удаленной ЭВМ. В этом случае в качестве прикладной программы можно рассматривать утилиту *ls* (в *UNIX*) или *dir* (в *MS-DOS*), а в качестве удаленного информационного ресурса — содержимое корневого каталога соответствующего магнитного диска.

2. Если пользователь хочет распечатать содержимое своего текстового файла на принтере, связанном с удаленной ЭВМ, то в качестве прикладной программы выступает утилита печати, а в качестве удаленного аппаратного ресурса — принтер.

3. Пусть пользователь хочет вывести текстовое сообщение на экран, принадлежащий другой ЭВМ. В этом случае в качестве прикладной программы можно рассматривать простую утилиту,

выполняющую перенос символьной строки с клавиатуры на экран. При этом экран является удаленным аппаратным ресурсом.

Во всех приведенных примерах прикладная программа представляет собой обычную локальную программу, не предназначенную для сетевого использования. Кроме того, каждой локальной программе требуется выполнить какие-то операции с удаленным файлом (устройство ввода-вывода — тоже файл). Для того чтобы подобная локальная программа могла получить доступ к удаленной файловой системе, сетевая ОС должна содержать модули, обеспечивающие взаимодействие между ними. В зависимости от того, как реализованы эти модули, будем различать два способа реализации сетевой ОС. В первом из этих способов оба модуля, обеспечивающие сетевое взаимодействие, входят в состав ядра ОС. Так как в данном методе учитываются особенности реализации файловой системы, то он будет рассмотрен позже в подразд. 6.3.

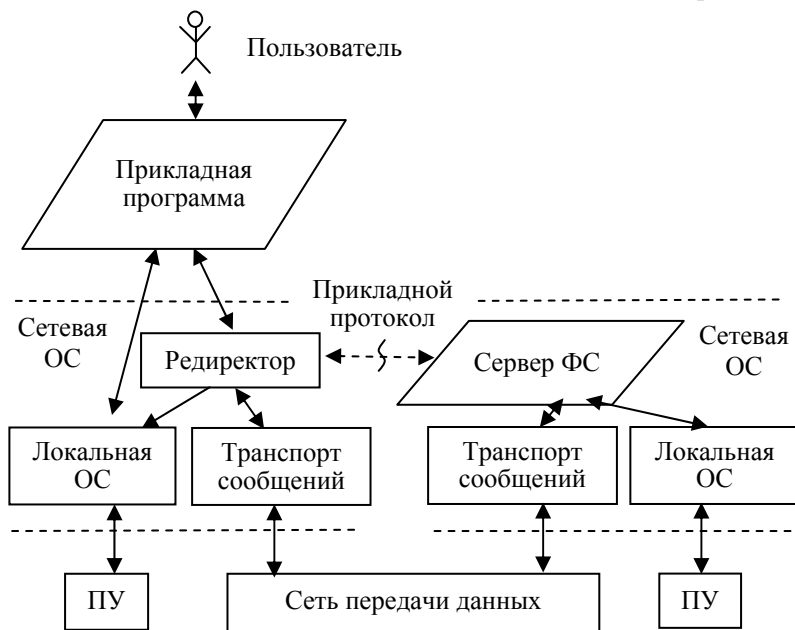
Другой способ реализации сетевой ОС предполагает, что, по крайней мере, один из взаимодействующих системных модулей находится вне ядра своей ОС. При этом в качестве такого модуля в узле-клиенте используется управляющая подпрограмма — редиректор, а в узле-сервере — процесс-сервер (рис. 24).

**Редиректор** — подпрограмма сетевой ОС, выполняющая обработку тех системных вызовов из прикладных программ, которые требуют выполнения операций с файлами (в том числе и с устройствами ввода-вывода). Если файл, упомянутый в данном вызове, является локальным (находится в том же узле), то системный вызов направляется для выполнения в свою локальную ОС. Если требуемый файл находится в каком-то другом узле сети, то редиректор посылает в этот узел сообщение для сервера файловой системы с просьбой выполнить требуемую операцию.

Реализация редиректора зависит от типа локальной ОС. Например, в среде *MS-DOS* редиректор может быть реализован в виде резидентной программы, перехватывающей системные вызовы (программные прерывания), предназначенные для инициирования файловой подсистемы. Если локальной ОС является *UNIX*, то редиректор может быть реализован как подпрограмма ядра, запускаемая из интерфейса системных вызовов при поступлении запроса на работу с файлами. При этом, как и другие подпрограммы ядра, редиректор подсоединяется к программе процесса, выдавшего системный вызов.

**Сервер файловой системы** — процесс (демон), предназначенный для выполнения удаленных системных вызовов. Этот процесс

инициируется при поступлении сообщения от какого то удаленного редиректора. Так как правомочность операций с файлами зависит от имени пользователя, то это имя является предметом



обсуждения между редиректором и сервером ФС. Именно от этого имени сервер будет выдавать системные вызовы для файловой подсистемы своей локальной ОС. Полученные результаты в виде сообщений будут пересылаться редиректору.

Рис. 24. Сетевое обслуживание локальной прикладной программы

Редиректор и сервер «беседуют» друг с другом в соответствии с некоторым прикладным протоколом. Это может показаться странным, так как обе эти программы являются системными, а редиректор даже принадлежит ядру. Причина этого заключается в том, что в данном случае термин «прикладной» означает «несетевой», что означает неучастие данного протокола в транспортировке информации по сети.

В рассмотренной выше схеме предполагалось, что конкретная ЭВМ является или узлом-сервером, или узлом-клиентом, участвующим в реализации одной из трех схем взаимодействия удаленных программных процессов. В действительности один и тот же узел может выполнять функции и клиента и сервера, одновременно

осуществляя обслуживание «чужих» прикладных программ и обращаясь в сеть с целью обслуживания «своих» программ. При этом различные клиенты и серверы в данном узле могут использовать разные схемы сетевого взаимодействия.

## 4. ПОДСИСТЕМА УПРАВЛЕНИЯ ПРОЦЕССАМИ

Данная подсистема занимает центральное место среди подсистем ОС, выполняя различные функции по обеспечению существования процессов. В подразд. 2.2 было приведено краткое определение процесса, согласно которому процесс есть одно выполнение последовательной программы. Более подробное определение: **процесс** — последовательная программа, располагающая окружением, достаточным для своего выполнения. **Окружение программы** образуют системные программы и системные структуры данных, обслуживающие данную программу. Так как эти программы и структуры данных реализуют виртуальную машину прикладной программы, то можно сказать, что **процесс** — последовательная программа, загруженная в свою виртуальную машину.

### 4.1. Состояния процесса

Являясь важнейшим объектом управления со стороны ОС, процесс может находиться в состояниях, изображенных на рис. 25. При этом дуги обозначают переходы между состояниями, а на дугах проставлены причины переходов. Эти причины бывают двух типов — системные вызовы, выдаваемые процессом-отцом или самим процессом, или события, происходящие внутри ядра ОС и приводящие к запуску каких-то подпрограмм ядра.

Создание процесса выполняет процесс-отец. Например, если вы запускаете какую-то программу из командной строки *UNIX*, то процессом-отцом является тот процесс-*shell*, в диалоге с которым вы запускаете данную программу. Для создания нового процесса процесс-отец использует системный вызов

*СОЗДАТЬ\_ПРОЦЕСС*( $|| i_p$ ) (на СИ — *fork*),

где  $i_p$  — номер нового процесса.

В результате выполнения ядром первой фазы данного системного вызова новый процесс оказывается в состоянии «Создан». Это переходное состояние: процесс уже существует, но еще не готов к запуску.

В результате второй фазы выполнения системного вызова процесс оказывается в состоянии «Готов». В этом состоянии он обладает всеми ресурсами, за исключением ЦП. В результате выпол-

нения подпрограммы ядра, называемой **диспетчером**, программа процесса начинает выполняться на ЦП.

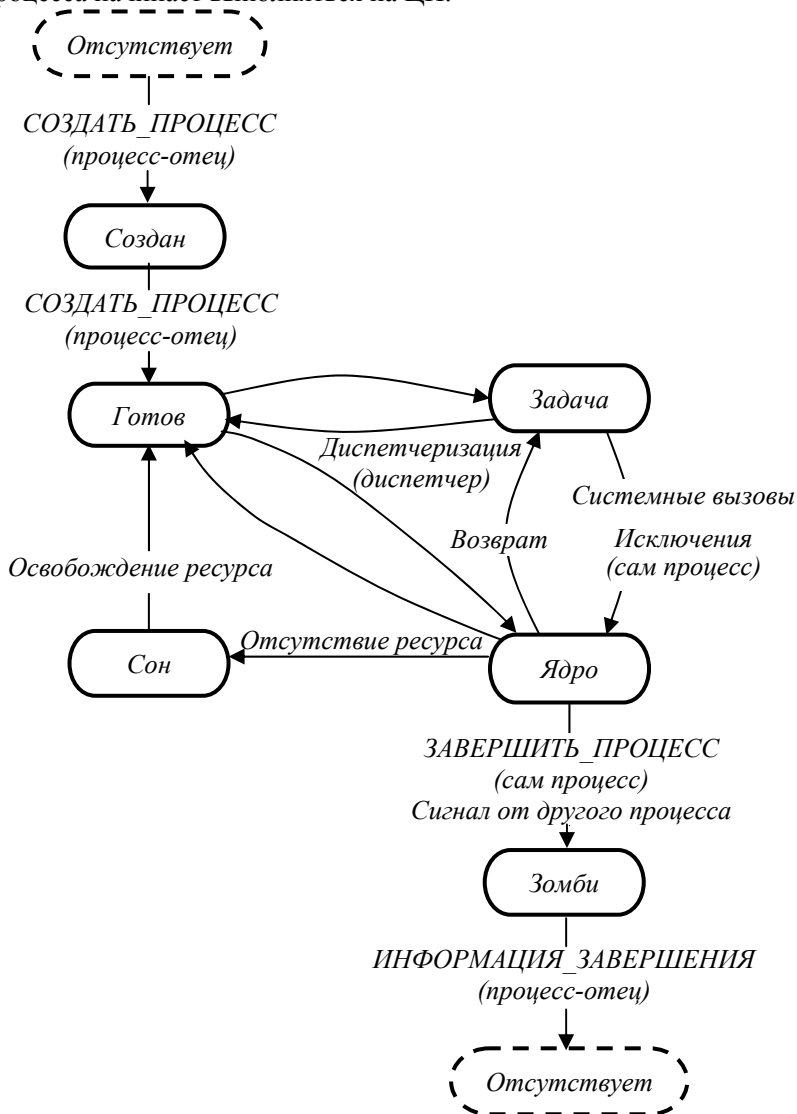


Рис. 25. Состояния процесса

Если на процессоре выполняются команды самой прикладной программы, то процесс находится в состоянии «Задача». Этому состоянию соответствует наименее приоритетное состояние аппаратуры ЦП.

Если очередная выполняемая команда программы есть или команда, приведшая к исключению (исключение — внутреннее аппаратное прерывание), или команда системного вызова, то на ЦП начинается исполняться соответствующая подпрограмма ядра ОС. (Напомним, что команда системного вызова представляет собой или команду программного прерывания *int*, или команду перехода *jmp*, каждая из которых делает переход на точку входа в ядро ОС.) В обоих указанных случаях инициирование подпрограммы ядра ОС производится в интересах процесса и, более того, это выполнение считается его частью. При выполнении подпрограмм ядра аппаратура ЦП находится в самом приоритетном состоянии, а сам процесс — в состоянии «Ядро».

Перечисленные выше команды перехода из состояния «Задача» в состояние «Ядро» предназначены для получения помощи прикладной программой со стороны ОС. Очень часто эта помощь не может быть оказана достаточно быстро, так как приходится ожидать завершения операции ввода-вывода или ожидать освобождения ресурса, требуемого для продолжения программы. Чтобы в этом случае процесс не занимал бесполезно ЦП, процесс переводится в состояние «Сон» с помощью внутренней подпрограммы ядра *sleep*.

Когда освобождается ресурс, которого ожидает «спящий» процесс, или завершается ожидаемая им операция ввода-вывода, выполняется внутренняя подпрограмма ядра — *wakeup*. Она переводит процесс в состояние «Готов», из которого он будет переведен впоследствии диспетчером в состояние «Ядро».

При завершении системного вызова процесс переходит из состояния «Ядро» в состояние «Задача», если на данный момент нет более приоритетного процесса, находящегося в состоянии «Готов». При наличии такого процесса выполняемый процесс переводится диспетчером в состояние «Готов». Переключение ЦП с одного процесса на другой производится диспетчером путем замены на ЦП аппаратного контекста одного процесса на аппаратный контекст другого процесса.

Причиной любого перехода процесса из состояния «Ядро» в состояния «Задача», «Готов», «Сон» является сам процесс. Никакой другой процесс не может вывести процесс из состояния



«Ядро». Исключениями являются обработчики наиболее важных аппаратных прерываний, выполнение которых не может быть отложено даже на короткий срок. После завершения обработки прерывания выполнение процесса продолжится опять в состоянии «Ядро». Более того, во время выполнения этих обработчиков прерванный процесс не выходит из состояния «Ядро», так как обработчики прерываний выполняются в его аппаратном контексте.

Напомним, что последовательное разделение ядра процессами производится с целью обеспечить целостность системных данных ядра. Одним из следствий данного принципа является то, что замена процесса, выполняемого на ЦП, происходит чаще всего при переходе процесса из состояния «Ядро» в состояние «Задача». Другим следствием является необходимость сокращения времени пребывания процесса в состоянии «Ядро». Для этого требуется оптимизировать выполнение системных функций по времени выполнения.

Переходы между состояниями «Готов», «Задача», «Ядро» и «Сон» будут продолжаться до тех пор, пока не будет выполнена подпрограмма ядра *exit*. Она может быть вызвана как в результате получения процессом какого-то сигнала, например *SIGKILL*, требующего завершения процесса, так и в результате выдачи самой прикладной программой процесса системного вызова

*ЗАВЕРШИТЬ\_ПРОЦЕСС* (*||*) (на СИ — *exit*).

В результате выполнения подпрограммы ядра *exit* процесс переходит в состояние «Зомби». В этом состоянии процесс фактически уже не существует. Все занимавшиеся им ресурсы освобождены, а его дочерние процессы становятся дочерними процессами процесса *init*, являющегося «прародителем» всех порождаемых процессов *UNIX*. Единственный неосвобождаемый ресурс — одна строка в системной таблице процессов *proc*, содержащая код завершения процесса и информацию о затратах времени ЦП на его выполнение.

Существование процесса прекратится полностью в результате выдачи процессом-отцом системного вызова

*ИНФОРМАЦИЯ\_ЗАВЕРШЕНИЯ* (*|| i<sub>p</sub>, k*) (на СИ — *wait*),

где *i<sub>p</sub>* — номер заверченного дочернего процесса; *k* — код завершения процесса.

Для выполнения этого системного вызова будет инициирована подпрограмма ядра *wait*, которая выдаст номер заверченного

дочернего процесса, его код завершения и уничтожит соответствующую строку в таблице *proc*. Если на момент выдачи этого системного вызова дочерний процесс еще не завершился, процесс-отец переходит в состояние «Сон», ожидая указанного события.

В некоторых *UNIX*-системах существует дополнительное состояние процесса — «Останов». В это состояние процесс переходит из состояния «Задача» или «Готов» в результате поступления сигнала *SIGSTOP*, выданного другим процессом, или сигнала *SIGTSTP*, выдаваемого в результате нажатия комбинации клавиш *<Ctrl>&<Z>*, или сигналов *SIGTTIN* и *SIGTTOU*, выдаваемых драйвером клавиатуры. Обратный переход из состояния «Останов» в состояние «Готов» производится в результате поступления сигнала *SIGCONT*.

## 4.2. Создание процесса

В результате создания процесса должно быть создано окружение программы, состоящее из переменных окружения процесса и следующих структур данных ядра ОС:

- 1) управляющей структуры *proc*;
- 2) управляющей структуры *user*;
- 3) структуры отображения памяти.

Первые две структуры вместе образуют блок управления (дескриптор) процесса. Структура *proc* используется ОС для выполнения манипуляций над процессом. Структуры *proc* для всех существующих в системе процессов объединены в единую **таблицу процессов системы**. Каждая *proc* является строкой этой таблицы. Системная переменная ***curproc*** содержит номер (индекс в таблице) того *proc*, который соответствует процессу, исполняемому в данный момент времени на ЦП.

Основное содержание структуры *proc*:

- 1) системное имя (номер) процесса;
- 2) номер процесса-отца;
- 3) номер сеанса, к которому принадлежит процесс;
- 4) номер группы процессов, к которой принадлежит процесс;
- 5) системное имя (номер) пользователя-владельца процесса;
- 6) информация о затратах времени ЦП на выполнение процесса (отдельно — в режиме «Задача» и в режиме «Ядро»);
- 7) сигналы, ожидающие доставки процессу;
- 8) текущее состояние процесса;
- 9) текущий приоритет процесса на получение времени ЦП;

10) системное имя (номер) события, возникновения которого ожидает процесс в состоянии «Сон»;

11) указатель (линейный адрес) на структуру *user*;

12) указатель на таблицу *LDT* (рассматривается в подразд. 5.2);

13) код завершения процесса, предназначенный для передачи в процесс-отец.

В отличие от структуры *proc*, которая используется ядром ОС, в основном, во время выполнения других процессов, системная структура *user* используется самим процессом (в режиме ядра). Эта структура содержит:

1) указатель на область памяти, содержащую **заголовок исполняемого файла**. Это служебная информация, которая предваряет код и данные прикладной программы и которая может использоваться как при создании процесса, так и впоследствии им самим (аналог *PSP* в *MS-DOS*);

2) указатель на **системный стек** — область памяти фиксированного размера, используемая в качестве стека при выполнении процесса в режиме «Ядро» (у каждого процесса свой системный стек);

3) указатель на область памяти, содержащую **аппаратный контекст** — состояние ЦП (содержимое его регистров) в момент прекращения выполнения данного процесса на ЦП. При возобновлении выполнения процесса содержимое данной области копируется в соответствующие регистры;

4) имя начального каталога пользователя. Это имя переписывается из файла */etc/passwd*;

5) имя текущего каталога пользователя. Относительно этого имени программа процесса может задавать относительные имена файлов;

6) диспозиция сигналов. Назначение данного поля будет рассмотрено в следующем разделе.

Кроме системных структур (*proc*, *user* и отображения памяти), доступ к которым процесс имеет только в состоянии «Ядро», при создании процесса инициализируются (заполняются) переменные окружения, наследуемые процессом от процесса-отца (см. п. 1.5.4).

Особенностью создания нового процесса в *UNIX* является то, что новый процесс, полученный в результате выполнения ядром системного вызова *СОЗДАТЬ\_ПРОЦЕСС*, является почти полной копией своего процесса-отца. При этом он имеет такую же программу, переменные окружения, структуры *proc* и *user*. Основное различие между процессами в том, что новый процесс имеет другое

имя (номер). Таким образом, результатом выполнения данного системного вызова является одновременное существование в данный момент двух процессов, выполняющих одну и ту же программу, причем в одной и той же ее точке: указатель команды *EIP* содержит адрес команды, которая следует в программе за командой, реализующей системный вызов *СОЗДАТЬ\_ПРОЦЕСС*.

Дальнейшее выполнение одной и той же программы процессом-отцом и дочерним процессом различно по той причине, что различается значение параметра  $i_p$ , возвращаемого системным вызовом в эти два процесса. При возврате в дочерний процесс этот параметр содержит не номер нового процесса, а число 0. Поэтому после команды системного вызова *СОЗДАТЬ\_ПРОЦЕСС*( $\|i_p$ ) программа должна содержать оператор условного перехода, выполняющий ветвление программы в зависимости от значения параметра  $i_p$ .

Далее дочерний процесс может продолжать выполнение копии программы процесса-отца или потребовать от ядра ОС выполнения своей собственной программы. Для этого ему достаточно передать ядру системный вызов

*ЗАГРУЗИТЬ\_ПРОГРАММУ* ( $I_F\|$ ) (на СИ — *exec*),

где  $I_F$  — имя файла, содержащего запусковую программу. В качестве такого файла может быть задан не только исполняемый файл, но и скрипт.

Особенностью данного системного вызова является то, что при его нормальном завершении управление не возвращается в ту точку прикладной программы, откуда был сделан вызов, так как вместо этой программы будет загружена новая программа, в которую и будет передано управление.

Выполнение данного системного вызова начинается с того, что проверяется наличие права доступа  $x$  (выполнение) процесса по отношению к загружаемому файлу. Этим правом может обладать или пользователь-владелец процесса, или группа пользователей, к которой принадлежит владелец процесса, или это право присуще всем пользователям файла. При отсутствии такого права выполняется возврат из системного вызова в старую программу. Иначе — проверяется наличие модификатора  $s$  у права доступа  $x$ . При наличии такого модификатора номер пользователя-владельца в структуре *proc* заменяется на номер пользователя-владельца загружаемого файла.

Далее определяется тип загружаемого файла. В исполняемом бинарном файле для этого используются первые два байта файла, содержащие так называемое *магическое число*. Если файл является скриптом, то его первая строка может задавать тип требуемого *shell* (см. подразд. 1.5.6). Если файл не является ни бинарным, ни скриптом, явно задающим *shell*, то будет загружен тот *shell*, который загружается по умолчанию в данной системе.

Сама загрузка программы, в отличие, например, от *MS-DOS*, не сводится к записи ее сегментов в ОП. Эти сегменты копируются из исполняемого файла в область ВП, называемую *областью свопинга*. Непосредственно в ОП записывается лишь самая начальная часть сегмента кода программы, с которой начнется выполнение программы в результате передачи управления или из ядра (при завершении системного вызова *ЗАГРУЗИТЬ\_ПРОГРАММУ*), или из динамического редактора связей (при использовании в программе *DLL*). Далее загрузка требуемых фрагментов программы из области свопинга будет производиться динамически, то есть во время выполнения программы. В том случае если программа процесса является реентерабельной, ее сегмент кода вообще не нуждается в записи в какую-либо память при условии, что эта программа уже выполняется каким-либо процессом.

Основная операция, выполняемая при загрузке программы, заключается в подготовке структур отображения памяти. Наличие данных структур, во-первых, позволяет использовать в программе логическую адресацию команд и данных, не зависящую от фактического размещения этих элементов в памяти. Во-вторых, эти структуры выполняют важную роль при защите информации в ОП. Подробно структуры отображения памяти рассматриваются в гл. 5.

### 4.3. Обработка сигналов

Вне зависимости от того, кто является источником сигнала (обработчик прерываний или процесс), его выдача сводится к установке одного бита в поле «сигналы» структуры *proc*. (Вспомним, что наряду с *user* это одна из двух основных управляющих структур процесса.) Длина этого поля в битах совпадает с числом типов сигналов, существующих в системе. Так как одному типу сигнала соответствует всего один бит, то до начала обработки сигнала и, следовательно, до сброса бита в поле «сигналы» все последующие сигналы данного типа будут процессом игнорироваться.

Наличие сигнала означает, что он должен быть обработан процессом. В общем случае существуют три варианта обработки сигнала:

1) обработка по умолчанию. Она выполняется одной из подпрограмм ядра и для большинства типов сигналов сводится к прекращению существования процесса;

2) игнорирование сигнала;

3) обработка сигнала собственной подпрограммой процесса.

В любой момент времени для каждого типа сигналов, поступающих в процесс, задан один (и только один) вариант обработки. Для этого используется строка *диспозиция сигналов* в управляющей структуре *user*. Каждое поле данной строки соответствует одному типу сигналов и содержит вариант его обработки. При этом если для обработки сигнала используется собственная (пользовательская) подпрограмма, то указанное поле содержит ее адрес в ОП.

Сразу же после создания процесса с помощью соответствующего системного вызова диспозиция сигналов нового процесса точно такая же, как и у процесса-отца (наследование диспозиции). Но если далее процесс выдаст системный вызов *ЗАГРУЗИТЬ\_ПРОГРАММУ*, то для всех сигналов устанавливается диспозиция «по умолчанию». Это объясняется тем, что все прежние обработчики сигналов уничтожены (как и все другие прикладные подпрограммы) в результате загрузки. Если такая начальная диспозиция не устраивает, то для ее изменения процесс выполняет системный вызов

*УСТАНОВИТЬ\_ОБРАБОТКУ\_СИГНАЛА* ( $I_s, i, A_a||$ ) (на СИ — *sigaction*),

где  $I_s$  — имя сигнала (номер или символьное имя);  $i$  — вариант обработки сигнала (по умолчанию, игнорирование, свой обработчик сигнала);  $A_a$  — стартовый адрес своего обработчика сигнала.

Воспользовавшись этим системным вызовом, программа процесса может сама задать вариант обработки любого своего входного сигнала, за исключением сигналов *SIGKILL* и *SIGSTOP*, для каждого из которых допустима только обработка по умолчанию (уничтожение процесса).

Установка в единицу бита в поле «сигналы» структуры *proc* не означает, что соответствующий сигнал начнет немедленно обрабатываться. Такая обработка может быть начата ядром в один из следующих моментов:

- 1) непосредственно перед переходом процесса из состояния «Готов» в состояние «Задача»;
- 2) непосредственно перед переходом процесса из состояния «Ядро» в состояние «Задача»;
- 3) непосредственно перед переходом процесса в состояние «Сон»;
- 4) непосредственно перед переходом процесса из состояния «Сон» в состояние «Готов».

В любой из этих моментов времени подпрограмма ядра *issig* проверяет наличие поступивших сигналов, читая соответствующее поле в структуре *proc*. Если сигнал обнаружен и если он не должен быть проигнорирован, то инициируется или системный обработчик сигнала, или собственная подпрограмма процесса. В последнем случае запуску подпрограммы процесса предшествует его перевод в состояние «Задача».

Если в момент появления сигнала процесс находился в состоянии «Ядро», то никакого воздействия на процесс сигнал не окажет до выхода из этого состояния. С другой стороны, сигнал не может появиться в то время, когда процесс находится в состоянии «Задача», так как источником сигнала в это время может быть только один из обработчиков прерываний. А любой из таких обработчиков прерываний может выполняться только в состоянии «Ядро».

Если запущенный обработчик сигнала не выполняет завершение процесса, то после выполнения этого обработчика выполнение процесса продолжится с той точки, в которой оно было прервано для обработки сигнала.

## 4.4. Диспетчеризация процессов

Наряду с ОП время ЦП является важнейшим ресурсом системы, подлежащим распределению между процессами, существующими в данный момент времени в системе и находящимися в состоянии «Готов». Система *UNIX* является системой *с разделением времени*, так как она предоставляет время ЦП процессам по очереди небольшими порциями (квантами). Что касается частоты предоставления квантов процессу и их фактической величины, то это зависит от типа процесса, предыдущей истории его обслуживания на ЦП, а также от типа других процессов, существующих в системе.

В зависимости от требуемой дисциплины диспетчеризации все существующие в *UNIX* процессы можно разбить на три группы:

1) **интерактивные процессы**. Они выполняются в интерактивном режиме (имеют доступ к терминалу), выполняя диалог с пользователем. Примеры: командные интерпретаторы (*shell*), текстовые редакторы. Так как пользователь постоянно находится за терминалом, то основным требованием интерактивного процесса к диспетчеризации является минимизация среднего времени реакции системы. **Время реакции** — время ожидания пользователем сообщения системы в ответ на завершение им ввода с клавиатуры. Для того чтобы пользователю не надоел диалог с системой, среднее время реакции не должно превышать 2–3 секунды;

2) **фоновые процессы**. Они не имеют непосредственного доступа к терминалу и не ведут диалог с пользователем. Примером являются процессы, выполняющие трудоемкие вычисления. Процессы такого типа предъявляют требования к общему объему времени ЦП, а не к динамике его назначения;

3) **процессы реального времени (РВ)**. В отличие от интерактивных и фоновых процессов, такие процессы обслуживают не людей-пользователей, а технологические процессы, выполняя автоматический съем информации и (или) автоматическую выдачу управляющих воздействий. Для каждого процесса РВ задается предельное время выполнения. Следствием этого являются жесткие требования к диспетчеризации таких процессов.

Структура данных ядра, которая находится в ведении диспетчера, называется **списком готовности (СГ)**. СГ — приоритетная очередь, в которую помещаются процессы при их переходе в состояние «Готов». **Приоритетная очередь** может рассматриваться как последовательное соединение обычных очередей, каждая из которых предназначена для размещения процессов с одинаковым приоритетом. Выборка элемента (процесса) из приоритетной очереди производится, как и из обыкновенной очереди — из начала списка. А размещение нового процесса в очереди производится в конце той подочереди, которая соответствует **приоритету диспетчеризации процесса**. Данный приоритет используется диспетчером при выборке процесса для исполнения на ЦП (не путать с аппаратными приоритетами, используемыми для защиты информации в ОП).

Приоритет диспетчеризации представляет собой целое неотрицательное число. В разных *UNIX*-системах это число выбирается по-разному. В некоторых из них чем число-приоритет выше, тем



оно «лучше», а в других, наоборот, «хуже». Для определенности далее рассматривается второй случай.

Текущая величина **PRI** приоритета процесса хранится в одном из полей управляющей структуры процесса *proc*. Значение величины **PRI** зависит от состояния процесса, а также от того, как процесс попал в это состояние. Специальная константа разделяет величины приоритетов **PRI** на два непересекающихся подмножества системных и прикладных приоритетов. Допустим, что величина этой константы — 76 (рис. 26). Тогда системный приоритет процесса находится в диапазоне  $0 \leq PRI \leq 76$ , а прикладной приоритет — в диапазоне  $76 < PRI \leq 140$ . В свою очередь системные приоритеты делятся на высокие ( $0 \leq PRI \leq 25$ ) и низкие ( $25 < PRI \leq 76$ ).

Процесс обладает системным приоритетом в следующих случаях: а) в состоянии «Ядро»; б) в состоянии «Сон»; в) в состоянии «Готов», если процесс перешел в это состояние из состояния «Сон». Покажем, что величина и использование системного приоритета в перечисленных случаях может быть различной.

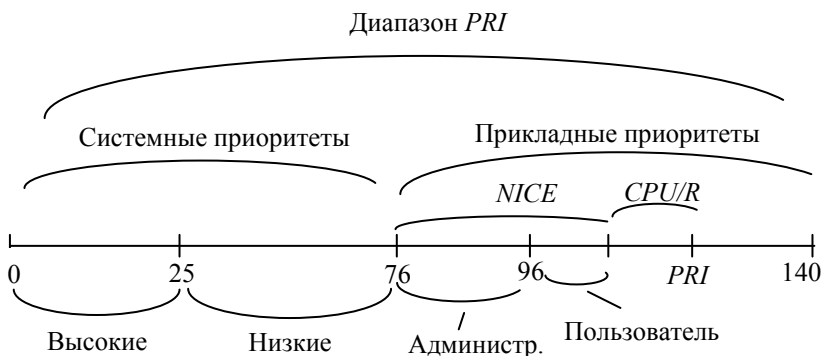


Рис. 26. Приоритеты процесса

Переход процесса в состояние «Ядро» является или результатом системного вызова, или результатом исключения, или результатом внешнего аппаратного прерывания. В любом из этих случаев процесс не снимается с ЦП, а переходит в другое состояние выполнения. Если процесс перешел в состояние «Ядро» в результате системного вызова или в результате исключения, то ему назначается достаточно низкий системный приоритет. Это не имеет никакого значения по отношению к другим процессам, так как пока процесс находится в состоянии «Ядро», никакой

другой процесс не может заменить его на ЦП. Данный системный приоритет используется лишь для обеспечения целостности данных ядра во время обработки внешних аппаратных прерываний.

Дело в том, что в отличие от исключений, происходящих вследствие выполнения программы самого процесса, внешние аппаратные прерывания никак не связаны с выполняемым на ЦП процессом. Для того чтобы обработчик внешнего прерывания не нарушил данные ядра, он снабжается своим приоритетом диспетчеризации, который сравнивается с приоритетом выполняемого на ЦП процесса. Уровень приоритета обработчика внешнего прерывания достаточен, чтобы прервать выполнение любого процесса в состоянии «Задача». Что касается состояния «Ядро», то приоритет процесса может быть как лучше, так и хуже приоритета обработчика. Это определяется тем, занимается ли в данный момент процесс обработкой данных ядра или нет. Если да, то процесс выполняет свою критическую секцию (см. п. 2.4.3) и его прерывать нежелательно. На время выполнения критической секции приоритет процесса улучшается настолько, чтобы он был лучше приоритетов опасных обработчиков прерываний. Затем восстанавливается прежнее значение приоритета.

Если при выполнении системного вызова процесс переводится в состояние «Сон», то при вызове внутренней процедуры ядра *sleep*, выполняющей такой перевод, в качестве ее входного параметра задается требуемый системный приоритет процесса. Величина этого приоритета зависит от типа операции ядра, завершения которой ожидает процесс. При этом каждой такой операции присущ фиксированный уровень системного приоритета процесса. Этот уровень приоритета процесса влияет на обработку сигналов, поступивших процессу в состоянии «Сон». При этом высокий системный приоритет временно запрещает, а низкий, наоборот, разрешает обработку поступивших сигналов.

После того как операция, ожидаемая процессом в состоянии «Сон», завершится, процесс окажется в состоянии «Готов» с тем системным приоритетом, который он получил при переходе в состояние «Сон». С этим приоритетом он конкурирует за время ЦП с другими процессами, находящимися в СГ, а также с процессом, выполняемым на ЦП в состоянии «Задача».

Если процесс находится на этапе выполнения прикладной программы, то его приоритет *PRI* рассчитывается по формуле

$$PRI = 76 + NICE + CPU/R,$$

где *NICE* — пользовательский приоритет; *CPU* — степень использования ЦП; *R* — коэффициент учета *CPU*. Подобно *PRI*, текущие значения *NICE* и *CPU* содержатся в структуре *proc* процесса.

**Пользовательский приоритет *NICE*** по умолчанию имеет значение 20. Именно на столько отличается число 96 (первоначальный прикладной приоритет процесса) от числа 76 (наилучший прикладной приоритет). Пользователь может влиять на *NICE* (а следовательно, и на *PRI*) у требуемого процесса с помощью команд *shell* — *nice* и *renice*. При этом следует отметить, что обычный пользователь может только ухудшать *NICE* у своих процессов. А администратор системы может не только ухудшать, но и улучшать *NICE* для любого процесса.

Команда *nice* используется для запуска нового процесса с приоритетом *NICE*, отличным от принятого по умолчанию. В следующем примере программа *prog5* (а точнее, соответствующий процесс) запускается с *NICE*, ухудшенным на 10:

```
$ nice -10 prog5
```

Пример команды, улучшающей *NICE* (команда доступна только администратору):

```
$ nice - -10 prog5
```

Команда *renice* позволяет изменить значение *NICE* для уже существующего процесса. В следующем примере у процесса с номером 168 значение *NICE* увеличивается на 20:

```
$ renice 20 168
```

***CPU* — степень использования ЦП**, которая увеличивается при выполнении процесса на ЦП. При этом через каждый тик таймера (10 или 20 мсек) диспетчер увеличивает *CPU* на единицу. Величина *CPU* для процессов, находящихся в СГ, периодически также изменяется диспетчером, но, во-первых, в сторону уменьшения, а во-вторых, это происходит гораздо реже. Что касается величины этого уменьшения, то в разных системах она определяется по-разному. Например, в некоторых системах *CPU* процесса в состоянии «Готов» корректируется каждую секунду в сторону уменьшения по формуле

$$CPU_{i,t} := CPU_{i,t-1} * (2 * n_{t-1}) / (2 * n_{t-1} + 1),$$

где  $CPU_{i,t}$  — величина *CPU* для *i*-процесса на *t*-й секунде времени;  $n_{t-1}$  — среднее число процессов, находившихся в СГ в течение последней секунды. Нетрудно заметить, что чем *n* больше, тем меньше улучшение приоритета каждого процесса в СГ.

**Коэффициент учета CPU R** используется для пересчета величины *CPU* в величину приоритета. Значение этого параметра зависит от типа системы *UNIX*. В рассматриваемом примере  $R = 8$ .

В подразд. 2.2 мы применяли команду *shell - ps* для получения краткой информации о процессах. Применение этой команды с флагом **-l** позволяет вывести на экран полную информацию о процессах, в том числе и их приоритеты. Пример выдачи приведен на рис. 27.

```
$ ps -l
```

F	S	UID	PID	PPID	CPU	PRI	NICE	TTY	TIME	CMD
1	S	vlad	701	683	0	8	0	P1	0:00.04	bash
1	R	vlad	712	701	137	133	20	P1	0:54.04	sh sc1
1	R	vlad	713	701	288	132	0	P1	2:37.91	sh sc1
1	O	vlad	720	701	0	96	0	P1	0:00:00	ps -l

Рис. 27. Пример выдачи на экран команды «ps -l»

В выдаче команды появились новые столбцы:

1) *F* — комбинация битов (записанная в восьмеричной системе), определяющая тип процесса (1 — программа процесса находится в ОП; 2 — системный процесс; 4 — программа процесса занимает фиксированное место в ОП (используется в процессах-драйверах); 10 — программа процесса выгружена из ОП; 20 — процесс трассируется другим процессом). Например, если  $F = 3$ , то это означает, что процесс системный и его программа находится в ОП;

2) *S* — состояние процесса (O — «Задача»; S — «Сон»; R — «Готов»; I — создается; Z — «Зомби»);

3) *UID* — имя пользователя-владельца процесса;

4) *PPID* — номер процесса-родителя;

5) *CPU* — степень использования ЦП;

6) *PRI* — текущий приоритет процесса;

7) *NICE* — относительный пользовательский приоритет процесса. Значение 0 в данном столбце означает, что пользовательский приоритет имеет значение по умолчанию — 20;

8) *ADDR* — адрес программы процесса (в ОП — для резидентных процессов; на диске (в области свопинга) — для остальных процессов);

9) *SZ* — размер программы процесса в блоках (по 512 байтов);

10) *WCHAN* — номер события, которого ожидает процесс в состоянии сна.

Последние три столбца из перечисленных на рис. 27 не показаны. Что касается остальных столбцов, то мы рассмотрим их более внимательно. Во-первых, заметим, что выдача команды *ps* отражает состояние процессов на момент времени, непосредственно предшествующий попаданию на ЦП той машинной команды программы *ps*, которая реализует системный вызов для вывода на экран. Во-вторых, на этот момент в системе существовали четыре процесса, принадлежащих данному пользователю *vlad*:

1) интерактивный процесс *shell* (программа *bash*), выполняющий обслуживание пользователя в оперативном режиме. На рассматриваемый момент времени данный процесс находился в состоянии «Сон», так как ожидал завершения своего дочернего процесса, выполняющего в оперативном режиме программу *ps*;

2) два фоновых процесса, порожденных оперативным *shell* в ответ на команды пользователя. Каждый из этих двух процессов выполняет свой *shell*, производящий интерпретацию одного и того же скрипта *sc1*, содержащего бесконечный цикл. Оба фоновых процесса запущены пользователем с разными пользовательскими приоритетами, но в один и тот же момент времени, с помощью следующего конвейера:

```
$ nice -20 sh sc1 | sh sc1 & ;
```

На момент получения выдачи программой *ps* оба процесса находились в состоянии «Готов»;

3) интерактивный процесс выполнения программы *ps* в оперативном режиме. На рассматриваемый момент времени данный процесс находился в состоянии «Задача».

Что касается приоритетов процессов, то только один из них обладает системным приоритетом (8). Это интерактивный *shell* с номером 701. Остальные процессы обладают лишь прикладными приоритетами — 133, 132, 96. Последнее из этих значений присуще интерактивному процессу, выполняющему программу *ps*. Что касается фоновых процессов 712 и 713, то имея близкие значения *PRI* (это следствие описываемой ниже работы диспетчера), повышенное значение пользовательского приоритета *NICE* в процессе 712 компенсируется более высоким значением *CPU* у процесса 713. Разница значений *CPU* у процессов является следствием разницы полученного ими времени ЦП (колонка *TIME* в выдаче команды *ps*).

Теперь мысленно предположим, что интерактивный процесс не перейдет в состояние «Ядро» (в результате системного вызова для вывода на экран), а останется на ЦП в состоянии «Задача». По истечении каждых 10 мс его *CPU* будет увеличено диспетчером на единицу. Так как *CPU* влияет на величину приоритета процесса с коэффициентом  $1/8$ , то приоритет *PRI* изменится на единицу через время:  $8 \cdot 10 \text{ мс} = 80 \text{ мс}$ .

Величина времени, в течение которого приоритет *PRI* процесса, исполняемого на ЦП, увеличивается на единицу, называется **квантом**. В рассматриваемом примере величина кванта равна 80 мс. По истечении кванта диспетчер сравнивает приоритет процесса, находящегося в состоянии «Задача», с приоритетом процесса, находящегося в начале СГ. При этом если приоритет исполняемого на ЦП процесса *PRI* больше, то этот процесс переходит с ЦП на то место в СГ, которое соответствует его приоритету *PRI*. А на ЦП ставится процесс из начала СГ. В нашем примере таким процессом будет процесс с номером 713. Напомним, что сама установка процесса на ЦП производится путем загрузки в регистры ЦП аппаратного контекста процесса.

Изложенная схема диспетчеризации учитывает интересы как интерактивных, так и фоновых (вычислительных) процессов. Интерактивные процессы большую часть своего времени находятся в состоянии «Сон» в ожидании завершения пользовательского ввода. Выходя из этого состояния, они имеют достаточно хороший системный приоритет и поэтому достаточно быстро начинают выполняться на ЦП. Что касается фоновых процессов, то когда они находятся подолгу в СГ, их приоритет улучшается и они гарантированно получают доступ к ЦП.

Если система рассчитана на выполнение не только интерактивных и вычислительных процессов, но и процессов реального времени, то пространство текущих приоритетов разделяют не на две, а на три части. Лучшая часть приоритетов отводится для процессов РВ, хуже — для системных приоритетов, еще хуже — для прикладных приоритетов. Приоритет процесса РВ фиксирован. Он не меняется ОС и одинаков для состояний процесса «Ядро» и «Задача». Поэтому на продолжительность выполнения процесса РВ никак не влияют не только интерактивные и вычислительные процессы, но и менее приоритетные процессы РВ.

Диспетчер иницируется одной из подпрограмм ядра, выполняющей одно из действий:

1) обработку системных вызовов, например, создание процесса. Перед тем как возвращать управление в прикладную программу процесса, обработка любого системного вызова предполагает вызов диспетчера;

2) обработку событий, связанных с освобождением ожидаемых ресурсов;

3) обработку отложенных вызовов. Каждый такой вызов представляет собой просьбу самого диспетчера о том, чтобы он был инициирован спустя заданный интервал времени. В частности, через каждую секунду диспетчер должен корректировать приоритеты процессов в СГ, через 10 мс корректировать приоритет исполняемого процесса, а по истечении кванта времени выполнять замену процесса на ЦП, если эта замена не произошла ранее по другой причине.

В основе обработки отложенных вызовов лежат прерывания от таймера. Это устройство играет большую роль в механизме функционирования всей системы.

## 4.5. Использование таймера для управления процессами

*Таймер* — аппаратное устройство, выдающее сигналы прерывания в ЦП через фиксированный промежуток времени, называемый *тиком*. В *UNIX* тик обычно равен 10 мсек, но может иметь и другое значение. Прерывания таймера имеют наивысший приоритет среди внешних маскируемых прерываний.

Обработчик прерываний таймера выполняет следующие действия:

1) обновление статистики использования процессора для текущего процесса. Отдельно подсчитываются затраты времени ЦП на выполнение процесса в режиме «Ядро» и в режиме «Задача»;

2) проверку превышения квоты процессорного времени, выделяемого процессу (с момента создания до момента завершения). В случае превышения выполняется отправка процессу соответствующего сигнала;

3) обновление системного времени (времени дня), а также некоторых других счетчиков;

4) обработку отложенных вызовов. *Отложенный вызов* — выполнение требуемой подпрограммы ядра через требуемое время. Заявки на отложенные вызовы поступают от подпрограмм ядра;

5) обработку алармов. **Аларм** — отправка сигнала процессу по истечении заданного интервала времени. Заявки на алармы могут поступать от любых процессов;

6) пробуждение тех процессов ядра, которые должны выполняться периодически, через заданный интервал времени.

Некоторые из перечисленных действий выполняются не на каждом тике. Если требуемое действие требует некоторых затрат времени ЦП, то оно реализуется не самим обработчиком прерываний таймера, а какой-то другой подпрограммой ядра, вызываемой с помощью отложенных вызовов. Любая подпрограмма ядра может сделать заявку на отложенный вызов себя или другой подпрограммы ядра, передав на вход подпрограммы обработки отложенных вызовов два параметра:  $A$  — стартовый адрес требуемой подпрограммы;  $t$  — величину требуемого промежутка времени в тиках.

Данная заявка помещается подпрограммой обработки отложенных вызовов в список, отсортированный по времени запуска. При этом каждый элемент содержит, кроме адреса  $A$ , разницу между временем вызова своей подпрограммы и временем вызова подпрограммы для предыдущего элемента. На каждом тике таймера содержимое первого элемента списка уменьшается на единицу. Как только это значение будет равно 0, производится вызов соответствующей подпрограммы, а также удаление первого элемента списка.

В отличие от отложенных вызовов, алармы предназначены для обслуживания не подпрограмм ядра (например диспетчера), а для обслуживания прикладных процессов. Для того чтобы процесс получил сигнал через требуемый интервал времени, программа этого процесса должна выполнить системный вызов

*ЗАДАТЬ\_ВРЕМЯ* ( $t$  ||) (на СИ — *alarm*),

где  $t$  — интервал времени (в секундах), по истечении которого процесс получит сигнал.

При выполнении данного системного вызова ядро создаст новую переменную — **таймер интервала** (не путать с аппаратным таймером), содержащую величину заданного интервала, измеренную в тиках. Данная переменная будет уменьшаться обработчиком прерываний таймера на единицу при обработке каждого тика. Как только таймер интервала станет равным нулю, соответствующему процессу будет послан сигнал *SIGALRM*. Следует отметить, что данный системный вызов не блокирует выполнение процесса до



получения сигнала через время  $t$ , а сразу возвращает ему управление. Чтобы обеспечить такое блокирование, системный вызов *ЗАДАТЬ\_ВРЕМЯ* следует использовать в программе процесса совместно с системным вызовом

*ПАУЗА*( $|$ ) (на СИ — *pause*).

Данный вызов приостанавливает выполнение вызывающего процесса до получения любого сигнала. В качестве такого сигнала особенно удобно использовать сигнал *SIGALRM*, выдаваемый системным вызовом *ЗАДАТЬ\_ВРЕМЯ*. Например, пара системных вызовов *ЗАДАТЬ\_ВРЕМЯ* и *ПАУЗА* используются при выполнении команды *shell – sleep*, выполняющей задержку программы на заданное число секунд (см. п. 1.5.5).

## 4.6. Информационные взаимодействия между процессами

### 4.6.1. Разделяемая память

Напомним, что использование несколькими процессами разделяемой памяти обеспечивает наиболее быстрое информационное взаимодействие между этими процессами. Реализация такого эффективного средства обеспечивается в основном используемой аппаратурой управления памятью, которая будет нами рассмотрена позже. Сейчас нашей задачей является рассмотрение той совокупности системных вызовов, которая позволит программе процесса использовать разделяемую память.

Любой процесс, желающий работать с разделяемым сегментом памяти, должен выполнить системный вызов

*СОЗДАТЬ\_РАЗДЕЛЯЕМЫЙ\_СЕГМЕНТ* ( $K, L, F || i$ ) (на СИ — *shmget*),

где  $K$  — ключ разделяемого сегмента;  $L$  — длина разделяемого сегмента в байтах;  $F$  — флаги, задающие права доступа к разделяемому сегменту (аналогично файлам);  $i$  — номер разделяемого сегмента. Он используется как программное и системное имя разделяемого сегмента и обладает свойством системной уникальности.

**Ключ** — численное имя объекта (в данном случае — разделяемого сегмента), используемое программистом в качестве первоначального программного имени сегмента. Ключ обладает свойством системной уникальности: среди всех разделяемых сегментов

в данной системе не должно существовать двух сегментов с одинаковым ключом. Кроме того, ключ должен быть известен всем процессам, работающим с данным разделяемым сегментом. Получение ключа, удовлетворяющего таким требованиям, представляет некоторую проблему. Аналогичная проблема возникает при использовании семафоров, а также очередей сообщений.

Проблема выбора ключа решается путем вызова процессом, создающим соответствующий объект (разделяемый сегмент, семафор или очередь), библиотечной подпрограммы, выполняющей генерацию ключа. Эта подпрограмма получает в качестве одного из своих параметров имя какого-то существующего файла, а в качестве другого параметра — какой-то символ. Действие данной подпрограммы основано на том, что каждый файл имеет уникальное системное имя (номер). На основе данного номера подпрограмма генерирует уникальный номер, корректируемый с помощью заданного символа. Эту же самую подпрограмму вызывает любой процесс, прежде чем выполнять операции с данным разделяемым сегментом. Естественно, что файл, имя которого используется для определения ключа разделяемого сегмента (семафора или очереди), должен существовать на протяжении всего времени существования этого сегмента (семафора или очереди), так как иначе процесс, начинающий работу с разделяемым сегментом, после уничтожения (или переименования) файла этот сегмент не найдет.

При выполнении этого и других системных вызовов по работе с разделяемой памятью ядро использует **таблицу разделяемой памяти**, которая содержит дескрипторы всех разделяемых сегментов системы. Каждый дескриптор содержит поле ключа, поле прав доступа, а также счетчик числа процессов, работающих в данный момент с разделяемым сегментом. Индекс дескриптора в таблице и выдается процессам в качестве номера разделяемого сегмента.

Выполняя системный вызов *СОЗДАТЬ\_РАЗДЕЛЯЕМЫЙ\_СЕГМЕНТ*, ядро начинает работу с того, что ищет в своей таблице разделяемой памяти строку с заданным полем ключа. Если ядро обнаружит, что дескриптор сегмента с заданным ключом в таблице разделяемой памяти отсутствует, то оно ищет в ОП сегмент заданной длины и создает для него дескриптор в таблице разделяемой памяти. Для заполнения полей этого дескриптора используются входные параметры системного вызова. Индекс

разделяемого сегмента возвращается в прикладную часть процесса в качестве программного имени сегмента.

Если при выполнении данного системного вызова ядро обнаружит, что в таблице разделяемой памяти уже существует дескриптор с заданным полем ключа, то выполняется проверка запрашиваемых прав доступа. В случае успешного ее завершения в процесс возвращается номер разделяемого сегмента.

Получив номер разделяемого сегмента, процесс должен выполнить его отображение на свою линейную виртуальную память (ЛВП). Для такого отображения используется системный вызов

*ПРИСОЕДИНИТЬ\_РАЗДЕЛЯЕМЫЙ\_СЕГМЕНТ* ( $i \parallel A$ ) (на СИ — *shmat*),

где  $i$  — номер разделяемого сегмента;  $A$  — начальный адрес размещения разделяемого сегмента в ЛВП процесса.

Выполняя данный вызов, ядро подыскивает подходящий участок ЛВП, создает для него дескриптор в *LDT*, создает для нового программного сегмента таблицу страниц, а дескриптор этой таблицы страниц помещает в каталог таблиц страниц процесса. Все перечисленные структуры управления памятью будут рассмотрены нами позже.

После успешного завершения данного системного вызова процесс может работать с разделяемым сегментом так же, как с обычной областью памяти, не используя никаких дополнительных системных вызовов. Интересно отметить, что процесс может отображать разделяемый сегмент на ЛВП не один, а любое число раз. В результате разные фрагменты программы могут работать с одной и той же областью реальной памяти, используя различающиеся между собой структуры данных.

Закончив работу с разделяемой памятью, процесс отсоединяет ее с помощью системного вызова

*ОТСОЕДИНИТЬ\_РАЗДЕЛЯЕМЫЙ\_СЕГМЕНТ* ( $A \parallel$ ) (на СИ — *shmdt*),

где  $A$  — начальный адрес размещения разделяемого сегмента в ЛВП процесса.

При выполнении данного вызова ядро уменьшает на единицу счетчик процессов в дескрипторе разделяемого сегмента (в таблице разделяемой памяти). Если значение счетчика становится нулевым, то дескриптор разделяемого сегмента уничтожается, а область ОП освобождается.

Соблюдение прав доступа к разделяемому сегменту явно недостаточно для обеспечения совместной работы нескольких процессов с одним и тем же сегментом памяти. Как было показано в п. 2.4.3, для этого требуется выполнить синхронизацию процессов, совместно использующих разделяемый сегмент, с помощью семафоров Дейкстры. В отличие от ранее рассмотренных системных вызовов  $p(S)$  и  $v(S)$ , используемые для работы с семафорами в системе *UNIX* системные вызовы весьма громоздки и нами рассматриваться не будут.

Кроме разделяемой памяти, требующей дополнительной синхронизации, ядро *UNIX* предоставляет другие средства межпроцессного информационного обмена, не требующие такой синхронизации. Два таких средства (каналы и именованные каналы) были рассмотрены в подразд. 2.5, а теперь рассмотрим еще два — очереди сообщений и сокеты.

#### 4.6.2. Очереди сообщений

В подразд. 2.5 было дано определение структурированного сообщения — пакета, представляющего собой последовательность байтов, предваряемую специальными служебными байтами. Таким образом, в отличие от неструктурированного потока данных, пакет представляет собой некоторую структуру данных, предназначенную для обмена между процессами. В нашем случае сообщение-пакет имеет три поля: 1) *тип сообщения* — целое положительное число; 2) длину сообщения в байтах (может быть нулевой); 3) собственно данные — любую последовательность байтов (не обязательно символы). Первые поля сообщения, содержащие служебную информацию, образуют *заголовок сообщения*.

Сообщение формируется процессом и записывается им с помощью соответствующего системного вызова в *очередь сообщений*. Прежде чем выполнять операции обмена с очередью сообщений, эта очередь должна быть создана с помощью системного вызова

*СОЗДАТЬ\_ОЧЕРЕДЬ\_СООБЩЕНИЙ* ( $K, F \parallel i$ ) (на СИ — *msgget*),

где  $K$  — ключ очереди сообщений, который используется в качестве первоначального программного имени очереди;  $F$  — флаги, задающие права доступа к очереди сообщений (аналогично файлам);  $i$  — номер очереди сообщений. Он используется как про-

граммное, так и системное имя очереди сообщений и обладает свойством системной уникальности.

Если очередь сообщений, которой соответствует заданный ключ, уже существует, то в результате выполнения данного системного вызова в программу возвратится номер этой очереди. Так как этот номер используется далее всеми процессами, работающими с очередью, в качестве ее программного имени, то, следовательно, каждый из этих процессов обязан выполнить системный вызов *СОЗДАТЬ\_ОЧЕРЕДЬ\_СООБЩЕНИЙ*.

При создании очереди сообщений для нее в пространстве памяти ядра создается блок управления, имеющий следующие поля: 1) права доступа; 2) длину очереди в байтах; 3) число сообщений в очереди; 4) указатель на первый элемент; 5) указатель на последний элемент. Теперь с очередью могут выполняться операции чтения и записи, подобно тому как эти операции выполняются с файлами. Подобно файлу, с каждой очередью связаны права доступа, регламентирующие доступ к очереди со стороны пользователя-владельца, со стороны пользователя-группы, а также со стороны других пользователей. Благодаря этим правам доступа процесс, создавший очередь, может, например, запретить другим процессам выполнять запись сообщений в очередь, а разрешить лишь чтение этих сообщений.

Как и блок управления очередью сообщений, сама эта очередь также размещается в пространстве памяти ядра. Это обеспечивает защиту очереди от несанкционированных операций со стороны процессов в состоянии «Задача». Логическому размещению очереди в пространстве ядра соответствует ее физическое размещение в ОП и в области свопинга ядра на магнитном диске. Какая часть очереди находится в ОП, а какая на диске, зависит от длины очереди, а также от текущего наличия аппаратных ресурсов в системе. В любом случае логическая организация очереди представляет собой связанный список, доступ к которому осуществляется через блок управления.

Процесс, выполняющий запись сообщения в очередь, обращается к ядру с помощью системного вызова

*ЗАПИСЬ\_СООБЩЕНИЯ\_ОЧЕРЕДЬ* ( $i, A, L ||$ ) (на СИ — *msgsnd*), где  $i$  — номер очереди сообщений;  $A$  — начальный адрес области памяти, в которой находится подготовленное сообщение;  $L$  — длина сообщения в байтах.

Процесс, выполняющий чтение сообщения из очереди, обращается к ядру с помощью системного вызова

*ЧТЕНИЕ\_СООБЩЕНИЯ\_ОЧЕРЕДЬ* ( $i, A, L, t ||$ ) (на СИ — *msgrcv*), где  $i$  — номер очереди сообщений;  $A$  — начальный адрес области памяти, в которую следует считать сообщение из очереди;  $L$  — длина сообщения в байтах;  $t$  — тип сообщения.

Обратим внимание, что процесс, выполняющий считывание записи из очереди сообщений, указывает тип сообщения. При выполнении данного системного вызова ядро ищет сообщение заданного типа в очереди и выдает его процессу. Поэтому фактически очередь сообщений является не обыкновенной, а приоритетной очередью. Операция чтения из такой очереди выдает то сообщение, которое поступило в очередь первым среди сообщений заданного типа. Наличие данного свойства приводит к тому, что информационный межпроцессный канал на базе очереди сообщений является мультиплексным (см. подразд. 2.5). Для мультиплексного канала характерно то, что в нем одновременно могут находиться данные нескольких типов. Причем передача данных каждого типа производится по информационному каналу логически параллельно с передачей данных других типов. В результате мультиплексный канал может рассматриваться как объединение нескольких логически параллельных моноканалов.

### Пример

В п. 2.4.3 нами был рассмотрен пример, в котором процесс-сервер выполнял запросы процессов-клиентов по распечатке текстовых файлов на принтере. При этом в качестве средства информационного взаимодействия между сервером и клиентами использовался разделяемый сегмент ОП. Рассмотрим реализацию этой же задачи с использованием очереди сообщений. Причем внесем в условие задачи следующее добавление: выполнив задание клиента, сервер посылает ему «отчет о проделанной работе».

Присвоим всем сообщениям клиентов тип 1, а каждому ответному сообщению сервера — тип, совпадающий с номером того процесса-клиента, которому это сообщение предназначено. Для того чтобы сервер мог определить номер процесса-клиента, сообщение клиента должно содержать этот номер. Возможное содержание очереди сообщений показано на рис. 28.

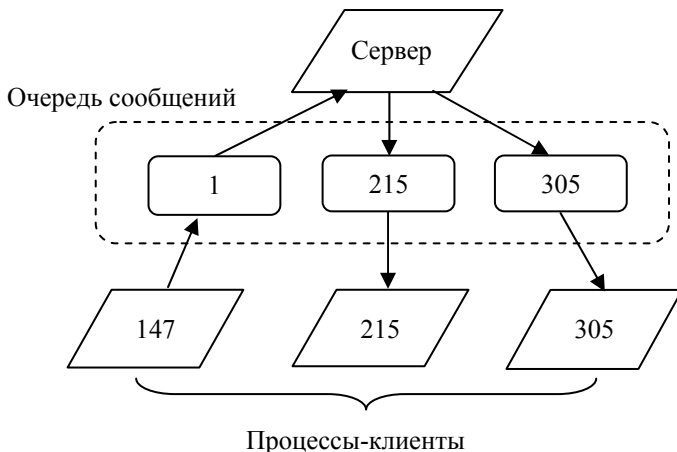


Рис. 28. Взаимодействие клиентов и сервера через очередь сообщений

В заключение сравним очередь сообщений с почтовым ящиком, в который процессы-писатели кладут письма-сообщения (при наличии соответствующих прав доступа), а процессы-читатели забирают эти сообщения. Причем множества писателей и читателей могут пересекаться или даже совпадать.

### 4.6.3. Сокеты

В *UNIX*-системах (причем не во всех) реализован метод межпроцессного информационного взаимодействия, основанный на использовании сокетов. Данный метод обладает большой гибкостью и позволяет создавать как устойчивые информационные каналы (виртуальные соединения), так и неустойчивые (датаграммные каналы), которые ориентированы на передачу неструктурированных сообщений. Соединяемые процессы могут находиться как на одной ЭВМ, так и в разных узлах сети. Такое многообразие получаемых информационных каналов выгодно отличает метод сокетов от ранее рассмотренных методов. Платой за универсальность является относительная сложность метода.

**Сокеты** — структуры данных ядра (очереди), предназначенные для создания информационных каналов между процессами. Подобно очереди сообщений, сокет может рассматриваться как почтовый ящик, предназначенный как для приема, так и для передачи информации. Но в отличие от очереди сообщений один сокет еще не образует информационного канала, а представляет собой лишь его окончание. Для создания информационного канала обязательно требуется наличие второго сокета, а также наличие механизма доставки информации между сокетами. Использование сокетов для информационного обмена между удаленными процессами будет рассмотрено нами в гл. 8, а пока остановимся на информационном взаимодействии локальных процессов (рис. 29).

Каждый сокет имеет системное имя (номер сокета в системе), а также может иметь два программных имени: 1) **локальное имя сокета**, используемое процессом-«владельцем» этого сокета; 2) **внешнее имя сокета**, используемое «чужими» процессами. Первое из этих имен должно быть уникально только среди имен сокетов, принадлежащих данному процессу. Второе имя должно быть уникальным в пределах всей ВС. Пользуясь этим именем, прикладная программа адресует свои сообщения не конкретному процессу, которого она «не знает», а его сокету, который, таким образом, играет роль «почтового ящика». Об этом же говорит то, что внешнее имя сокета обычно называют **адресом сокета**.

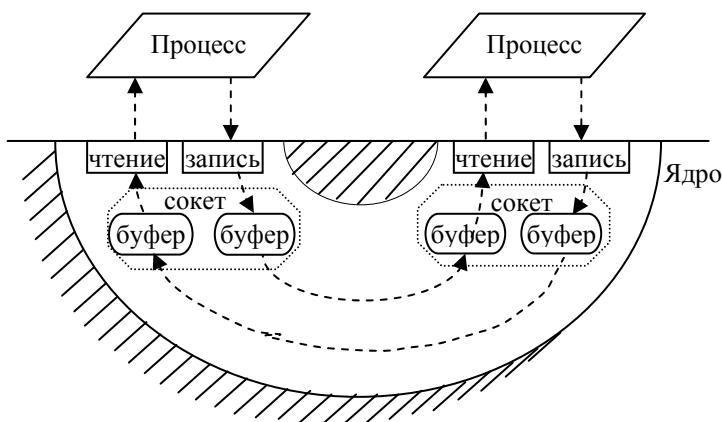


Рис. 29. Информационный канал с использованием сокетов



Если сокеты используются для взаимодействия процессов, выполняемых на одной ЭВМ, то для выбора имен этих сокетов используется тот же подход, что и для файлов. Так как имя-путь файла уникально в пределах своей ЭВМ, то такое же имя может использоваться для идентификации сокета «чужими» процессами. При этом сокет «маскируется» под файл, так как внешнее имя сокета включается в файловую структуру системы, но никакой файл этому имени, конечно, не соответствует. Как и для файла, процесс-владелец обращается к своему сокету, используя его локальное имя — *номер файла-сокета*.

Рассмотрим системные вызовы, используемые для реализации датаграммного информационного канала. Допустим, что такой канал используется для взаимодействия процесса-клиента с процессом-сервером. Как и ранее, предположим, что сервер выполняет вывод на печать текстовых файлов, заданных клиентами. После распечатки файла сервер посылает клиенту «отчет». Возможный порядок выдачи системных вызовов сервером и клиентом во времени приведен на рис. 30.

Для создания своего сокета любой процесс использует системный вызов

*СОЗДАТЬ\_СОКЕТ* (*d* || *i*) (на СИ — *socket*),

где *d* — *комм.* *ий домен*. Это множество, к которому относится внешнее имя сокета. Основные значения *комм.*: *AF\_UNIX* — множество имен файлов (используется для связи локальных процессов), *AF\_INET* — множество имен файлов Internet (используется для связи удаленных процессов), *AF\_IPX* — множество имен файлов IPX (используется для связи удаленных процессов). *СОЗДАТЬ\_СОКЕТ* — системный вызов, создающий сокет. *СВЯЗАТЬ\_СОКЕТ\_АДРЕС* — системный вызов, связывающий сокет с адресом. *СВЯЗАТЬ\_СОКЕТ\_АДРЕС* — системный вызов, связывающий сокет с адресом. *ПОЛУЧИТЬ\_ДАТАГРАММУ* — системный вызов, получающий датаграмму. *ПОСЛАТЬ\_ДАТАГРАММУ* — системный вызов, посылающий датаграмму. *ЗАКРЫТЬ\_СОКЕТ* — системный вызов, закрывающий сокет.

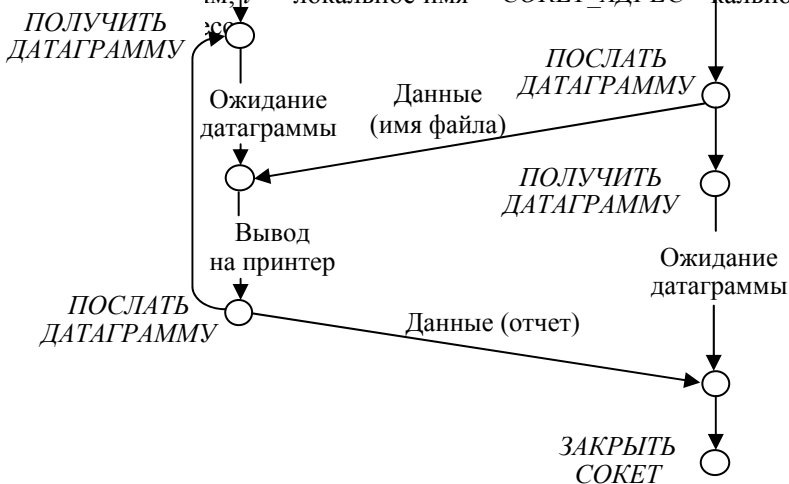


Рис. 30. Пример создания и использования датаграммного канала

После того как процесс (сервер или клиент) создал свой сокет с параметрами  $d = AF\_UNIX$ ,  $u = SOCK\_DGRAM$ , он связывает этот сокет с его внешним именем (адресом), используя системный вызов

*СВЯЗАТЬ\_СОКЕТ\_АДРЕС* ( $i, A ||$ ) (на СИ — *bind*),

где  $A$  — адрес (внешнее имя-путь) сокета.

В результате выполнения этих двух системных вызовов и сервером и клиентом, датаграммный канал готов к работе. При этом каждый из созданных сокетов может участвовать в работе не одного, а многих датаграммных информационных каналов. Последнее свойство очень важно для сервера, так как позволяет многим клиентам выполнять информационный обмен с ним, используя единственный внешний адрес его сокета.

Инициатива начать информационный обмен всегда принадлежит клиенту. Для этого клиент должен заранее «знать» адрес сокета сервера. Используя этот адрес, клиент передает ядру следующий системный вызов

*ПОСЛАТЬ\_ДАТАГРАММУ* ( $i_c, b, n, A_s ||$ ) (на СИ — *sendto*),

где  $i_c$  — номер сокета клиента;  $b$  — адрес прикладного буфера, из которого должна быть взята посылаемая датаграмма;  $n$  — длина датаграммы в байтах;  $A_s$  — адрес сокета сервера.

Обработка этого системного вызова ядром сводится к тому, что подпрограмма «Запись» (см. рис. 29) помещает текст датаграммы в выходной буфер своего сокета, из которого она будет перемещена во входной буфер требуемого сокета.

Что касается сервера, то первоначально он не знает не только адреса сокетов клиентов, но даже и количество этих клиентов. Поэтому единственное, что может делать в такой ситуации сервер, — это ожидать поступления очередной датаграммы, используя системный вызов

*ПОЛУЧИТЬ\_ДАТАГРАММУ* ( $i_s, b, n || A_c$ ) (на СИ — *recvfrom*),

где  $i_s$  — номер сокета сервера;  $b$  — адрес прикладного буфера, в который должна быть помещена датаграмма;  $n$  — длина датаграммы в байтах;  $A_c$  — адрес сокета клиента.

Обратим внимание, что в результате завершения данного системного вызова сервер «знает» внешнее имя (адрес) сокета клиента, пославшего датаграмму. Пользуясь этим адресом, сервер может посылать ответные датаграммы, используя для этого системный вызов *ПОСЛАТЬ\_ДАТАГРАММУ*. Естественно, что для приема

этой датаграммы клиент должен выполнить системный вызов *ПОЛУЧИТЬ\_ДАТАГРАММУ*.

После того как датаграммный информационный канал будет не нужен, процесс-клиент должен выполнить системный вызов

*ЗАКРЫТЬ\_СОКЕТ* ( $i ||$ ) (на СИ — *close*),

где  $i$  — номер сокета.

Использование сокетов позволяет создавать не только датаграммные, но и устойчивые информационные каналы. Переделаем рассмотренный выше пример так, чтобы для связи между клиентом и сервером использовался информационный канал в виде виртуального соединения. Возможная последовательность во времени выдачи системных вызовов приведена на рис. 31.

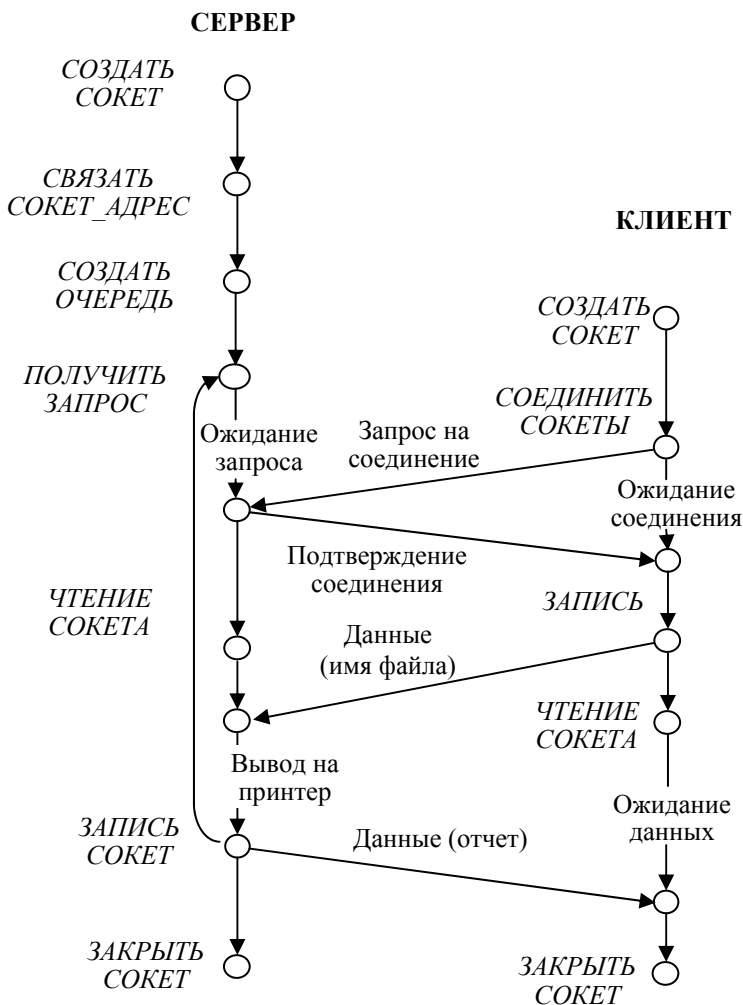


Рис. 31. Пример создания виртуального соединения

Процесс-сервер начинает создание виртуального канала с того, что выполняет системные вызовы *СОЗДАТЬ СОКЕТ* с параметром  $u = \text{SOCK\_STREAM}$  и *СВЯЗАТЬ СОКЕТ АДРЕС*. Полученный сокет имеет адрес (внешнее имя) и предназначен для того, чтобы принимать запросы от клиентов на создание виртуальных соединений. Далее сервер должен выполнить следующий системный вызов, который информирует ядро о том, что входной буфер сокета представляет собой очередь запросов на создание виртуальных соединений:

*СОЗДАТЬ\_ОЧЕРЕДЬ* ( $i_s ||$ ) (на СИ — *listen*),

где  $i_s$  — номер сокета сервера.

На другом конце создаваемого устойчивого канала процесс-клиент начинает с того, что выполняет системный вызов *СОЗДАТЬ СОКЕТ* с параметром  $u = \text{SOCK\_STREAM}$ . Далее он должен выполнить системный вызов

*СОЕДИНИТЬ СОКЕТЫ* ( $i_c, A_s ||$ ) (на СИ — *connect*),

где  $i_c$  — номер сокета клиента;  $A_s$  — адрес сокета сервера.

В результате данного системного вызова серверу будет послано сообщение-запрос на установление виртуального соединения, которое будет помещено во входной буфер (очередь) соответствующего сокета  $A_s$ .

Сокет  $i_s$  будем называть **главным сокетом** сервера. Он используется сервером только для приема запросов на создание виртуальных соединений. Никакой передачи данных через этот сокет сервер не производит. Для работы в самих создаваемых виртуальных соединениях серверу требуются новые сокеты. Причем для каждого соединения требуется отдельный сокет. Следующий системный вызов сервер должен выполнять в цикле (для каждого элемента входной очереди сокета  $i_s$ ):

*ПОЛУЧИТЬ\_ЗАПРОС* ( $i_s || i$ ) (на СИ — *accept*),

где  $i_s$  — номер главного сокета сервера;  $i$  — новый сокет сервера.

При выполнении данного системного вызова ядро выбирает первый элемент входной очереди главного сокета  $i_s$  и создает для этого элемента новый сокет, являющийся окончанием (со стороны сервера) нового виртуального соединения. Другим окончанием (со стороны клиента) данного информационного канала является сокет клиента. После этого сокету клиента посылается подтверждение о создании виртуального соединения. Обратим внимание, что для

послания этого подтверждения не требуется адрес сокета клиента, так как на момент этого послания виртуальное соединение уже создано.

Для выполнения информационного обмена по виртуальному соединению и клиенту, и серверу достаточно знать лишь локальное имя своего сокета:  $i$  — для сервера,  $i_c$  — для клиента. Знание адреса (внешнего имени) чужого сокета не требуется. Для передачи и приема сообщений по виртуальному соединению используются системные вызовы, аналогичные тем, что используются для работы с файлами:

*ЗАПИСЬ СОКЕТ* ( $i, B, n \parallel n_y$ ) (на СИ — *write*),

*ЧТЕНИЕ СОКЕТА* ( $i, B, n \parallel n_y$ ) (на СИ — *read*),

где  $i$  — номер сокета, используемого для связи с каналом;  $B$  — адрес прикладного буфера, в который читается строка байтов из канала (при вводе) или из которого строка записывается в канал (при выводе);  $n$  — число читаемых (записываемых) байтов;  $n_y$  — число фактически считанных (записанных) байтов.

После того как виртуальное соединение будет не нужно, процесс-клиент и процесс-сервер должны выполнить системный вызов

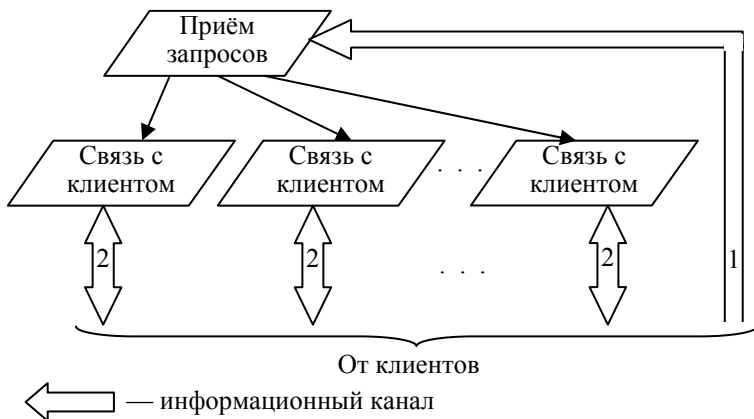
*ЗАКРЫТЬ СОКЕТ* ( $i \parallel$ ) (на СИ — *close*),

где  $i$  — номер сокета.

Обратим внимание, что на рис. 31 вызов *ЗАКРЫТЬ СОКЕТ* относится не к главному сокету, работа с которым продолжается в бесконечном цикле, а к сокету, принадлежащему конкретному виртуальному соединению.

В рассмотренном примере процесс-сервер создает один сокет для приема запросов на создание виртуальных соединений (главный сокет), а затем для каждого поступившего запроса создает еще один сокет, являющийся окончанием соответствующего виртуального соединения. Кроме того, этот процесс выполняет обслуживание готовых виртуальных соединений, принимая поступающие заявки на обслуживание (имена файлов), производя распечатку файлов, а затем отправляя соответствующие отчеты. Подобная однопроцессная модель сервера оправдана только в случае небольшого объема работы по обслуживанию каждого виртуального соединения. Намного более изящной является многопроцессная модель сервера, согласно которой главный процесс сервера не выполняет обслуживание отдельных виртуальных соединений, создавая для обслуживания каждого из них новый дочерний процесс (рис. 32). Особенно полезна многопроцессная модель сервера при

достаточно большой трудоемкости обслуживания одного запроса, позволяя процессу-серверу выполнять лишь координацию обслуживания поступающих запросов, не занимаясь деталями такого



обслуживания.

Рис. 32. Простая многопроцессная модель сервера:

1 — запросы на виртуальное соединение; 2 — виртуальное соединение

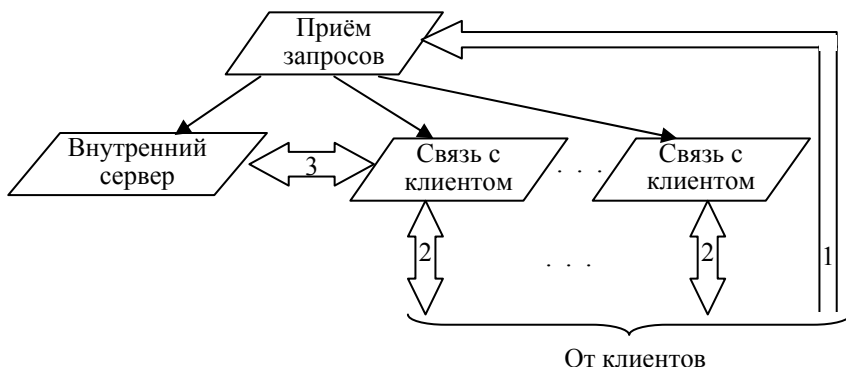
Запишем упрощенную программу главного сервера:

```
СОЗДАТЬ_СОКЕТ
СВЯЗАТЬ_СОКЕТ_АДРЕС
СОЗДАТЬ_ОЧЕРЕДЬ
цикл
    ПОЛУЧИТЬ_ЗАПРОС
    СОЗДАТЬ_ПРОЦЕСС
конец цикла
```

Программа дочернего процесса выполняет операции с готовым виртуальным соединением, используя системные вызовы *ЗАПИСЬ\_СОКЕТ* и *ЧТЕНИЕ\_СОКЕТА*. При этом номер сокета, используемый в данных системных вызовах для доступа к виртуальному соединению, дочерний процесс получает «по наследству» от главного серверного процесса среди номеров других (настоящих) открытых файлов.

На рис. 33 приведена более сложная многопроцессная модель сервера, отличающаяся от модели на рис. 30 тем, что внутри сервера выделен свой серверный процесс, выполняющий обслуживание процессов, занятых обслуживанием виртуальных соединений.

В ведении внутреннего сервера может быть, например, обслуживание запросов по доступу к общей базе данных. Ведение диалога с процессом-клиентом и преобразование полученных запросов в стандартную форму выполняет процесс, ответственный за соответствующее виртуальное соединение. Для создания внутрисерверного информационного канала (канал 3 на рис. 33) могут быть



использованы не только сокеты, но и очередь сообщений.

Рис. 33. Многопроцессная модель сервера с внутренним сервером

В заключение отметим, что все рассмотренные ранее однопроцессные серверы также могут быть реализованы с помощью одной из многопроцессных моделей.

## 5. УПРАВЛЕНИЕ ОПЕРАТИВНОЙ ПАМЯТЬЮ И ПРОЦЕССОРОМ

### 5.1. Задачи управления памятью

Важнейшими аппаратными ресурсами ВС являются ЦП и ОП. Управление этими ресурсами выполняется как программно — подпрограммами ядра ОС, так и аппаратно — аппаратурой самого ЦП. При этом программное управление ЦП и ОП использует и дополняет соответствующее аппаратное управление.

В предыдущих разделах пособия были выявлены основные задачи по управлению ЦП:

1) диспетчеризация процессов — распределение времени ЦП между процессами. Данная задача решается ядром ОС;

2) замена на ЦП аппаратного контекста процесса. Такая замена приводит к смене выполняемого на ЦП процесса. Ее реализация требуется для решения предыдущей задачи диспетчеризации процессов. Данная задача решается аппаратно;

3) переключение ЦП из непривилегированного в привилегированный режим и обратное переключение. Такое переключение требуется для реализации системных вызовов, приводящих к переводу процесса из состояния «Задача» в состояние «Ядро», а также для обратных переходов. Данная задача решается аппаратно;

4) поддержка прерываний. Механизм прерываний чрезвычайно важен для организации взаимодействия между аппаратурой ВС и ее программным обеспечением. Данная задача решается совместно аппаратурой ЦП и подпрограммами ядра ОС.

Первая из перечисленных задач (диспетчеризация процессов) была рассмотрена нами ранее в подразд. 4.4, а остальные задачи будут представлены в данном разделе после рассмотрения задач управления ОП.

Напомним, что реальная ОП любой ВС представляет собой линейную последовательность пронумерованных ячеек. Номер ячейки памяти называется ее реальным адресом. Благодаря наличию подсистемы управления памятью, входящей в состав ядра ОС (см. рис. 21), а также *аппаратуры управления памятью*, любая программа имеет дело не с реальной, а с *виртуальной ОП*. Предоставление такой ОП требует выполнения следующих трех функций:



1) преобразование виртуальных адресов в реальные адреса ячеек ОП;

2) защита областей памяти одного процесса от воздействия других процессов;

3) распределение ОП между процессами.

При выполнении этих функций роль ОС и аппаратуры управления памятью различна. Так как преобразование адресов выполняется во времени очень часто (при выполнении любой машинной команды, имеющей дело с ячейками ОП), то это преобразование реализовано аппаратно. Роль ОС сводится только к инициализации регистров и таблиц, используемых для аппаратного преобразования адресов и входящих в состав аппаратуры управления памятью.

Защита памяти процесса от воздействия других процессов основана на контроле правомочности выполнения каждой операции с ОП. Так как частота таких операций очень высока, то аналогично функции преобразования адресов, контроль правомочности операций с ОП выполняется полностью аппаратно. Роль ОС сводится к выполнению двух функций, дополняющих аппаратный контроль. Во-первых, она осуществляет инициализацию регистров и таблиц, используемых для такого контроля. Во-вторых, в случае нарушения правомочности доступа к памяти ОС обрабатывает исключение (внутреннее аппаратное прерывание), выдаваемое аппаратурой управления памятью.

В отличие от двух других функций, функция распределения памяти выполняется во времени достаточно редко и поэтому может быть реализована программно, то есть ядром ОС. Но если мы хотим обеспечить превышение объема виртуальной памяти программы над объемом реальной ОП, выделенной ей, то должны использовать дополнительные элементы аппаратуры управления памятью. Таким образом, выбор аппаратуры управления памятью оказывает решающее влияние на реализацию всех трех функций управления ОП. При решении своих задач ОС опирается на имеющуюся в ВС аппаратуру управления памятью и использует имеющиеся в ней регистры, таблицы и исключения.

В общем случае аппаратура управления памятью включает аппаратуру управления сегментами и аппаратуру управления страницами. Применение этих типов аппаратуры приводит к тому, что преобразование каждого виртуального программного адреса в реальный адрес производится не на одном, а на двух этапах (рис. 34). Благодаря такому 2-этапному преобразованию адресов

каждая из трех перечисленных задач управления ОП также разбивается на две подзадачи.



Рис. 34. Состав аппаратуры управления памятью

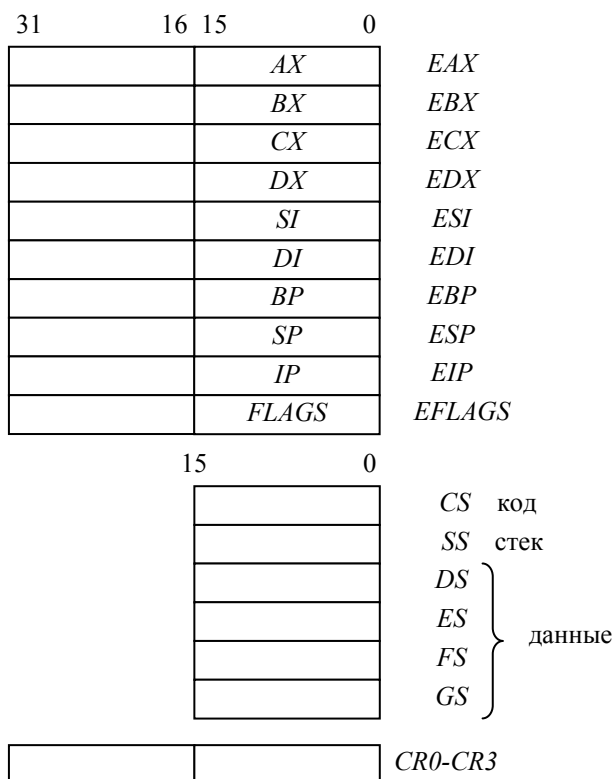
Дальнейшее изложение решения задач по управлению ОП и ЦП основано на использовании в качестве ЦП микропроцессора *i80386*. Данный процессор занимает очень важное место в историческом ряде процессоров фирмы *Intel* по следующим причинам. Во-первых, это первый 32-битный процессор. Его регистры, за исключением некоторых регистров управления памятью, рассматриваемых нами позже, приведены на рис. 35.

32-битные рабочие регистры *EAX*, *EBX*, *ECX*, *EDX* имеют в качестве младших половин регистры *AX*, *BX*, *CX*, *DX*, доставшиеся «в наследство» от процессора *i8086*. Это же относится и к регистрам-указателям *ESI*, *EDI*, *EBP*, *ESP*, *EIP*, а также к регистру флагов *EFLAGS*. Что касается сегментных регистров, то они по-прежнему остались 16-битными. Их число увеличилось на два (*FS* и *GS*). Новыми по сравнению с *i8086* являются также управляющие 32-битные регистры *CR0*, *CR1*, *CR2*, *CR3*. Эти регистры, в отличие от регистра *EFLAGS*, отражают не текущее состояние ЦП, определяемое программой, выполняемой в данный момент времени на ЦП, а долговременное состояние, общее для всех выполняемых на процессоре программ.

Во-вторых, начиная с *i80386* аппаратно поддерживается виртуальная память, имеющая объем во много раз больший, чем реальная ОП.

В-третьих, в *i80386* реализованы достаточно надежные аппаратные методы защиты информации в ОП.

Несмотря на то что последующие модели процессоров фирмы *Intel* многократно превосходят *i80386* по скорости вычислений, в них по-прежнему реализуются идеи, полученные при его проектировании. Указанное превосходство в скорости достигается, в основном, за счет «количественных» факторов: увеличения числа команд, размера и количества регистров, емкости буферов, увеличения тактовой частоты.

Рис. 35. Регистры процессора *i80386*

Процессор *i80386* имеет три режима работы: 1) реальный режим; 2) защищенный режим; 3) режим *V86*. В **реальном режиме** процессор оказывается сразу же после включения питания или в случае сбоя. В этом режиме процессор очень похож на *i8086*, выполняя все программы для него (в том числе и саму *MS-DOS*). Отличия: 1) выше скорость выполнения машинных команд; 2) длина всех регистров (кроме сегментных) увеличена до 32 битов.

Режим *V86* (виртуальный процессор *i8086*) имитирует для каждой из нескольких параллельно выполняемых *DOS*-программ режим работы *i8086*.

Бит 0 в *CR0* управляет переключением режима процессора: 1 — защищенный режим; 0 — реальный режим. Следующая группа

машинных команд (используется ассемблерная форма записи) выполняет переключение ЦП из реального в *защищенный режим*:

```
mov AX, CR0  
or  AX, 1  
mov CR0, AX
```

В защищенном режиме процессор полностью преобразуется:

1) адресное пространство ОП увеличивается до 4 Гбайт или более, где  $1 \text{ Г} = 1 \text{ К}^3$ ,  $1 \text{ К} = 1024$ ;

2) аппаратно поддерживается мультипрограммность: каждой прикладной программе назначается своя область памяти, аппаратно защищаемая от воздействия других прикладных программ. Особо защищается от воздействия прикладных программ область ядра ОС.

## 5.2. Сегментная виртуальная память

### 5.2.1. Преобразование адресов

Благодаря наличию *аппаратуры управления сегментами* любой процесс имеет в своем распоряжении *сегментную виртуальную память* — конечное множество непрерывных областей памяти (сегментов), доступных программе процесса. В этом случае адрес ячейки ОП всегда задается в программе в виде пары ( $S$ ,  $L$ ), где  $S$  — идентификатор сегмента, а  $L$  — смещение относительно начала сегмента.

При работе ЦП в реальном режиме  $S$  представляет собой начальный адрес размещения сегмента в ОП (точнее — номер начального параграфа). Так как реальный режим ЦП используется лишь при загрузке мультипрограммной системы, то далее речь будет идти, в основном, о защищенном режиме.

При работе в защищенном режиме  $S = (T, I)$ , где  $T$  — тип таблицы дескрипторов (0 —  $GDT$ , 1 —  $LDT$ );  $I$  — индекс дескриптора сегмента в таблице  $GDT$  или  $LDT$ . Этот индекс есть смещение (в байтах) относительно начала таблицы. Идентификатор сегмента  $S$  для защищенного режима обычно называют *селектором сегмента*.

**$GDT$**  — глобальная таблица дескрипторов. В этой таблице находятся дескрипторы сегментов ядра. Эта таблица общая для всех процессов. При этом процесс, выполняемый в данный момент на ЦП, теоретически может инициировать любой из этих сегментов. Но если он находится в состоянии «Задача», то может

вызывать лишь некоторые сегменты ядра, называемые **вентилими**. Вызов любого другого сегмента приведет к возникновению исключения и, как следствие, к уничтожению процесса. Если вызван вентиль, то процесс переходит в состояние «Ядро» и ему доступны все сегменты в *GDT*.

**LDT** — локальная таблица дескрипторов. Для каждого процесса существует своя единственная *LDT*. В ней находятся дескрипторы сегментов процесса, используемые им в состоянии «Задача». Количество сегментов в *LDT* для каждого процесса свое. Наиболее часто в эту таблицу включают сегменты: 1) сегмент кода; 2) сегмент данных; 3) сегмент стека. Реже — сегменты *DLL* и сегменты разделяемой памяти.

Поле *B* дескриптора любого сегмента содержит базовый линейный адрес сегмента — адрес размещения в линейной памяти первой ячейки сегмента. Если в ВС нет аппаратуры управления страницами или она выключена (путем сброса старшего бита в регистре *CR0*), то в качестве линейной памяти, используемой для размещения сегментов процесса, выступает реальная ОП. В этом случае любой адрес  $(S, L) = ((T, I), L)$ , содержащийся в поле машинной команды, преобразуется при попадании этой команды на ЦП в реальный адрес ОП по схеме, изображенной на рис. 36:

$$R = B + L,$$

где *B* — базовый адрес сегмента, дескриптор которого находится или в таблице *GDT* (если  $T = 0$ ), или в таблице *LDT* (если  $T = 1$ ) со смещением *I* байт относительно начала таблицы.

Теперь дополним изложенную выше схему адресации ячеек ОП в защищенном режиме некоторыми существенными деталями: рассмотрим достаточно подробно структуру дескрипторов сегментов, а также используемые для работы с этими дескрипторами регистры и машинные команды. Начнем с вопроса о том, где размещается сегментный виртуальный адрес ячейки ОП —  $(S, L)$ .

Напомним, что в процессоре *i8086* идентификатор сегмента *S* есть номер начального параграфа сегмента. При этом *S* содержится в том сегментном регистре, который используется для адресации рассматриваемой ячейки ОП. Каждая машинная команда «знает», какой сегментный регистр используется для адресации операнда этой команды, если этим операндом является ячейка ОП. Например, команда безусловного перехода *jmp* выполняет переход на ячейку памяти, для адресации которой используется регистр сегмента кода *CS*. Это справедливо и для адресации ячеек ОП в

реальном режиме процессора *i80386*. В защищенном режиме ЦП для хранения  $S$  также используется один из сегментных регистров. Но так как теперь  $S = (T, I)$ , то и содержимое сегментного регистра в защищенном режиме будет другим (рис. 37). Поле  $RPL$  используется для аппаратной защиты информации в ОП и будет рассмотрено нами в п. 5.2.3.

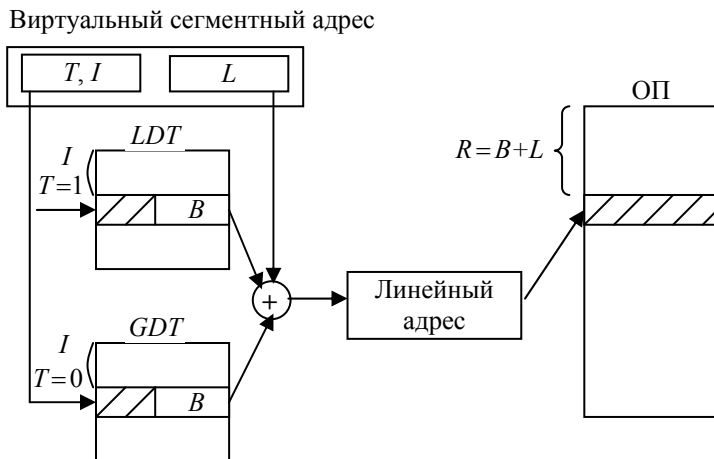


Рис. 36. Преобразование виртуального сегментного адреса в реальный при отсутствии аппаратуры управления страницами:  
 $T$  — тип таблицы дескрипторов (0 —  $GDT$ , 1 —  $LDT$ );  $I$  — индекс дескриптора сегмента в  $GDT$  или  $LDT$ ;  $L$  — смещение относительно начала сегмента;  
 $B$  — базовый линейный адрес сегмента;  $R$  — реальный адрес первой ячейки сегмента

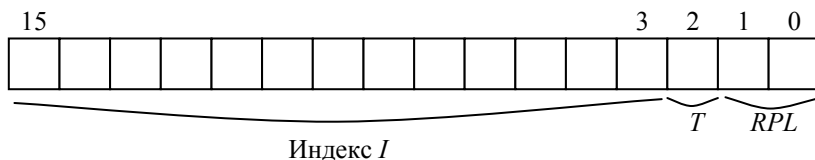


Рис. 37. Структура сегментного регистра в защищенном режиме:  
 $I$  — индекс дескриптора сегмента в таблице  $GDT$  или  $LDT$ ;  
 $T$  — тип таблицы дескрипторов (0 —  $GDT$ , 1 —  $LDT$ );  
 $RPL$  — уровень запрашиваемых привилегий

Обратим внимание, что размер поля индексов  $I$  в сегментном регистре составляет 13 бит. Поэтому максимальная длина (в бай-

тах) *GDT* или *LDT* равна  $2^{13} = 8192$ . Так как длина одного дескриптора равна 8 байтам, то максимальное число дескрипторов в таблице *GDT* или *LDT* будет:  $8192/8 = 1024 = 1 \text{ К}$ . Такое максимальное число сегментов памяти может иметь конкретная прикладная программа, или ядро ОС.

На рис. 38 приведена структура дескриптора сегмента (строка *GDT* или *LDT*). Его длина — 8 байтов, из которых базовый линейный адрес сегмента (*B*) занимает 4 байта, или 32 бита. Так как  $2^{32} = 4 \text{ Г}$ , то общий объем линейного виртуального адресного пространства составляет 4 Гбайт.

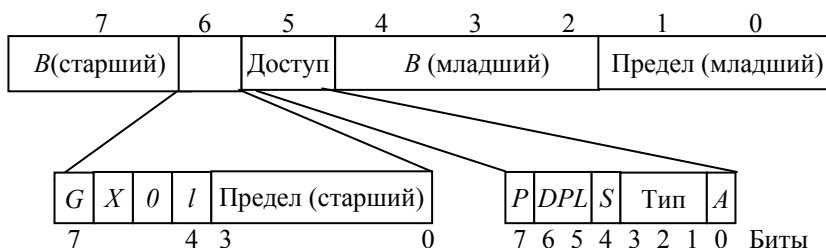


Рис. 38. Структура дескриптора сегмента

Поле **предел** содержит размер сегмента в байтах, уменьшенный на 1. В реальном режиме размер сегмента ОП составляет 64К. В защищенном режиме можно задавать любую длину сегмента от 1 байта до  $2^{20} = 1 \text{ М}$  байтов или страниц. Бит *G* — **бит granularity**. Если  $G = 1$ , то поле предела задает предельный размер сегмента в страницах (по 4096 байтов), иначе — в байтах.

Бит *X* задает размерность машинных команд, выполняемых ЦП. Если  $X = 1$ , то выполняются 32-битные команды, иначе — 16-битные. (16-битные команды остались «в наследство» от процессора *i8086*.) Бит *l* используется операционной системой для своих нужд.

Поле (байт) **доступ** ограничивает операции, которые можно выполнять над сегментом. При этом бит *S* — признак системного сегмента. Если этот бит сброшен в 0, то сегмент системный. Поле *DPL* используется для организации защиты сегмента. Его назначение будет рассмотрено в п. 5.2.3. Биты *P* и *A* используются для организации сегментного свопинга. Эти биты будут рассмотрены в п. 5.2.2. Содержание поля **тип** определяется назначением сегмента. На рис. 39 приведен байт доступа для сегмента кода, содержа-

щего машинные команды программы. Здесь бит  $C$  — бит подчинения (рассматривается в п. 5.4.1).  $R$  — бит разрешения чтения сегмента (1 — чтение разрешено, 0 — чтение запрещено). Что касается записи, то в сегмент кода она запрещена всегда (в отличие от реального режима).

7	6	5	4	3	2	1	0
$P$	$DPL$	1	1	$C$	$R$	$A$	

Рис. 39. Байт доступа для сегмента кода

На рис. 40 приведен байт доступа для сегмента данных. Здесь бит  $D$  задает направление расширения сегмента. Обычный сегмент данных расширяется в направлении старших адресов ( $D = 0$ ). Если же в сегменте расположен стек, расширение происходит в обратном направлении ( $D = 1$ ).  $W$  — бит разрешения записи в сегмент (1 — запись допустима, 0 — нет). Если программа попытается записать в сегмент при  $W = 0$ , то ее выполнение будет прервано. Для системных сегментов поле «тип» принимает другие значения.

7	6	5	4	3	2	1	0
$P$	$DPL$	1	0	$D$	$W$	$A$	

Рис. 40. Байт доступа для сегмента данных

Так как  $GDT$  содержит дескрипторы сегментов ядра ОС, эта таблица заполняется при инициализации ОС. Что касается  $LDT$ , то она содержит дескрипторы сегментов только одного процесса и ее создание инициируется процессом-отцом при выполнении системного вызова *СОЗДАТЬ ПРОЦЕСС*. Если предположить, что мы занимаемся созданием нового процесса самостоятельно, не используя данный системный вызов, то для задания  $LDT$  дочернего процесса наша программа могла бы использовать, например, следующие операторы ассемблера:

; Первая строка любой  $GDT$  или  $LDT$  содержит нули

*Ldt db 0,0,0,0,0,0,0*

; Дескриптор сегмента кода (выполнение, чтение, 32-битные команды)

*db 0FFh, 0FFh, 0, 0, 0, 0FAh, 4Fh, 0*

; Дескриптор сегмента данных (рост вверх, чтение, запись)

*db 0FFh, 0FFh, 0, 0, 0, 0F2h, 4Fh, 0*



При рассмотрении данного примера следует учесть, что в операторе *db* младший байт дескриптора расположен слева, а на рис. 36 — наоборот, справа. В качестве базового линейного адреса для обоих сегментов выбран нулевой адрес. Что касается предельной длины сегментов, то она тоже одинакова (1 Мбайт). Следовательно, оба дескриптора описывают один и тот же сегмент линейной памяти. В первом случае этот сегмент рассматривается как сегмент кода, а во втором — как сегмент данных. В результате процесс имеет по отношению к своей собственной памяти неограниченные права доступа.

Анализ данного примера позволяет сделать еще один вывод. Так как базовый линейный адрес сегмента нулевой, то в данном примере речь может идти только о виртуальном линейном пространстве, для отображения которого на реальные адреса требуется аппаратура управления страницами. (В самом начале реальной ОП находится таблица векторов прерываний, используемая ЦП в реальном режиме работы.)

Область памяти, в которой находится *LDT* будущего процесса, должна быть зарегистрирована процессом-отцом в таблице *GDT* как особый системный сегмент памяти. Для того чтобы выполнить запись соответствующего дескриптора в таблицу *GDT*, эту таблицу необходимо найти в памяти. Это можно сделать, прочитав содержимое регистра ***GDTR***. Этот 6-байтовый регистр содержит 4-байтовый базовый реальный адрес таблицы *GDT* и 2-байтовый размер этой таблицы.

*GDTR* — очень важный регистр, используемый самим ЦП для адресации ячеек памяти. Допустим, например, что в сегментном регистре кода *CS* бит  $T=0$ . Тогда для определения адреса следующей выполняемой команды ЦП просуммирует содержимое *GDTR* с содержимым поля индекса в регистре *CS*, а затем извлечет из найденного дескриптора базовый линейный адрес сегмента кода, который и будет просуммирован со смещением из регистра *EIP*.

Для заполнения регистра *GDTR* программы ядра ОС используют специальную машинную команду *lgdt*. Она выполняется, например, после добавления в *GDT* нового дескриптора *LDT*.

Если в сегментном регистре бит  $T=1$ , то дескриптор искомого сегмента находится в *LDT*. Так как в любой момент времени на ЦП выполняется только один программный процесс, то только одна таблица *LDT* должна быть «видима» процессором. Для этого ЦП имеет 2-байтовый регистр ***LDTR***. Этот регистр содержит индекс дескриптора текущей таблицы *LDT* в таблице *GDT*.

Специальная машинная команда *lldt* выполняет замену содержимого регистра *LDTR*. Записав с ее помощью в *LDTR* индекс дескриптора *LDT* в *GDT*, мы сделаем данную *LDT* текущей. Теперь любой сегментный регистр с битом  $T = 1$  будет указывать на один из сегментов, определенных (с помощью своих дескрипторов) именно в этой таблице. При этом следует отметить, что кроме *LDTR* в ЦП имеется внутренний программно недоступный регистр, содержащий базовый линейный адрес текущей *LDT*, а также ее размер. Запись в него ЦП осуществляет при выполнении команды *lldt* путем копирования полей адреса и размера из дескриптора *LDT* в таблице *GDT*. Наличие такого внутреннего регистра позволяет ЦП достаточно быстро вычислять линейные адреса ячеек в сегментах, определенных в текущей *LDT*.

## 5.2.2. Распределение памяти

В отличие от процессора *i8086*, наличие аппаратуры управления сегментами в процессорах, начиная с *i80386*, обусловлено не стремлением расширить объем адресуемой памяти, а вызвано следующими причинами. Во-первых, благодаря дескрипторам сегментов решается задача аппаратной защиты информации в ОП. При этом возводится достаточно надежный «забор» между сегментами памяти разных процессов. Методы реализации такой защиты будут рассмотрены в п. 5.2.3.

Во-вторых, сегментная организация памяти существенно упрощает использование одной и той же области памяти несколькими процессами. В качестве таких областей могут рассматриваться реентерабельные программы, *DLL*, а также разделяемые области данных. Например, неизменяемый код реентерабельной программы выделяется в каждом из процессов в отдельный сегмент кода. (На аппаратном уровне для сегмента кода обеспечивается запрет какой-либо записи.) При создании первого процесса, использующего данную реентерабельную программу, производится загрузка этой программы в память (ОП + область свопинга), а в *LDT* процесса помещается дескриптор сегмента кода, содержащий виртуальный линейный адрес загруженной программы. При создании всех последующих процессов в их *LDT* также помещаются дескрипторы сегментов кода, содержащие виртуальные линейные адреса реентерабельного сегмента. При отсутствии аппаратуры управления страницами эти линейные адреса будут совпадать, а при наличии такой аппаратуры виртуальные линейные адреса

реентерабельного сегмента кода будут, скорее всего, разными. Так как сегменты данных у каждого из процессов свои, то никакого влияния процессов друг на друга нет.

Реентерабельный код *DLL* также записывается в отдельный сегмент кода, который может использовать любой процесс. Для этого требуется поместить дескриптор сегмента в *LDT* процесса. Очевидно, что каждый процесс должен иметь свой экземпляр сегмента данных *DLL*, а его дескриптор — в своей *LDT*. Использование для информационного взаимодействия между процессами разделяемого сегмента данных предполагает размещение в *LDT* каждого процесса своего дескриптора этого сегмента. Подобное размещение выполняет ядро при выполнении системного вызова *СОЗДАТЬ\_РАЗДЕЛЯЕМЫЙ\_СЕГМЕНТ*.

Перечисленные достоинства сегментной организации памяти уменьшают потребности процессов в ОП, но не устраняют такую потребность совсем. Рассмотрим распределение ОП при наличии аппаратуры управления сегментами и отсутствии аппаратуры управления страницами. Решение этой задачи возможно двумя способами.

Простейший подход предполагает, что суммарный объем сегментной виртуальной памяти всех процессов не может превышать объем ОП. В этом случае для выделения памяти новому процессу ОС рассматривает свободные участки памяти, оставшиеся после первоначальной загрузки в ОП ядра ОС, а также программ предыдущих процессов. Например, допустим, что перед созданием процесса 4 ОП имеет состояние, изображенное на рис. 41.

Реальный адрес	ОП
0 Мбайт	Ядро ОС
1 Мбайт	
2 Мбайт	
3 Мбайт	Процесс 1
3,5 Мбайт	Процесс 3
4 Мбайт	Свободно
5 Мбайт	Процесс 2
6 Мбайт	
7 Мбайт	Свободно
8 Мбайт	
9 Мбайт	

Рис. 41. Пример распределения памяти

Если общая длина программы процесса 4 превышает 4,5 Мбайт, то в данный момент требуемая память выделена быть не может и создание процесса откладывается до ее появления. Если длина программы не превышает суммарного объема свободной ОП, то возможны два варианта. Во-первых, программа может быть легко загружена в память, если для каждого сегмента программы имеется не меньший свободный участок ОП. Если это не так, то ОС должна «сдвинуть» в памяти ранее загруженные программы процессов. Например, пусть программа 4 имеет 2 сегмента длиной по 2 Мбайт. Тогда без перемещения программы 2 в сторону больших или меньших адресов программа 4 загружена быть не может.

Изложенная простейшая схема распределения ОП исходит из предположения, что суммарный объем виртуальной памяти процессов не превышает суммарного объема реальной ОП. (При этом разделяемые сегменты кода или данных учитываются лишь один раз.) Но аппаратура управления сегментами в *i80386* позволяет отойти от этого предположения и загружать в память меньшего объема большую по размеру программу. Иными словами, виртуальная сегментная память процесса может быть больше, чем выделенная процессу реальная ОП.

Реализация подобного подхода основана на наличии в дескрипторе каждого сегмента битов *P* (*бит присутствия*) и *A* (*бит обращения к сегменту*), а также на наличии у процессора исключения **отказ сегмента**. Совместная работа аппаратуры и ядра ОС заключается в следующем.

Когда на ЦП попадает адрес ячейки в виде  $((T, I), L)$ , то аппаратно проверяется бит *P* в искомом дескрипторе. Если  $P = 1$  (сегмент в ОП), то выполнение программы продолжается, иначе возникает исключение **отказ сегмента**. Его обработчик подкачивает требуемый сегмент из ВП. Так как для размещения этого сегмента может потребоваться откачка другого сегмента, то для определения откачиваемого сегмента может помочь бит *A*. Если он сброшен, то к сегменту не было обращения и он может быть откачен.

Перекачка сегментов между ОП и ВП называется **сегментным свопингом**. Распределение ОП, основанное на свопинге, обладает тем существенным недостатком, что вследствие неодинаковой длины сегментов размещение одного сегмента может потребовать

откачку и (или) перемещение в ОП не одного, а нескольких других сегментов.

### 5.2.3. Защита информации в оперативной памяти

Защита информации является необходимым условием функционирования мультипрограммной системы. В любой ВС информация находится в ОП и на ПУ. Так как оперативная память распределяется между процессами в виде сегментов, то требуется исключить воздействие процессов на сегменты памяти, не принадлежащие им. Более того, основной целью сегментного управления памятью как раз и является защита информации в ОП.

От кого должен быть защищен сегмент нашей программы? От чужих сегментов кода и иногда от своих. Основной принцип защиты следующий. Существует ряд ограничений, при нарушении любого из которых возникает исключение (внутреннее аппаратное прерывание), обработчик которого прекращает выполнение текущей программы, то есть уничтожает соответствующий программный процесс. Наиболее широко используется исключение № 0Dh — *общее нарушение защиты (GP)*. Перечислим ограничения, нарушения которых приводят к исключениям.

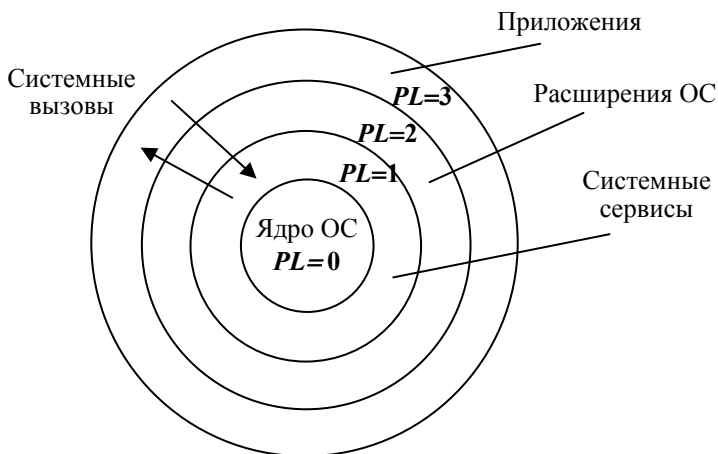
1. Ограничение числа таблиц дескрипторов сегментов, доступных процессу. Любой прикладной или системный обрабатывающий процесс имеет доступ только к двум таблицам: *GDT* (она доступна всем процессам) и к своей *LDT*.

Несмотря на то что в *GDT* находятся дескрипторы для всех *LDT*, существующих в данный момент времени в системе, выполнить загрузку адреса «чужой» *LDT* в *LDTR* текущий процесс не сможет из-за ограничения 6. Поэтому сегменты, описанные в «чужих» *LDT*, для процесса недоступны.

2. Ограничение числа доступных сегментов данных. Доступность для прикладного процесса таблицы *GDT* вовсе не означает доступность сегментов, дескрипторы которых находятся в *GDT*. Это объясняется разницей запрашиваемого и имеющегося уровней привилегий. *Уровень привилегий PL* — целое число от 0 до 3. 0 — самый привилегированный уровень, 3 — самый непривилегированный уровень. Эти уровни можно наглядно изобразить в виде *колец защиты* (рис. 42). На этом рисунке показано, как могут быть распределены по кольцам защиты прикладные и системные программы ВС. В *UNIX*-системах из четырех уровней привилегий

используются только два — 0 и 3. Уровень 0 присущ процессу в состоянии «Ядро», а уровень 3 — в состоянии «Задача».

Основной принцип ограничения числа доступных сегментов данных следующий. Любой сегмент кода, выполняемый на ЦП, имеет свой уровень привилегий *CPL*. В ходе этого выполнения обычно требуется использовать ячейки с данными, принадлежащие другим сегментам, каждый из которых также имеет свой уровень привилегий — *DPL*. Подобное использование возможно только в том случае, если выполняется условие:  $CPL \leq DPL$ . Иначе возникает исключение, обработчик которого прекращает выполнение процесса, содержащего сегмент кода. Например, сегменты из ядра ОС могут обращаться к любой ячейке любого сегмента из *GDT* или



*LDT*, так как  $CPL = 0$ . Теперь рассмотрим вопрос о том, где хранятся уровни привилегий.

Рис. 42. Возможное распределение программ ОС по кольцам защиты

Подпрограмма ядра ОС, выполняющая системный вызов *СОЗДАТЬ\_ПРОЦЕСС*, создает для нового процесса таблицу *LDT* и помещает дескриптор этой таблицы в *GDT*. При этом в поле *DPL* (см. рис. 38) каждого дескриптора в *LDT* помещается уровень привилегий соответствующего сегмента. Допустим теперь, что в результате диспетчеризации новый процесс переводится из состояния «Готов» в состояние «Задача». Иными словами, программа процесса начинает выполняться на ЦП. Подобная передача управления программе производится загрузкой в регистр *CS* индекса

дескриптора сегмента кода в таблице *LDT*. Кроме того, в качестве поля *RPL* в *CS* (см. рис. 37) копируется *DPL* из дескриптора сегмента кода. Теперь до тех пор пока выполняются машинные команды из этого сегмента кода, его *RPL* является текущим уровнем привилегий *CPL*. Следует отметить, что прикладной процесс не может изменить свой *CPL*, то есть его программа не может выполнить запись в поле *RPL* регистра *CS*. Но прочитать это поле программа всегда может.

Теперь допустим, что сегмент кода нашего процесса хочет обрабатывать какие-то ячейки из какого-то сегмента данных. Тогда одна из команд должна загрузить индекс, соответствующий положению в *LDT* или в *GDT* дескриптора этого сегмента, в соответствующий сегментный регистр данных, например в *DS*. Что касается поля *RPL* (см. рис. 37) в этом регистре, то записываемое туда значение называется **уровнем запрашиваемых привилегий**. Команда загрузки в сегментный регистр данных (*DS*, *ES* и т.д.) индекса и *RPL* завершится успешно только в том случае, если выполняется условие

$$(DPL)_{\text{в дескрипторе сегмента данных}} \geq \max [(CPL)_{\text{в } CS}, (RPL)_{\text{в } DS}].$$

При этом очевидно, что запрашивать уровень привилегий лучше, чем *CPL*, бесполезно. Если указанное условие не выполняется, то загрузка в *DS* (и выполнение всей программы) завершится исключением *GP*.

3. Подобно тому как процессу в состоянии «Задача» доступны не все сегменты данных, перечисленные в *GDT*, ему доступны и далеко не все сегменты кода. Иначе бесконтрольные вызовы подпрограмм ядра ОС могут привести к разрушению как самой ОС, так и к разрушению прикладных программ.

С другой стороны, полный запрет вызова подпрограмм ядра прикладной программой также недопустим, так как программа должна иметь возможность обращаться к ОС с целью получения помощи от нее. Для этого прикладной программе должны быть доступны те подпрограммы ядра ОС, которые специально предназначены для обслуживания прикладных программ. Так как процесс в состояниях «Задача» и «Ядро» использует разные сегменты кода, то, во-первых, межсегментные переходы обязательно должны существовать, а во-вторых, эти переходы должны быть ограничены.

4. Ограничение длины доступных сегментов. После того как загрузка сегментного регистра завершилась успешно, процесс

(а точнее — его сегмент кода) получает доступ к ячейкам этого сегмента. Для адресации искомой ячейки при этом используется регистр-смещение. Меняя его содержимое, можно обрабатывать различные ячейки сегмента.

Как показано на рис. 38, одно из полей дескриптора сегмента содержит **предел** — размер соответствующего сегмента. Попытка использовать в текущей команде адрес-смещение, превышающее предел сегмента, неизбежно приведет к исключению.

5. Ограничение формы доступа к сегменту. Основные формы доступа — «чтение», «запись» и «выполнение». Для сегментов данных возможными формами доступа являются «чтение» и «запись», а для сегмента кода — «выполнение» и «чтение».

Для конкретного сегмента могут быть разрешены не все возможные формы доступа. Для этого в дескрипторе сегмента кода (см. рис. 39) бит  $R$  разрешает ( $R = 1$ ) или запрещает ( $R = 0$ ) чтение из сегмента кода. В дескрипторе сегмента данных (см. рис. 40) бит  $W$  разрешает ( $W = 1$ ) или запрещает ( $W = 0$ ) запись в сегмент данных.

6. Ограничение множества допустимых машинных команд. Находясь в сегменте кода с уровнем привилегий  $CPL$ , мы можем выполнять только такие команды, которые соответствуют этому уровню.

В частности, при  $CPL = 0$  допустимы все команды используемого ЦП. Некоторые из этих команд разрешены только в этом приоритетном кольце. Они называются **привилегированными командами**. Перечислим их:

- $lgdt$  — загрузка  $GDTR$ ;
- $lldt$  — загрузка  $LDTR$ ;
- $lidt$  — загрузка регистра таблицы дескрипторов прерываний  $IDTR$ ;
- $ltr$  — загрузка регистра задачи  $TR$ ;
- $lmsw$  — загрузка регистра состояния машины (это 16 младших битов регистра  $CR0$ );
- $clts$  — сброс флага переключения задачи;
- $hlt$  — останов ЦП.

Кроме привилегированных команд, выполнение некоторых других команд также ограничено. Сюда относятся команды ввода-вывода и команды для работы с флагом разрешения прерываний  $IF$ . Контроль над этим флагом очень важен, так как программа, сбросившая его и вошедшая в бесконечный цикл, в принципе не может быть снята с ЦП никаким способом кроме перезагрузки ОС.



Это обусловлено тем, что завершение процесса с заиклившейся программой выполняет или обработчик прерываний клавиатуры (при нажатии специальной комбинации клавиш), или обработчик прерываний таймера (по истечении кванта времени процесса). Так как прерывания обоих этих типов являются маскируемыми, то при  $IF = 0$  они запрещены.

Изменение флага  $IF$  выполняют две команды: *sti* и *cli*. Их применение в программе процесса допустимо при выполнении условия:  $CPL \leq IOPL$ , где  $IOPL$  — число в битах 12 и 13 регистра флагов *EFLAGS*. Программы, работающие не в нулевом кольце, не могут модифицировать поле  $IOPL$ . Поэтому при выполнении команд, загружающих *EFLAGS* (команды *iret* и *popf*) не в нулевом кольце, поле  $IOPL$  не модифицируется. При этом также не меняется флаг  $IF$ . Команды *iret* и *popf* являются примерами **чувствительных команд**, функции которых зависят от текущего уровня привилегий процесса.

7. Ограничение доступа к периферийным устройствам. Доступ к ПУ программа осуществляет через порты, используя команды *in*, *out*, *ins* и *outs*. Контроль над использованием этих команд важен не только для защиты информации, расположенной на ПУ (например на диске), но и необходим для защиты информации в ОП. Это обусловлено тем, что с помощью этих команд можно, например, перепрограммировать контроллеры прерываний и прямого доступа в память (ПДП), что откроет путь к непосредственной модификации содержимого ОП по реальным адресам.

Ограничение использования команд ввода-вывода основано на совместном использовании упомянутого выше поля  $IOPL$  в регистре флагов *EFLAGS* и битовой карты ввода-вывода. **Битовая карта ввода-вывода** — битовая строка, в которой каждый бит соответствует одному порту ввода-вывода. Причем номер бита в карте равен номеру (адресу) порта. Каждому процессу назначается своя карта ввода-вывода.

Если  $CPL \leq IOPL$ , то на операции ввода-вывода не накладывается никаких ограничений. Если  $CPL > IOPL$ , то ЦП проверяет тот бит в карте ввода-вывода процесса, который соответствует адресуемому порту. Если этот бит равен 0, то текущая команда ввода-вывода выполняется. Иначе возникает исключение. Оно возникает и в том случае, если адресуемому порту не соответствует никакой бит в карте ввода-вывода (из-за ее короткой длины).

## 5.3. Линейная виртуальная память

### 5.3.1. Преобразование адресов

Если включена аппаратура управления страницами (установкой в 1 старшего бита регистра  $CR0$ ), то ОС записывает в дескрипторы сегментов не линейные реальные адреса, а **линейные виртуальные адреса**. Вопрос о том, как ОС выбирает эти адреса, мы рассмотрим позже, а пока остановимся на преобразовании этих адресов в реальные аппаратурой управления страницами.

Допустим, что вся реальная ОП разделена на участки фиксированной длины, называемые **физическими страницами**. (При использовании в качестве ЦП *i80386* длина страницы составляет 4096 байтов.) Программы, выполняемые на ЦП, также считаются разделенными на участки этой же длины, называемые **логическими страницами**. Преобразование логической страницы в физическую выполняют совместно ОС и аппаратура управления страницами. При этом ОС выполняет инициализацию таблиц, используемых для аппаратного преобразования линейных адресов, а также выполняет главную часть страничного свопинга.

Допустим пока, что свопинг не требуется, так как каждой логической странице соответствует своя физическая страница. В этом случае каждый линейный виртуальный адрес, выдаваемый аппаратурой управления сегментами, преобразуется в реальный адрес по схеме на рис. 43. Так как длина линейного виртуального адреса — 32 бита, то общий объем линейного виртуального адресного пространства составляет  $2^{32} = 4$  Гбайт.

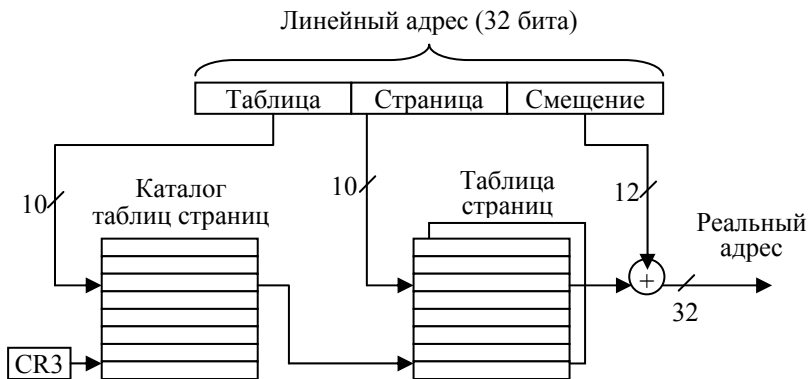


Рис. 43. Преобразование линейного виртуального адреса в реальный

Как видно из рис. 43, 32 бита линейного адреса аппаратно разделяются на три части. Старшие 10 битов используются как индекс поиска (здесь индекс — номер строки) в таблице, называемой **каталогом таблиц страниц**. Каждая строка этой таблицы содержит дескриптор одной из таблиц страниц. Максимальное число строк (дескрипторов) в таблице-каталоге — 1024 ( $2^{10}$ ). Причем количество самих каталогов не ограничено. В любой момент времени существует каталог, называемый **текущим каталогом** (не путать с текущим каталогом файлов). Начальный реальный адрес этого каталога содержится в регистре *CR3*. Смена содержимого этого регистра приводит к смене текущего каталога.

На рис. 44 приведена структура дескриптора таблицы страниц. Поясним содержание полей этого дескриптора:

1) в битах 12–31 находятся старшие 20 битов реального адреса в ОП таблицы страниц, соответствующей дескриптору. Младшие 12 битов этого адреса таблицы страниц всегда равны нулю. Поэтому в ОП таблица страниц может находиться только по адресу, кратному числу 4096 ( $2^{12}$ ), то есть на границе физической страницы;

2) *AVL* — биты, используемые ОС по своему усмотрению;

3) *D* — бит устанавливается в 1, если была выполнена запись в таблицу страниц;

4) *A* — бит доступа, он устанавливается в 1 ЦП перед выполнением операции чтения (записи) из таблицы (в таблицу) страниц;

5) *U* — бит принадлежности таблицы страниц (0 — таблица страниц ядра, 1 — таблица страниц прикладной программы);

6) *W* — бит разрешения записи (1 — запись в таблицу страниц разрешена, 0 — запись не разрешена);

7) *P* — бит присутствия в памяти (1 — таблица страниц находится в ОП, 0 — нет).

31		12	11	10	9	8	7	6	5	4	3	2	1	0
Реальный адрес таблицы страниц				<i>AVL</i>	0	<i>D</i>	<i>A</i>	0	<i>U</i>	<i>W</i>	<i>P</i>			

Рис. 44. Структура дескриптора таблицы страниц

Следующие 10 битов линейного адреса являются индексом в таблице страниц, выбранной с помощью старших десяти битов адреса. **Таблица страниц** состоит из 1024 **дескрипторов страниц**, каждый из которых описывает одну логическую страницу. Дескриптор страницы имеет тот же состав полей, что и дескриптор таблицы страниц (см. рис. 44). Единственное уточнение: старшие 20 битов дескриптора страницы содержат 20 старших битов (из 32)

начального реального адреса соответствующей физической страницы.

Младшие 12 битов линейного виртуального адреса (см. рис. 43) содержат смещение адресуемой ячейки ОП относительно начала страницы. В результате аппаратного суммирования этого смещения с начальным реальным адресом соответствующей физической страницы получается искомый реальный адрес ячейки ОП.

### 5.3.2. Распределение памяти

Как следует из подразд. 5.2, аппаратура управления сегментами обеспечивает два свойства линейных виртуальных адресов: 1) каждый такой адрес не может быть больше 4 Гбайт; 2) любой виртуальный сегмент отображается на непрерывный участок линейной памяти. С учетом этих свойств сохраняется большая свобода выбора начальных линейных адресов сегментов, записываемых ОС в дескрипторы сегментов (поле *B* на рис. 36, 38). Эти адреса выбираются ОС следующим образом.

Во-первых, программа каждого процесса «загружается» в собственную линейную виртуальную память размером 4 Гбайт. **Линейная виртуальная память (ЛВП)** — абстракция, используемая не самой программой (она работает с сегментной виртуальной памятью), а операционной системой.

Во-вторых, в эту ЛВП «загружается» и сама ОС. В *UNIX* используется распределение линейной виртуальной памяти процесса, изображенное на рис. 45. При этом 3 Гбайт с меньшими адресами отводятся самой прикладной программе, а 1 Гбайт с большими адресами — для ядра ОС. Причем системные *DLL* обычно располагаются вверху прикладной части ЛВП.

В-третьих, если в ходе выполнения процесса ему понадобится дополнительная ОП, то запрашиваемый объем будет выделен ОС из подходящей свободной области ЛВП динамически. (Всякое назначение памяти процессу, осуществляемое в ходе его выполнения, называется **динамическим распределением памяти**. **Статическое распределение** — память выделяется при создании процесса.)

В-четвертых, ОС выполняет «загрузку» сегментов процесса в ЛВП путем записи начального линейного виртуального адреса каждого сегмента в его дескриптор. При этом сегменты ядра (их дескрипторы находятся в *GDT*) всегда «загружены» в ЛВП по одним и тем же адресам.

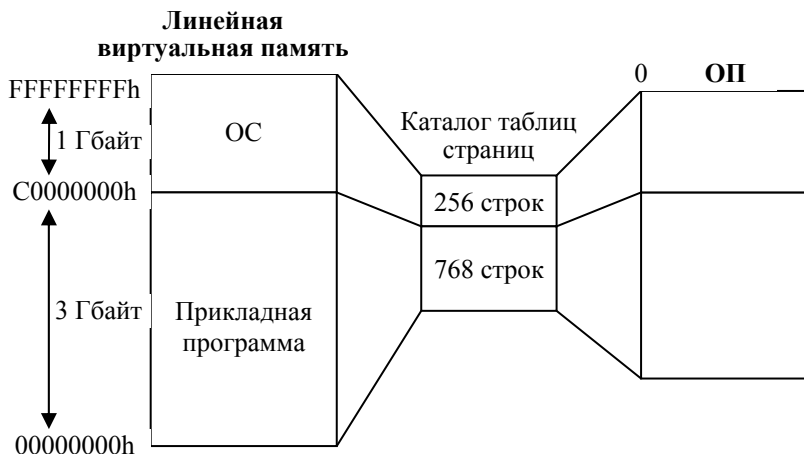


Рис. 45. Отображение линейной виртуальной памяти на реальную ОП

В-пятых, содержимое ЛВП используется ОС для выполнения записей в каталог таблиц страниц и в сами таблицы страниц (инициализация таблиц). Один каталог страниц обычно используется для адресации памяти одного процесса. При этом из 1024 строк каталога 256 строк соответствуют ОС, а 768 строк — прикладной программе. (При смене на ЦП выполняемого процесса 256 строк в каталоге не меняются, а заменяются лишь 768 строк.) Если 4 Мбайта ЛВП, соответствующие данной строке каталога, содержат какую-то информацию, то в эту строку ОС помещает указатель на соответствующую таблицу страниц. Иначе строка каталога содержит пустой указатель.

Таблица страниц, соответствующая непустой строке каталога, может содержать до 1024 строк, каждая из которых содержит дескриптор одной логической страницы программы. Так как объем одной страницы составляет 4096 байт, то максимальный объем ЛВП, соответствующий одной таблице страниц, составляет 4 Мбайт. Любая часть этого объема может соответствовать «пустым» страницам. Так как ОС ведет учет распределения ЛВП, то «пустые» страницы ей известны и она не создает для них дескрипторы в таблице страниц.

Естественно, что при создании процесса ОС распределяет ему не только ЛВП, но и реальную ОП. Не предоставив процессу хотя бы небольшое число физических страниц, нельзя обеспечить его выполнение. При этом системная часть программы процесса (256

строк в каталоге таблиц страниц) всегда отображается на одну и ту же часть реальной ОП (с меньшими адресами). А прикладная часть ЛВП отображается на ту совокупность физических страниц, которые ОС выделила процессу. Страничное распределение ОП обладает тем существенным достоинством, что вследствие одинаковой длины страниц любая логическая страница может быть загружена в любую физическую страницу. Поэтому не только страницы процесса, но и страницы логического сегмента не образуют непрерывный раздел ОП, а располагаются в произвольных местах памяти.

Благодаря *свопингу страниц* число логических страниц процесса может быть значительно больше, чем выделенное ему число физических страниц. Для организации такого свопинга аппаратура процессора *i80386* предоставляет в помощь ОС исключение *отказ страницы*, а также специальные биты в дескрипторах страниц, аналогичные соответствующим битам в дескрипторе таблицы страниц (см. рис. 44):

- 1) *P* — бит присутствия страницы в памяти (1 — страница в ОП; 0 — нет);
- 2) бит *D* — устанавливается в 1, если была выполнена запись в страницу;
- 3) *AVL* — три бита, которые ОС может использовать по своему усмотрению.

Совместная работа аппаратуры и ядра ОС при реализации страничного свопинга заключается в следующем. Выполнив разделение очередного виртуального линейного адреса, аппаратура ЦП проверяет бит *P* в искомом дескрипторе страницы. Если *P* = 1, то выполнение программы продолжается, иначе возникает исключение «отказ страницы». Его обработчик подкачивает требуемую страницу из ВП.

Обычно страница загружается на место ранее загруженной страницы, которая или копируется из ОП на диск (если бит *D* = 1) или нет (бит *D* = 0). Для определения откачиваемой страницы могут использоваться различные критерии. Например, в качестве такой страницы может быть выбрана та, к которой было сделано наименьшее число обращений. В качестве счетчика числа обращений может быть использовано поле *AVL* дескриптора страницы.

Распределение ОП, основанное на страничном свопинге, обладает тем существенным достоинством по сравнению с сегментным

свопингом, что вследствие одинаковой длины страниц размещение одной страницы в ОП может быть выполнено вместо любой другой. Поэтому при размещении в памяти новой страницы не требуется выполнять каких-либо перемещений ранее загруженных страниц.

Что касается защиты информации в ОП, то аппаратура управления страницами в *i80386* предоставляет для этого единственное средство: при выполнении любой машинной команды, выполняющей запись в ОП, аппаратно проверяется бит  $W$  в дескрипторе той страницы, в которую выполняется запись. При  $W = 1$  команда записи производится, а при  $W = 0$  возникает исключение. Эффективность данного средства существенно ниже защитных действий, выполняемых аппаратурой управления сегментами (см. п. 5.2.3).

## 5.4. Переключение процессора

Переключение ЦП требуется выполнять в трех случаях:

- 1) при выполнении межсегментных переходов в пределах текущего процесса;
- 2) для смены выполняемого процесса;
- 3) для обработки прерываний.

Все эти переключения ЦП выполняются чисто аппаратными средствами.

Роль программы сводится к выдаче команды передачи управления, инициирующей переключение ЦП.

### 5.4.1. Межсегментные переходы внутри процесса

Внутрисегментные переходы в защищенном режиме ЦП выполняют те же машинные команды передачи управления, что и в реальном режиме: *jmp*, *call*, *ret*. Для этого, как и в реальном режиме, в качестве параметра команд *jmp* и *call* задается внутрисегментное смещение, команда *ret* получает это смещение из стека. В результате выполнения команды заданное смещение будет записано в качестве нового содержимого указателя команды *EIP*. Единственное отличие: при выполнении внутрисегментного перехода в защищенном режиме ЦП проверяет, не вышел ли адрес перехода за пределы данного сегмента.

Эти же команды переходов могут быть использованы и для межсегментных переходов в пределах текущего процесса. Требуе

пояснения термин «эти же команды». Речь идет о совпадении ассемблерных мнемоник команд, но не о машинных кодах операций, которые для внутрисегментных и межсегментных переходов будут различны.

При межсегментной передаче управления меняется не только содержимое *EIP*, но и *CS*. Далее мы будем называть сегмент, из которого делается переход, **исходным сегментом**, а сегмент, в который переход делается, — **целевым сегментом**. В основе ограничений на межсегментные переходы лежит сравнение текущего уровня привилегий *CPL* (совпадает с *DPL* исходного сегмента) и *DPL* целевого сегмента. Если результаты сравнения неудовлетворительны, возникает исключение и процесс уничтожается.

Межсегментные переходы с помощью команд *jmp* и *call* можно выполнить двумя способами. Первый способ — прямой вызов целевого сегмента кода. Он предполагает задание в качестве операнда команды *jmp* (*call*) селектора этого сегмента кода. Второй способ — косвенный вызов целевого сегмента при использовании вентильного вызова. Сначала рассмотрим прямые вызовы.

Реализация межсегментных прямых переходов схематично показана на рис. 46. На выполнение прямого вызова целевого сегмента влияет *бит подчинения C* — бит 2 в байте доступа дескриптора целевого сегмента (см. рис. 39). Если  $C = 0$ , то целевой сегмент называется **несогласованным**. Такой сегмент может быть вызван только из выполняемого сегмента, имеющего не меньшие привилегии, чем целевой сегмент:  $CPL \leq DPL$ .

Если процесс находится в состоянии «Задача», то его программа выполняется в кольце 3, и поэтому она не может вызывать несогласованные сегменты, находящиеся в кольце 0. Это позволяет блокировать несанкционированные вызовы модулей ядра ОС. Поэтому в системе *UNIX* прямые межсегментные переходы используются или для передачи управления между прикладными сегментами (процесс в состоянии «Задача»), или между системными сегментами (процесс в состоянии «Ядро»). При возврате из целевого сегмента с помощью команды *ret* также проверяется уровень привилегий того сегмента, в который делается переход. Так как такой переход возможен только в сегмент, имеющий не лучший уровень привилегий, то это предотвращает использование команды *ret* для несанкционированного доступа к модулям ядра.



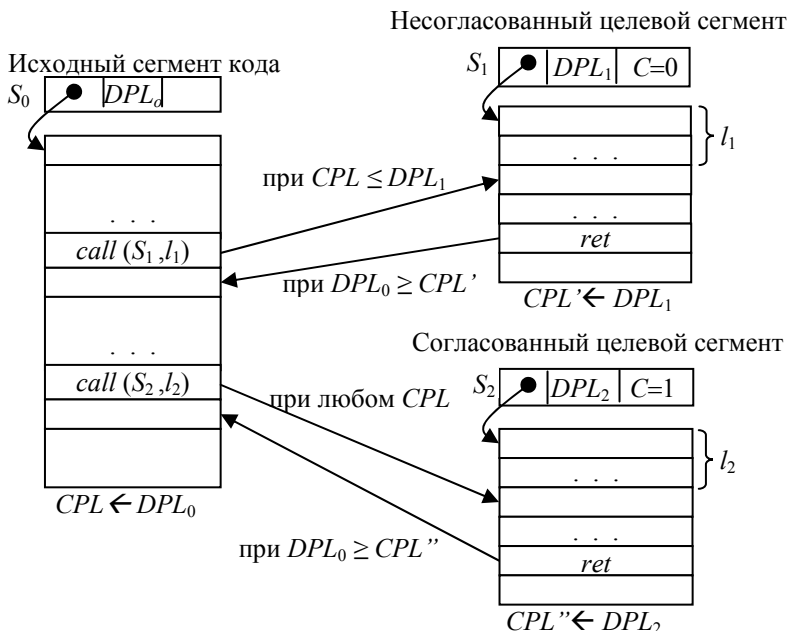


Рис. 46. Прямые межсегментные переходы

Один из способов предоставления обслуживания прикладным программам со стороны модулей ядра ОС состоит в размещении этих модулей в **согласованных сегментах**. Байт доступа дескриптора такого сегмента имеет бит  $C = 1$ . Согласованный сегмент можно вызывать из программных сегментов, находящихся на любом кольце защиты (см. рис. 46). Но всегда после своего вызова такой целевой сегмент будет выполняться с привилегиями исходного сегмента. Так как фактического переключения ЦП в более приоритетный режим не происходит, то данный метод имеет ограниченное применение для реализации системных вызовов *UNIX*.

Для реализации системных вызовов удобно использовать метод косвенных переходов, основанный на применении вентилей вызова. Каждый **вентиль вызова** представляет собой особый системный дескриптор (рис. 47).

В вентиле вызова селектор задает целевой сегмент, а смещение указывает на точку входа в этот сегмент. Благодаря смещению один и тот же сегмент может иметь несколько точек входа. Поле *DPL* вентиле вызова задает минимальный уровень привилегий, необходимый для получения доступа через этот вентиль. Например, если  $DPL = 3$ , то любая программа может получить доступ к целевому сегменту, несмотря на то что приоритет этого сегмента наивысший (0). Благодаря заданию в вентиле смещения обеспечивается гарантия того, что не будет осуществлен доступ к запретной части сегмента.

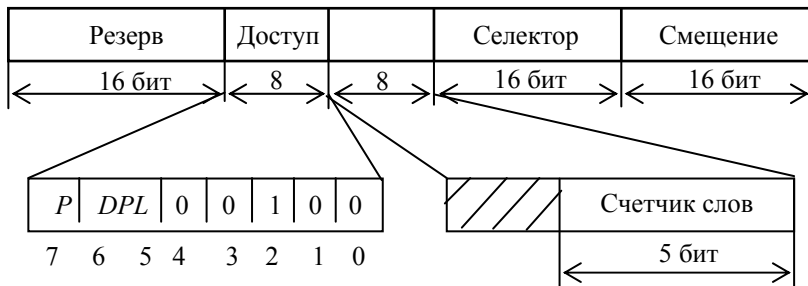


Рис. 47. Структура вентиле вызова

На рис. 48 схематично показана реализация косвенного перехода к целевому сегменту с помощью команды *call* и обратного перехода к исходному сегменту с помощью команды *ret*. Данная схема не содержит сведений о передаче информации на вход целевого сегмента, то есть параметров соответствующего системного вызова. Для такой передачи используется следующий подход: так как исходный и целевой сегменты выполняются в разных кольцах защиты, то они используют разные стеки. При вызове целевого сегмента через вентиль вызова ЦП аппаратно копирует из стека вызывающей программы столько 16-битных слов, сколько задано в 5-битном счетчике слов в вентиле вызова. Максимальное число слов — 31.

Вентиль вызова представляет собой идеальное средство для реализации системных вызовов. С одной стороны, все точки входа в ядро могут быть представлены в *GDT* с помощью своих вентилях входа, а с другой — исключается возможность несанкционированного доступа к внутренним модулям ядра.

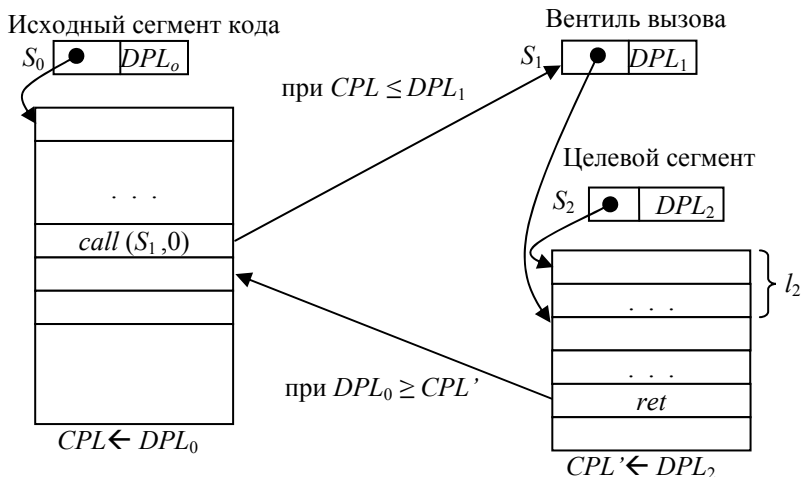


Рис. 48. Косвенный переход через вентиль вызова

### 5.4.2. Аппаратное переключение процессов

Инициирование процесса диспетчером, то есть перевод этого процесса из состояния «Готов» в состояние «Задача», производится путем записи в регистры ЦП значений, образующих аппаратный контекст процесса. Так как в состав этих регистров входят и адресные регистры (например сегментные регистры), то замена аппаратного контекста процесса приводит к замене полного контекста процесса, выполняемого на ЦП. (**Полный контекст процесса** образуют его аппаратный контекст и содержимое его виртуальной памяти.) В *i80386* замена текущего аппаратного контекста реализована в виде неделимой аппаратной операции, называемой **переключением задач**. Как будет показано далее, такая операция может быть инициирована или командой *jmp*, или командой *call*.

Для выполнения переключения задач ЦП должен, во-первых, записать куда-то аппаратный контекст исполнявшегося процесса, а во-вторых, записать в регистры ЦП контекст иницируемого процесса. Для запоминания аппаратных контекстов процессов используются специальные системные сегменты памяти — **сегменты состояния задачи (TSS)**. Как и дескрипторы других системных сегментов, дескрипторы *TSS* находятся в *GDT*. Структура *TSS* показана на рис. 49. Рассмотрим содержимое полей этой структуры.

31

0

База карты ввода-вывода	0	T
0	LDTR	
0	GS	
0	FS	
0	DS	
0	SS	
0	CS	
0	ES	
EDI		
ESI		
EBP		
ESP		
EBX		
EDX		
ECX		
EAX		
EFLAGS		
EIP		
CR3		
0	SS2	
ESP2		
0	SS1	
ESP1		
0	SS0	
ESP0		
0	LINK	
Любая информация ОС		
Карта ввода-вывода		

Рис. 49. Сегмент состояния задачи TSS

База карты ввода-вывода содержит адрес-смещение этой карты относительно начала TSS. Карта ввода-вывода завершается байтом FFh. Предел сегмента TSS записывается с учетом этого байта.

Бит  $T$  — бит трассировки. Если он установлен, то переключение на данный процесс приведет к тому, что завершение каждой машинной команды процесса будет приводить к возникновению отладочного исключения, которое можно использовать в каком-то отладчике.

Поле  $LDTR$  нужно потому, что каждая задача имеет свою таблицу  $LDT$ , и при переключении ЦП на эту задачу в  $LDTR$  должен быть загружен указатель на эту таблицу.

Следующие поля используются для хранения содержимого других регистров. При этом поле  $CR3$  содержит указатель на каталог таблиц страниц. Поэтому каждая задача изолирована от других задач не только за счет своего набора сегментов, но и за счет своего набора страниц.

Аппаратура *i80386* позволяет выполнять один и тот же процесс попеременно в разных кольцах защиты. (Напомним, что в системе *UNIX* используются два таких кольца: 0 — режим ядра, 1 — режим задачи.) Для этого процесс должен иметь несколько стеков — по одному для каждого режима. Указатель на стек для  $i$ -го кольца содержится в паре полей:  $SSi$  и  $ESPi$ . При переходе процесса в кольцо  $i$  содержимое этих полей загружается в регистры  $SS$  и  $ESP$ .

Для инициирования одного процесса из другого могут использоваться команды дальних переходов *jmp* и *call*. Для этого в качестве адреса-сегмента должен быть задан селектор того  $TSS$ , который соответствует запускаемому процессу. Если переключение ЦП происходит в результате выполнения им команды *call*, то в поле  $LINK$   $TSS$  вызванного процесса помещается селектор  $TSS$  вызывающего процесса. Кроме того, устанавливается флаг  $NT = 1$  (флаг вложенной задачи) в поле  $EFLAGS$   $TSS$  вызванного процесса.

Для возврата из вызванного процесса в вызывающий используется команда *iret*. При выполнении этой команды проверяется флаг  $NT$ . Если он сброшен, то команда *iret* выполняется как обычно, осуществляя межсегментный переход в контексте выполняемого процесса. Если же флаг  $NT = 1$ , то производится переключение ЦП на контекст вызвавшего процесса при использовании поля  $LINK$ .

Отметим следующее серьезное различие между вызовами процедуры и процесса. В то время как после возврата из процедуры при ее повторном вызове мы опять попадем в ее начальную точку входа, при повторном вызове процесса мы передадим управление команде, следующей за *iret*. Это происходит потому, что при

переключении ЦП на контекст другого процесса в *TSS* сохраняется содержимое регистров *CS* и *IP* на момент переключения. Поместив после *iret* команду безусловного перехода *jmp* на начало программы процесса, мы сделаем его вызов похожим на вызов процедуры.

Если для переключения аппаратного контекста используется команда *jmp*, то флаг *NT* в *TSS* нового процесса сбрасывается. Для возврата в старый процесс также используется команда *jmp*, в качестве операнда которой используется селектор старого *TSS*.

Подобно тому как для вызова подпрограмм могут быть использованы вентили вызова, для инициирования процессов могут использоваться **вентили задач**. Структура вентиля задачи аналогична структуре вентиля вызова (см. рис. 47). Единственное отличие — два младших бита байта доступа в вентиле задач содержат двоичное число 01.

При переключении ЦП с одного процесса на другой процессору требуется знание не только селектора *TSS* нового процесса, но и селектора *TSS* текущего процесса (который пока еще находится на ЦП), для того чтобы запомнить аппаратный контекст этого процесса в его *TSS*. Для хранения селектора *TSS* текущего процесса в ЦП имеется специальный 16-битный регистр — *TR* (*Task Register*).

### 5.4.3. Обработка прерываний

**Прерыванием** называется навязанное принудительно центральному процессору прекращение выполнения текущей программы и переход им на выполнение подпрограммы, которая называется **обработчиком прерываний**. В работе любой ВС (за исключением простейших) прерывания играют исключительно важную роль. Приводимые ниже примеры иллюстрируют это.

Допустим, что на ЦП выполняется прикладная программа, вошедшая в бесконечный цикл вследствие ошибки программирования. Тогда для прекращения выполнения этой программы достаточно нажать ту комбинацию клавиш, которая специально предназначена для этой цели в используемой ВС. Например, в *UNIX* это одновременное нажатие клавиш *<Ctrl>&<C>*, *<Ctrl>&<\>* или *<Ctrl>&<Z>*. Следствием этого является выдача интерфейсным устройством клавиатуры аппаратного сигнала прерывания, который поступает в ЦП по ОШ. Процессор прерывает выполнение заикливающейся программы и начинает выполнять

обработчик прерываний клавиатуры. Эта подпрограмма распознает код нажатой комбинации клавиш, например `<Ctrl>&<C>`, и обеспечивает завершение программы. (Обработчик прерываний клавиатуры прекращает выполнение текущей программы не сам, а использует для этого сигнал *SIGINT* — см. п. 2.4.2.)

Прерывания от клавиатуры являются примером **внешних аппаратных прерываний**. Другим примером таких прерываний являются прерывания от таймера (см. подразд. 4.5). Сигналы внешних аппаратных прерываний выдаются в ЦП различными ПУ (а точнее — их ИУ), которым требуется внимание со стороны программ ЦП. Эти сигналы передаются в ЦП по шине управления, которая входит в состав ОШ. Характерной особенностью внешних аппаратных прерываний является то, что их первичной причиной являются процессы, асинхронные (независимые) по отношению к текущей работе ЦП. Примером такого асинхронного процесса является деятельность пользователя по нажатию клавиш, приводящая к возникновению прерываний от клавиатуры и от мыши. Аппаратные процессы, протекающие в ПУ, также могут являться первичными источниками внешних аппаратных прерываний.

Характерной особенностью сигналов внешних аппаратных прерываний является то, что они должны обрабатываться в реальном времени, то есть интервал времени между моментом появления сигнала прерывания и завершением его обработки не должен превышать предельно допустимой величины, которая зависит от типа сигнала прерывания. Например, для таймера этот интервал не должен превышать один тик. Задержка завершения обработки на большую величину приведет к потере тика и, следовательно, к нарушению правильности системного времени.

Внешние аппаратные прерывания разделяются на маскируемые и немаскируемые прерывания. **Немаскируемые прерывания** — наиболее важные прерывания, обработка которых не может быть отложена ни на какое время. Сюда относятся прерывания, возникающие при сбоях питания. Соответствующие сигналы прерываний выдаются в ЦП аппаратными схемами контроля питания. **Маскируемые прерывания** — прерывания, обработка которых может быть отложена на время, требуемое ЦП для выполнения какой-то другой, более важной операции. Запрет всех маскируемых прерываний выполняет машинная команда *cli*, а разрешение — команда *sti*. Первая из этих команд сбрасывает, а вторая устанавливает флажок разрешения прерываний *IF* в регистре *EFLAGS*.

Кроме внешних аппаратных прерываний существуют также **внутренние аппаратные прерывания**, называемые обычно **исключениями**. Источником аппаратного сигнала исключения является одна из аппаратных схем самого ЦП. Сигнал выдается в том случае, если при выполнении на ЦП очередной машинной команды возникла ситуация, требующая помощи со стороны системных программ. Например, если при выполнении на ЦП команды *div* (деление) получилось частное, величина которого превышает предельно допустимую емкость рабочего регистра, то возникает исключение «деление на ноль». Обработчик данного исключения посылает сигнал *SIGFPE*, стандартная обработка которого приводит к завершению процесса.

В качестве второго примера рассмотрим исключение «трасировка». Данное исключение будет происходить при выполнении на ЦП каждой машинной команды, если установлен флаг *TF* в регистре *EFLAGS*. В результате этого наша прикладная программа будет «спотыкаться» — по завершении каждой команды программы будет инициироваться обработчик исключения, выполняющий посылку сигнала *SIGTRAP*. Обработчик данного сигнала выполняется в контексте текущего процесса и поэтому может вывести на экран информацию о состоянии этого контекста. В частности, на экран может быть выведено содержимое регистров, образующих аппаратный контекст процесса.

Третий тип прерываний — **программные прерывания** (термин неудачный, но общепринятый). Причиной такого прерывания является сама программа (отсюда и название), а именно — попадание на ЦП машинной команды *int*. Подобно исключениям, программные прерывания являются синхронными с текущей работой ЦП, так как обслуживают выполняемую на нем программу. Но в отличие от исключений, моменты возникновения которых автору прикладной программы заранее не известны, команды *int* помещаются программистом в текст программы сознательно с целью выполнения системных вызовов. Заметим, что в однопрограммной ВС, например, на основе *MS-DOS*, не существует иного способа вызова из прикладной программы системных управляющих подпрограмм (ОС или *BIOS*) кроме применения команды программного прерывания *int*.

Все типы прерываний в системе пронумерованы. **Номер прерывания** выполняет роль не только системного, но и программного имени сигнала прерывания, используемого в качестве параметра команды *int*. Это позволяет вызывать подпрограмму обработчика



прерываний не по ее начальному адресу (как в команде *call*), а по номеру, что намного удобнее. Кроме того, замена того или иного обработчика прерываний не требует внесения изменений в тексты прикладных программ, так как номер прерывания остается прежним. Естественно, что при передаче сигналов аппаратных прерываний передача номера прерывания требует намного меньше линий, чем потребовалось бы для передачи адреса. Примеры номеров прерываний:

- 0 — деление на нуль;
- 1 — трассировка;
- 2 — немаскируемое прерывание.

Для того чтобы ЦП мог выполнить преобразование номера прерывания в начальный адрес соответствующего обработчика прерываний, используется **вектор прерываний** — небольшая область данных, содержащая этот адрес. Все векторы прерываний сведены в единую **таблицу векторов прерываний**. При поступлении сигнала прерывания в ЦП его аппаратура определяет по номеру прерывания расположение соответствующего вектора прерываний, считывает из него стартовый адрес обработчика прерываний и загружает его в регистры *CS* и *EIP*, выполняя тем самым инициирование обработчика прерываний. Перед этим прежние содержимые *CS* и *EIP* запоминаются в стеке. Там же запоминается и содержимое на момент прерывания регистра *EFLAGS*. В конце программы обработчика прерываний обычно находится команда *iret*, выполняющая загрузку в регистры *CS, EIP* и *EFLAGS* значений, находящихся в памяти стека. Результат выполнения этой команды: следующей командой, выполняемой на ЦП, будет очередная команда прерванной программы. Изложенная общая схема обработки прерываний справедлива и для реального, и для защищенного режимов работы ЦП. Но в каждом из этих режимов имеются важные особенности.

В реальном режиме, который «достался в наследство» от процессора *i8086*, объем адресного пространства ОП составляет всего 1 Мбайт, а все регистры ЦП 16-битные. Длина вектора прерываний составляет 4 байта. Первые два байта содержат смещение, а вторые — сегмент стартового адреса обработчика прерываний. Таблица векторов прерываний занимает первые 1024 байта ОП, и их никогда нельзя использовать для других целей. Следовательно, общее число векторов прерываний — 256. Таково максимальное число типов прерываний, которые могут существовать в системе.

Перед тем как инициировать обработчик прерываний, в ЦП аппаратно сбрасываются флаг трассировки *TF* и флаг разрешения прерываний *IF*. Поэтому если конкретный обработчик прерываний должен допускать прерывание своей работы со стороны другого прерывания, то он должен разрешить маскируемые прерывания командой *sti*.

Обработка прерываний в защищенном режиме ЦП имеет существенные отличия от реального режима. Перечислим основные из этих отличий.

1. Значительно расширен состав исключений. Некоторые из новых исключений:

*0Ah* — недействительный сегмент состояния задачи *TSS*;

*0Bh* — исключение *отказ сегмента* (см. п. 5.2.2);

*0Dh* — исключение *общее нарушение защиты* (см. п. 5.2.3);

*0Eh* — исключение *отказ страницы* (см. п. 5.3.2).

Следствием появления новых исключений является то, что одни и те же номера прерываний в реальном и защищенном режимах процессора обозначают разные типы прерываний.

2. При инициировании в защищенном режиме подпрограммы обработки исключения не изменяется флаг *IF*. Поэтому в мультипрограммной системе возникновение исключения в одном процессе никак не влияет на выполнение других процессов. Например, это не влияет на переключение процессов диспетчером. (Напомним, что в основе такого переключения лежат сигналы прерываний от таймера.)

3. В защищенном режиме используется другая таблица векторов прерываний — ***IDT*** (дескрипторная таблица прерываний). Элементами *IDT*, то есть векторами прерываний, являются 8-байтовые системные дескрипторы — вентили прерываний и вентили задач. Структура вентилей приведена на рис. 47. В младших двух битах байта доступа вентилей исключения содержится двоичное число 11, а для других типов прерываний — число 10. Напомним, что в вентиле задачи это число равно 01.

Наличие в *IDT* и вентилей прерываний, и вентилей задач приводит к тому, что, в отличие от реального режима, обработчиками прерываний могут быть не только подпрограммы (они инициируются через вентили прерываний), но и процессы (инициируются через вентили задач). Естественно, что все обработчики прерываний, выполняемые в защищенном режиме, должны принадлежать ядру.

4. В отличие от реального режима, таблица векторов прерываний *IDT* может располагаться в любом месте пространства ОП, распределяемом ядру. На ее положение в памяти указывает 6-байтовый регистр *IDTR*, имеющий структуру, аналогичную регистру *GDTR* (см. п. 5.2.1). Для загрузки этого регистра используется машинная команда *lidt*.

Регистр *IDTR* обычно загружают в реальном режиме, перед переходом в защищенный режим. Но это можно делать и в защищенном режиме, находясь в нулевом кольце защиты.

5. Повторная запускаемость машинных команд, приведших к исключению. В реальном режиме обработчик исключения (как и любого другого прерывания) всегда возвращает управление на ту команду прерванной программы, которая следует за командой, приведшей к исключению. В защищенном режиме обработчики многих исключений возвращают управление не на следующую, а на саму команду, приведшую к исключению. Для того чтобы обработка исключения завершилась повторным выполнением команды, при возникновении исключения в стек аппаратно заносится адрес не следующей, а текущей команды.

Например, пусть текущая машинная команда выполняет какую-то операцию с ячейкой ОП. Если логическая страница программы, содержащая требуемую переменную, отсутствует в ОП, то выполнение данной команды завершится исключением *0Eh* (*отказ страницы*). Обработчик данного исключения обеспечит подкачку требуемой страницы из области свопинга на диске в ОП. После этого логично повторить выполнение той команды, которая привела к исключению.

# 6. УПРАВЛЕНИЕ ФАЙЛАМИ

## 6.1. Виртуальная файловая система

### 6.1.1. Логические файлы

Файловая подсистема ОС предназначена для обслуживания процессов по информационному обмену с периферийными устройствами, прежде всего с устройствами ВП. В *UNIX* эта подсистема является частью ядра ОС (см. подразд. 3.3) и имеет укрупненную структуру, приведенную на рис. 50. Как видно из данного рисунка, файловая подсистема включает виртуальную файловую систему, программные части реальных файловых систем, а также КЭШ.

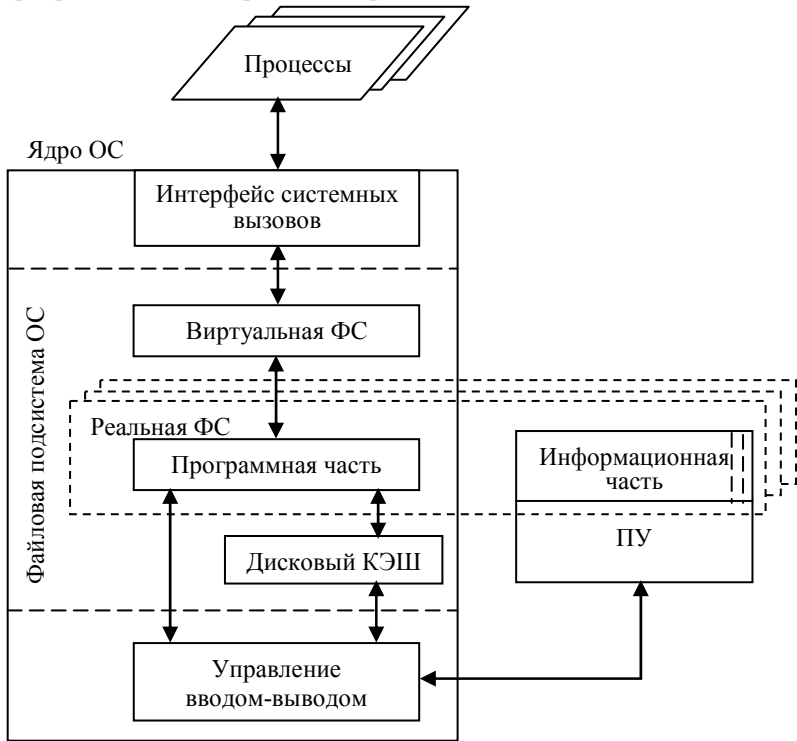


Рис. 50. Структура файловой подсистемы ОС

В подразд. 1.2 мы рассмотрели файлы и файловую структуру системы с точки зрения пользователя. Теперь расширим эти представления до уровня, используемого в программах процессов. Так

как программисты являются частным случаем пользователей, то такое расширение является вполне обоснованным.

Во-первых, заметим, что программа процесса имеет дело не с реальным, а с логическим файлом. В отличие от реального файла, представляющего собой уникально поименованную битовую строку на конкретном носителе, *логический файл* не связан с конкретным носителем информации, а его программное имя не является уникальным в пределах всей системы.

Несмотря на то что и физический, и соответствующий ему логический файл содержат одну и ту же битовую строку, разбиение этой строки на записи в обоих файлах различно. В то время как записи физического файла выбираются исходя из удобства их размещения на носителе, разбиение логического файла на записи производится по смысловому признаку. Например, если логический файл содержит сведения о сотрудниках какой-то организации, то одна запись такого файла содержит информацию об одном сотруднике.

Многие прежние ОС поддерживали разбиение логических файлов на логические записи. В результате программа могла, например, выполнить чтение или запись логической записи, используя для ее идентификации или номер записи в файле, или ее ключ (здесь ключ — символьное имя записи). Современные ОС, в том числе *UNIX*, не поддерживают разбиение логических файлов на записи. Они позволяют программе работать с логическим файлом так, как с длинной последовательностью байтов. Естественно, что разбиение логического файла на записи есть. Но это разбиение известно только самой обрабатывающей программе. Для ОС оно неизвестно, и ее информационное общение с программой сводится к обмену между ними цепочками байтов, которые ОС или помещает в файл, или считывает из него.

В подразд. 1.2 была рассмотрена файловая структура системы, объединяющая в единое целое все файлы системы. (Напомним, что для *UNIX* файловая структура представляет собой единое дерево.) С учетом того, что нигде в системе данная структура целиком не хранится, будем называть ее *логической файловой структурой*. Если считать, что элементами логической файловой структуры являются логические файлы, то получим *логическую файловую систему* — представление совокупности файлов с точки зрения пользователя-программиста.

Заметим, что термин «файловая система» является общепринятым, но не однозначным. Точно так же принято называть совокупность подпрограмм, выполняющих обработку соответствующих файлов. Во избежание путаницы будем называть ту совокупность подпрограмм, которая выполняет обработку логических файлов, *виртуальной файловой системой*.

После того как интерфейс системных вызовов выполнит «техническую» подготовку поступившего в ядро системного вызова, виртуальная ФС приступает к его действительному выполнению. Наличие этой подсистемы позволяет использовать в программах процессов стандартную совокупность системных вызовов для обработки файлов, независимую ни от физической реализации файлов, ни от физической реализации соответствующей файловой системы.

### 6.1.2. Открытие файла

Наиболее интересным системным вызовом, выполняемым виртуальной файловой системой, является открытие файла. Данный системный вызов выполняет связывание физического файла на ПУ с его логическим заменителем. При этом для получения такой связи используются почти все управляющие структуры данных виртуальной файловой системы.

Для открытия существующего файла программа процесса должна содержать системный вызов

*ОТКРЫТЬ\_ФАЙЛ* ( $I_f, r \parallel i$ ) (на СИ — *open*),

где  $I_f$  — символьное имя файла одного из трех типов (простое, относительное или абсолютное), рассмотренных в подразд. 1.2. Это имя используется для идентификации физического файла, задавая его расположение в логической файловой структуре;  $r$  — требуемый режим работы с файлом (чтение, запись, чтение и запись, добавление в конец файла);  $i$  — программное имя (номер) файла, уникальное для данного процесса. Это имя логического файла, которое может использоваться далее программой процесса для выполнения операций с этим файлом.

Получив данный системный вызов, виртуальная файловая система ищет требуемый файл, обращаясь за помощью к реальной файловой системе. Подробно реальные файловые системы рассматриваются в подразд. 6.2, а пока лишь заметим, что основной функцией такой системы является выполнение операций с физическими файлами. Кроме того, в данном разделе мы не будем обра-

щать особого внимания на тот факт, что в любой ФС существует не одна, а несколько реальных файловых систем. К этому вопросу мы вернемся в подразд. 6.3.

В зависимости от заданного символьного имени файла виртуальная ФС начинает поиск требуемого файла с анализа или корневого каталога системы (задано абсолютное имя файла), или корневого каталога данного пользователя (относительное имя файла начинается с символа «~»), или текущего каталога данного пользователя (простое или относительное имя файла). При этом имена текущего каталога и корневого каталога данного пользователя берутся из соответствующих полей в структуре *user*, входящей в состав дескриптора процесса (см. подразд. 4.2). В любом случае поиск искомого файла начинается с анализа того *vnode*, который соответствует исходному каталогу.

**Vnode** (от *virtual inode* — виртуальный индексный дескриптор) — часть дескриптора файла, содержащая ту информацию о реальном файле, состав которой универсален для любого типа файлов. Перечислим наиболее интересные поля *vnode*:

1) число ссылок на данный *vnode*. Это суммарное число открытий файла во всех процессах на данный момент времени;

2) указатель на тот элемент списка монтирования (рассматривается в подразд. 6.3), который соответствует реальной файловой системе, содержащей искомый физический файл;

3) указатель на тот элемент списка монтирования, который соответствует подключенной реальной ФС (если *vnode* соответствует каталогу, который является точкой монтирования);

4) номер **реального дескриптора файла** — *inode*. Подробно *inode* рассматриваются в подразд. 6.2, а пока лишь заметим, что это вторая часть дескриптора файла (первая — *vnode*) и что в любой информационной части реальной ФС все *inode* пронумерованы. Номер *inode* является уникальным числом в пределах данной информационной части реальной ФС;

5) указатель на *inode* в **таблице активных *inode***, расположенной в ОП. Эта таблица принадлежит реальной ФС и содержит только те *inode*, которые соответствуют файлам, открытым хотя бы в одном процессе;

6) указатель на перечень операций, который может выполняться над данным *vnode*. Состав этих операций стандартный. Но фактическая реализация каждой такой операции зависит от типа реальной ФС, в которой находится соответствующий физический файл. Поэтому данное поле содержит указатель на массив других

указателей, каждый из которых представляет собой начальный адрес той процедуры реальной ФС, которая выполняет соответствующую операцию над физическим файлом;

7) тип файла, которому соответствует данный *vnode*: обычный файл, каталог, специальный файл устройства, символическая связь, сокет, удаленный файл.

Обратим внимание на поле 2, содержащее указатель на реальную ФС в списке монтирования, и поле 4, содержащее номер *inode* в реальной ФС. Пара этих чисел является уникальным системным именем файла на данный момент времени. Другим уникальным системным именем файла является номер самого *vnode* в **таблице *vnode*** (рис. 51). Размер этой таблицы определяет максимальное число файлов, которые могут быть открыты в системе. При этом каждому открытому файлу соответствует один *vnode*, независимо от того, в скольких процессах сколько раз был открыт данный файл. Данное имя файла обеспечивает быстрый доступ к содержимому *vnode* файла, но оно является временным: после того как файл будет закрыт во всех процессах, соответствующий *vnode* будет освобожден и может быть выделен другому файлу. В отличие от него, рассмотренное ранее системное имя файла (указатель на реальную ФС в списке монтирования и номер *inode*) является более долговременным и может использоваться для идентификации файла до тех пор, пока данная реальная ФС (информационная часть) не будет отсоединена от файловой структуры системы.

Определив из *vnode* исходного каталога его системное имя (указатель на реальную ФС в списке монтирования и номер *inode*), виртуальная ФС использует это системное имя каталога, а также символическое имя искомого файла в качестве входных параметров при вызове процедуры реальной ФС. Эта процедура выполняет **трансляцию имени файла** — получение на основе символического имени файла его системного имени (указатель на реальную ФС в списке монтирования и номер *inode*). Заметим, что это одна из тех процедур реальной ФС, доступ к которой выполняется с помощью указателя на перечень операций, расположенного в поле *vnode*. Сама трансляция имени сводится к пошаговому «движению» по цепочке каталогов до тех пор, пока или не будет достигнут требуемый файл, или при прохождении очередного каталога будет выяснено, что права доступа к нему данного пользователя не отвечают минимально необходимым требованиям (см. подразд. 3.2). Так как каталог является разновидностью файла, то для его просмотра также требуется выполнить операцию «открытие



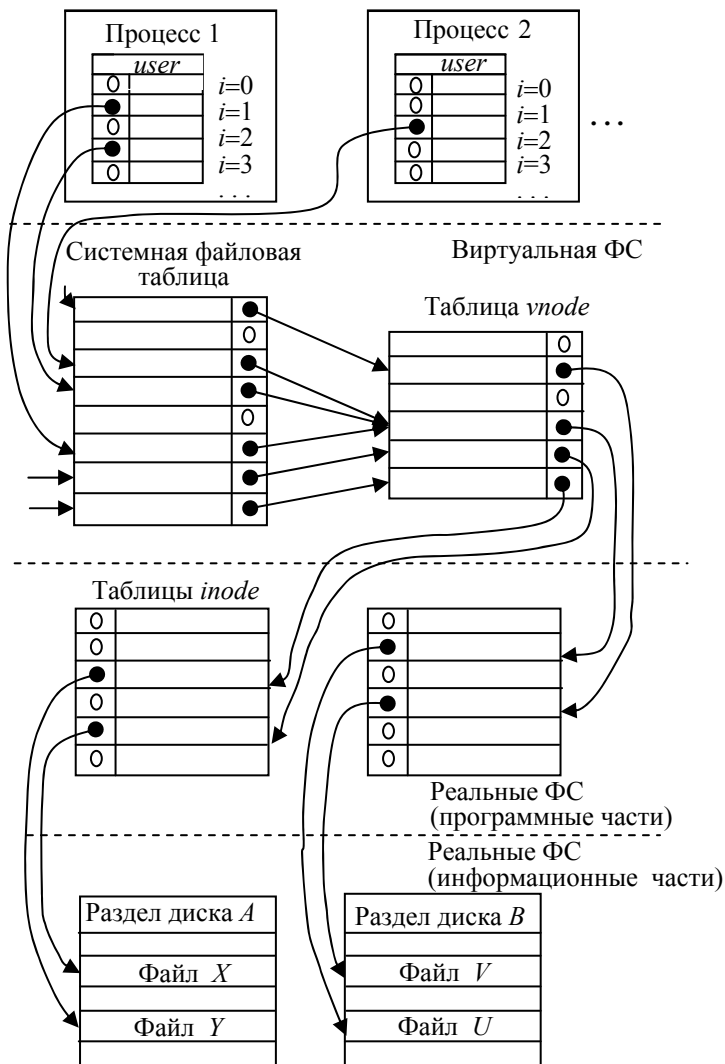


Рис. 51. Системные структуры данных для доступа процессов к файлам

файла». Но в отличие от открытия обычного файла, открытие каталога выполняется в рамках ядра и скрыто от прикладной программы, выдавшей системный запрос.

Если при чтении последнего каталога, заданного в символьном имени файла, будет найдено искомое простое имя файла, то процедура трансляции имени возвратит в виртуальную ФС системное имя файла (указатель на реальную ФС в списке монтирования и номер *inode*). Далее виртуальная ФС использует это системное имя файла для поиска его *vnode* в своей таблице.

Если требуемый *vnode* обнаружен, то число ссылок в его соответствующем поле увеличивается на единицу. Если виртуальная ФС не обнаружила *vnode* открываемого файла в своей таблице, то она создает новый *vnode*. Для этого ищется первый свободный элемент в таблице *vnode*, и его порядковый номер становится номером нового *vnode*. При заполнении полей этого *vnode* виртуальная ФС использует информацию, возвращенную ей процедурой реальной ФС.

После того как *vnode* открываемого файла или найден, или создан заново, виртуальная ФС добавляет новую запись в свою **системную файловую таблицу**. Каждый процесс имеет в этой таблице свои записи, каждая из которых соответствует одному открытому процессом файлу. Если один и тот же файл открыт процессом несколько раз, то каждому открытию соответствует своя запись в системной файловой таблице. Перечислим поля этой записи:

1) режим доступа к файлу (чтение, запись, чтение и запись, добавление в конец файла);

2) **текущее значение файлового указателя** — номер байта файла, начиная с которого будет выполняться следующая операция информационного обмена с файлом (чтение или запись). Сразу после открытия файла эта переменная указывает, в зависимости от режима открытия, на самый первый или самый последний байт файла;

3) указатель на *vnode* файла.

Далее виртуальная ФС находит первую свободную строку в **таблице открытых файлов процесса**, являющейся частью структуры *user* процесса, а затем заполняет эту строку. Данная строка имеет всего два поля: указатель на соответствующую строку в системной файловой таблице и **флаг наследования открытого файла**. Если этот флаг установлен, то при создании дочернего процесса

(с помощью соответствующего системного вызова) данный файл закрывается и его номер не наследуется дочерним процессом.

После того как строка в таблице открытых файлов процесса заполнена, ее номер  $i$  возвращается системным вызовом в прикладную программу процесса в качестве программного имени открытого файла. Это имя действует в пределах данного процесса, а также наследуется (если это не запрещено флагом наследования) его дочерними процессами.

Использование для открытия файла строки таблицы открытых файлов процесса с наименьшим номером нашло применение, в частности, для перенаправления ввода-вывода. Например, если пользователь запустил с командной строки какую-то программу, указав при этом замену стандартного вывода (то есть экрана) на какой-то файл, то процесс-*shell*, выполняющий обработку этой команды, во-первых, закроет файл с программным именем 0. Во-вторых, он откроет заданный файл. При открытии файла будет выбран свободный элемент таблицы открытых файлов процесса с наименьшим номером, то есть с номером 0. Так как структура *user*, включающая таблицу открытых файлов процесса, наследуется дочерним процессом, то весь его вывод на экран будет записан в требуемый файл. Естественно, что процесс-*shell* после запуска нового процесса должен «восстановить справедливость», закрыв файл и открыв экран. Заметим, что многие файлы, которые *shell* открывает для себя, не предназначены для использования в дочерних процессах. Для таких файлов выполняется установка флага наследования.

На рис. 51 один и тот же файл  $U$  открыт и в процессе 1, и в процессе 2. Причем в процессе 1 он открыт дважды: под именем 1 и под именем 3.

### 6.1.3. Другие операции с файлами

**1. Создание файла.** Для создания нового файла любой процесс использует системный вызов

*СОЗДАТЬ\_ФАЙЛ* ( $I_f, D \parallel i$ ) (на СИ — *creat*),

где  $I_f$  — символьное имя файла;  $D$  — задаваемые права доступа к файлу (права доступа были рассмотрены нами в подразд. 3.2);  $i$  — программное имя (номер) файла, уникальное для данного процесса.

Данный системный вызов не только создает новый файл, но и выполняет его открытие для записи, возвращая в качестве

выходного параметра внутрипроцессный номер файла. В качестве владельцев файла записываются владелец-пользователь и владелец-группа того процесса, который выдал данный системный вызов. Если файл с именем  $I_f$  уже существует, то его длина обнуляется, а его права доступа и владельцы остаются прежними.

Виртуальная файловая система начинает выполнение данного системного вызова с того, что определяет наличие файла с заданным именем  $I_f$ . При этом поиск требуемого файла выполняется с помощью реальной файловой системы точно так же, как и при открытии файла. В результате поиска или определяется *inode* файла, или делается вывод об отсутствии файла с заданным именем. В первом случае виртуальная файловая система выполняет обнуление длины файла (с помощью процедуры реальной файловой системы), а затем создает новый *vnode* в первой свободной строке своей таблицы *vnode*. После этого заполняется строка в системной файловой таблице, а также заполняется строка в таблице открытых файлов процесса, являющейся частью его структуры *user*. Номер этой строки возвращается в программу процесса в качестве программного имени файла  $i$ .

Если файл с заданным именем  $I_f$  не обнаружен, виртуальная файловая система приступает к его созданию. Для этого процедуры реальной файловой системы создают для нового файла *inode* и помещают указатель на этот *inode* в родительский каталог. Номер *inode* и указатель на него в таблице активных *inode* возвращаются в виртуальную файловую систему, которая использует эти параметры для создания нового *vnode*. Затем создаются новые записи в системной файловой таблице и в таблице открытых файлов процесса.

**2. Дублирование программного имени (номера).** В результате выполнения данного системного вызова файл, открытый ранее программой процесса, получает еще одно имя (номер). В отличие от повторного открытия файла, старый и новый номера файла указывают на один и тот же логический файл. Это проявляется, в частности, в существовании единственного файлового указателя.

Дублирование имени файла выполняет системный вызов

*ДУБЛИРОВАТЬ\_НОМЕР\_ФАЙЛА* ( $i_1 || i_2$ ) (на СИ — *dup*),

где  $i_1$  — программное имя (номер) ранее открытого файла;  $i_2$  — дополнительное программное имя файла.

При выполнении данного системного вызова виртуальная файловая система создает новую запись в таблице открытых фай-

лов процесса, но не создает новой записи в системной файловой таблице.

**3. Перемещение файлового указателя.** Эта операция выполняется для того, чтобы последующая операция чтения из файла или записи в него выполнялась бы, начиная с требуемого байта файла.

Перемещение файлового указателя выполняет системный вызов

*УСТАНОВИТЬ\_ФАЙЛОВЫЙ\_УКАЗАТЕЛЬ* ( $i, l, t \parallel L$ ) (на СИ — *lseek*),

где  $i$  — программный номер файла;  $l$  — требуемая величина смещения указателя;  $t$  — тип смещения файлового указателя (0 — от текущего положения; 1 — от конца файла; 2 — от начала файла);  $L$  — полученное значение файлового указателя.

При выполнении данного системного вызова виртуальная ФС корректирует содержимое поля «текущее значение файлового указателя» в той строке системной файловой таблицы, на которую указывает указатель в  $i$ -й строке таблицы открытых файлов процесса.

Интересно отметить, что новое значение файлового указателя может выйти за пределы файла. В этом случае в файле образуется «дыра» — незаполненная последовательность байтов. В действительности содержимое «дыры» заполняется нулями.

**4. Чтение из файла.** Операция чтения начинает выполняться с того байта файла, номер которого содержит файловый указатель. После завершения операции значение файлового указателя увеличивается на число считанных байтов.

Для выполнения чтения из файла используется системный вызов

*ЧТЕНИЕ\_ИЗ\_ФАЙЛА* ( $i, n, A \parallel N$ ) (на СИ — *read*),

где  $i$  — программный номер файла;  $n$  — количество читаемых байтов;  $A$  — начальный адрес прикладной области в ОП, в которую производится чтение;  $N$  — количество фактически считанных байтов.

При выполнении данного системного вызова виртуальная ФС считывает из  $i$ -й строки таблицы открытых файлов процесса указатель на строку в системной файловой таблице, а затем читает из этой строки текущее значение файлового указателя. Кроме того, из этой же строки системной файловой таблицы считывается указатель на *vnode*, из которого, в свою очередь, считывается указатель на *inode*. Этот указатель используется виртуальной ФС для задания

физического файла при иницировании той процедуры реальной ФС, которая производит чтение из заданного физического файла с указанного места заданного числа байтов.

**5. Запись в файл.** Операция записи выполняется с того байта файла, номер которого содержит файловый указатель. После завершения операции значение файлового указателя увеличивается на число записанных байтов.

Для выполнения записи в файл используется системный вызов

*ЗАПИСЬ\_В\_ФАЙЛ( $i, n, A ||$ )* (на СИ — *write*),

где  $i$  — программный номер файла;  $n$  — количество записываемых байтов;  $A$  — начальный адрес прикладной области в ОП, из которой производится запись в файл.

Данный системный вызов выполняется виртуальной ФС совместно с реальной ФС аналогично тому, как выполняется чтение из файла.

**6. Закрывание файла.** Данная операция выполняет действия, обратные операции открытия файла. При этом уничтожается логический файл с заданным программным номером, а соответствующий физический файл остается без изменений.

Для выполнения закрытия файла используется системный вызов

*ЗАКРЫТЬ\_ФАЙЛ( $i ||$ )* (на СИ — *close*),

где  $i$  — программный номер файла.

При выполнении данного системного вызова виртуальная ФС считывает из  $i$ -й строки таблицы открытых файлов процесса указатель на строку системной файловой таблицы. После этого  $i$ -я строка таблицы открытых файлов процесса помечается как свободная. Далее из найденной строки системной файловой таблицы производится считывание указателя на *vnode* файла, после чего строка системной файловой таблицы также помечается как свободная.

Далее уменьшается на 1 содержимое поля *vnode* — «число ссылок на данный *vnode*». Если полученное число больше нуля, то выполнение данного системного вызова завершено. Иначе — виртуальная ФС освобождает *vnode*, предварительно считав из его поля указатель на строку в таблице активных *inode*. Далее производится вызов той процедуры реальной ФС, которая по заданному указателю на строку в таблице активных *inode* освобождает эту строку.

**7. Уничтожение файла.** Фактически результатом этой операции является уничтожение жесткой связи между физическим файлом и одним из каталогов. Само уничтожение файла выполняется только в том случае, если уничтожаемая жесткая связь была единственной для данного файла.

Для уничтожения файла программа процесса использует системный вызов

*УНИЧТОЖИТЬ\_ФАЙЛ* ( $I_f ||$ ) (на СИ — *unlink*),

где  $I_f$  — символьное имя файла.

При выполнении данного системного вызова виртуальная ФС находит в своей таблице или создает новый *vnode* для родительского каталога по отношению к уничтожаемому файлу. Далее она вызывает процедуру реальной ФС с целью выполнить корректировку этого каталога, передав на вход вызываемой процедуры указатель на соответствующий *inode*. Процедура реальной ФС не только корректирует каталог, убрав из него информацию о файле, но и проверяет поле в *inode*, содержащее число оставшихся жестких связей с файлом. Если это число равно нулю, то производится фактическое уничтожение файла путем освобождения его *inode* (на диске), а также освобождения занимаемой этим файлом памяти диска.

**8. Отображение файла на ОП.** Применение данного системного вызова позволяет прикладной программе выполнять последующие операции с содержимым файла на устройстве ВП так, как выполняются действия над обыкновенной областью ОП, не используя рассмотренные выше системные вызовы чтения из файла и записи в него.

Отображение заданного файла на область ОП выполняет системный вызов

*ОТОБРАЗИТЬ\_ФАЙЛ*( $i, l, n || A$ ) (на СИ — *map*),

где  $i$  — программный номер файла (для получения данного номера отображаемый файл должен быть предварительно открыт в программе процесса);  $l$  — величина смещения в файле на диске;  $n$  — количество отображаемых байтов файла;  $A$  — начальный адрес прикладной области в ОП, на которую производится отображение файла.

Выполняя данный системный вызов, ФС выполняет округление величины  $n$  в большую сторону до ближайшей границы логической страницы. Далее вызывается подсистема управления

памятью с целью выделения такой области в прикладной части линейного виртуального адресного пространства процесса, которая достаточна для размещения отображаемой части файла (с учетом округления длины). Напомним, что в данном случае речь идет не о выделении реальной ОП, а лишь о создании таблицы страниц и записи в нее дескрипторов страниц.

Само же назначение реальной ОП (физической страницы) производится в результате страничного свопинга, реализация которого не зависит от того, что находится в странице ОП — фрагмент отображаемого файла или обычные данные программы. Единственное отличие: при свопинге отображения файла информационный обмен производится не с областью свопинга на системном устройстве ВП, а с тем дисковым файлом, отображение которого выполнено.

Следует добавить, что применение отображения файла не позволяет изменить длину этого файла.

## 6.2. Реальные файловые системы

### 6.2.1. Критерии оценки файловых систем

Рассмотренная выше виртуальная ФС обеспечивает стандартный программный интерфейс, но с реальными (физическими) файлами эта подсистема ФС не работает. Эту функцию выполняет реальная ФС (программная часть). Наличие пояснения в круглых скобках обусловлено тем, что используемый в литературе термин «реальная ФС» существенно неоднозначен. Далее будем называть *программной частью реальной ФС* совокупность подпрограмм ядра ОС, предназначенных для выполнения действий над информационной частью реальной ФС. При этом под *информационной частью реальной ФС* будем понимать совокупность физических файлов, а также управляющих структур данных, расположенных в непрерывной части носителя ВП, называемой в *UNIX разделом*.

В отличие от программной части реальной ФС, существующей в системе (в ядре) в единственном экземпляре, число информационных частей реальной ФС может быть любым. Каждая такая часть предназначена для покрытия одного непрерывного фрагмента логической файловой структуры системы, расположенного в разделе носителя ВП. Вопросы объединения реальных ФС (информационных частей) в единую логическую структуру будут рас-



смотрены в подразд. 6.3, а пока будем рассматривать каждую такую ФС в качестве отдельной информационной структуры.

Любая *UNIX*-система поддерживает несколько типов реальных ФС. Выбор для конкретного носителя (а точнее — для раздела носителя) типа реальной ФС зависит от требований, предъявляемых к ней пользователем. Каждое из этих требований может рассматриваться в качестве одного из критериев выбора реальной ФС. Перечислим основные из этих критериев:

1) **функциональность** — пригодность реальной ФС выполнять те или иные функции. Например, различной функциональностью обладают распределенные и локальные реальные ФС. По этому же критерию различаются те реальные ФС, которые поддерживают наличие различных прав доступа у разных пользователей и которые не поддерживают наличие таких прав;

2) **производительность** — средняя скорость информационного обмена с физическими файлами, принадлежащими реальной ФС;

3) **надежность** — способность реальной ФС выполнять свои функции в полном или частичном объеме после завершения воздействия на ВС факторов, не соответствующих требованиям нормальной эксплуатации системы. К таким факторам относятся сбои в ВС, например, в результате внезапной потери питания системы, а также механические повреждения носителя ВП;

4) **используемость ВП** — отношение общей длины физических файлов, содержащихся в информационной части реальной ФС к объему соответствующего раздела носителя;

5) **предельная длина имени файла** — максимальная длина простого имени файла;

6) **предельное число файлов** — максимальное количество физических файлов, которые могут находиться в одной информационной части реальной ФС;

7) **сложность алгоритмов** — сложность алгоритмов подпрограмм, принадлежащих программной части реальной ФС.

При выборе реальной ФС, как и при выборе любой другой системы, набор критериев, используемых для оценки вариантов системы, является противоречивым. Например, производительность многих типов реальных ФС зависит от используемости ВП: при высокой степени использования ВП производительность реальной ФС существенно падает. Аналогично повышение надежности реальной ФС обычно связано с дублированием информации,

хранящейся на носителе, что приводит к снижению реальной производительности системы и к снижению используемости ВП.

Интересно выполнить сравнение распределенной и локальной реальных ФС. По критерию функциональности явно выигрывает распределенная ФС, так как она позволяет своим информационным частям размещаться на удаленных ЭВМ. Но по критериям надежности, сложности алгоритмов и особенно по критерию производительности локальная реальная ФС явно предпочтительней. Так как логика использования распределенной реальной ФС имеет некоторые отличия от логики работы распределенной (сетевой) ОС, рассмотренной в подразд. 3.4, то в подразд. 6.3 мы вернемся к рассмотрению распределенной реальной ФС.

При создании реальной ФС ее проектировщики находят в процессе проектирования некоторый компромисс между предъявляемыми к ней требованиями. Иными словами, оценки разрабатываемого варианта системы по перечисленным выше критериям должны быть, с точки зрения ее разработчиков, оптимальными. Здесь термин «оптимальный» существенно отличается от соответствующего понятия в математике, которое предполагает поиск наилучшего решения лишь по одному критерию, а остальные критерии, как правило, записываются в качестве ограничений математической модели.

Задача выбора реальной ФС пользователем системы может рассматриваться как упрощенный вариант задачи проектирования. Поэтому при решении этой задачи должны быть известны хотя бы приближенные оценки сравниваемых типов реальных систем по перечисленным выше критериям. С учетом того что общее количество реальных ФС, поддерживаемых в настоящее время различными *UNIX*-системами, достаточно велико, в настоящем пособии не ставится задача рассмотрения всех этих реальных ФС. Вместо этого далее рассматриваются некоторые принципы проектирования таких систем, позволяющие существенно улучшить их оценки по основным из перечисленных выше критериев.

При рассмотрении принципов проектирования реальных ФС далее используются три типа таких систем: *s5fs*, *ffs* и *fat*. Первые две из этих систем являются для *UNIX* «родными». Несмотря на то что эти ФС, особенно *s5fs*, разработаны давно, они до сих пор поддерживаются многими *UNIX*-системами. Присущая этим ФС простота делает их удобными для нашего рассмотрения.

Третья из перечисленных реальных ФС (*fat*) является для *UNIX* «чужой», так как она разрабатывалась для операционных систем *MS-DOS* и *WINDOWS*. Но учитывая ее очень широкое распространение, она входит в состав реальных ФС, поддерживаемых *UNIX*. Обратим внимание, что только информационные части *fat*-систем, принадлежащих *UNIX*, аналогичны информационным частям таких систем, принадлежащих *MS-DOS* и *WINDOWS*. Что касается программных частей, то эти части реальной ФС сильно различаются в зависимости от типа ОС. Кроме того, заметим, что название *fat* является собирательным, объединяющим три реальных ФС: *fat12*, *fat16* и *fat32*.

### 6.2.2. Физическое размещение информации на носителе

Оценки реальной ФС по критериям производительности, надежности, а также используемости ВП в значительной степени зависят от физического размещения информации, используемого в данной системе. Так как наиболее распространенными носителями ВП являются магнитные диски, то далее будет рассмотрено размещение информации на этих носителях. В данном подразделе рассмотрим общие свойства таких носителей, а в последующих подразделах рассмотрим влияние этих свойств на реальную ФС.

Упрощенное представление магнитного диска приведено на рис. 52. Этот диск состоит из одной, двух или большего числа металлических или стеклянных пластин, вращающихся вокруг общей оси. Одна или обе поверхности каждой пластины покрыты тонким однородным слоем магнитного материала. Подобная поверхность пластины условно разделена на тонкие кольца, называемые *дорожками*. Все дорожки на каждой пластине пронумерованы, причем дорожка 0 находится около внешнего края пластины. Все дорожки диска, имеющие одинаковый номер, принадлежат одному *цилиндру*, номер которого определяется номерами входящих в него дорожек. Например, цилиндр 0 объединяет дорожки с номерами 0 (рис. 52).

Для чтения (записи) информации с пластины (на пластину) используется один или несколько элементов, называемых *головками*. Количество головок зависит от используемого варианта конструкции дисководов. В первом из этих вариантов каждую поверхность пластины обслуживает единственная головка, которая

может перемещаться по радиусу диска дискретными шагами. При этом каждый шаг соответствует одной дорожке пластины. После того как головка «зависнет» над требуемой пластиной, она может выполнять информационный обмен с этой дорожкой. Так как все головки диска связаны жестко в единую «гребенку», то при установке одной из головок на требуемую дорожку все остальные головки «зависнут» над дорожками этого же цилиндра. Следствием этого является то, что переход между дорожками одного цилиндра не требует каких-либо механических перемещений головок. Для такого перехода достаточно лишь выполнить электронное переключение головок. Другой вариант конструкции дискового устройства предполагает, что каждую дорожку поверхности пластины обслуживает своя отдельная головка. В этом случае никакого перемещения головок не производится, так как для выбора требуемой дорожки достаточно лишь электронно включить требуемую головку.

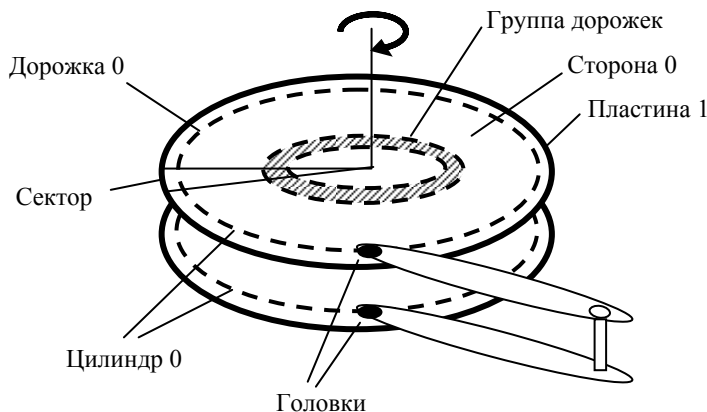


Рис. 52. Упрощенное представление магнитного диска

Что касается информационного обмена с дорожкой, то он производится следующим образом. Во-первых, заметим, что магнитная поверхность дорожки состоит из маленьких фрагментов, называемых **магнитными доменами**. Во время движения дорожки мимо головки (вследствие вращения диска) магнитное поле головки может повлиять на полярность того домена, который находится в данный момент времени рядом с головкой. Благодаря этому магнитный домен может использоваться для записи одного

бита информации. При этом одна полярность домена кодирует 0, а другая — 1. Во время чтения информации с диска, наоборот, полярность текущего домена влияет на магнитное поле головки. Это состояние магнитного поля считывается в электронную цепь и распознается ею как 0 или 1. Несмотря на то что длины дорожек на пластине разные, все они содержат одинаковое количество доменов, кодирующих биты.

Поверхность пластины, как и любой круг, можно разделить на угловые секторы (см. рис. 52). Возьмем размер углового сектора таким, чтобы он «вырезал» из каждой дорожки на поверхности пластины дугу, содержащую 512 бит. Эта дуга дорожки называется **сектором**. Как видно из рис. 52, один угловой сектор «вырезает» секторы одинаковой длины (измерение в битах) не только на одной поверхности одной пластины, но и на всех поверхностях всех пластин. Разбиение дорожки диска на секторы производится во время *низкоуровневого форматирования диска*. Во время этого форматирования начало каждого сектора помечается специальной последовательностью битов. Кроме того, для каждого сектора записывается его номер.

С точки зрения ОС сектор является минимальной единицей информационного обмена между диском и ОП. Для выполнения такого обмена каждый сектор задается своим адресом (номер цилиндра; номер поверхности; номер сектора).

Среднее время чтения (записи) одного сектора  $T_c$  можно найти по формуле

$$T_c = T_d + T_v + T_r,$$

где  $T_d$  — среднее время установки головки на требуемую дорожку. Типичное значение этого времени для современных дисков составляет от 5 до 10 мс;  $T_v$  — среднее время вращения диска до достижения требуемого сектора. Нетрудно заметить, что это время выполнения диском половины своего оборота, которое в свою очередь определяется скоростью вращения диска:  $T_v = 1/2n$ , где  $n$  — скорость вращения диска (об./сек). Гибкие магнитные диски имеют скорость вращения 5–10 об./сек, а жесткие — 90–170 об./сек. Например, при скорости вращения 170 об./сек среднее время вращения диска до достижения требуемого сектора составляет примерно 3 мс;  $T_r$  — время непосредственного считывания (записи) одного сектора. Так как это время поворота диска на заданный угловой сектор, то его можно определить по формуле

$$T_r = 1/(n * L),$$

где  $L$  — число секторов на дорожке. Например, если диск имеет на каждой дорожке 320 секторов, а его скорость вращения  $n = 170$  об./сек, то для него  $T_r = 1/(170 * 320) = 0,018$  мс.

Приведенные формулы могут быть использованы для приблизительной оценки производительности различных реальных ФС. Но при этом надо учесть, что в качестве единицы информационного обмена с диском любая реальная ФС использует не сектор, а более крупную величину — блок. **Блок** — часть дорожки диска, состоящая из одного или нескольких секторов. Длина блока для каждой информационной части реальной ФС является величиной фиксированной и равняется размеру сектора диска, умноженному на степень двойки: 512 байт  $\times N$ , где  $N = 1, 2, 4, \dots$ . Заметим, что блок в *MS-DOS* и *WINDOWS* называется **кластером**. В отличие от сектора имя блока не связано с его физическим размещением на диске, а представляет собой порядковый номер в пределах данной информационной части реальной ФС. Блок является той единицей пространства ВП, которая используется ОС при распределении этого пространства между файлами.

После того как проведено разбиение диска на разделы (если это требуется), для каждого раздела производится свое отдельное **высокоуровневое форматирование**. Во время этого форматирования в разделе размещаются управляющие структуры той реальной ФС, одна из информационных частей которой будет размещаться в данном разделе. При этом среди прочей управляющей информации записывается размер блока, выбранный для данного раздела. Заметим, что размеры блоков разных разделов диска могут быть разными.

Допустим, что размер блока составляет 4K (8 секторов). С учетом того что секторы одного блока расположены последовательно на одной и той же дорожке, для приведенных выше численных параметров среднее время чтения (записи) одного блока составит  $T_b$ :

$$T_b = T_d + T_v + 8 * T_r = 10 \text{ мс} + 3 \text{ мс} + 8 * 0,018 \text{ мс} = 13,144 \text{ мс}.$$

Очень часто требуется считать (записать) не один блок файла, а целую последовательность таких блоков. В этом случае затраты на обработку каждого блока приведенного выше среднего времени приведут к очень большому суммарному времени на обработку данного файла. Например, пусть требуется считать 1000 блоков

файла. Если эти блоки разбросаны (в результате распределения ВП файлу) по всему разделу, то общее время чтения будет очень велико:

$$T_1 = 1000 * T_b = 1000 * 13,144 \text{ мс} = 13,144 \text{ с.}$$

Для увеличения производительности реальной ФС необходимо локализовать размещение блоков файла на диске так, чтобы они находились если не в пределах одного цилиндра, то в пределах близких цилиндров. Например, если все 1000 блоков нашего файла расположены на 4 дорожках одного и того же цилиндра, то среднее время их чтения или записи составит:

$$\begin{aligned} T_2 &= T_d + 4 * T_v + 1000 * 8 * T_r = \\ &= 10 \text{ мс} + 4 * 3 \text{ мс} + 1000 * 8 * 0,018 \text{ мс} = 0,166 \text{ с.} \end{aligned}$$

Полученное время чтения файла  $T_2$  меньше времени  $T_1$  соответствующего чтения при произвольном размещении блоков на диске почти в 80 раз. Времена  $T_1$  и  $T_2$  соответствуют крайним случаям размещения блоков файла на диске. Первый из этих случаев имеет место при реализации наиболее простых алгоритмов динамического распределения ВП между файлами. Второй случай возможен только при статическом распределении.

**Статическое распределение ВП** — вся память назначается файлу при его создании. В этом случае нетрудно обеспечить размещение блоков небольшого файла на одном цилиндре, а большого — на нескольких соседних цилиндрах. Большим недостатком статического распределения является очень низкая используемость ВП для многих типов файлов. Причиной этого является необходимость одновременного выделения файлу такой памяти, значительная часть которой не понадобится для размещения данных файла в ближайшем будущем. Другая значительная часть выделенной ВП, возможно, вообще останется невостребованной, так как для многих файлов нельзя заранее точно предсказать их длину. С учетом данного недостатка статическое распределение дисковой памяти в настоящее время используется лишь для *cd*-дисков, так как длины файлов, размещаемых на таких дисках, заранее точно известны.

Что касается магнитных дисков, то размещаемые на них реальные ФС используют только **динамическое распределение ВП** между файлами. При таком распределении блоки ВП назначаются файлу не при его создании, а в процессе его эксплуатации: если при получении от виртуальной ФС указания выполнить запись в файл некоторого числа байтов реальная ФС обнаружила, что для записи

этих байтов не хватает места на ранее выделенных файлу блоках, то эта реальная ФС (программная часть) выделяет этому файлу новый блок диска. Простейшие алгоритмы динамического распределения не учитывают ранее проведенного назначения блоков диска файлу, динамически выдавая ему любой свободный блок раздела диска. Такие алгоритмы реализованы, например, в реальных ФС *s5fs* и *fat*. Эти реальные ФС обладают хорошей используемостью ВП (для распределения пригоден любой свободный блок раздела), но весьма низкой производительностью.

Гораздо лучшую производительность имеет реальная ФС *ffs*. Для этой системы характерно то, что при высокоуровневом форматировании раздел диска, занимаемый этой системой, делится на несколько групп цилиндров. **Группа цилиндров** — несколько соседних цилиндров (см. рис. 52). В начале этой группы находится управляющая информация, в том числе дисковые блоки управления файлами — *inode*. При создании файла его *inode* помещается в одну из групп цилиндров. При выборе этой группы используются соображения:

- 1) если создается не каталог, а файл данных, то желательно поместить *inode* файла в ту группу цилиндров, куда ранее был помещен родительский каталог. Это позволяет повысить производительность реальной ФС при обслуживании многих программ, выполняющих обработку файлов из одного каталога;

- 2) если создается каталог, то его *inode* желательно поместить в группу цилиндров, отличную от группы цилиндров, в которой находится родительский каталог. Это позволяет улучшить равномерность распределения данных по диску.

Если при выполнении очередной операции записи в файл ему требуется выделить новый блок памяти, то этот блок выбирается, по возможности, из той же группы цилиндров, где находится *inode* файла. Это позволяет сократить не только время перехода между блоками данных, но и время перехода от *inode* файла к его блокам данных.

Если раздел диска, занимаемый системой *ffs*, включает всего одну группу цилиндров, то эта система не обладает лучшей производительностью по сравнению с *s5fs*. При наличии нескольких групп цилиндров система *ffs* имеет достаточно хорошую производительность лишь при ограниченной используемости ВП (не более 90 %). При более высокой степени загрузки диска распределение свободных блоков фактически производится случайным



образом, то есть без соблюдения перечисленных выше рекомендаций.

В заключение данного подраздела рассмотрим вопрос о выборе размера блока (кластера) с точки зрения производительности реальной ФС и присущей ей используемости ВП. С учетом того что содержимое блока всегда находится на одной дорожке (или в одном цилиндре), увеличение размера блока приводит к увеличению производительности системы, так как при одних и тех же накладных расходах времени ( $T_d + T_v$ ) удается выполнить больший информационный обмен между ОП и диском. Но с другой стороны, увеличение размера блока в системах *s5fs* и *fat* приводит к существенному снижению используемости ВП, так как в этих системах блок является минимальной единицей распределяемой ВП. И поэтому даже очень небольшому файлу выделяется целый блок диска. В результате при наличии большого числа таких файлов в системе значительный объем дисковой памяти фактически не используется. Во-вторых, каждому файлу, занимающему более одного блока, в среднем соответствует половина неиспользуемого дискового блока, в котором находятся последние байты файла.

Для того чтобы обеспечить хорошую используемость ВП при использовании блоков большого размера, в реальной ФС *ffs* применяется **фрагментация блоков** — логическое разбиение блока на два, четыре, восемь или более фрагментов одинаковой длины. При этом минимально допустимый размер фрагмента определяется размером сектора (512 байтов). В результате такой фрагментации единицей распределения ВП становится фрагмент. Что касается блока, то он по-прежнему остается единицей информационного обмена между диском и ОП.

### 6.2.3. Каталоги

Напомним (см. подразд. 1.2), что каталог представляет собой служебный файл, содержащий сведения о других файлах, в том числе, возможно, и о других каталогах. Благодаря наличию каталогов все физические файлы, принадлежащие одной и той же информационной части реальной файловой системы, оказываются логически связанными в единую информационную структуру в виде дерева. Одна логическая запись (элемент) каталога содержит сведения об одном дочернем файле или каталоге.

**Система *s5fs*.** В реальной файловой системе *s5fs* одна логическая запись каталога содержит минимум информации о дочернем файле (каталоге):

1) номер *inode* файла (2 байта). Этот номер представляет собой уникальное имя файла в пределах конкретной информационной части реальной ФС. Нулевой номер *inode* обозначает удаленный файл. Поэтому при удалении файла реальная ФС (программная часть) заменяет номер его *inode* на 0;

2) простое имя файла (14 байтов). Первый элемент каталога содержит имя «.», обозначающее сам этот каталог. Второй элемент каталога содержит имя «..», обозначающее родительский каталог по отношению к рассматриваемому каталогу.

Длины этих полей логической записи каталога определяют две характеристики всей реальной ФС:

1) предельное число файлов:  $2^{16} - 1 = 65536 - 1 = 65535$ ;

2) предельная длина имени файла — 14 символов.

**Система *ffs*.** Для преодоления второго ограничения реальная файловая система *ffs* использует следующую структуру каталога:

1) номер *inode* файла (2 байта);

2) длина (в байтах) поля, содержащего простое имя файла (2 байта);

3) длина (в байтах) простого имени файла (2 байта);

4) простое имя файла (поле имеет переменную длину до 255 байтов). Данное поле дополняется нулями до 4-байтной границы.

При удалении файла вся логическая запись каталога, соответствующая удаляемому файлу, подсоединяется к концу поля с именем файла в предыдущей логической записи этого же каталога. При этом корректируется длина данного поля.

Что касается ограничения на предельное число файлов, то оно легко преодолевается удлинением того поля элемента каталога, которое отводится под номер *inode*. Удлинение этого поля всего на один бит увеличивает предельное число файлов в два раза. В системе *ffs* такое увеличение не делается.

**Система *fat*.** В отличие от файловых систем *s5fs* и *ffs*, каталоги которых содержат минимум информации о дочерних файлах, каждый элемент каталога в системе *fat* фактически содержит дескриптор (блок управления) дочернего файла. Перечислим поля 32-байтного элемента каталога в системе *fat12*:

1) простое имя файла (11 байтов, из которых 3 байта отводятся под расширение имени);

2) флаги атрибутов файла (1 байт). Например, бит 0 — файл «только для чтения» (данный файл нельзя открыть для записи), бит 4 — файл является каталогом;

3) свободное поле (10 байтов);

- 4) время создания или последнего изменения (2 байта);
- 5) дата создания или последней модификации (2 байта);
- 6) номер начального блока (кластера) файла (2 байта);
- 7) размер файла (4 байта).

Как видно из данного перечня полей каталога, предельная длина имени файла в *fat12* (и в *fat16*) всего 11 символов, из которых 8 символов — предельная длина собственно имени файла, а 3 символа — предельная длина расширения имени файла. Для преодоления этого ограничения в реальной ФС *fat32* каждому файлу соответствуют несколько 32-байтовых записей каталога. Структура первой из них аналогична записи каталога в *fat12* и в *fat16*, а в последующих записях содержится длинное (до 255 байтов) простое имя файла. Таким образом, один и тот же файл, принадлежащий информационной части *fat32*, может обрабатываться как программной частью *fat12* или *fat16* (используется короткое имя файла), так и программной частью *fat32* (используется длинное имя файла).

Обратим внимание, что среди перечисленных полей элемента каталога нет перечня прав доступа к дочернему файлу со стороны различных типов пользователей. Это является следствием того, что операционные системы *MS-DOS* и *WINDOWS*, для которых *fat* «родная», являются однопользовательскими. Поэтому, выполняя запросы виртуальной ФС, программная часть *fat* возвращает ей для каждого файла одинаковый набор прав доступа, например *rwix* — (см. подразд. 3.2).

## 6.2.4. Управляющие структуры данных

Информационная часть реальной ФС включает физические файлы, размещенные на носителе ВП или в его разделе, а также управляющие структуры данных, находящиеся на этом же носителе (разделе носителя). Реальные ФС, разработанные для *UNIX*, содержат управляющие структуры данных:

- 1) **суперблок** — дескриптор информационной части реальной ФС. Этот блок управления содержит информацию, необходимую для управления всей информационной частью реальной ФС;

- 2) массив *inode*. Каждый *inode* представляет собой дескриптор какого-то одного физического файла, принадлежащего данной информационной части реальной ФС. При открытии файла *inode* считывается с носителя в ОП и становится элементом таблицы активных *inode*.

**Система *s5fs*.** Каждая информационная часть этой реальной ФС имеет всего один суперблок, который находится в начале раздела и содержит поля:

- 1) тип реальной ФС (*s5fs*);
- 2) размер информационной части реальной ФС в блоках;
- 3) размер массива *inode*;
- 4) число свободных блоков;
- 5) число свободных *inode*;
- 6) размер блока (512, 1024, 2048,...);
- 7) список номеров свободных *inode*;
- 8) список номеров свободных блоков.

Размер списка номеров свободных *inode* во много раз меньше предельного числа файлов, которое для *s5fs* равно 65535. Поэтому при исчерпании этого списка программная часть реальной ФС просматривает массив *inode* с целью поиска его свободных элементов (каждый свободный *inode* имеет специальную отметку в своем первом поле). Найденные номера помещаются в список номеров свободных *inode*.

Что касается списка номеров свободных блоков, то его приходится хранить целиком. Первой причиной этого является то, что блоки не имеют специальных отметок об их занятости. Другой причиной являются большие затраты времени на считывание блоков для их просмотра из-за их значительных размеров. Для того чтобы не ограничивать размеры списка номеров свободных блоков, поступают следующим образом. Во-первых, сам суперблок содержит лишь начальную часть этого списка. Первый элемент этой части списка, в отличие от других элементов, содержит не номер свободного блока, а номер блока, содержащего продолжение списка. Если требуется, то первый элемент этого блока с продолжением также содержит номер блока, используемого для продолжения списка и т.д.

Дескриптор физического файла *inode* в реальной ФС *s5fs* содержит поля:

- 1) тип файла, права доступа;
- 2) число ссылок на файл, совпадающее с числом его имен в логической файловой структуре системы;
- 3) имена пользователя-владельца и пользователя-группы;
- 4) размер файла в байтах;
- 5) время последнего доступа к файлу;
- 6) время последней модификации;

7) время последней модификации *inode* (кроме модификации полей 5 и 6);

8) массив номеров блоков, занимаемых данными файла.

Последнее поле содержит массив из 13 элементов, первые десять из которых содержат номера первых десяти блоков файла. Последние три элемента используются, соответственно, для косвенной, двойной косвенной и тройной косвенной адресации блоков файла. Косвенная адресация заключается в том, что 11-й элемент массива содержит номер того блока раздела, который содержит номера блоков, занимаемых файлом. Двойная косвенная адресация реализуется за счет того, что 12-й элемент массива содержит номер блока, используемого для хранения номеров блоков, косвенно адресующих блоки файла. Тройная косвенная адресация заключается в том, что 13-й элемент массива содержит номер блока, в котором находятся номера блоков, используемых для двойной косвенной адресации блоков файла.

**Система *ffs*.** В этой реальной ФС каждая информационная часть имеет с целью повышения надежности не один экземпляр суперблока, а несколько — по одному суперблоку на каждую группу цилиндров. Для того чтобы не вносить в каждую копию суперблока текущие изменения (во время этой операции может произойти сбой в системе), суперблок содержит лишь постоянную информацию о разделе. Поэтому суперблок *ffs* не имеет таких полей, присущих суперблоку *5fs*, как «число свободных блоков» или «число свободных *inode*».

Вслед за суперблоком каждая группа цилиндров содержит битовую карту блоков и список свободных *inode*, расположенных в этой группе цилиндров. Список свободных *inode* содержит перечень номеров свободных элементов в массиве *inode*. Размер данного массива позволяет иметь один *inode* на каждые 2 кбайт дисковой памяти, принадлежащей данной группе цилиндров. **Битовая карта блоков** — последовательность битов, длина которой (в битах) определяется общим числом фрагментов блоков, имеющихся в данной группе цилиндров. При этом каждому фрагменту блока соответствует один бит в данной битовой карте (1 — фрагмент занят, 0 — свободен). Подобное распределение управляющей информации между группами цилиндров не только повышает производительность реальной ФС, но и существенно улучшает ее надежность, так как порча такой информации в одной

группе цилиндров не влияет на доступ к данным в других группах цилиндров.

**Система *fat*.** Суперблок (в *MS-DOS* и *WINDOWS* этот термин не используется) этой реальной ФС находится в начале самого первого сектора раздела. Перечислим лишь его первые поля:

- 1) имя изготовителя и версия (8 байтов);
- 2) длина сектора в байтах (2 байта);
- 3) длина блока (кластера) в секторах (1 байт);
- 4) число копий таблицы *fat*.

В системе *fat* массив *inode* не используется, так как дескрипторы файлов находятся в каталогах. Следствием этого является ненужность списка свободных *inode*. Отсутствует также список свободных блоков раздела. Функции этого списка, а также функцию учета блоков, выделенных каждому файлу, выполняет таблица *fat* (*file allocation table*), название которой используется в качестве названия всей ФС. Таблица *fat* имеет столько 12-, 16- или 32-битных элементов, сколько блоков раздела могут распределяться между файлами. Иными словами, *fat* представляет собой уменьшенную модель распределяемой части раздела диска. Ее наличие позволяет размещать файл в разрывной области ВП. Для этого каждому файлу ставится в соответствие вспомогательный линейный связанный список, построенный из элементов таблицы *fat*.

Вспомним, что одно из полей записи каталога, описывающей файл, содержит номер его начального блока и, следовательно, номер соответствующего элемента в таблице *fat*. Содержимым этого элемента является номер следующего элемента связанного списка, который совпадает с номером следующего блока файла. Если элемент *fat*-таблицы соответствует последнему блоку файла, то он содержит специальное число (*FFFh*, *FFFFh* или *FFFFFFFFh*).

Если элемент *fat*-таблицы соответствует свободному блоку раздела, то он также содержит специальное число (*000h*, *0000h* или *00000000h*). Это число используется программной частью системы *fat* для определения номера блока, который может быть распределен файлу. Для повышения надежности в разделе диска обычно содержится не один, а два экземпляра таблицы *fat*.

### 6.3. Объединение реальных файловых систем

Огромное дерево логической файловой структуры системы состоит из фрагментов, физически реализованных на разных устройствах ВП или разных разделах этих устройств. Каждый такой фрагмент имеет древовидную логическую структуру и реализован в виде отдельной информационной части реальной ФС. Первоначально файловая структура системы включает лишь корневую реальную ФС (информационную часть), расположенную на системном устройстве ВП, к которой постепенно подсоединяются другие информационные части реальных ФС. Операция подсоединения одной информационной части реальной ФС к файловой структуре системы называется **монтированием**.

Например, пусть требуется выполнить подсоединение реальной ФС, расположенной в разделе диска с именем */dev/rz0b* (это имя специального файла раздела), к корневой реальной ФС, расположенной в разделе */dev/rz0a* (рис. 53). Для выполнения такого монтирования, во-первых, требуется выбрать в корневой ФС **точку монтирования** — каталог, к которому будет непосредственно подсоединена монтируемая ФС. Обязательным требованием к такому каталогу является его неиспользуемость в качестве точки монтирования для другой ФС. Кроме того, желательно, чтобы этот каталог был пуст. В примере в качестве точки монтирования используется каталог */home/vlad/prog*.

Для того чтобы виртуальная ФС выполнила монтирование, она должна быть инициализирована с помощью системного вызова

**МОНТИРОВАТЬ** ( $t, I_u, I_k ||$ ) (на СИ — *mount*),

где  $t$  — тип монтируемой реальной ФС;  $I_u$  — имя-путь специального файла, соответствующего устройству или разделу устройства, на котором расположена монтируемая информационная часть реальной ФС;  $I_k$  — имя-путь того каталога, который является точкой монтирования.

Первичным источником данного системного вызова является системная обрабатывающая программа *mount*, запускаемая суперпользователем из командной строки *UNIX* и имеющая те же входные параметры, что и системный вызов.

Что касается выполнения системного вызова **МОНТИРОВАТЬ**, то виртуальная ФС начинает с того, что находит *vnode*, соот-

ветствующий тому каталогу «старой» реальной ФС, который выбран в качестве точки монтирования. Далее находится тот элемент коммутатора файловых систем, который соответствует

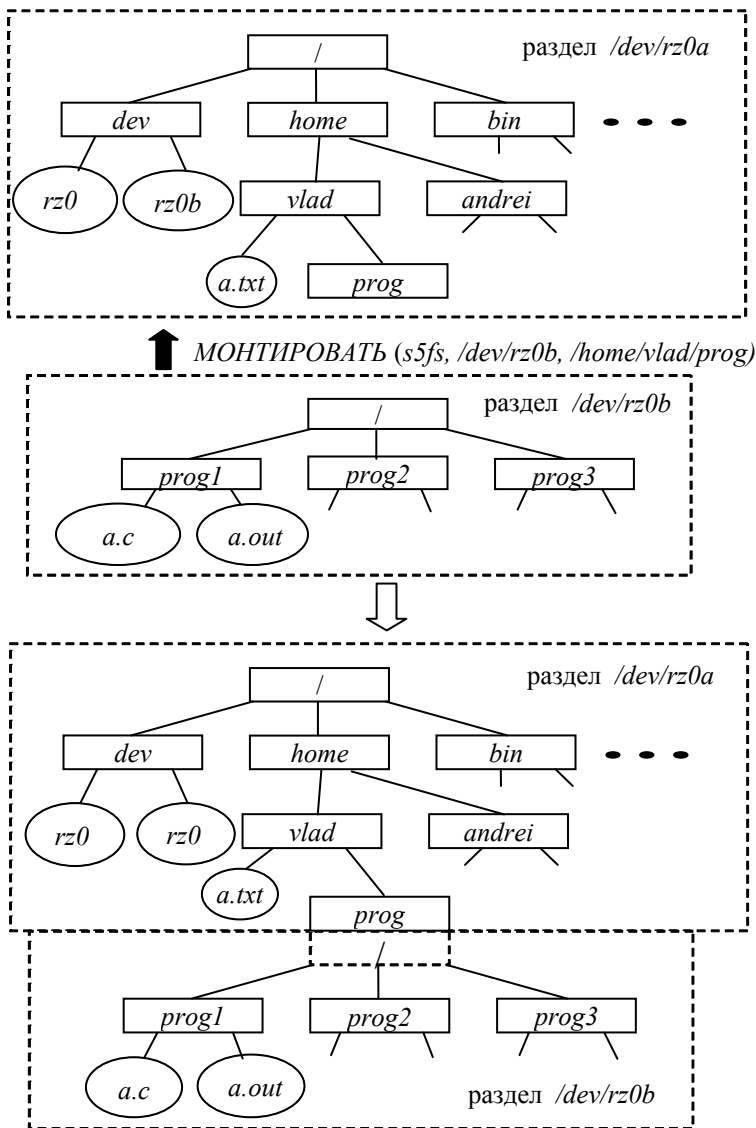




Рис. 53. Пример монтирования реальной файловой системы

заданному типу  $t$  реальной ФС. **Коммутатор файловых систем** — массив, элементами которого являются описатели точек входа в программные части реальных ФС. Каждый элемент этого коммутатора соответствует одному типу реальных ФС и содержит поля:

- 1) тип реальной ФС;
- 2) адрес процедуры инициализации реальной ФС;
- 3) указатель на вектор операций реальной ФС.

Используя поле 2 найденного элемента коммутатора, виртуальная ФС вызывает процедуру инициализации реальной ФС. Эта процедура, во-первых, выполняет считывание в ОП суперблока монтируемой информационной части реальной ФС. Во-вторых, данная процедура добавляет новый элемент в **список монтирования** — односвязанный список, элементами которого являются дескрипторы смонтированных информационных частей реальных ФС. Элемент этого списка содержит поля:

- 1) указатель на следующий элемент в списке монтирования;
- 2) указатель на вектор операций реальной ФС. Это копия соответствующего поля элемента коммутатора файловых систем;
- 3) номер  $vnode$  того каталога, который является точкой монтирования;
- 4) размер блока монтируемой системы;
- 5) указатель на суперблок смонтированной реальной ФС.

Таким образом, в отличие от коммутатора файловых систем, элементы которого используются для доступа к программным частям реальных ФС, элементы списка монтирования используются для доступа к информационным частям этих ФС. Напомним, что в системе может быть только одна программная часть реальной ФС, но может существовать любое число ее информационных частей. Первым элементом списка монтирования всегда является корневая реальная ФС.

После того как в список монтирования добавлен новый элемент, указатель на этот элемент помещается в одно из полей того  $vnode$ , который соответствует каталогу, являющемуся точкой монтирования. Кроме того, создается новый  $vnode$  для корневого элемента монтируемой реальной ФС.

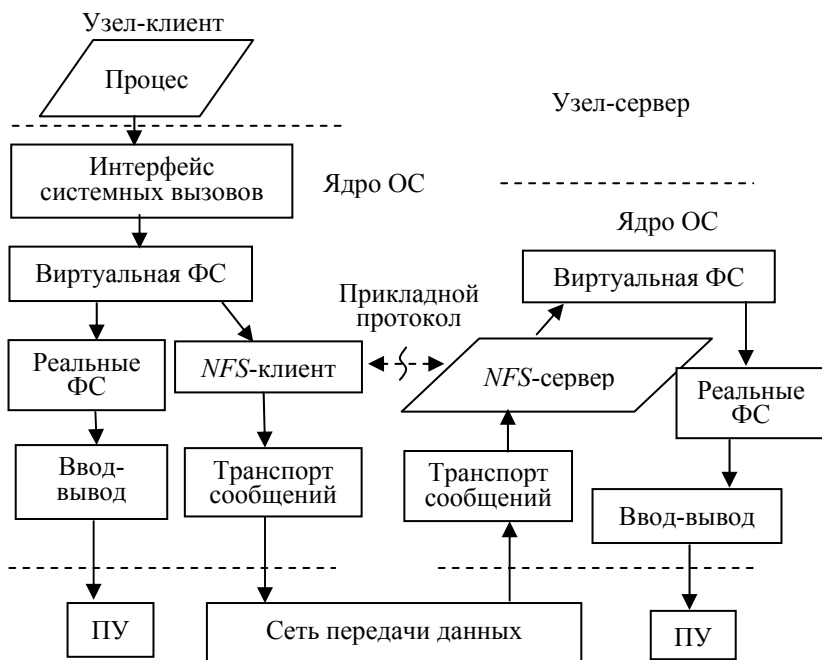
Интересно отметить, что в результате монтирования происходит логическое слияние двух каталогов, один из которых является точкой монтирования, а второй есть корень монтируемой

ФС. Подобное слияние происходит лишь с точки зрения пользователя (на логической файловой структуре системы), так как каждому каталогу по-прежнему соответствует своя физическая реализация в своем разделе диска, а также свой отдельный *vnode*. Данное свойство влияет, в частности, на выполнение трансляции имен файлов. Если при трансляции очередного имени файла реальная ФС (программная часть) обнаружит, что достигнута точка монтирования, то она возвратит в виртуальную ФС не системное имя искомого файла, а указатель на подключенную ФС в списке монтирования. Далее виртуальная ФС должна инициализировать одну из процедур этой ФС с целью продолжения трансляции имени файла в его системное имя.

Существенные отличия имеет операция монтирования сетевой операционной системы *NFS*, упрощенная схема выполнения которой рассматривается далее. Во-первых, заметим, что эта реальная ФС является примером распределенной программы, включающей одну или несколько серверных частей и одну или несколько клиентских частей. При этом серверные части *NFS* находятся в узлах-серверах. Любой узел-сервер имеет один или несколько каталогов, каждый из которых является корнем поддерева файловой структуры, «экспортируемого» в узлы-клиенты. Для того чтобы программные процессы, существующие в узле-клиенте, могли выполнять операции с «экспортируемыми» файлами, содержащие эти файлы поддерева должны быть смонтированы с файловой структурой узла-клиента.

Получив системный вызов *МОНТИРОВАТЬ*, виртуальная ФС в узле-клиенте начинает с того, что по первому параметру системного вызова определяет тип монтируемой системы — *NFS*. Далее в список монтирования добавляется новый элемент (информационная часть *NFS*), а указатель на этот элемент помещается в тот *vnode*, который соответствует точке монтирования. После этого виртуальная ФС вызывает процедуру инициализации реальной ФС *NFS*. Данная процедура (как и другие процедуры *NFS*) выполняет свои функции, обмениваясь сообщениями с удаленной серверной частью *NFS*. При этом адрес требуемого узла-сервера определяется из параметра системного вызова *МОНТИРОВАТЬ*, который соответствует разделу монтируемой системы. При монтировании удаленной ФС этот параметр содержит не имя раздела, а полное сетевое имя того каталога, который является корнем монтируемого поддерева файловой структуры.

Запрос с просьбой разрешить монтирование посылается в требуемый узел-сервер с помощью модуля транспорта, имеющегося в узле-клиенте (рис. 54). Приход этого сообщения инициирует процесс ядра «*NFS-сервер*», который большую часть времени находится в заблокированном состоянии. Реализация *NFS-сервера* в виде процесса обусловлена тем, что программа процесса, в отличие от процедур, не должна «подсоединяться» к программе какого-то другого процесса, выдавшего системный вызов. Реализация процесса «*NFS-сервер*» вне ядра, то есть в виде демона, привела бы к существенному снижению производительности сетевой ФС, так как данный процесс был бы вынужден конкурировать за время ЦП с обычными прикладными процессами. Существенным отличием *NFS-сервера* от *NFS-клиента* и от программных частей других



реальных ФС является то, что не виртуальная ФС инициирует этот модуль, а наоборот. При этом *NFS-сервер* играет роль своеобразного имитатора интерфейса системных вызовов.

Рис. 54. Распределенная реальная файловая система *NFS*

При получении сообщения, содержащего просьбу о монтировании, *NFS*-сервер передает в свою виртуальную ФС команду проверить правомочность запрашиваемой операции. Если проверка оказалась успешной, виртуальная ФС возвращает в *NFS*-сервер содержимое того *vnode*, который соответствует корню монтируемого поддерева. Далее это содержимое посылается *NFS*-клиенту в виде сообщения по сети. Получив это сообщение, *NFS*-клиент завершает монтирование, создав свой *inode* для корня монтируемого поддерева файловой структуры, расположенного в другом узле сети. Данный *inode* включает среди прочей информации содержимое удаленного *vnode*.

Напомним, что среди прочей информации *vnode* имеет два поля, содержимое которых выполняет роль уникального имени файла в пределах данного узла-сервера: 1) указатель на элемент в списке монтирования; 2) номер *inode*. Пара этих чисел используется для указания требуемого файла в любом обращении *NFS*-клиента к *NFS*-серверу. Например, если прикладной процесс в узле-клиенте выдал запрос на открытие файла, расположенного в узле-сервере, то как и при открытии локального файла, в узле-клиенте создается *vnode* файла и делаются записи в системную файловую таблицу и в таблицу открытых файлов процесса. Что касается *inode* файла, то для его создания *NFS*-клиент посылает *NFS*-серверу сообщение с просьбой выполнить трансляцию имени требуемого файла. При этом в качестве имени файла *NFS*-клиент передает смещение относительно корня смонтированного поддерева, а в качестве имени корня поддерева передается указанная выше пара чисел.

Заметим, что при открытии файла по запросу удаленного клиента *NFS*-сервер не делает никакие записи в системные таблицы, а ограничивается лишь выдачей *vnode* того файла, трансляцию имени которого запросил *NFS*-клиент. Содержимое двух полей этого *vnode* используется далее *NFS*-клиентом при выполнении всех системных вызовов, требующих выполнения информационного обмена с данным файлом (например, чтение файла). Несмотря на то что программа прикладного процесса использует обычное логическое имя (номер) файла, в протоколе общения между *NFS*-клиентом и *NFS*-сервером используется только указанное выше системное имя файла. Этот же протокол регламентирует передачу в виде сообщений содержимого самого файла. При записи это содержимое передается от *NFS*-клиента к *NFS*-серверу. При чтении

файла направление передачи будет, наоборот, от *NFS*-сервера к *NFS*-клиенту.

## 6.4. Кэширование блоков данных

Как видно из рис. 50, реальная ФС может взаимодействовать с управлением вводом-выводом не непосредственно, а через **дисковый КЭШ** — программный модуль, включающий буфер для информационного обмена между устройствами ВП (дисками) и областями оперативной памяти процессов, а также подпрограммы для работы с этим буфером. Основная идея применения КЭШа заключается в том, что его буфер содержит копии наиболее используемых секторов дисков. Поэтому вместо того чтобы с помощью подсистемы управления вводом-выводом выполнять чтение (запись) реальных секторов диска, реальная ФС выполняет с помощью подпрограмм КЭШа чтение (запись) тех элементов его буфера, которые соответствуют требуемым секторам диска.

Так как скорость переноса информации между ячейками ОП в тысячи раз превосходит скорость информационного обмена между диском и ОП, то применение КЭШа позволяет существенно повысить производительность любой реальной ФС (в том числе сетевой). Так как в среднем на 90 % реальная ФС выполняет информационный обмен не с устройством ВП, а с КЭШем, то увеличение производительности происходит примерно на порядок.

Существенным недостатком применения КЭШа является уменьшение надежности ФС из-за того, что текущее содержимое некоторых элементов КЭШа отличается от содержимого соответствующих секторов диска. Поэтому если в данный момент времени в ВС произойдет сбой, то фактическое содержимое информационных частей реальных ФС будет отличаться от их требуемого содержимого.

Существуют два основных подхода к реализации дискового КЭШа: 1) традиционный подход; 2) реализация с помощью подсистемы ядра, выполняющей управление ОП. Несмотря на то что современные ОС используют в основном второй из этих подходов, многие из них продолжают поддерживать и первый подход.

### 6.4.1. Традиционный подход к реализации дискового КЭШа

В данном подходе буфер КЭШа занимает существенную часть реальной ОП ядра ОС. Каждый элемент этого буфера предназначен для размещения одного сектора диска (512 байтов) и имеет свой дескриптор:

*ДЕСКРИПТОР\_ЭЛЕМЕНТА\_БУФЕРА* =  $\{f, u_a, u_b, u_c, u_d, ta, ti, L_b, A_b, i_s\}$ ,

где  $f$  — флаги состояния элемента буфера;  $u_a, u_b$  — указатели на последующий и предыдущий элементы буфера в очереди элементов, ожидающих операции ввода/вывода с конкретным типом устройств;  $u_c, u_d$  — указатели на последующий и предыдущий элементы буфера в списке свободных элементов;  $ta, ti$  — старший (*major*) и младший (*minor*) номера устройства (старший номер задает тип устройства, а младший номер определяет то однотипное устройство, на котором расположен дисковый сектор, соответствующий данному элементу буфера. При этом в качестве устройства может выступать или отдельный дисковод, или раздел диска);  $L_b$  — размер буфера;  $A_b$  — начальный адрес размещения элемента буфера в ОП. Здесь речь идет о линейном виртуальном адресе первого байта элемента буфера;  $i_s$  — порядковый номер сектора на носителе ВП (на диске или разделе диска).

Поясним назначение этих полей. Поле  $f = \{f_1, f_2, f_3, f_4\}$  флагов состояния элемента буфера содержит флаги:

$f_1$  — флаг активности элемента буфера. Активность означает, что данный элемент буфера в настоящий момент времени уже участвует в какой-то операции информационного обмена с диском и поэтому не может участвовать в других операциях. Поэтому те процессы, которые выдали системные вызовы на выполнение этих операций, временно блокируются, то есть переводятся в состояние «сон»;

$f_2$  — флаг готовности элемента буфера. Установка данного флага означает, что текущая операция с элементом буфера (ввод с диска или вывод на диск) завершена;

$f_3$  — тип требуемой операции с элементом буфера (ввод или вывод);

$f_4$  — флаг модификации элемента буфера. Установка данного флага означает, что в элемент буфера была произведена запись нового содержимого. Поэтому в данный момент времени существует различие между этим содержимым и содержимым соответствующего сектора диска.

Так как операции ввода/вывода для каждого типа устройств ВП (как и любых ПУ) выполняет своя подпрограмма (драйвер устройства), то для организации линейного двусвязанного списка ожидания к этому драйверу одно из полей дескриптора элемента очереди содержит два указателя на соседние элементы в этом списке. При этом для выбора требуемого драйвера используется старший номер устройства. В том случае если драйвер обслуживает несколько однотипных устройств, то он сам определяет номер требуемого устройства, читая младший номер устройства в поле дескриптора элемента буфера.

Перечисленные поля дескрипторов элементов буфера КЭШа используются подпрограммами КЭШа при выполнении запросов программных частей реальных ФС на выполнение операций информационного обмена с устройствами ВП. При этом многие из этих полей используются для организации интерфейса между КЭШем и подсистемой ввода-вывода.

Например, запрос на чтение блока от реальной ФС содержит параметры: 1) системное имя устройства, в качестве которого используется пара чисел (старший номер устройства; младший номер устройства); 2) номер требуемого блока на устройстве; 3) длину блока информационной части реальной ФС (в секторах); 4) начальный виртуальный адрес той области ОП процесса, в которую требуется выполнить считывание блока. Получив данный запрос, программа КЭШа определяет системное имя первого сектора требуемого блока в виде тройки чисел (старший номер устройства; младший номер устройства; номер сектора на устройстве). Далее выполняется поиск дескриптора элемента буфера, поля которого содержат указанную тройку чисел. Если поиск завершился успешно, то содержимое заданного числа элементов буфера КЭШа переписывается в прикладную область процесса.

Если буфер КЭШа не содержит требуемый сектор, то программа КЭШа делает запрос к подсистеме ввода-вывода с целью выполнить считывание этого сектора с диска. При этом также считывается несколько последующих секторов. Это позволяет, во-первых, получить весь блок, требуемый реальной ФС, а во-вторых, в буфере КЭШа создается запас копий дисковых секторов, которые с большой вероятностью в ближайшем будущем будут востребованы программой процесса.

Заметим, что для формирования запроса к подсистеме ввода-вывода на считывание сектора программа КЭШа создает деск-

риптор для нового элемента буфера. При этом часть полей этого дескриптора содержит детали запроса на чтение сектора: старший и младший номера устройства; порядковый номер сектора на устройстве; тип требуемой операции с элементом буфера; начальный адрес элемента буфера в ОП. После того как подсистема ввода-вывода завершит требуемую операцию, в дескрипторе элемента буфера будет установлен флаг готовности.

Что касается нового элемента буфера, предназначенного для размещения сектора, считываемого с диска, то он выбирается из начала списка свободных элементов. Данный список представляет собой двухсвязанную очередь, образованную с помощью двух указателей, находящихся в одном из полей элемента буфера КЭШа. Список свободных элементов содержит все элементы буфера КЭШа, у которых сброшен флаг активности и которые, следовательно, в данный момент времени не участвуют ни в одной операции информационного обмена с диском. Если элемент буфера КЭШа участвует в какой-то такой операции, то сразу же после ее завершения он будет помещен в конец списка свободных элементов. В случае начала другой такой операции этот элемент опять покинет список на время выполнения операции.

Допустим, что из реальной ФС поступил запрос выполнить запись какого-то блока файла на диск. В этом случае программа КЭШа определяет наличие соответствующих элементов в своем буфере. Если такие элементы уже есть, то в качестве их содержимого записывается информация из прикладной области, заданной в запросе реальной ФС. Иначе перед тем как выполнять запись, осуществляется выборка требуемого числа элементов из списка свободных элементов. В любом случае производится установка флага модификации элемента буфера. Так как содержимое этих модифицированных элементов буфера отличается от соответствующих дисковых секторов, то их необходимо время от времени переписывать на диск. Подобную операцию будем называть *синхронизацией КЭШа*.

Нетрудно заметить, что для повышения надежности ФС необходимо проводить синхронизацию КЭШа как можно чаще. С другой стороны, для повышения производительности системы такую синхронизацию следует выполнять реже. Инициирование синхронизации КЭШа выполняется тремя способами. При первом из них причиной синхронизации является системный вызов

*СИНХРОНИЗИРОВАТЬ\_КЭШ (||)* (на СИ — *sync*).



При получении данного системного вызова программа КЭШа передает в подсистему ввода-вывода для записи на диск все модифицированные элементы буфера КЭШа. При этом для каждого переданного элемента выполняется сброс флага модификации. Источником данного системного вызова является утилита *supc*, запускаемая по команде суперпользователя.

Второй причиной синхронизации КЭШа является процесс ядра, входящий в состав КЭШа, который инициируется через фиксированные промежутки времени с помощью отложенных вызовов (см. подразд. 4.5). Данный процесс ядра обеспечивает сброс на диск всех модифицированных элементов буфера КЭШа.

Третий способ синхронизации КЭШа заключается в том, что системный вызов открытия файла, выданный прикладным процессом, в качестве входного параметра содержит флаг «синхронный». В этом случае все операции записи в открытый таким образом файл приведут к немедленной записи данных на диск.

### **6.4.2. Использование подсистемы управления памятью**

Любая UNIX-система имеет подсистему управления памятью, принципы работы которой были рассмотрены нами в гл. 5. Входящие в эту подсистему подпрограммы, а также аппаратные средства, предназначенные для поддержки линейной виртуальной памяти, фактически реализуют дисковое кэширование. Это обусловлено, во-первых, тем, что реализация линейной виртуальной памяти предполагает отображение файла, содержащего загрузочный модуль программы, на множество логических страниц созданного процесса. При этом часть этих логических страниц (и соответствующих им фрагментов файла) загружается в физические страницы ОП. Во-вторых, логические страницы программы, содержащие ее данные, перемещаются в результате страничного свопинга между ОП и областью свопинга на системном устройстве ВП. В-третьих, система управления памятью используется для реализации отображения файлов на области виртуальной ОП (см. п. 6.1.3).

Для того чтобы организовать КЭШ на основе подсистемы управления памятью, достаточно организовать отображение дисковых файлов не на прикладную часть линейной виртуальной ОП (как при выполнении системного вызова отображения файла), а на системную часть этой памяти. Это отображение скрыто от про-

граммы процесса, которая по-прежнему общается с ФС с помощью рассмотренных ранее системных вызовов. Для выполнения таких отображений создается специальный системный сегмент памяти — **сегмент отображений**, в который помещаются данные всех файлов, открытых всеми процессами. Кроме того, подсистема управления памятью создает **таблицу отображений**, элементами которой являются дескрипторы отображений. **Дескриптор отображения** содержит поля:

- 1) номер *vnode* файла;
- 2) величину смещения в файле, начиная с которого данные файла отображаются на ОП;
- 3) смещение относительно начала сегмента отображений — относительный адрес, начиная с которого располагается область отображения файла.

Величина одной области отображения файла рассчитана на размещение 8096 его последовательных блоков. В данном случае речь идет лишь о виртуальной памяти. Что касается реальной памяти, то она выделяется по мере надобности в результате работы страничного свопинга. Работа с перечисленными системными структурами данных скрыта от программы процесса и производится следующим образом. Допустим, что из программы процесса поступил системный вызов выполнить чтение из файла или запись в него. Виртуальная ФС по программному номеру файла определяет строку в таблице открытых файлов процесса и считывает из этой строки указатель на запись в системной файловой таблице. Далее из этой записи считываются текущее значение файлового указателя файла и номер его *vnode*, которые используются виртуальной ФС при обращении к подсистеме управления памятью. Эта подсистема ядра ищет в таблице отображений такой дескриптор отображения с заданным *vnode*, который содержит величину смещения в файле, соответствующую текущему значению файлового указателя. Данное соответствие не означает равенство, так как каждый дескриптор отображения позволяет отображать часть файла длиной 8096 секторов диска. Если текущее значение файлового указателя не выходит за пределы этой области, то, следовательно, требуемое место файла отображается на участок сегмента отображений. Если подходящий дескриптор отображения не найден, то система управления памятью создает новый дескриптор отображения, используя в качестве величины смещения в файле текущее значение файлового указателя.

Предельная длина одной области отображения, равная  $8096 \times 512$  байтов, обусловлена требованием использования единственной таблицы страниц для установки соответствия между этой виртуальной областью и реальной ОП. Напомним, что таблица страниц содержит 1024 дескрипторов страниц, каждая из которых имеет длину 4096 байтов. Поэтому общая длина области виртуальной ОП, поддерживаемая одной таблицей страниц, составляет  $1024 \times 4096 = 8096 \times 512$  байтов = 4096 кбайтов. Выделение данной виртуальной памяти, конечно, не означает назначения реальной ОП, так как лишь некоторым из 1024 логических страниц области отображения файла будут соответствовать реальные физические страницы ОП.

При выполнении системного вызова чтения файла происходит отображение требуемых логических страниц на физические страницы ОП. При этом назначение очередной физической страницы происходит вследствие исключения «отказ страницы». После того как требуемый фрагмент файла будет переписан в одну или несколько физических страниц ОП, принадлежащих ядру, производится его копирование в прикладную память процесса, адрес которой был задан в качестве параметра системного вызова.

При выполнении системного вызова записи в файл производится копирование прикладной области, заданной одним из входных параметров системного вызова, в подходящую область отображения. При выполнении этого копирования логические страницы области отображения, соответствующие записываемым участкам файла, отображаются на реальные страницы ОП. Что касается записи этих страниц в файл на диске, то эта запись производится позднее и одновременно для всех страниц: по мере того как физические страницы, занимаемые областью отображения, требуются для размещения каких-то иных данных, их содержимое выводится программой свопинга в дисковый файл.

## 7. ПОДСИСТЕМА ВВОДА-ВЫВОДА

### 7.1. Предоставляемый интерфейс

Никакая прикладная или системная обрабатывающая программа не может выполняться без операций с ПУ. Кроме стандартных устройств ввода-вывода (например экрана и клавиатуры) и внешней памяти (дисководов) существует огромное количество нестандартных периферийных устройств. Подобные устройства используются, например, для управления технологическими процессами. Сюда относятся различные датчики (устройства ввода), а также различные задвижки и вентили (устройства вывода). В состав любой ОС входит *подсистема ввода-вывода (ПВВ)*, выполняющая запросы программных процессов по их взаимодействию с ПУ.

В состав ПВВ входит модуль управления драйверами, а также сами драйверы. Функцией *модуля управления драйверами* является согласование вышестоящих подсистем ядра (ФС, КЭШ, подсистемы управления памятью) с драйверами. Каждый *драйвер* выполняет управление периферийными устройствами одного типа. В ОС обязательно имеются драйвер экрана, драйвер клавиатуры, драйвер дисковода (для каждого типа дисководов свой драйвер) и многие другие драйверы. На рис. 55 приведена укрупненная схема управления ПУ, рассмотрение которой весьма полезно для выявления функций ПВВ.

Во-первых, обратим внимание, что в состоянии «Задача» процесс не имеет непосредственного доступа к ПВВ. Такой доступ возможен только через файловую систему. Причем доступ к устройствам ввода-вывода выполняется с помощью операций над специальными файлами устройств (расположены в каталоге */dev*). А доступ к устройствам внешней памяти возможен двумя способами: 1) с помощью операций над файлами данных; 2) с помощью операций с файлами устройств. Второй способ используется, например, при копировании содержимого одного диска на другой диск, так как в этом случае файловое содержимое копируемого диска не представляет интереса.

Для работы с файлами данных программа процесса в состоянии «Задача» использует системные вызовы, рассмотренные в предыдущем разделе. Схожие системные вызовы используются для работы с файлами устройств:

*ОТКРЫТЬ\_ФАЙЛ\_УСТРОЙСТВА* (*I* || *i*) (на СИ — *open*);  
*ЗАКРЫТЬ\_ФАЙЛ\_УСТРОЙСТВА* (*i* || ) ( — *close*);

ЧТЕНИЕ\_ФАЙЛА\_УСТРОЙСТВА ( $i, A, n \parallel$ ) (  $-/-$  read);  
 ЗАПИСЬ\_ФАЙЛ\_УСТРОЙСТВА ( $i, A, n \parallel$ ) (  $-/-$  write),

где  $I$  — имя файла-устройства (примеры таких имен: 1)  $/dev/cd0$  — CD-ROM с номером 0; 2)  $/dev/lp3$  — параллельный порт с номером 3);  $i$  — программное имя (номер) файла, уникальное для данного процесса (после открытия файла-устройства номер  $i$  используется в программе процесса как идентификатор этого файла);  $A$  — начальный адрес прикладного буфера, в который производится чтение из файла-устройства или из которого производится запись на устройство;  $n$  — число байтов, считываемых с устройства или записываемых на него.

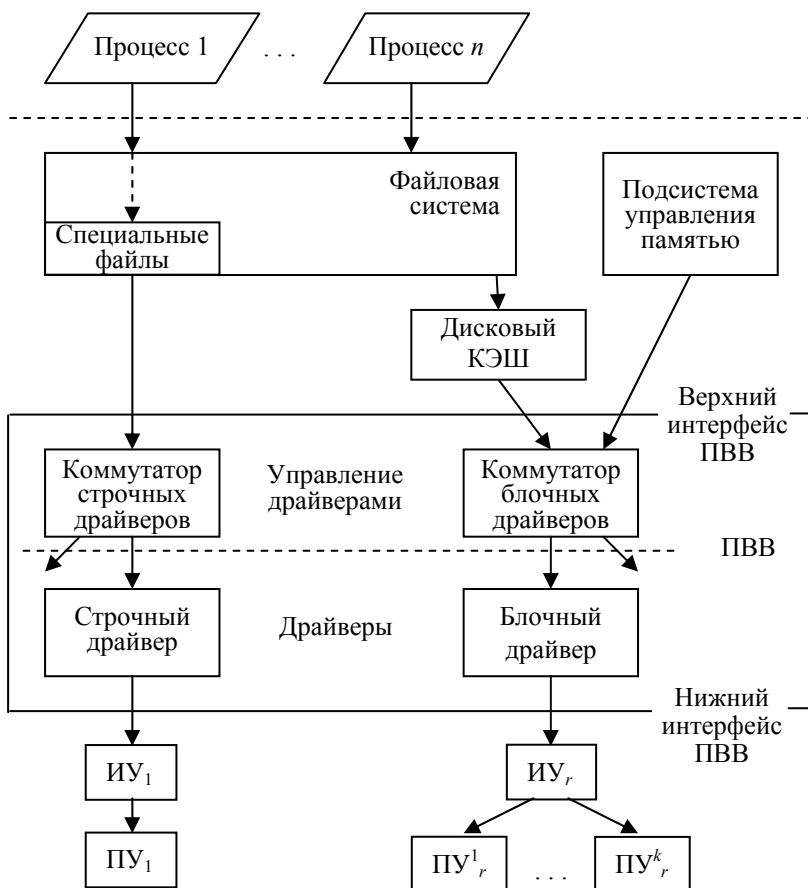


Рис. 55. Укрупненная схема управления периферийными устройствами

Как следует из предыдущего раздела, при выполнении операций с файлами данных ФС взаимодействует с ПВВ не непосредственно, а через дисковый КЭШ. Третьей подсистемой ОС, которая может непосредственно обращаться к ПВВ, является подсистема управления памятью. Если ФС обращается к ПВВ не через КЭШ, а напрямую (используя специальные файлы), то она пользуется услугами строковых драйверов. **Строковый драйвер** позволяет вышестоящим программам, то есть ФС при работе с файлами устройств, выполнять ввод-вывод строк байтов произвольной длины.

В отличие от строкового, **блочный драйвер** выполняет запросы дискового КЭШа или подсистемы управления памятью по вводу-выводу блоков — последовательностей байтов фиксированной длины. Обычно длина блока равна длине сектора диска (512 байтов), умноженной на степень двойки. Например, для подсистемы управления памятью блок соответствует физической странице, длина которой может составлять  $512 \cdot 8 = 4096$  байтов.

При рассмотрении дискового КЭШа в п. 6.4.2 нами был рассмотрен дескриптор элемента буфера КЭШа. Данный дескриптор содержит описание той работы по вводу-выводу блока, которую должна выполнить ПВВ по запросу КЭШа. Что касается самого этого запроса, то он содержит лишь указатель на соответствующий дескриптор элемента буфера КЭШа. Подсистема управления памятью для каждой физической страницы ОП записывает свой дескриптор со структурой, аналогичной структуре дескриптора элемента КЭШа. Для выполнения откатки (подкачки) физической страницы указатель на ее дескриптор передается в ПВВ.

Что касается вызова строкового драйвера, то, формируя его, специальная ФС использует параметры системного вызова (число передаваемых байтов и адрес прикладной области), а также поля в *inode* специального файла:

- 1) тип драйвера (строковый или блочный);
- 2) старший и младший номера устройства.

Таким образом, при обращении к ПВВ любая из вышестоящих подсистем (ФС, КЭШ, подсистема управления памятью или специальная ФС) использует для задания требуемого устройства пару чисел — старший (*major*) и младший (*minor*) номера устройства. Старший номер задает тип устройства, а младший номер определяет конкретное однотипное устройство. При этом в каче-

стве устройства может выступать или устройство ввода-вывода, или отдельный дисковод, или раздел диска. В последнем случае младший номер устройства содержит в закодированном виде и номер дисковода, и номер раздела на диске.

Старший номер устройства не только определяет тип требуемого устройства, но и совместно с типом драйвера (строковый или блочный) определяет драйвер, используемый для управления устройствами этого типа. При этом данный номер используется модулем управления драйверами для выделения записи, соответствующей искомому драйверу, в одной из двух структур данных:

$$\begin{aligned} \text{КОММУТАТОР\_СТРОЧНЫХ\_ДРАЙВЕРОВ} &= \{\dots, S^k, \dots\} \\ \text{КОММУТАТОР\_БЛОЧНЫХ\_ДРАЙВЕРОВ} &= \{\dots, B^k, \dots\}, \end{aligned}$$

где  $S^k$  — совокупность точек входа для строчного драйвера со старшим номером  $k$ ;  $B^k$  — совокупность точек входа для блочного драйвера со старшим номером  $k$ .

Для блочного драйвера запись  $B^k$  представляет собой структуру

$$B^k = \{b_o^k, b_c^k, b_s^k, b_i^k\},$$

где  $b_o^k$  — точка входа (адрес) интерфейсной процедуры драйвера «Открыть устройство» (каждый вызов этой процедуры приводит к созданию внутри драйвера нового экземпляра буфера, что позволяет драйверу выполнять одновременное обслуживание нескольких программных процессов. Для идентификации созданного буфера драйвер и вышестоящая программа могут использовать младший номер устройства);  $b_c^k$  — точка входа интерфейсной процедуры драйвера «Закрыть устройство»;  $b_s^k$  — точка входа интерфейсной процедуры драйвера «Чтение-запись блока»;  $b_i^k$  — точка входа интерфейсного модуля драйвера «Обработчик прерываний». На этот вход подается сигнал прерывания от устройства. Кроме того, данный вход можно использовать для программной имитации сигнала прерывания. Для этого достаточно сделать безусловный переход на адрес входа, предварительно поместив в стек адрес возврата из прерывания.

Каждому строчному драйверу соответствует своя запись  $S^k$  в коммутаторе строчных драйверов:

$$S^k = \{s_o^k, s_c^k, s_r^k, s_w^k, s_p^k, s_i^k\},$$

где  $s_o^k$  — точка входа (адрес) интерфейсной процедуры драйвера «Открыть устройство»;  $s_c^k$  — точка входа интерфейсной процедуры

драйвера «Закрыть устройство»;  $s_r^k$  — точка входа интерфейсной процедуры драйвера «Чтение с устройства»;  $s_w^k$  — точка входа интерфейсной процедуры драйвера «Запись на устройство»;  $s_p^k$  — точка входа интерфейсной процедуры драйвера «Опрос устройства» (данная процедура обычно используется для управления устройствами, не имеющими сигналов прерываний);  $s_i^k$  — точка входа интерфейсного модуля драйвера «Обработчик прерываний». На этот вход подается сигнал прерывания от устройства.

## 7.2. Классификация драйверов

Деление драйверов на строковые и блочные соответствует первому уровню классификации драйверов, приведенной на рис. 56. На втором уровне драйверы разделяются в зависимости от типа устройств, управляемых ими. При этом различаются драйверы псевдоустройств, символьных и блочных устройств. Символьное устройство может выполнять только побайтовый обмен информацией с ОП. Примерами таких устройств являются клавиатура, экран, мышь и другие устройства ввода-вывода. Псевдоустройство имитирует работу реального устройства, например терминала (см. подразд. 3.1). Как символьное устройство, так и псевдоустройство позволяют реализовать лишь строковый верхний интерфейс драйвера.

Блочное устройство выполняет информационный обмен с ОП блоками фиксированной длины. К таким устройствам относятся практически все устройства ВП. Управление блочным устройством предоставляет для вышестоящих программ не только блочный, но и строковый верхний интерфейс. Услугами блочного верхнего интерфейса пользуются КЭШ и подсистема управления памятью, а услугами строкового верхнего интерфейса — ФС при выполнении запросов программ процессов по работе с файлами устройств.

В качестве следующего признака классификации драйверов на рис. 56 используется число уровней в структуре драйвера. Одноуровневый драйвер состоит из нескольких процедур, вызываемых по запросу ФС модулем управления драйверами (используя соответствующий коммутатор драйверов). На время выполнения все эти процедуры становятся частью программы того процесса, который выдал системный вызов по работе с устройством, управляемым данным драйвером. При этом данные процедуры выполняются, естественно, в режиме ядра.



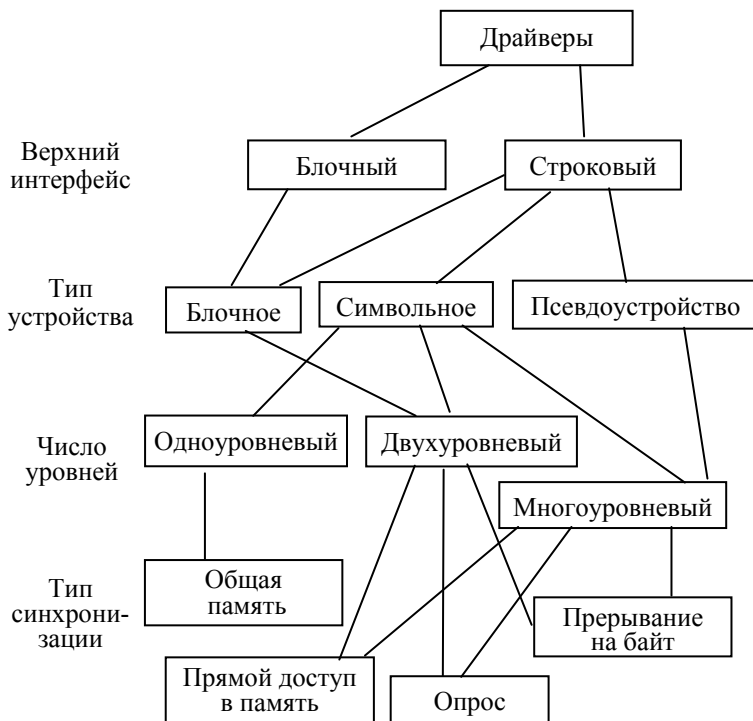


Рис. 56. Классификация драйверов

Процедуры двухуровневого драйвера делятся на две группы: процедуры верхнего уровня драйвера и процедуры нижнего уровня драйвера. Подобно одноуровневому драйверу, процедуры верхнего уровня выполняются в контексте процесса, выдавшего системный вызов. А процедуры нижнего уровня драйвера инициируются или обработчиком прерываний от управляемого устройства, или обработчиком прерываний таймера. При этом процедуры разных уровней никак не связаны между собой по управлению. Если требуется информационное взаимодействие между уровнями, то для его реализации используются общие структуры данных (буферы).

Многоуровневый драйвер представляет собой цепочку двухуровневых и одноуровневых драйверов, каждый из которых может иметь свои внутренние буферы. Подобная структура драйвера позволяет возложить на него дополнительные функции — управление потоком данных и мультиплексирование.

В качестве последнего признака классификации драйверов будем использовать тип синхронизации между программой драйвера и управляемой им аппаратурой (ПУ + ИУ). Сущность этой синхронизации состоит в следующем. Любой драйвер должен выполнять три основных функции взаимодействия с ПУ:

1) подготовка ПУ и, возможно, самого драйвера к последующим операциям по обмену данными — выполнение команды «Открыть устройство»;

2) инициирование очередной операции ПУ по вводу (выводу) единицы информации;

3) обеспечение завершения очередной операции ввода-вывода.

Первые две функции выполняются модулями верхнего уровня драйвера в контексте того процесса, в интересах которого выполняется информационный обмен с ПУ. Что касается третьей функции, то она выполняется нижним уровнем драйвера в контексте ядра, недоступном для процессов. Реализация этой функции зависит от типа используемой синхронизации.

Существуют типы синхронизации: опрос; прерывание на байт; прямой доступ в память; общая память. Разница между этими типами синхронизации заключается в том, как «узнает» программа драйвера о завершении текущей операции информационного обмена с ПУ. В методе опроса это делает само ядро, периодически опрашивая состояние устройства. В методе прерываний на байт это делает ИУ, выдавая в ЦП сигнал прерывания по завершению периферийным устройством очередной операции ввода-вывода байта данных. В методе прямого доступа в память сигнал прерывания выдается в ЦП при завершении ввода-вывода не одного байта, а целого блока данных. Использование общей памяти для управления устройством делает синхронизацию на программном уровне между драйвером и ПУ вообще ненужной.

Выбор типа синхронизации зависит, прежде всего, от типа управляемого устройства. Для большинства символьных устройств применимы два типа синхронизации — опрос или прерывание на байт. При управлении экраном используемый тип синхронизации — общая память. Для управления блочным устройством обычно используется прямой доступ в память. Особенности каждого метода синхронизации будут рассмотрены ниже при рассмотрении одноуровневых и двухуровневых драйверов.

### 7.3. Аппаратный интерфейс

Несмотря на то что конструкция драйвера в значительной степени определяется типом ПУ, фактическое взаимодействие драйвера происходит не с ПУ, а с его ИУ. Поэтому остановимся на вопросе интерфейса между этими модулями. Допустим, что ЭВМ имеет структуру с общей шиной (см. рис. 1). С помощью одного ИУ к ОШ подсоединяется или одно, или несколько однотипных ПУ. Простейшее ИУ состоит из одного или более регистров, в состав которых обязательно входит *регистр состояния и управления RS*. Этот регистр играет для ИУ и подключенных к нему ПУ ту же роль, что регистр *EFLAGS* играет для ЦП, то есть фактически является их блоком управления.

Количество регистров состояния и управления, а также назначение их битов для разных ИУ существенно отличаются. Биты этого регистра (регистров) делятся на управляющие биты и биты состояния. *Управляющие биты* устанавливаются программами, которые исполняются на ЦП, и предназначены для задания режима работы ПУ. Примером управляющих битов являются биты функции, используемые для уточнения операции (функции), выполняемой устройством.

*Биты состояния* устанавливаются ПУ или его ИУ. Их назначение заключается в том, чтобы сообщить программе ЦП (драйверу) о текущем состоянии ПУ. Примером бита состояния является флаг ошибки. Его установка говорит о том, что при выполнении последней операции ввода-вывода произошла ошибка. Нулевое содержимое этого бита сообщает об отсутствии ошибки.

Кроме *RS* многие ИУ имеют *буферный регистр данных (RD)*, предназначенный для временного хранения одного или двух байтов данных, передаваемых на ПУ при выводе или принимаемых с ПУ при вводе.

При использовании простейшего ИУ всю работу по управлению ПУ выполняет сам ЦП. При этом он отвлекается от выполнения самих программ, в результате чего значительно снижается его фактическая производительность. Применение более сложных ИУ позволяет возложить на них значительную часть работы по управлению ПУ, которая теперь, в принципе, может выполняться параллельно работе ЦП. Наиболее сложным ИУ является *контроллер* — специализированный процессор ввода-вывода. (Как и всякое ИУ, контроллер имеет свои *RS* и *RD*.)

Регистры ИУ, непосредственно доступные из программ ЦП, называются **портами**. Портами являются многие *RS* и *RD* (но не все), а также служебные регистры, используемые для программного доступа к тем *RS* и *RD*, к которым нет непосредственного доступа. Каждый порт в ЭВМ имеет свой уникальный номер, называемый **адресом порта**. Для некоторых типов ЦП программные операции с портами (чтение и запись) выполняются точно так же, как и операции с ячейками ОП. При этом для адресации портов используется часть адресного пространства ОП, в которой каждый реальный адрес соответствует не ячейке ОП, а порту.

Для процессоров *Intel* используется другой подход, при котором существуют два адресных пространства — одно для ячеек ОП, а второе — для портов. При этом адрес порта — число в диапазоне от 0 до 65535, а сам порт представляет собой 8-битный регистр. Следствием наличия двух адресных пространств является использование для работы с портами специальных команд ЦП: *in* — для ввода из порта и *out* — для вывода в порт.

## 7.4. Одноуровневые драйверы

Особенностью одноуровневого драйвера является то, что все его процедуры выполняются в контексте того процесса, который выдал системный вызов для работы с устройством, обслуживаемым данным драйвером. Использование такой простой структуры объясняется возможностью непосредственного доступа программ ЦП к внутренней памяти ИУ благодаря отображению этой памяти на виртуальное адресное пространство ОП. Такая непосредственная передача информации между ИУ и прикладной памятью процесса делает ненужным использование промежуточных буферов и, как следствие, исключает потребность промежуточного копирования. К сожалению, ИУ лишь немногих устройств, к которым относятся различные экраны (мониторы), допускают отображение своей внутренней памяти на ОП.

Допустим, что для вывода на экран строк символов программы процессов используют системные вызовы записи в специальные файлы */dev/mnj*, где *j* — номер экрана. В этом случае для управления экранами в текстовом режиме может использоваться драйвер, упрощенная логическая схема которого приведена на рис. 57. (Используемое при этом упрощение заключается в том, что в действительности портом ИУ экрана является не *RS* (а точнее, несколько *RS*), а два других, специально предназначенных для

этого регистра.) Что касается графического режима экрана, то он обеспечивается другим драйвером, доступ к которому программа процесса имеет через другой специальный файл.

Логическая схема на рис. 57 представляет собой совокупность двух интерфейсных (то есть доступных для вызова извне драйвера) логических процедур «Открыть», «Запись строки байт». Кроме того, эта схема включает внутреннюю структуру данных — массив, каждый элемент которого представляет собой текущие координаты курсора для одного из экранов. Рассмотрим работу этих модулей драйвера во взаимодействии с другими аппаратными и программными модулями.

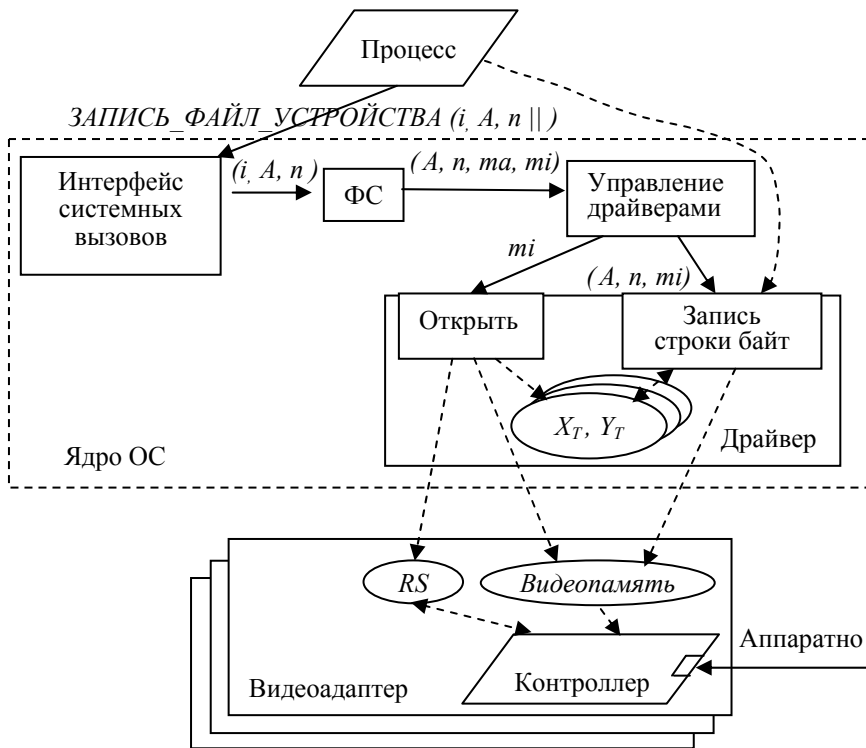


Рис. 57. Логическая структура драйвера экрана:

$i$  — программный номер файла-устройства;  $A$  — начальный адрес прикладного буфера;  $n$  — число выводимых байтов;  $ma$  — старший номер устройства;  $mi$  — младший номер устройства;  $RS$  — регистр состояния и управления;  $X_T, Y_T$  — текущие координаты курсора

Подобно работе с обычными файлами, работа со специальным файлом (файлом-устройством) начинается с открытия этого файла в программе процесса. Для этого может использоваться, например, следующий системный вызов:

*ОТКРЫТЬ\_ФАЙЛ\_УСТРОЙСТВА ( /dev/mn3 || i).*

Данный вызов преобразуется интерфейсом системных вызовов в команду для виртуальной ФС, которая начинает выполнять открытие файла обычным образом. При этом проверяются права доступа к файлу и создаются необходимые записи в системных таблицах. Так как файл специальный, то для работы с ним привлекается *специальная реальная ФС*. Эта ФС находит в *inode* файла его атрибуты — старший и младший номера устройства. Далее эти числа используются при формировании команды ФС для модуля управления драйверами.

Используя старший номер устройства, модуль управления драйверами находит в коммутаторе строковых драйверов вход в драйвер экрана и в этом входе находит строку с адресом процедуры «Открыть». При вызове данной процедуры ей на вход передается младший номер устройства.

Процедура «Открыть» начинает с того, что выполняет отображение видеопамати требуемого экрана на область памяти ядра. При этом видеопамати каждого экрана соответствует своя отдельная область. В п. 6.1.3 рассматривалось отображение на память ядра областей ВП по запросам прикладных программ. Несмотря на то что реализация отображения видеопамати имеет существенные отличия, она преследует ту же цель — обеспечить работу программы (драйвера) с видеопаматью точно так же, как с обычной ОП. В частности, полученное отображение видеопамати используется самой процедурой «Открыть» для того, чтобы заполнить экран символами пробела с требуемым цветом фона.

Далее процедура «Открыть» по заданному номеру устройства *mi* находит адрес порта *RS* требуемого экрана (каждый экран имеет свой *RS*) и выполняет запись в его биты управления с целью: 1) задания номера дисплейной страницы (этот термин будет пояснен позже), выводимой на экран; 2) задания первоначальных координат курсора; 3) задания формы курсора. *Курсор* — светящийся прямоугольник, генерируемый аппаратно. Обычно курсор указывает на ту позицию экрана, в которую будет помещен следующий символ, выводимый на экран. Но так как положение курсора и вывод символов на экран никак не связаны аппаратно, то син-

хронизацией между ними должен заниматься сам драйвер. Выполнив первоначальную установку курсора, процедура «Открыть» помещает его первоначальные координаты в переменные  $X_T$  и  $Y_T$ .

Видеопамять, а также контроллер экрана входят в состав ИУ экрана, называемого **видеоадаптером**, в качестве примера которого мы будем рассматривать видеоадаптер *CGA*. Являясь одним из первых видеоадаптеров (создан в 1981 году), *CGA* имитируется аппаратно многими другими, более современными видеоадаптерами. Видеопамять в *CGA* имеет размер 16 кбайтов. Каждый символ занимает в ней два байта. В младшем из этих байтов содержится код *ASCII* символа (информация о том, что выводить), а в старшем байте — атрибуты символа (информация о том, как выводить). Структура байта атрибутов приведена на рис. 58. Задавая в байте атрибутов комбинации битов красного, зеленого и синего цветов, а также интенсивности, можно получить другие цвета, например сиреневый ( $R = 1, G = 0, B = 1, I = 0$ ) или желтый ( $R = 1, G = 1, B = 0, I = 1$ ).

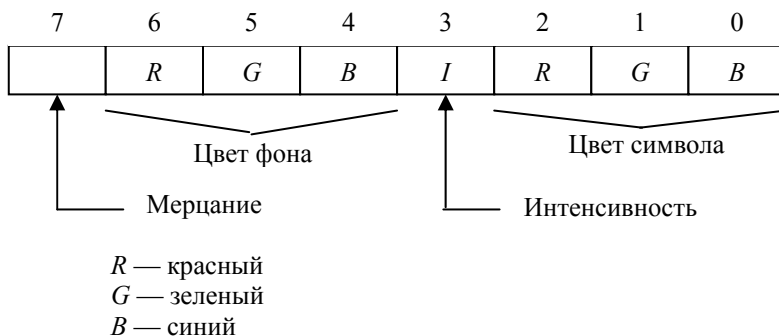


Рис. 58. Структура байта атрибутов

На экране одновременно могут находиться 2000 символов (80\*25). Эти символы занимают 4000 байтов (около 4 кбайтов) видеопамяти. Такая область памяти, занимаемая данными для вывода экрана, называется **дисплейной страницей**. Нетрудно подсчитать, что в видеопамяти *CGA* объемом 16 кбайтов могут разместиться 4 дисплейных страницы. При этом данные экрана располагаются в памяти построчно (начиная с верхнего левого угла), а небольшие пустоты между страницами усекаются аппаратно.

Аппаратура адаптера периодически считывает содержимое видеопамати (а точнее — дисплейной страницы) и помещает его на экран. Электронный луч, управляемый системой отклонения, пробегает по экрану строка за строкой слева направо и сверху вниз (развертка). При этом контроллер включает и выключает интенсивность луча, повторяя «узор» битов в видеопамати. За секунду электронный луч 50 раз пробегает по всему экрану (кадру). Между кадрами луч должен из правого нижнего угла вернуться в левый верхний угол. Это движение называется *обратным ходом луча*.

Запустив устройство (экран), процедура «Открыть» возвращает управление в модуль управления драйверами. Дальнейший возврат управления продолжится по цепочке модулей и завершится возвратом в прикладную программу процесса. При этом в качестве параметра будет передан программный номер  $i$  файла-устройства.

Допустим, что через некоторое время процесс передал в ядро системный вызов для вывода требуемого числа байтов  $n$  из своей прикладной области на экран:

*ЗАПИСЬ\_ФАЙЛ\_УСТРОЙСТВА ( $i, A, n ||$ ).*

Подобно обычным файлам, ФС находит *inode* файла, а затем использует его для определения номеров устройства  $ma$  и  $mi$ . Эти номера используются, в свою очередь, для выдачи запроса к модулю управления драйверами. Этот модуль находит требуемый вход в коммутаторе строковых драйверов и вызывает из этого входа процедуру «Запись строки байтов».

Данная процедура находит по заданному номеру  $mi$  отображение видеопамати требуемого экрана, а затем переписывает в область отображения заданную строку символов из прикладной памяти процесса, помещая между записываемыми символами байты атрибутов. При этом для определения начального адреса в области отображения используются текущие координаты курсора  $X_T$  и  $Y_T$ . После завершения записи в видеопамать процедура «Запись строки байтов» корректирует содержимое этих переменных, а также изменяет расположение курсора на экране, выполнив корректировку  $RS$ . После завершения данной процедуры управление возвратится в прикладную программу на команду, следующую за командой системного вызова для вывода на экран.

Одноуровневый драйвер управления экраном в текстовом режиме обычно существует не самостоятельно, а входит в состав многоуровневого драйвера терминала. Заметим, что прикладные программы осуществляют доступ к нему через специальный файл терминала.



## 7.5. Двухуровневые драйверы

На рис. 59 приведена укрупненная двухуровневая структура строкового драйвера.

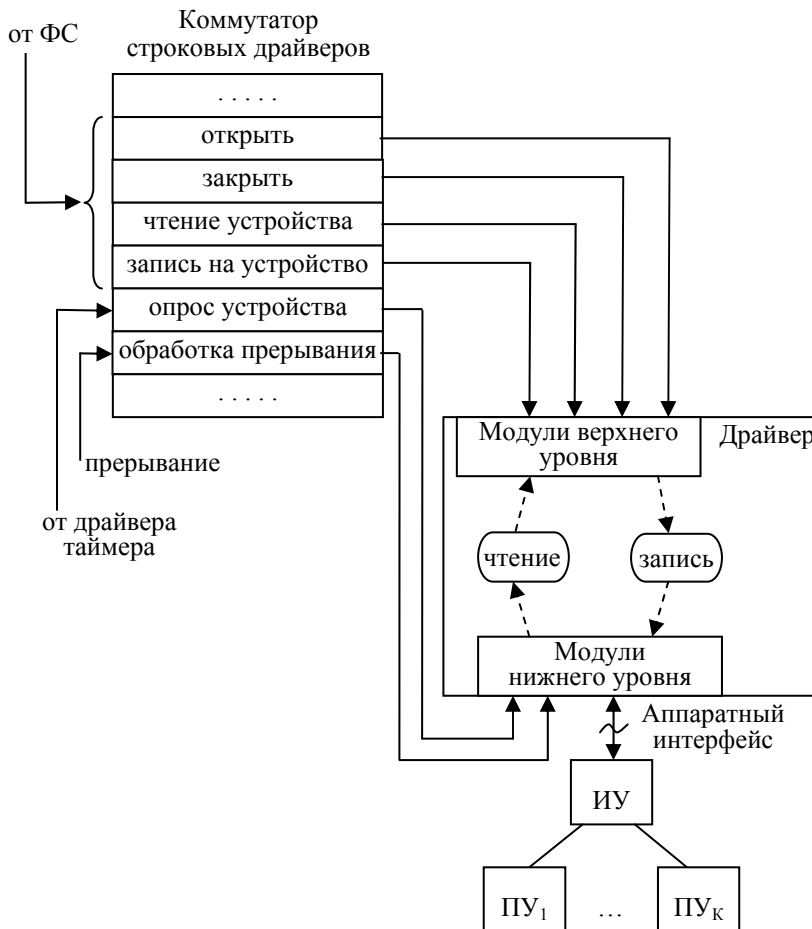


Рис. 59. Укрупненная структура строкового драйвера

Данная структура включает программные модули верхнего уровня: 1) «Открыть устройство»; 2) «Закрыть устройство»; 3) «Чтение с устройства»; 4) «Запись на устройство»; модули нижнего уровня: 1) «Опрос устройства»; 2) «Обработчик прерываний». Причем модули разных уровней никак не связаны между собой по

управлению. Они взаимодействуют только через общие структуры данных — *буфер чтения* и *буфер записи*. Эти буферы реализуются как очереди, элементами которых являются или вводимые/выводимые символы, или области памяти фиксированной длины, содержащие эти символы. В первом случае буфер реализуется в виде несвязанной, а во втором — в виде связанной очереди.

Несмотря на то что модули обоих уровней относятся к ядру, они работают с различным объемом линейной виртуальной памяти. Модули верхнего уровня имеют дело со всеми 4 Гбайт ЛВП, так как они выполняются в контексте процесса. Поэтому они могут выполнять перенос данных между прикладными буферами (находятся в прикладной части ЛВП) и буферами драйвера (в системной части ЛВП). Модули нижнего уровня работают только с 1 Гбайт системной части ЛВП.

Состав модулей драйвера и их алгоритмы зависят от типа ПУ и выбранного типа синхронизации. Далее эти особенности строковых драйверов рассматриваются для двух различных методов синхронизации — опроса и прерывания на байт. При выполнении этого рассмотрения мы будем использовать не реальное, а гипотетическое (вымышленное) ПУ — устройство ввода перфоленты. Подобный выбор обусловлен тем, что управление реальным устройством требует рассмотрения второстепенных деталей, присущих только данному типу ПУ и не используемых при построении драйверов других устройств.

На рис. 60 приведен пример типичного  $RS$ , который будет использоваться нами в дальнейшем. В этом  $RS$  биты  $b_0$ – $b_5$  — управляющие биты, устанавливаемые программами ЦП:

- 1) если  $b_0 = 1$ , то разрешается работа ПУ по вводу (выводу) единицы информации;
- 2)  $b_1$ – $b_2$  используются для уточнения операции (функции), выполняемой устройством;
- 3) если  $b_3 = 1$ , то в ЦП может быть выдан сигнал прерывания;
- 4)  $b_4$ – $b_5$  используются для задания номера устройства.

На рис. 60 биты  $b_4$ – $b_7$  — биты состояния ПУ, устанавливаемые ПУ или его ИУ:

- 1)  $b_7$  описывает состояние буферного регистра данных. Для устройства ввода  $b_7 = 1$  означает, что  $RD$  заполнен данными, которые могут быть перенесены в ЦП. Для устройства вывода это означает, что выходной буфер данных сейчас пуст и ждет, когда ЦП загрузит в него новые данные;

2)  $b_6$  — флаг ошибки. Его установка говорит о том, что при выполнении последней операции ввода-вывода произошла ошибка. Нулевое содержимое этого бита сообщает об отсутствии ошибки;

3)  $b_4$ – $b_5$  используются для задания номера устройства. Таким образом, эти биты являются и битами состояния, и битами управления.

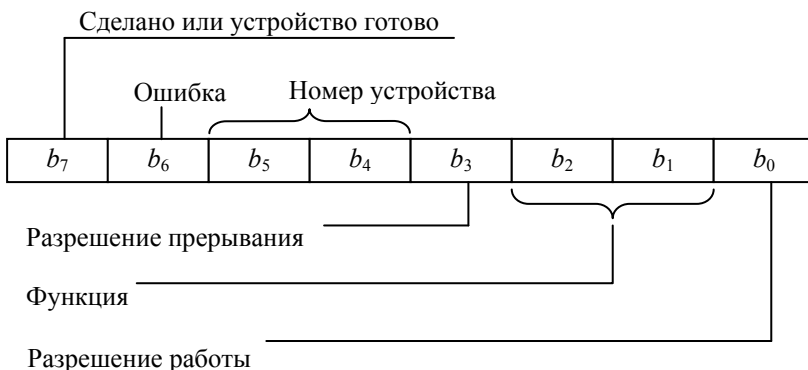


Рис. 60. Пример регистра состояния и управления *RS*

### 7.5.1. Двухуровневый драйвер с опросом

Реализацию данного драйвера рассмотрим на следующем гипотетическом примере. Пусть требуется разработать драйвер для управления работой считывателя перфоленты. Допустим, что файл-устройство, соответствующий требуемому считывателю перфоленты, имеет имя */dev/pl3*. Структура *RS* приведена на рис. 60. Код вводимого символа помещается в *RD*.

На рис. 61 приведена логическая схема драйвера. Она представляет собой совокупность трех интерфейсных логических процедур: «Открыть», «Чтение строки байт», «Опрос». Кроме того, эта схема включает две внутренние структуры данных: а) совокупность буферов чтения устройств; б) массив флагов открытия устройств. Рассмотрим работу этих модулей драйвера во взаимодействии с другими аппаратными и программными модулями.

Работа со специальным файлом (файлом-устройством) начинается с открытия этого файла в программе процесса с помощью системного вызова

*ОТКРЫТЬ\_ФАЙЛ\_УСТРОЙСТВА (/dev/pl3 || i).*

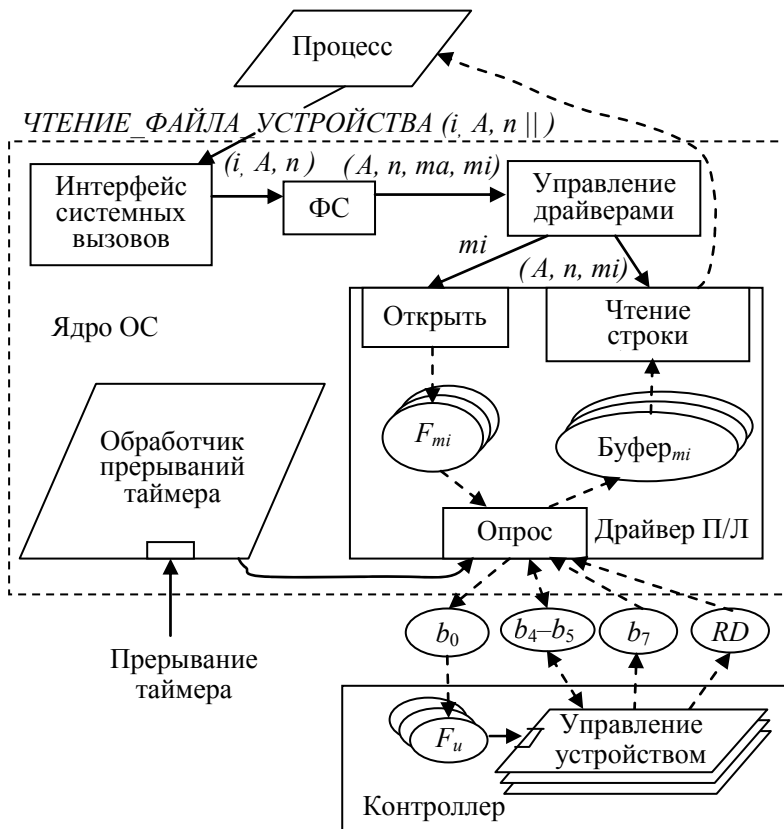


Рис. 61. Логическая структура драйвера с опросом:

$i$  — программный номер файла-устройства;  $A$  — начальный адрес прикладного буфера;  $n$  — число читаемых байтов;  $ma$  — старший номер устройства;  $mi$  — младший номер устройства;  $F_{mi}$  — флаг открытия  $mi$ -го устройства;  $b_0$  — бит  $RS$  разрешения работы;  $b_4-b_5$  — номер устройства;  $b_7$  — бит готовности в  $RS$ ;  $RD$  — буферный регистр данных;  $F_u$  — флаг запуска устройства

Данный вызов преобразуется интерфейсом системных вызовов в команду для виртуальной ФС, которая начинает выполнять открытие файла обычным образом. При этом проверяются права доступа к файлу и создаются необходимые записи в системных таблицах. Специальная реальная ФС находит в *inode* файла его атрибуты — старший и младший номера устройства. Эти числа

используются при формировании команды ФС для модуля управления драйверами.

Используя старший номер устройства, модуль управления находит в коммутаторе строковых драйверов вход в драйвер чтения с перфоленты и находит в этом входе строку с адресом процедуры «Открыть». При вызове данной процедуры ей на вход передается параметр — младший номер устройства.

Процедура «Открыть» проверяет установку флага открытия, соответствующего устройству с номером  $mi$ . Так как устройство ввода с перфоленты является последовательно используемым (см. подразд. 2.3), то оно не может быть открыто более одного раза. Поэтому если флаг  $F_{mi}$  установлен, то процедура «Открыть» блокирует вызывающий процесс (переводит в состояние «Сон»). Если же флаг  $F_{mi}$  сброшен, то процедура «Открыть» устанавливает этот флаг и запускает требуемое устройство, установив в  $1\ b_0$ , записав перед этим в  $b_4\text{--}b_5$  номер устройства  $mi$ . Каждая запись в бит  $b_0$  приводит к копированию его содержимого во флаг  $F_u$ , который управляет работой одного из устройств. Кроме того, процедура «Открыть» сбрасывает бит  $RS\ b_3$ , запрещая тем самым прерывания от устройства ввода с перфоленты, которые в данном драйвере не используются.

Как показано на рис. 61, упрощенная логическая структура контроллера представляет собой совокупность логических процессов, каждый из которых предназначен для управления одним устройством. В состоянии бездействия такой процесс находится в своей точке входа, ожидая установки своего управляющего флага  $F_u$ . Установка этого флага приводит к тому, что контроллер включает двигатель устройства, обеспечивающий перемотку перфоленты на один шаг. После этого очередной байт переписывается с перфоленты в буферный регистр данных  $RD$ . (Каждый байт хранится на перфоленте в виде восьми отверстий (битов), расположенных перпендикулярно направлению ленты.) Кроме того, логический процесс контроллера записывает номер своего устройства в биты  $b_4\text{--}b_5\ RS$ , а также производит установку бита готовности  $b_7$ . После этого данный логический процесс возвращается в точку своего входа.

Запустив устройство, процедура «Открыть» возвращает управление в модуль управления драйверами. Дальнейший возврат управления продолжится по цепочке модулей и завершится возвратом в прикладную программу процесса. При этом в качестве параметра будет передан программный номер  $i$  файла-устройства.

Допустим, что через некоторое время процесс передал в ядро системный вызов для чтения требуемого числа байтов  $n$  с перфоленты:

*ЧТЕНИЕ\_ФАЙЛА\_УСТРОЙСТВА* ( $i, A, n \parallel$ ).

Подобно обычным файлам, ФС находит *inode* файла, а затем использует его для определения номеров устройства  $ta$  и  $ti$ . Эти номера используются, в свою очередь, для выдачи запроса к модулю управления драйверами. Этот модуль находит требуемый вход в коммутаторе строковых драйверов и вызывает из этого входа процедуру «Чтение строки байтов».

Данная процедура выбирает из множества буферов драйвера тот буфер, который соответствует заданному номеру  $ti$ . Если количество символов, находящихся в этом буфере, не менее требуемого числа  $n$ , то первые  $n$  символов извлекаются из буфера драйвера и записываются в прикладную область памяти процесса с заданным начальным адресом  $A$ . Напомним, что модули верхнего уровня драйвера, как и модули ядра, соединяющие процесс с драйвером, выполняются в контексте этого процесса. Это обеспечивает данным модулям доступ не только к памяти ядра, но и к прикладной памяти процесса. После завершения записи в прикладную область памяти управление возвращается в прикладную часть процесса — на машинную команду, расположенную сразу же за командой системного вызова.

Если на момент выполнения процедуры «Чтение строки байтов» буфер драйвера не содержит требуемого числа символов, то данная процедура переписшет имеющиеся в буфере символы в прикладную область процесса, а затем выполнит блокирование текущего процесса, переводя его в состояние «Сон». Разблокирование процесса произойдет позже, когда процедура «Опрос» или обнаружит наличие в буфере недостающего числа символов, или обнаружит, что буфер полон. В любом случае выполнение процесса продолжится с процедуры «Чтение строки байтов», которая не только переписшет недостающие символы из буфера драйвера в прикладную область, но и выполнит включение устройства (установкой бита  $b_0$ ).

Основной функцией процедуры «Опрос» является прием символов, читаемых устройствами ввода с перфоленты и записываемых затем контроллером в регистр  $RD$ . Данная процедура иницируется через постоянные промежутки времени обработчиком прерываний таймера (обычно через тик) и выполняется в контексте этого обработчика, не имея никакого доступа к прикладной памя-

ти текущего процесса. При каждом инициировании процедура «Опрос» переписывает символ из  $RD$  в буфер драйвера. Если после записи символа в буфер окажется, что буфер полон, то данная процедура выключает устройство (сбросом бита  $b_0$ ).

Так как процедуры «Чтение строки байтов» и «Опрос» работают с общими структурами данных, выполняясь в контексте разных процессов, то они не только кооперируются, но и конкурируют из-за этих структур данных и нуждаются в синхронизации (см. п. 2.4.3). Подобная синхронизация обеспечивается запретом на время выполнения процедуры «Чтение строки байтов» всех маскируемых прерываний (в том числе от таймера).

### 7.5.2. Двухуровневый драйвер с прерыванием на байт

Основная идея построения драйверов на основе использования сигналов прерывания заключается в том, чтобы информация о завершении устройством текущей операции ввода-вывода поступала в драйвер не по инициативе драйвера, то есть в результате опроса, а по инициативе самого устройства (или его ИУ). Рассмотрим реализацию такого драйвера опять на примере управления гипотетическим устройством ввода с перфоленты.

Логическая схема драйвера представляет собой активный пакет, состоящий из двух интерфейсных логических процедур, логического процесса и двух структур данных (рис. 62). Работа модулей верхнего уровня драйвера мало отличается от работы соответствующих модулей в драйвере с опросом. Отличия заключаются в том, что процедура «Открыть» не сбрасывает, а, наоборот, устанавливает бит разрешения прерываний  $b_3$ .

Что касается нижнего уровня драйвера, то образующий этот уровень обработчик прерываний иницируется сигналом прерывания, поступившим в ЦП из контроллера ввода с перфоленты. Этот сигнал прерывания поступает на вход обработчика в виде управляющего воздействия через коммутатор строковых драйверов (вход драйвера «обработка прерывания») или через дескрипторную таблицу прерываний  $IDT$  (см. п. 5.4.3). Второй из этих вариантов инициирования обработчиков прерываний реализуется в большем числе систем. После своего инициирования обработчик прерываний переписывает код символа из  $RD$  в буфер драйвера и, если

буфер не полон, устанавливает бит запуска устройства  $b_0$  и бит разрешения прерываний  $b_3$ . Иначе эти биты сбрасываются.

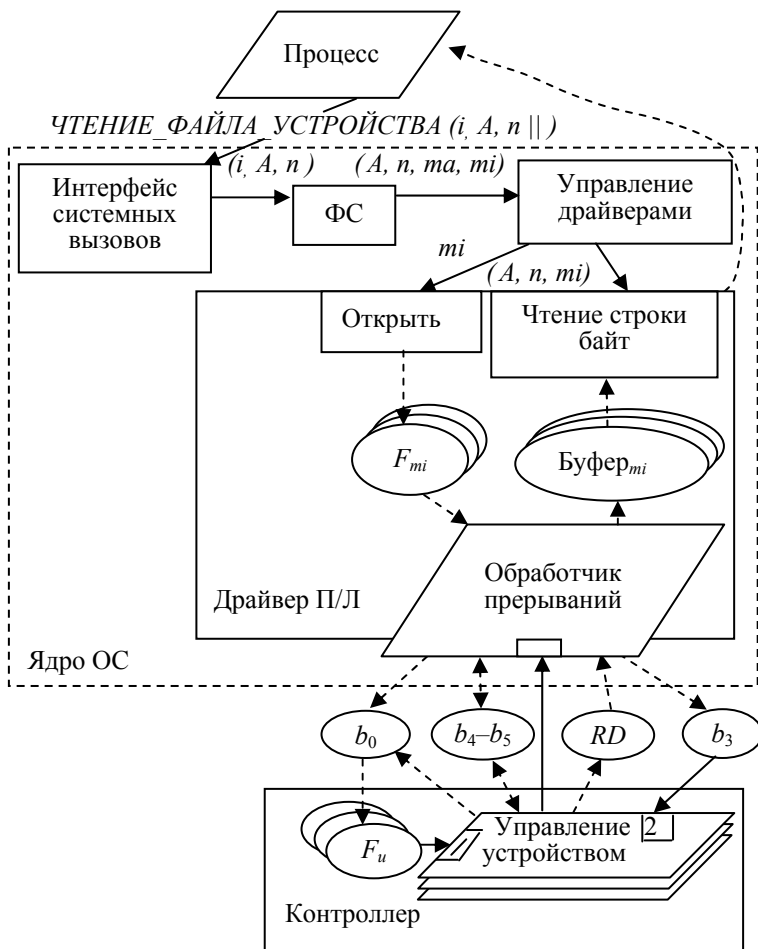


Рис. 62. Логическая структура драйвера с прерыванием:

$i$  — программный номер файла-устройства;  $A$  — начальный адрес прикладного буфера;  $n$  — число читаемых байтов;  
 $ta$  — старший номер устройства;  $mi$  — младший номер устройства;  
 $F_{mi}$  — флаг открытия  $mi$ -го устройства;  $b_0$  — бит  $RS$  разрешения работы;  
 $b_3$  — бит разрешения прерываний в  $RS$ ;  $b_4-b_5$  — номер устройства;  
 $RD$  — буферный регистр данных;  $F_u$  — флаг запуска устройства



Как и для драйвера с опросом, логическая схема контроллера представляет собой совокупность логических процессов, каждый из которых выполняет управление одним устройством. Если устройство не работает, его управляющий процесс находится в состоянии «Вход 1», ожидая установки флага запуска устройства  $F_u$ . Значение  $b_0 = 1$  приводит к установке этого флага для устройства с заданным номером, что инициирует работу контроллера по вводу очередного символа. Логический процесс контроллера помещает код символа в  $RD$  (сбрасывая при этом бит  $b_0 RS$ ) и оказывается в состоянии «вход 2». Так как бит разрешения прерывания  $b_3 = 1$ , то контроллер инициируется по данному входу и выдает сигнал прерывания. Обработчик прерываний считывает код символа из  $RD$  в буфер драйвера. Если при этом буфер оказывается полным, то управление сразу возвращается прерванной программе. Иначе сначала установкой  $b_0 = 1$  инициируется работа контроллера по вводу следующего символа.

В рассмотренном изложении предполагалось, что аппаратный сигнал внешнего прерывания поступает от ИУ сразу в ЦП. Но в большинстве ВС такие сигналы поступают сначала в **программируемый контроллер прерываний**. Данный контроллер упорядочивает поступление сигналов внешних прерываний в ЦП.

В качестве примера рассмотрим контроллер прерываний *i8259A*. Данный контроллер имеет 8 входов —  $IRQ0, IRQ1 \dots IRQ7$ , позволяющих принимать запросы на прерывание от восьми ИУ:

$IRQ0$  — таймер;

$IRQ1$  — клавиатура;

$IRQ2$  — второй контроллер прерываний;

$IRQ3$  — последовательный порт  $COM2$ ;

$IRQ4$  — последовательный порт  $COM1$ ;

$IRQ5$  — параллельный порт  $LPT2$ ;

$IRQ6$  — гибкий диск;

$IRQ7$  — параллельный порт  $LPT1$ .

Данные входы имеют приоритеты: чем меньше номер входа, тем приоритет выше. Поэтому самыми приоритетными являются сигналы прерываний от таймера. Приоритет сигнала используется контроллером прерываний для определения возможности направления этого сигнала в ЦП. При этом если в контроллер одновременно поступило несколько сигналов прерываний, то из них в ЦП будет отправлен тот сигнал, приоритет которого выше. Кроме того, если ЦП уже занят обработкой какого-то прерывания, то

новый сигнал будет отправлен контроллером прерываний только в том случае, если его приоритет выше, чем у того прерывания, обработчик которого выполняется на ЦП.

Номер прерывания, соответствующий конкретному ИУ, можно определить следующим образом: к базовому номеру, соответствующему контроллеру прерываний (8), следует прибавить номер входа в этот контроллер. Например, номер прерывания от клавиатуры:  $N = 8 + 1 = 9$ .

Как и любое ИУ, контроллер прерываний имеет порты, которые используются программами (драйверами) для управления им. Эти порты имеют адреса  $20h$  и  $21h$ . Порт  $21h$  используется, в частности, для маскирования (то есть для запрета) прерываний от устройств требуемого типа. Для этого требуется записать единицу в тот бит порта, который соответствует номеру входа для устройства. Например, следующие две команды выполняют маскирование прерываний от клавиатуры:

```
mov AL, 00000010b ; Вход для клавиатуры — IRQ1
out 21h, AL
```

Существуют другие команды управления контроллером прерываний, позволяющие размаскировать прерывания от любого ИУ (независимо от маскирования в ЦП), а также изменять приоритеты ИУ. Аналогичные команды используются и для управления вторым (ведомым) контроллером прерываний, который подсоединяется к входу  $IRQ2$  ведущего контроллера. Поэтому ИУ, обслуживаемые ведомым контроллером, выдают сигналы прерываний с приоритетами, расположенными между приоритетами клавиатуры и последовательного порта  $COM2$ . Для управления ведомым контроллером используются порты  $A0h$  и  $A1h$ , аналогичные портам  $20h$  и  $21h$  для ведущего контроллера. Базовый номер прерываний, соответствующий ведомому контроллеру прерываний, равен  $70h$ .

В заключение приведем два типа устройств, обслуживаемых ведомым контроллером прерываний (с указанием его входов):

```
IRQ13 — математический сопроцессор;
IRQ14 — жесткий магнитный диск.
```

Благодаря коммутаторам драйверов любая вышестоящая система (ФС, дисковый КЭШ или подсистема управления памятью) имеет стандартный интерфейс для доступа к любым драйверам. Допускается построение строчно-блоковых драйверов, имеющих свои записи в обоих коммутаторах. Поведение такого драйвера

(а точнее — используемый набор интерфейсных процедур) зависит от того, через какой коммутатор был сделан доступ к драйверу.

### **7.5.3. Блочные драйверы с прямым доступом в память**

Основное отличие блочных и строковых драйверов заключается в том, как они выполняют обслуживание поступающих запросов на ввод-вывод данных. Строковые драйверы используют для этого две отдельные интерфейсные процедуры «Чтение строки байт» и «Запись строки байт», каждая из которых использует отдельный буфер-очередь драйвера для размещения символьных строк, обслуживаемых в порядке поступления. Каждый блочный драйвер использует для обслуживания поступающих запросов на ввод-вывод данных единственную интерфейсную процедуру «Чтение-запись блока», которая помещает все запросы в единственную очередь (рис. 63), элементами которой являются дескрипторы буферов блоков, предназначенных для вывода на устройство или читаемых с него. При этом отметим, что блочный драйвер не имеет собственного буфера, так как в качестве буферов блоков используются или элементы буфера КЭШа (см. п. 6.4.2), или физические страницы ОП (см. п. 5.3.1).

Вызов процедуры «Чтение-запись блока» происходит следующим образом: дисковый КЭШ или подсистема управления памятью вызывает процедуру «Управление драйверами», передав ей на вход единственный параметр — указатель на дескриптор буфера блока. Этот дескриптор содержит детали требуемой операции информационного обмена с устройством. Структура дескриптора буфера блока была рассмотрена нами в п. 6.4.2. Одно из полей этой структуры содержит старший номер устройства и используется модулем управления драйверами для поиска требуемой точки входа в коммутаторе блочных драйверов. Далее модуль управления драйверами инициирует процедуру «Чтение-запись блока» в требуемом драйвере, передав ей указатель на дескриптор буфера блока.

Процедура «Чтение-запись блока» начинает свое выполнение с того, что помещает новый запрос в очередь на обслуживание. Элементами этой очереди являются дескрипторы буферов блоков. (Напомним, что два поля в каждом дескрипторе представляют собой указатели на соседние элементы очереди.) Далее процедура «Чтение-запись блока» выбирает один элемент из очереди и при-

ступает к его обслуживанию. При этом важно подчеркнуть, что выборка запросов из очереди производится не в порядке поступления, а в соответствии с некоторой стратегией эффективности. При этом под **эффективностью** обычно понимается общее время выполнения всех заявок в очереди, и, следовательно, целесообразно выбирать заявки из «очереди» в таком порядке, при котором данное время будет минимальным. Указанное общее время зависит в основном от времени перемещения головок дискового двигателя. От дискового КЭШ, или, что сначала выполнить чтение от подсистемы управления памятью цилиндра, а уж затем обрабатывать далеко расположенный блок, несмотря на то что соответствующий Коммутатор блочных о, поступила гораздо раньше.

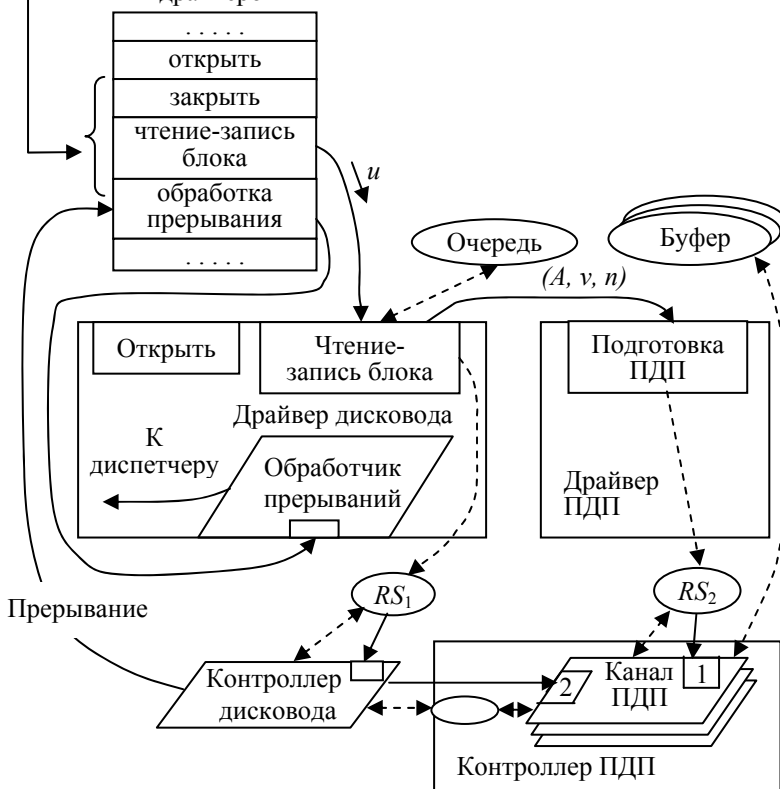


Рис. 63. Логическая схема блочного драйвера с ПДП:

$u$  — указатель на дескриптор буфера блока;  $A$  — реальный адрес буфера блока;  $v$  — тип операции (0 — чтение, 1 — запись);  
 $n$  — число читаемых (записываемых) байтов

Интересно отметить взаимосвязь между критерием эффективности работы драйвера и критерием производительности реальной ФС (см. п. 6.2.1). Организация информации на носителе в соответствии с требованием производительности реальной ФС лишь создает предпосылки для повышения фактической средней скорости информационного обмена между носителем ВП и ОП. Действительное увеличение данной скорости произойдет только в том случае, если драйвер обеспечивает эффективность этого обмена.

Выбрав очередной элемент из очереди, процедура «Чтение-запись блока» вызывает процедуру «Подготовка ПДП», входящую в состав драйвера ПДП, передав ей на вход параметры:

1)  $A$  — реальный адрес буфера блока. Для расчета данного адреса процедура «Чтение-запись блока» использует одно из полей дескриптора буфера блока, содержащее линейный виртуальный адрес  $A_b$ . Сам расчет производится так, как это делает ЦП. Отличие состоит в том, что расчет реального адреса выполняется не аппаратно, а программно;

2)  $v$  — тип операции (0 — чтение, 1 — запись);

3)  $n$  — число читаемых (записываемых) байтов.

Целью применения **прямого доступа в память (ПДП)** является существенное снижение затрат времени ЦП на обслуживание информационного обмена между устройством ВП и ОП за счет того, что ЦП не отвлекается для передачи каждого байта, а обрабатывает лишь завершение передачи всего блока. Для реализации данного метода синхронизации требуется наличие специального контроллера ПДП. В процессе ввода-вывода с использованием ПДП обмен байтами данных между ИУ и ОП производится не через регистр  $RD$  и регистры ЦП, а через этот контроллер. При этом роль ЦП ограничивается выдачей разрешений на занятие ОШ (а точнее — шины данных). Обычно контроллер ПДП рассчитан на одновременное обслуживание нескольких ИУ, каждое из которых подключается к нему набором проводников. Часть контроллера ПДП, предназначенная для обслуживания одного ИУ, называется **каналом ПДП**. В соответствии с классификацией информационных каналов из подразд. 2.5 канал ПДП представляет собой устойчивый

полудуплексный моноканал, предназначенный для передачи потока данных.

На рис. 63 приведена упрощенная логическая схема контроллера ПДП, согласно которой он представляет собой совокупность параллельных логических процессов (каналов ПДП) и имеет всего один регистр состояния и управления  $RS_2$ . В нерабочем состоянии канал ПДП ожидает записи иницирующего значения в регистр  $RS_2$ , находясь в состоянии «Вход 1». Процедура «Подготовка ПДП» не только помещает это значение в  $RS_2$ , но и записывает в этот регистр те параметры  $A$ ,  $v$ ,  $n$ , которые были переданы ей при вызове. Считав эти параметры, канал ПДП переходит в состояние «Вход 2», ожидая поступления команды от контроллера дисководов начать передачу информации.

Передавая содержимое некоторых полей дескриптора буфера блока в процедуру «Подготовка ПДП», процедура «Чтение-запись блока» помещает преобразованное содержимое некоторых других полей этого дескриптора в регистр  $RS_1$ :

1) номер однотипного устройства, обслуживаемого данным контроллером (и драйвером) дисководов. Данный номер получается дешифрацией младшего номера устройства;

2) физический адрес сектора на устройстве (номер цилиндра; номер поверхности; номер сектора). Для получения этих параметров процедура «Чтение-запись блока» использует поле дескриптора, содержащее порядковый номер сектора на диске;

3) тип операции (чтение или запись).

Так как контроллер дисководов обслуживает подключенные к нему дисководы только последовательно, то он изображен на логической схеме в виде одного логического процесса. Находясь в состоянии своего входа, этот логический процесс ожидает записи иницирующего значения в свой регистр состояния и управления  $RS_1$ . После того как процедура «Чтение-запись блока» запишет в  $RS_1$  перечисленные выше значения, контроллер дисководов начинает работу по записи или чтению требуемого сектора диска. Для этого он сначала передает команду требуемому дисководу, а затем иницирует подключенный к нему канал ПДП. После этого между двумя контроллерами начинается побайтовый обмен данными, считываемыми с диска или записываемыми на него.

По завершении передачи сектора диска контроллер дисководов выдает в ЦП сигнал прерывания. Обработчик этого прерывания, входящий в состав драйвера дисководов, выполняет деблокирование

процесса, ожидающего в состоянии «Сон» завершения операции чтения или записи дискового блока. После этого управление передается диспетчеру процессов, который или передает управление разбуженному процессу, или помещает его в список готовности.





## 8. ПРИКЛАДНОЙ УРОВЕНЬ СЕТИ

### 8.1. Общая структура сетевого приложения

Прикладной уровень сети образуют *сетевые приложения* — распределенные обрабатывающие программы. Каждая распределенная программа одновременно выполняется на нескольких ЭВМ, являющихся конечными узлами сети, называемыми *хостами*. Кроме хостов сеть обычно имеет внутренние узлы, обеспечивающие транспортировку информации между хостами. Так как внутренние узлы сети «не видны» из сетевых приложений, то мы их рассмотрим в другом разделе. Напомним, что распределенная программа обязательно имеет серверную часть и одну или несколько клиентских частей. При этом каждая из этих частей выполняется на своем хосте.

На рис. 64 приведена укрупненная структура распределенной программы, имеющей одну серверную и две клиентские части. В свою очередь, клиентская часть сама состоит из двух частей — интерфейсной и протокольной. При этом функцией протокольной части является обмен данными с удаленным сервером, имеющим для этого свою протокольную часть. Обе протокольные части взаимодействуют друг с другом согласно общему прикладному протоколу. Интерфейсная часть клиента обеспечивает его взаимодействие со своим пользователем, выполняя несетевую обработку команд пользователя и представляя для него на экране результирующую информацию клиента. Иными словами, интерфейсная часть клиента предоставляет в распоряжение пользователя пользовательский интерфейс.

Серверная часть распределенной программы также имеет две части — протокольную и управления данными. В то время как протокольная часть обеспечивает обмен данными с удаленными клиентами, вторая часть предоставляет эти данные для протокольной части. В зависимости от приложения управление данными включает управление или базами данных, или файлами, или другими информационными структурами.

Например, для рассмотренной в подразд. 3.1 распределенной программы *rlogin*, обеспечивающей доступ удаленного пользователя в *UNIX*-систему, протокольную часть серверной части содержит *rlogin*-сервер. А роль подсистемы управления данными выполняет псевдотерминал. Что касается самих этих данных, то это множество файлов в *UNIX*-системе, доступных по правам доступа

для конкретного удаленного пользователя. Подобные файлы содержат или загрузочные модули программ, или данные, обрабатываемые этими программами. Доступ к этим данным псевдотерминал выполняет через программу *shell*.

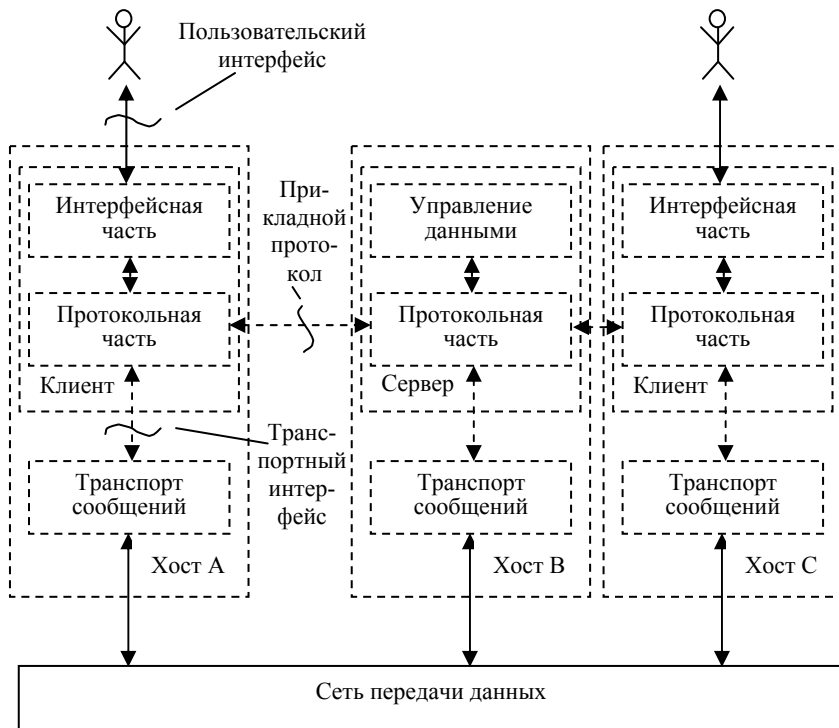


Рис. 64. Распределенная программа из трех частей

Протокольные части клиента и сервера «беседуют» друг с другом, используя для передачи своих сообщений модули транспорта сообщений, входящие в состав ОС. Благодаря этому взаимодействующие процессы имеют в своем распоряжении **транспортный информационный канал**. Непосредственное взаимодействие протокольной части клиента (или сервера) со своим модулем транспорта производится по правилам **транспортного интерфейса** (см. рис. 64). Так как при разработке любого сетевого приложения обязательно решаются вопросы выбора транспортного информационного канала и транспортного интерфейса, необходимо остановиться на этих вопросах более подробно.

## 8.2. Выбор транспортного информационного канала

В отличие от локальных информационных каналов, рассмотренных ранее, транспортный канал является *протяженным информационным каналом*, так как соединяет программные процессы, удаленные в пространстве. Для характеристики протяженных информационных каналов могут использоваться признаки, приведенные в подразд. 2.5 для локальных информационных каналов:

1) *направление передачи данных*. Информационные каналы делятся на симплексные, полудуплексные и дуплексные;

2) *устойчивость информационного канала*. Различают виртуальные соединения (устойчивые каналы) и датаграммные каналы (неустойчивые каналы);

3) *структурированность передаваемых данных*. Различают информационные каналы, передающие потоки данных и передающие сообщения;

4) *параллельность передаваемых данных*. Различают моноканалы и мультиплексные каналы.

Исходя из перечисленных признаков, транспортный информационный канал является дуплексным моноканалом, выполняющим передачу потоков данных. Он может быть реализован как виртуальное соединение и как датаграммный канал.

Кроме тех характеристик, которые используются для описания локальных каналов, для оценки протяженных каналов используются дополнительные характеристики. Перечислим только те из них, которые используются в качестве критериев при выборе транспортного канала:

1) *достоверность передачи данных* — вероятность искажения или потери единицы информации, передаваемой по каналу. В качестве единицы информации для транспортного канала целесообразно брать сообщение. Некоторые приложения отличают искажение сообщения от его потери, так как имеют возможность восстанавливать первоначальное содержимое сообщения, используя избыточные данные. В общем случае такое различие нецелесообразно;

2) *порядок приема информации*. По этому признаку транспортные каналы делятся на каналы, которые обеспечивают пра-

вильный порядок приема информации, и такие, которые этот порядок не обеспечивают;

3) **задержка** — время (в секундах) между началом передачи сообщения по каналу и моментом завершения приема этого сообщения;

4) **скорость передачи информации в канал** — предельное число единиц информации, гарантированно передаваемых в канал в течение одной секунды. В качестве единицы информации при этом используется 1 кбит или 1 Мбит. Термин «гарантированно» вовсе не означает, что переданная информация успешно достигнет удаленного конца канала. Это лишь говорит о том, что канал примет эту информацию для передачи.

Рассмотрим перечисленные характеристики протяженного информационного канала более внимательно. Во-первых, заметим, что обеспечить достаточно высокую достоверность передачи данных можно двумя путями: а) благодаря использованию для связи между хостами высококачественной сети передачи данных; б) путем выполнения повторных передач данных в случае их потерь и искажений. Первый из этих путей обусловлен тем, что транспортный канал физически «прокладывается», используя линии связи и внутренние узлы сети, соединяющие эти линии. В том случае если используются линии связи с высокой надежностью передачи данных, а внутренние узлы сети не перегружаются поступающими в них данными, и поэтому эти данные не отбрасывают, то достоверность передачи сообщений по транспортному каналу будет достаточно высока. В действительности лишь близко расположенные хосты соединяются (и то далеко не всегда) высококачественными сетями передачи данных. Что касается больших сетей, наибольшая из которых — *Internet*, то они не обеспечивают высокую достоверность передачи данных без использования повторных передач.

В принципе, повторную передачу сообщений можно выполнять и на прикладном уровне, то есть силами самого сетевого приложения. Для этого осуществляются следующие действия:

1) в передающем хосте приложение создает копию сообщения, отправляемого по транспортному каналу. Кроме того, сообщение снабжается порядковым номером, а также вспомогательной кодовой последовательностью, полученной в результате некоторой обработки сообщения. Все это вместе с сообщением передается по транспортному каналу;

2) в приемном хосте проверяется факт получения сообщения (по порядковому номеру) и проверяется его правильность (по соответствию сообщения и его кодовой последовательности). В зависимости от результатов проверки в обратном направлении посылается или служебное сообщение, подтверждающее успешный прием — **положительная квитанция**, или сообщение об ошибке — **отрицательная квитанция**. В конкретном прикладном протоколе, как правило, используется лишь один тип квитанций;

3) при получении отрицательной квитанции или длительном неполучении положительной квитанции передающий прикладной процесс повторяет передачу сообщения.

Порядок получения сообщений на приемном конце транспортного канала может не соответствовать порядку их отправки на передающем конце, вследствие того что разные сообщения могут пересылаться через сеть разными путями. Поэтому сообщение, отправленное позже, может прийти в хост назначения раньше. Восстановить правильный порядок среди переданных сообщений может само сетевое приложение. Для этого достаточно в приемной части приложения (клиенте или сервере) создать буфер, рассчитанный на несколько сообщений. При поступлении следующего сообщения оно помещается на то место в буфере, которое соответствует порядковому номеру сообщения (напомним, что этот номер передается вместе с сообщением).

Несмотря на то что приложение может само (за счет усложнения прикладного протокола) обеспечить высокую достоверность передачи данных, а также их правильный порядок, в том случае если эти характеристики следует обеспечить, лучше поручить это транспортному уровню. Разработанные один раз транспортные модули могут выполнять требуемое обслуживание многих приложений, что существенно снизит затраты на их разработку.

Заметим, что оценка транспортного канала по критерию достоверности передачи данных конфликтует с оценками канала по критериям задержки и скорости передачи информации в канал. Наличие первого конфликта очевидно, так как высокая достоверность передачи данных обычно достигается за счет повторных передач, что неизбежно увеличивает задержку некоторых сообщений, а также среднюю величину задержки. Что касается второго конфликта, то он не так очевиден, и мы вернемся к нему чуть позже.

Перечисленные критерии оценки транспортных каналов используются для выбора того канала, который наиболее удовлет-

воряет требованиям конкретного приложения. Любое сетевое приложение использует только однотипные транспортные каналы, созданные на основе одного и того же транспортного протокола. Среди многих транспортных протоколов наиболее распространены те протоколы, которые используются для передачи информации между приложениями в оконечных узлах сети *Internet* — **TCP** (*Transmission Control Protocol* — протокол управления передачей) и **UDP** (*User Datagram Protocol* — протокол пользовательских датаграмм). Первый из этих протоколов предназначен для организации транспортных информационных каналов в виде виртуальных соединений, а второй — в виде датаграммных каналов.

Так как транспортный канал на основе протокола *TCP* представляет собой виртуальное соединение, то он обеспечивает хорошую достоверность передачи сообщений между удаленными частями приложения, а также правильный порядок их приема. Что касается задержки, то для некоторых сообщений она будет гораздо выше, чем для других, из-за повторных передач. Кроме того, в случае перегрузок в приемном хосте или во внутренних узлах сети, через которые передается сообщение, протокол *TCP* «тормозит» поступление новых данных от приложения на передающем хосте. В результате скорость передачи информации в канал может стать весьма низкой.

Транспортный канал на основе протокола *UDP* является датаграммным каналом и не обеспечивает поэтому для приложения ни хорошую достоверность передаваемых сообщений-датаграмм, ни правильный порядок их приема. Первое обусловлено тем, что в случае потери датаграммы внутри сети или на приемном хосте никто внутри транспортного канала не занимается повторной передачей этой датаграммы. Второе свойство обусловлено тем, что транспортный канал на основе *UDP* не занимается «сортировкой» принимаемых датаграмм по времени их отправления, а просто передает их приложению по мере их получения из сети.

Характеристики — задержка и скорость передачи информации в канал — для датаграммного канала на основе протокола *UDP* существенно лучше, чем для виртуального соединения на основе протокола *TCP*. Это объясняется тем, что датаграммный транспортный канал не занимается ни повторной передачей сообщений-датаграмм, ни «торможением» их приема от приложения в передающем хосте. Поэтому передающая часть приложения направляет свои сообщения в транспортный канал с достаточно

высокой скоростью, но без уверенности в том, что они действительно дойдут до принимающей части приложения, тем более в правильном порядке.

Выбор транспортного канала для разрабатываемого сетевого приложения зависит от типа этого приложения. Будем различать следующие типы таких приложений:

1) приложения с передачей файлов. В общем случае такое приложение обслуживает пользователя, находящегося возле клиентского хоста. При этом производится пересылка файлов от серверной части приложения к клиентской части и (или) наоборот. Если размер файла достаточно велик, то прежде чем передавать информацию по транспортному каналу, сервер (клиент) делит файл на части-сообщения. Окончательное использование принятой информации в принимающей части приложения производится только после получения всего файла;

2) потоковые приложения. Принципиальное отличие такого приложения от приложений с передачей файлов — в том, что информация, передаваемая от сервера к клиенту, используется последним по частям, по мере поступления, не дожидаясь завершения приема всей информации. Подобным свойством обладают мультимедийные приложения, выполняющие передачу аудио- и (или) видеоданных в реальном времени. Примером является Интернет-телефония — устная беседа двух пользователей сети *Internet*, разделенных, возможно, многими тысячами километров. При этом как только любой из пользователей произнесет несколько звуков, эти звуки в закодированном виде сразу же передаются по сети. Затем они достигают приемного хоста и декодируются в нем приложением, становясь доступными для удаленного слушателя;

3) управляющие приложения. Такое сетевое приложение предназначено для сбора информации об удаленном объекте и (или) для оказания воздействия на этот объект с целью изменения его поведения. В качестве такого объекта может выступать технологический процесс, соединенный с хостом, или даже сам хост. Примером такого приложения, выполняющего управление удаленным хостом, является программа *rlogin*.

Перечисленные группы приложений предъявляют разные требования к транспортным каналам. Приложение с передачей файлов, как правило, очень критично к достоверности передачи данных, но не критично к задержке и к скорости передачи информации в канал. Это обусловлено тем, что при передаче

многих файлов недопустимы не только потери отдельных сообщений (фрагментов файла), но и искажения отдельных битов файла. В качестве примеров можно привести файлы, содержащие загрузочные модули программ, или файлы, содержащие банковскую информацию.

С другой стороны, пользователи, обслуживаемые приложением с передачей файлов, вполне допускают некоторый рост задержки. Подобный рост, естественно, приводит к увеличению всего времени реакции приложения на команду пользователя. Поэтому хотя приложение и не критично к величине задержки, ее снижение весьма желательно. Что касается скорости передачи информации в канал, то она влияет на пользователя приложения с передачей файлов лишь потому, что влияет на величину задержки.

С учетом сказанного при разработке приложений с передачей файлов в основном используют транспортные каналы на основе протокола *TCP*. Напомним, что это виртуальные соединения, обеспечивающие надежную передачу сообщений, а также правильный порядок их приема. Что касается задержки передачи сообщения, а также скорости передачи информации в канал, то никаких гарантий на их величины протокол *TCP* не предоставляет.

Потоковые приложения, выполняющие передачу аудио- и видеoinформации в реальном времени, не критичны к потере некоторой части (до 10 %) своих сообщений. Подобные потери приводят лишь к допустимому ухудшению качества звука или к возникновению небольших искажений на экране. С другой стороны, потоковые приложения весьма критичны к величине задержки и к скорости передачи информации в транспортный канал. Это объясняется тем, что большая задержка в передаче некоторых сообщений приводит к тому, что эти сообщения становятся ненужными («хороша ложка к обеду»). Кроме того, недопустимо какое-либо «торможение» в приеме транспортным каналом сообщений, содержащих аудио- и (или) видеoinформацию реального времени. Подобное замедление передачи неизбежно приведет к существенным искажениям звука и (или) изображения в приемном хосте.

С учетом сказанного при разработке мультимедийных потоковых приложений в основном используют транспортные каналы на основе протокола *UDP*. Напомним, что это датаграммные каналы, не обеспечивающие ни надежную передачу сообщений, ни правильный порядок их приема. С другой стороны, эти каналы имеют сравнительно небольшую задержку при передаче сооб-



щений-датаграмм, а также позволяют приложению передавать информацию в транспортный канал с постоянной скоростью.

Что касается управляющих приложений, то все они требуют от транспортного канала высокой достоверности передачи данных. Требование к задержке зависит от типа приложения. Если такое приложение обслуживает человека-пользователя, то к величине задержки не предъявляются жесткие требования. Если же приложение выполняет управление технологическим процессом, то допустимая величина задержки определяется скоростью изменений, протекающих в управляемом процессе: чем выше скорость, тем допустимая величина задержки меньше.

Если управляющее приложение не предъявляет жестких требований к величине задержки, то в качестве транспортного канала можно использовать *TCP*-соединение, реализованное в любой сети передачи данных, в том числе и *Internet*. Если же приложение требует, чтобы величина задержки не превышала предельную заданную величину, то в качестве сети передачи данных приходится использовать высококачественную сеть передачи данных. Для наиболее ответственных управляющих приложений в качестве такой сети могут использоваться выделенные каналы связи.

## 8.3. Транспортные интерфейсы

### 8.3.1. Транспортные порты

Транспортный интерфейс образуют системные вызовы, пользуясь которыми серверная и клиентские части сетевого приложения используют транспортные каналы, реализующие транспортный протокол *UDP* или *TCP*. С учетом того что точное описание транспортного интерфейса в этих протоколах отсутствует, существует некоторая свобода выбора типа транспортного интерфейса. Прежде чем рассматривать конкретный тип транспортного интерфейса, рассмотрим понятие транспортного порта, существенное для любого транспортного интерфейса.

**Транспортный порт** — структуры данных, образующие оконечности транспортных каналов, принадлежащие одной части (клиентской или серверной) конкретного приложения. Часто слово «транспортный» опускают, говоря просто «порт». Но с учетом того что термин «порт» имеет и другие значения (вспомним, например, про порты ввода-вывода), такое сокращение следует использовать осторожно.

Подобно другим объектам, управляемым ОС, все транспортные порты, принадлежащие конкретному хосту, пронумерованы, причем раздельно для каждого типа портов (*UDP* и *TCP*). Номер порта (*UDP* или *TCP*) — 16-битовое число, принимающее значение от 0 до 65535. Первые номера портов от 0 до 1023 зарезервированы за портами серверов распространенных приложений, называемых **стандартными портами**.

Некоторые из стандартных *UDP*-портов:

7 — «эхо» (сообщение, переданное клиентом серверу по сети, пересылается сервером обратно. Используется для анализа работы сети);

11 — список активных пользователей;

17 — цитата дня;

37 — текущее время (пересылается от сервера к клиенту в виде 32-битного числа, содержащего число секунд, прошедших с фиксированной секунды, — нулевая секунда 1 января 1990 года);

53 — сервер доменных имен (рассматривается в подразд. 8.4);

69 — сервер упрощенной передачи файлов *TFTP*.

Некоторые из стандартных *TCP*-портов:

20 — порт *FTP*-сервера для передачи данных;

21 — порт *FTP*-сервера для передачи управляющей информации;

53 — сервер доменных имен.

Большие номера портов выделяются ОС динамически по запросам локальных (то есть находящихся на этом же хосте) программ-клиентов. Программы-серверы обычно ограничиваются использованием одного (реже двух) стандартных портов. Но если этого недостаточно, они также обращаются к ОС с просьбой выделить дополнительные порты.

Для того чтобы серверная (или клиентская) часть приложения могла получать через свой порт сообщения от удаленной клиентской (или серверной) части, этот порт должен иметь внешний адрес, включающий, кроме номера порта, сетевой адрес того хоста, на котором порт находится. Общепринято использовать в качестве сетевого адреса хоста *IP*-адрес (*Internet Protocol* — межсетевой протокол, протокол сети *Internet*). ***IP-адрес*** представляет собой 32-битное (4-байтовое) число, уникальное для каждого хоста сети. Данный адрес принято изображать в виде четырех десятичных чисел, разделенных точками и представляющих значения каждого из байтов, например: 195.120.16.7. Числа *IP*-адреса, рассматрива-

емые слева направо, последовательно уточняют расположение адресуемого хоста в сети *Internet*.

Из всех существующих реализаций транспортных интерфейсов наиболее распространены интерфейсы на основе сокетов. Ранее мы уже рассматривали применение сокетов для создания локальных информационных каналов, связывающих процессы, выполняющиеся на одной и той же ЭВМ. Сейчас расширим это представление для связи удаленных частей сетевых приложений.

### 8.3.2. Интерфейс сокетов *UDP*-канала

Напомним, что сокет представляет собой окончание информационного канала, состоящее из двух очередей. Сейчас в качестве такого канала мы рассматриваем транспортный *UDP*-канал. Этот канал состоит из двух удаленных *UDP*-портов, каждый из которых содержит всего один сокет.

Как и для локального канала, каждый сокет имеет локальное системное имя (номер сокета в хосте), а также может иметь два программных имени: 1) **локальное имя сокета**, используемое той частью сетевого приложения, которая является «владельцем» этого сокета; 2) **внешнее имя сокета**, используемое удаленными частями приложения. Напомним, что внешнее имя сокета обычно называют **адресом сокета**. Как и для локальных каналов, первое из этих имен представляет собой «номер открытого файла». Второе имя должно быть уникальным в пределах всей сети передачи данных. В качестве такого имени (адреса) используется внешнее (сетевое) имя того порта, которому принадлежит данный сокет. Напомним, что внешнее имя порта включает сетевой *IP*-адрес хоста, а также номер *UDP*-порта.

Все, что было сказано ранее о создании на основе сокетов локального датаграммного информационного канала, относится и к созданию *UDP*-канала. При этом следует учесть лишь небольшие изменения:

1) при создании своего сокета (с помощью системного вызова *СОЗДАТЬ\_СОКЕТ*) в качестве коммуникационного домена (это множество имен, к которому принадлежит внешнее имя сокета) следует задать *AF\_INET* — множество имен *Internet*;

2) в качестве адреса сокета в системных вызовах *СВЯЗАТЬ\_СОКЕТ\_АДРЕС*, *ПОСЛАТЬ\_ДАТАГРАММУ*, *ПОЛУЧИТЬ\_ДАТАГРАММУ* используется пара (*IP*-адрес хоста, номер порта). Причем в системном вызове *СВЯЗАТЬ\_СОКЕТ\_АДРЕС* номер порта задает-

ся явно в виде числа (для *UDP*-порта, принадлежащего серверной части приложения), или этот номер проставляется самой ОС при выполнении данного вызова (для клиентских частей приложения). Никакого отдельного системного вызова для определения номера порта нет. В вызове *ПОСЛАТЬ\_ДАТАГРАММУ* номер порта всегда задается самим приложением. Это или номер стандартного *UDP*-порта, или номер порта был получен ранее (вместе с *IP*-адресом в результате приема датаграммы с помощью системного вызова *ПОЛУЧИТЬ\_ДАТАГРАММУ*).

### 8.3.3. Интерфейс сокетов *TCP*-канала

В отличие от *UDP*-порта, *TCP*-порт, принадлежащий серверной части приложения, содержит в общем случае не один, а несколько сокетов. Один из них является главным сокетом и используется для приема запросов на создание виртуальных соединений. Остальные сокеты являются окончаниями созданных виртуальных соединений и используются для приема-передачи данных по этим каналам. Таким образом, один и тот же *TCP*-порт сервера одновременно используется для создания виртуальных соединений и участвует в работе этих соединений. *TCP*-порт клиента включает всего один сокет, выполняющий роль окончания виртуального соединения.

Для создания главного сокета сервера и сокета клиента используются системные вызовы *СОЗДАТЬ\_СОКЕТ*. Все остальные сокет сервера создаются во время выполнения системного вызова *ПОЛУЧИТЬ\_ЗАПРОС*. Все, что было сказано ранее о создании на основе сокетов локального виртуального соединения, относится и к созданию *TCP*-канала. При этом следует учесть лишь небольшие изменения:

1) при создании сокетов с помощью системного вызова *СОЗДАТЬ\_СОКЕТ* в качестве коммуникационного домена следует задать *AF\_INET* — множество имен *Internet*;

2) в качестве адреса сокета в системных вызовах *СВЯЗАТЬ\_СОКЕТ\_АДРЕС* и *СОЕДИНИТЬ\_СОКЕТЫ* используется пара (*IP*-адрес хоста, номер порта). Причем в системном вызове *СВЯЗАТЬ\_СОКЕТ\_АДРЕС* номер порта задается явно в виде числа (для стандартных *TCP*-портов), или этот номер проставляется самой ОС при выполнении данного вызова. В вызове *СОЕДИНИТЬ\_СОКЕТЫ* номер порта всегда задается самим приложением-клиентом как номер стандартного *TCP*-порта сервера.

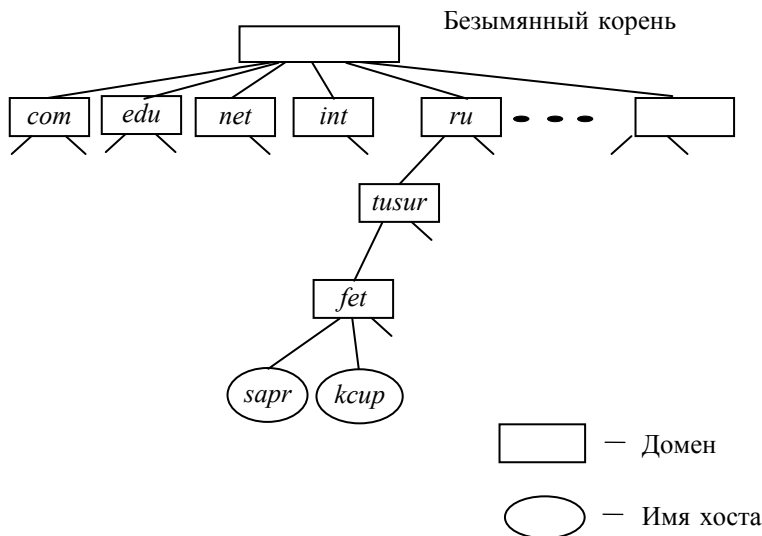
## 8.4. Трансляция имен хостов

Во многих приложениях первоисточником *IP*-адресов хостов, используемых в качестве обязательных частей адресов *UDP*- и *TCP*-портов, является пользователь клиентской части приложения. То есть прежде чем выдать в ОС системные вызовы по созданию и использованию транспортного канала, приложение получает от пользователя сетевое имя того хоста, на котором находится серверная часть приложения. Использование в качестве пользовательского сетевого имени *IP*-адреса (как и любого другого численного имени) очень неудобно для пользователя. Поэтому в качестве пользовательских сетевых имен хостов принято использовать символьные имена хостов, называемые доменными именами.

**Доменное имя хоста** — последовательность меток, разделенных точками, которая зарегистрирована в системе доменных имен *DNS* (*DNS* — *Domain Name System*). Здесь **метка** — последовательность латинских букв, цифр, других символов, за исключением точки. Например, один из хостов на кафедре КСУП (компьютерные системы в управлении и проектировании) в Томском государственном университете систем управления и радиоэлектроники (ТУСУР) имеет доменное имя *kcup.fet.tusur.ru*. В состав этого имени входят метки *kcup*, *fet*, *tusur*, *ru*. Приведенное доменное имя хоста является «листом» в дереве имен *Internet*, содержащем доменные имена всех хостов, подключенных к сети *Internet*. Фрагмент этого дерева приведен на рис. 65.

Структура дерева имен *Internet* очень похожа на структуру файловой системы. Точно так же вверху дерева находится корень, который, правда, не имеет имени и поэтому в имени хоста отсутствует. На следующем уровне дерева находятся имена доменов верхнего (первого) уровня. Здесь **домен** — множество имен сети *Internet*. Перечислим некоторые имена доменов первого уровня:

- com* — коммерческие организации;
- edu* — образовательные учреждения;
- net* — основные центры поддержки сети *Internet*;
- int* — международные организации;
- fr* — организации, расположенные во Франции;
- ru* — организации, расположенные в России.

Рис. 65. Фрагмент дерева доменных имен *Internet*

Так как домен является множеством, то он может быть разбит на подмножества (поддомены). Например, одним из поддоменов домена *ru* является *tusur.ru*. Обратим внимание, что в отличие от имени-пути файла имя домена верхнего уровня (*ru*) записывается не в левой, а в правой части доменного имени. Аналогично одним из поддоменов *tusur.ru* является *fet.tusur.ru* (ФЭТ — один из корпусов ТУСУРа). Домен *fet.tusur.ru* может иметь свои поддомены, а также хосты, в том числе и *kcup.fet.tusur.ru*.

Рассмотрим кратко вопрос о том, кто занимается построением дерева имен *Internet*. Во-первых, данный процесс происходит не стихийно, а организованно. Во-вторых, в этом участвует не одна, а много организаций. Главная из этих организаций — *ICANN* (*Internet Corporation for Assigned Names and Numbers*). Она назначает имена доменов первого и второго уровней, а также утверждает организации, каждая из которых ответственна за свой домен второго уровня. Одной из таких организаций является ТУСУР. Он занимается декомпозицией (детализацией) домена *tusur.ru*, не отчитываясь за результаты перед вышестоящей организацией.

Имя любого домена, являющегося «потомком» домена *tusur.ru*, ТУСУР может передать другой организации, которая и будет теперь отвечать за декомпозицию этого домена. Например, имя *kcup.fet.tusur.ru* передано кафедре КСУП, которая вправе присвоить его одному из своих хостов или использовать его в качестве имени своего домена.

В отличие от *IP*-адреса хоста, отражающего фактическое подключение этого хоста к сети передачи данных, доменное имя хоста в общем случае таким свойством не обладает. Например, все хосты, имена которых принадлежат домену *fet.tusur.ru*, подключены или к сети корпуса ФЭТ, или к локальным сетям, являющимся подсетями указанной сети. С другой стороны, хосты, принадлежащие одному и тому же домену *net*, могут быть расположены в разных концах света.

Размер дерева доменных имен чрезвычайно велик и продолжает быстро увеличиваться. Вспомним, что гораздо меньшее по размеру дерево файловой структуры локальной *UNIX*-системы не хранится в одном месте этой системы как единое целое. Для системы доменных имен *DNS*, используемой в условиях сети, распределенный характер тем более важен. При этом распределение дерева имен производится не по устройствам, а по узлам сети.

Каждая программа, называемая **сервером *DNS***, содержит фрагмент дерева доменных имен, представляющий собой верхнюю часть одного из его поддеревьев. При этом для каждого доменного имени сервер *DNS* содержит соответствующий *IP*-адрес. Кроме того, каждый сервер *DNS* содержит *IP*-адреса нескольких других *DNS*-серверов, к которым относятся: а) корневой сервер; б) вышестоящий («родительский») *DNS*-сервер, содержащий «родительский» домен для поддерева данного сервера; в) «дочерние» *DNS*-серверы, которые содержат декомпозицию нижестоящих доменов данного сервера. Таким образом, все *DNS*-серверы оказываются связанными в единое дерево, корнем которого является **корневой сервер *DNS***. Листьями данного дерева являются **локальные *DNS*-серверы**. Каждый хост «знает» *IP*-адрес своего локального сервера, так как этот адрес прописывается вручную при настройке ОС. Обычно это самый близкий *DNS*-сервер, расположенный в той же локальной сети, что и хост.

Перейдем к рассмотрению обслуживания сетевых приложений со стороны *DNS*-серверов. Любое сетевое приложение (а точнее —

клиентская часть приложения) может стать *DNS*-клиентом, если включит в состав своей программы специально предназначенную для этого библиотечную функцию (процедуру). В *UNIX*-системе это функция *gethostbyname*, имеющая один параметр — указатель на символьную строку, содержащую исходное *DNS*-имя. Данная функция возвращает значение — указатель на структуру, содержащую список из *IP*-адресов, соответствующих заданному *DNS*-имени. Если ни один *IP*-адрес не найден, то возвращается указатель *NULL*.

Программный код функции *gethostbyname* таков, что он содержит системные вызовы для создания *UDP*-канала между хостом и *UDP*-портом номер 53 своего локального сервера, а также для отправки по этому каналу датаграммы-запроса, содержащей исходное *DNS*-имя. Получив запрос от приложения, локальный сервер действует следующим образом: если полученное *DNS*-имя имеется в базе данных самого локального сервера, то приложению возвращается искомый *IP*-адрес. В противном случае локальный сервер проверяет, является ли поступившее от приложения *DNS*-имя «родственным» имени того домена, за который отвечает данный сервер. (У «родственных» имен совпадают правые части.) Если имена родственны, то поступившее *DNS*-имя направляется вышестоящему *DNS*-серверу. При этом локальный сервер выступает в роли клиента, а вышестоящий *DNS*-сервер — в роли сервера. Если локальный сервер выяснит, что *DNS*-имена «неродственны», то он передаст поступившее от приложения *DNS*-имя корневому серверу. Так как корневой сервер «знает» *IP*-адреса всех *DNS*-серверов, которые ответственны за декомпозицию доменов второго уровня, то он возвратит в локальный сервер *IP*-адрес искомого *DNS*-сервера. Пользуясь этим *IP*-адресом, локальный сервер вновь повторит свой запрос. Полученный в результате ответ он возвратит приложению. Перейдем к рассмотрению вопроса о том, как локальный и другие серверы выполняют поступающие к ним запросы.

Каждый запрос на обслуживание, поступающий в *DNS*-сервер от приложения или от другого сервера, содержит кроме *DNS*-имени тип записи, уточняющий то значение, которое сервер должен возвратить клиенту. Перечислим наиболее интересные типы записи:

— *A* (сокращение от *address*) — возвращаемое значение представляет собой *IP*-адрес, соответствующий заданному *DNS*-имени;



– ***MX*** (сокращение от *Mail eXchanger*) — возвращаемое значение представляет собой *IP*-адрес, соответствующий заданному *DNS*-имени и используемый для приема электронной почты. Применение данного типа позволяет использовать для приема электронной почты и для других приложений (в том числе и *Web*) два разных хоста, имеющих одинаковое *DNS*-имя, но разные *IP*-адреса;

– ***CNAME*** — переданное в запросе *DNS*-имя представляет собой псевдоним настоящего (канонического) имени хоста. Возвращаемое значение представляет собой *IP*-адрес, соответствующий каноническому имени. Например, если хост *kcup.fet.tusur.ru* используется в качестве *Web*-сервера, то в качестве псевдонима для этого хоста может использоваться *www.kcup.fet.tusur.ru*. Естественно, что для задания хоста пользователь может использовать не только псевдоним, но и каноническое имя.

Получив запрос от клиента, *DNS*-сервер выполняет поиск в своей базе данных, в которой хранится информация о каждом *DNS*-имени из того поддерева имен *DNS*, за которое отвечает данный сервер. Каждая запись в базе данных сервера содержит: 1) *DNS*-имя; 2) тип записи; 3) значение. При этом поля «*DNS*-имя» и «тип записи» используются в качестве исходной информации для поиска требуемого содержимого третьего поля.

Заметим, что пользователь сетевого приложения часто может использовать не только полные *DNS*-имена, но и сокращения от этих имен. Например, сокращениями имени хоста *kcup.fet.tusur.ru* являются *kcup.fet.tusur*, *kcup.fet* и *kcup*. Так как *DNS*-серверы имеют дело исключительно с полными *DNS*-именами, то всю работу по преобразованию сокращенного *DNS*-имени в полное имя выполняет *DNS*-клиент (более точно — процедура *gethostbyname*). Например, пусть пользователь использовал в диалоге с сетевым приложением *DNS*-имя *kcup*. Тогда, обратившись к *DNS*-серверу, *DNS*-клиент получит значение *NULL*. После этого *DNS*-клиент последовательно увеличивает длину *DNS*-имени за счет подсоединения к сокращенному *DNS*-имени суффиксов *fet.tusur.ru*, *tusur.ru*, *ru* до тех пор, пока не получит от своего локального сервера искомый *IP*-адрес. Естественно, что перечисленные суффиксы должны быть заранее введены в качестве констант, доступных процедуре *gethostbyname*. Заметим, что в приведенном примере первое же присоединение суффикса *fet.tusur.ru* приведет к получению полного *DNS*-имени.

Для повышения эффективности использования системы *DNS* используются два приема — тиражирование корневого *DNS*-сервера и кэширование имен. Рассмотрим эти приемы.

Так как корневой *DNS*-сервер используется при преобразовании в *IP*-адреса многих *DNS*-имен, то если бы этот сервер был в сети *INTERNET* один, нагрузка на него была бы весьма велика. Кроме того, передача данных между *DNS*-серверами и единственным корневым сервером привела бы к излишней загрузке магистральных каналов связи. При этом выход из строя корневого сервера неизбежно привел бы к плохим последствиям для всей сети *INTERNET*. Во избежание перечисленных трудностей используется не один, а много экземпляров корневого сервера, расположенных в различных географических точках. Каждый такой экземпляр представляет собой полную копию главного корневого сервера и выполняет обслуживание тех *DNS*-серверов, которые расположены в данном районе.

Кэширование *DNS*-имен позволяет существенно сократить объем данных, передаваемых по сети при выполнении *DNS*-запросов. В основе данного метода лежит тот факт, что многие *DNS*-имена применяются пользователями сетевых приложений многократно. Поэтому каждый *DNS*-сервер временно хранит (обычно 48 часов) те *DNS*-записи, которые он получает от других *DNS*-серверов в ходе обслуживания поступающих к нему запросов. Если в течение временного хранения *DNS*-записи в данный *DNS*-сервер поступит запрос, содержащий те же *DNS*-имя и тип записи, то *DNS*-сервер сразу же возвратит искомый *IP*-адрес, не обращаясь за помощью к другим *DNS*-серверам. Естественно, что при этом будет обнулен счетчик прошедшего времени хранения.

## 8.5. Приложения для передачи файлов

### 8.5.1. Приложения на основе протокола *FTP*

Очень распространены сетевые приложения для передачи файлов, которые используют прикладной протокол ***FTP*** (*File Transfer Protocol* — протокол передачи файлов). В отличие от рассмотренного выше приложения *DNS*, такое приложение обслуживает не только программы, но и пользователей. Поэтому оно предоставляет для такого обслуживания не только программный, но и пользовательский интерфейсы. Ранее была рассмотрена общая структура подобных сетевых приложений.

Обычно на одном и том же хосте имеются и клиентская, и серверная части *FTP*-приложения, которые никак не связаны друг с другом по управлению. При этом клиентская часть выполняет запросы пользователей и программ по выполнению файлового обмена с другими хостами. А серверная часть выполняет запросы удаленных клиентов по доступу к файлам на своем хосте. Сущность обслуживания, предоставляемого *FTP*-приложением своему пользователю, кратко заключается в том, что по команде пользователя производится или копирование удаленного файла в локальную файловую структуру, или, наоборот, копирование локального файла в файловую структуру на удаленном хосте. Для выполнения любой из этих операций требуется, чтобы права доступа данного пользователя к файлу на удаленном хосте соответствовали типу этой операции. Естественно, что для получения каких-то особых прав доступа пользователь должен, во первых, заранее быть зарегистрирован на удаленном хосте (получить символическое имя и пароль), а во вторых, в начале работы с *FTP*-клиентом пользователь должен пройти процедуру аутентификации — правильно назвать свое имя и пароль. В том случае если пользователь не зарегистрирован на удаленном хосте, он может получить минимальные права доступа к файлам этого хоста, используя стандартное имя — *anonymous*, а в качестве пароля — *quest*.

Сам протокол *FTP* задает только команды, которыми обмениваются протокольные части клиента и сервера, но не определяет пользовательский интерфейс. Поэтому клиентские части *FTP*-приложений, выполняющиеся на разных хостах, могут иметь разные наборы пользовательских команд. Ниже перечислены некоторые из наиболее используемых таких команд:

1) *open* — подготовиться к пересылке файла (файлов) между локальным и удаленным хостами. При выполнении данной команды *FTP*-клиент запрашивает у пользователя *DNS*-имя удаленного хоста (если при вводе команды *shell - ftp* пользователь не ввел в качестве параметра *IP*-адрес этого хоста). Кроме того, *FTP*-клиент выполняет аутентификацию пользователя, запрашивая у него имя и пароль;

2) *close* — команда, обратная *open*. Она уничтожает транспортные каналы между локальным и удаленным хостами. Применение данной команды не означает завершение работы *FTP*-

клиента, так как пользователь может выдать новую команду *open* для этого же или для другого удаленного хоста;

3) **get** — передать копию файла с удаленного хоста на локальный хост. В качестве первого параметра команды пользователь может задать имя требуемого файла на удаленном хосте, а в качестве второго параметра — имя файла-копии на локальном хосте. Если второй параметр отсутствует, то имя файла-копии будет таким же, как и у исходного файла. Если отсутствует и первый параметр, то *FTP*-клиент сам запросит у пользователя имя копируемого файла;

4) **mget** — передать с удаленного хоста на локальный хост копии нескольких файлов;

5) **put** — передать копию файла с локального хоста на удаленный хост. В качестве первого параметра задается имя файла на локальном хосте, а в качестве второго параметра — имя файла-копии на удаленном хосте;

6) **mput** — передать с локального на удаленный хост копии нескольких файлов;

7) **pwd** — определить имя-путь текущего каталога на удаленном хосте. Текущий каталог на удаленном хосте используется точно так же, как и текущий каталог на локальном хосте — для сокращения задаваемых имен файлов;

8) **cd** — сделать переход в другой текущий каталог на удаленном хосте;

9) **ls** — вывести на экран содержимое требуемого каталога на удаленном хосте;

10) **quit** — завершить работу *FTP*-клиента.

Каждой команде пользователя соответствует одна или несколько *FTP*-команд, передаваемых от *FTP*-клиента к *FTP*-серверу. Получив очередную команду, сервер посылает клиенту ответ. Одной из особенностей *FTP*-протокола является то, что и команды клиента, и ответы сервера представляют собой символьные строки в коде *ASCII*, которые при желании могут быть выведены на экран пользователя и прочитаны им. Перечислим некоторые команды пользователя и соответствующие им *FTP*-команды:

1) при получении команды пользователя *open* клиент посылает серверу следующие команды:

а) **USER** *имя\_пользователя* — передает серверу имя пользователя;

б) **PASS** *пароль* — передает серверу пароль пользователя;

2) при получении команды пользователя *ls* клиент посылает серверу команду **LIST** — запрос на пересылку содержимого текущего каталога;

3) при получении команды пользователя *get* клиент посылает серверу команду **RETR имя\_файла** — запрос на пересылку копии заданного файла с удаленного хоста на локальный хост. При получении от пользователя команды *mget* клиент посылает серверу несколько команд **RETR** — по одной на каждый копируемый файл;

4) при получении команды пользователя *put* клиент посылает серверу команду **STOR имя\_файла** — запрос на пересылку копии заданного файла с локального на удаленный хост. При получении от пользователя команды *mput* клиент посылает серверу несколько команд **STOR** — по одной на каждый копируемый файл.

Прежде чем обмениваться *FTP*-командами и ответами на них, клиент и сервер создают управляющее *TCP*-соединение. Инициатором создания такого соединения является клиент, а точнее — главный клиентский процесс. Запрос на создание управляющего соединения этот процесс посылает в *TCP*-порт с номером 21, принадлежащий хосту с требуемым *FTP*-сервером. При этом *IP*-адрес требуемого хоста сервера известен или из команды пользователя, запускающей *FTP*-клиент, или в результате преобразования *DNS*-сервером того *DNS*-имени, которое сообщил пользователь.

В *FTP*-сервере главный процесс выполняет «прослушивание» *TCP*-порта с номером 21 с целью считывания запросов на создание управляющих *TCP*-соединений. Считав подобный запрос от очередного клиента, главный процесс сервера создает не только управляющее *TCP*-соединение, но и дочерний процесс, предназначенный для обслуживания этого виртуального соединения. Само обслуживание заключается в том, что дочерний процесс принимает по управляющему *TCP*-соединению *FTP*-команды от главного процесса клиента, а затем посылает по этому транспортному каналу свои ответы.

Для передачи данных какого-то файла (по команде **RETR** или **STOR**), а также для передачи содержимого какого-то каталога (по команде **LIST**) между дочерним процессом сервера и главным процессом клиента создается дополнительное *TCP*-соединение передачи данных. Инициатором создания такого виртуального соединения является сервер, в котором для передачи данных используется *TCP*-порт с номером 20. Так как инициатор создания *TCP*-соединения должен заранее знать не только *IP*-адрес, но и

номер *TCP*-порта удаленного партнера, то этот номер, полученный ранее клиентом динамически от своей локальной ОС, передается им серверу (при передаче команд *RETR*, *STOR*, *LIST*).

Кроме того, для выполнения приема и передачи данных по этому *TCP*-соединению и главный процесс клиента, и дочерний процесс сервера порождают свои дочерние процессы. Таким образом, в отличие от обычной модели клиент-сервер (см. п. 4.6.3), в которой клиент есть один процесс, а сервер — 2-уровневое дерево процессов, в *FTP*-приложении клиент имеет 2-х, а сервер — 3-уровневую структуру. Несмотря на подобное усложнение структур обеих частей приложения, использование отдельных *TCP*-соединений для передачи команд и для передачи данных приносит существенную пользу, так как позволяет существенно упростить алгоритмы программ процессов. Кроме того, управляющее соединение и соединение передачи данных могут использоваться параллельно, что предоставляет пользователю дополнительные возможности. Например, пользователь может прервать своей командой передачу файла.

Полезно сравнить *FTP*-приложение с рассмотренной в подразд. 6.3 сетевой операционной системой *NFS*. В отличие от *FTP*-приложения, каждая часть (клиент или сервер) системы *NFS* реализована не вне ядра своей ОС, а внутри него. Несмотря на это, система *NFS* также является сетевым приложением, так как не занимается транспортировкой информации по сети. В отличие от *FTP*-приложения, система *NFS* «прозрачна» для пользователя. То есть для того чтобы работать с удаленным файлом, ни от пользователя, ни от программиста, разрабатывающего программу, не требуется выдачи никаких дополнительных команд или системных вызовов. После того как выполнено монтирование удаленной файловой системы, работа с ней выполняется точно так же, как и с локальной ФС.

Кроме того, система *NFS* отличается от *FTP*-приложения по своим функциям. В то время как *FTP*-приложение выполняет «перекачку» файлов целиком, система *NFS* позволяет пользователям и программам работать с удаленными файлами точно так же, как и с локальными файлами. При этом может быть считан или скорректирован любой фрагмент файла без «перекачки» всего файла.

Так как программная реализация протокола *FTP* достаточно сложна, то для сокращения длины программы иногда используют

другой, простейший протокол передачи файлов **TFTP** (*Trivial File Transfer Protocol*). В отличие от *FTP*-приложения, *TFTP*-приложение использует для связи между своими клиентской и серверной частями не несколько параллельных *TCP*-соединений, а единственный *UDP*-канал (номер порта сервера — 69). Инициатором передачи файла, как всегда, является клиентская часть приложения. Получив запрос от какой-то программы на своем хосте, клиент посылает серверу сообщение-команду на чтение или запись указанного в команде файла.

Так как аутентификация пользователя в протоколе *TFTP* не предусмотрена, то запрашиваемая клиентом операция должна требовать лишь минимальные права доступа к файлу на удаленном хосте. Если это выполняется, то сервер начинает передачу (или прием) файла, имя которого указано в команде клиента. Сама передача данных файла производится блоками по 512 байтов. Так как *UDP*-канал не обеспечивает надежную передачу данных, то этим занимается само *TFTP*-приложение.

После передачи очередного блока передающая часть приложения (клиент или сервер) включает таймер интервала, передав ядру ОС заявку на аларм (см. подразд. 4.5). Если в течение указанного интервала времени передающая часть приложения не получит положительную квитанцию от принимающей части, то она выполнит повторную передачу блока файла. Принимающая часть приложения в случае успешного приема очередного блока файла отсылает передающей части положительную квитанцию. При этом если длина принятого блока оказалась менее 512 байтов, то делается вывод об успешном завершении передачи файла. Если принятый блок файла содержит ошибку, то прием файла также завершается.

Наибольшее применение *TFTP*-приложение нашло для начальной загрузки небольших бездисковых хостов. После включения питания на таком хосте аппаратно иницируется программа *TFTP*-клиента, «зашитая» в ПЗУ такого хоста. Этот клиент начинает с того, что посылает запрос на чтение исполняемого файла, содержащего ядро загружаемой ОС, в *TFTP*-сервер, расположенный в этой же локальной сети. После того как этот файл успешно принят, ему передается управление, и дальнейшую работу по подготовке данной системы к работе выполняет уже сама ОС.

К достоинствам подобной загрузки небольших ЭВМ, подключенных к сети, во-первых, относится небольшой объем ПЗУ:

кроме *TFTP*-клиента оно содержит лишь небольшую программу, реализующую протокол *UDP*. Во вторых, существенно повышается гибкость всей ВС, элементами которой являются бездисковые хосты, так как для замены программного обеспечения на этих хостах достаточно заменить лишь файл (или файлы) на одном хосте, содержащем *TFTP*-сервер. Никакие изменения в ПЗУ клиентов при этом не требуются.

### 8.5.2. Электронная почта

**Электронная почта** — одно из самых используемых сетевых приложений. Первоначально единственным ее назначением была передача текстовых сообщений между удаленными пользователями. Со временем пользователи получили возможность передавать наряду с текстами любые данные, хранящиеся на хосте-отправителе в виде файлов: картинки, двоичные коды программ и т.д. Эти файлы «прикрепляются» пользователем-отправителем к тексту его почтового сообщения. Среди многих вариантов реализации электронной почты наиболее распространена та, которая предназначена для обслуживания пользователей сети *Internet*. Особенности ее работы рассмотрим на следующем примере.

Допустим, что Вова, являющийся пользователем хоста *kcip.fet.tusur.ru*, решил отправить электронное письмо Пете, являющемуся пользователем хоста *sapr.fet.tusur.ru*. Для того чтобы подобное информационное взаимодействие было возможно, и Вова, и Петя должны иметь каждый свой собственный электронный **почтовый ящик** — область на носителе ВП, предназначенную для приема почтовых сообщений, присылаемых данному пользователю. Каждый почтовый ящик имеет символьное имя, уникальное для всей сети *Internet*. Это имя состоит из двух частей, разделенных символом «@» (коммерческое *at*): справа — *DNS*-имя того узла сети, на котором находится почтовый ящик, а слева — локальное имя почтового ящика в этом узле.

В нашем примере почтовые ящики имеют имена: *vova@mail.ru* и *petja@tusur.ru*. Нетрудно заметить, что *DNS*-имена, входящие в состав указанных имен почтовых ящиков, отличаются от *DNS*-имен тех хостов, на которых работают пользователи Вова и Петя. Возможны две причины такого несоответствия.

Во-первых, почтовый ящик обязательно находится в том же узле сети, где находится и **почтовый сервер** — программа, выполняющая сетевое обслуживание программ, посылающих сообщения



в почтовый ящик или читающих сообщения из него. Так как моменты времени, когда требуется подобное обслуживание, заранее не предсказуемы, почтовый сервер должен быть постоянно готов выполнить запросы на это обслуживание. И как следствие, ЭВМ, на которой выполняется почтовый сервер, должна быть не только постоянно включенной, но и подключенной к сети *Internet*. Так как большинство хостов не могут быть постоянно включенными и подключенными к *Internet*, то их пользователи создают свои почтовые ящики на удаленных хостах, специально предназначенных для выполнения почтовых серверов. *DNS*-имя такого почтового сервера и записывается в качестве правой части имен почтовых ящиков, обслуживаемых данным почтовым сервером.

Во-вторых, даже если почтовый сервер и выполняется на хосте пользователя, *DNS*-имена самого хоста и почтового сервера могут быть разные. Это обусловлено тем, что делая *DNS*-запрос, *DNS*-клиент сопровождает его указанием типа *A* или *MX*. Первый тип задается всеми приложениями, за исключением электронной почты, которая использует второй тип (см. подразд. 8.4). Поэтому одному и тому же *IP*-адресу хоста могут соответствовать два разных *DNS*-имени.

Вернемся к примеру. На рис. 66 сплошными стрелками показан путь, который проходит сообщение от Вовы до Пети, а пунктирными стрелками показан путь ответного сообщения — от Пети до Вовы. Отправку своего письма Вова начинает с того, что запускает на своем хосте программу — *почтовый агент*, который является одновременно клиентом двух (не считая *DNS*) сетевых приложений. Из этих клиентов при отправке письма используется *SMTP*-клиент. ***SMTP*** (*Simple Mail Transfer Protocol*) — простой протокол передачи электронной почты. Получив от Вовы через интерфейсную часть сообщение-письмо, а также адрес назначения *petja@tusur.ru* *SMTP*-клиент посылает это сообщение *SMTP*-серверу, входящему в состав почтового сервера, имеющего *DNS*-имя — *mail.ru*. Естественно, что перед отправкой сообщения *SMTP*-клиент временно становится *DNS*-клиентом для того, чтобы получить *IP*-адрес почтового сервера.

Получив сообщение-письмо, *SMTP*-сервер передает его *SMTP*-клиенту, расположенному в этом же почтовом сервере. Далее *SMTP*-клиент посылает это письмо тому *SMTP*-серверу, который входит в состав почтового сервера с требуемым *DNS*-именем — *tusur.ru*. Получив письмо, *SMTP*-сервер определяет из него имя

одного из своих почтовых ящиков (*petja*) и помещает в него письмо. Далее письмо лежит без движения в почтовом ящике до тех пор, пока Петя не захочет его прочитать.

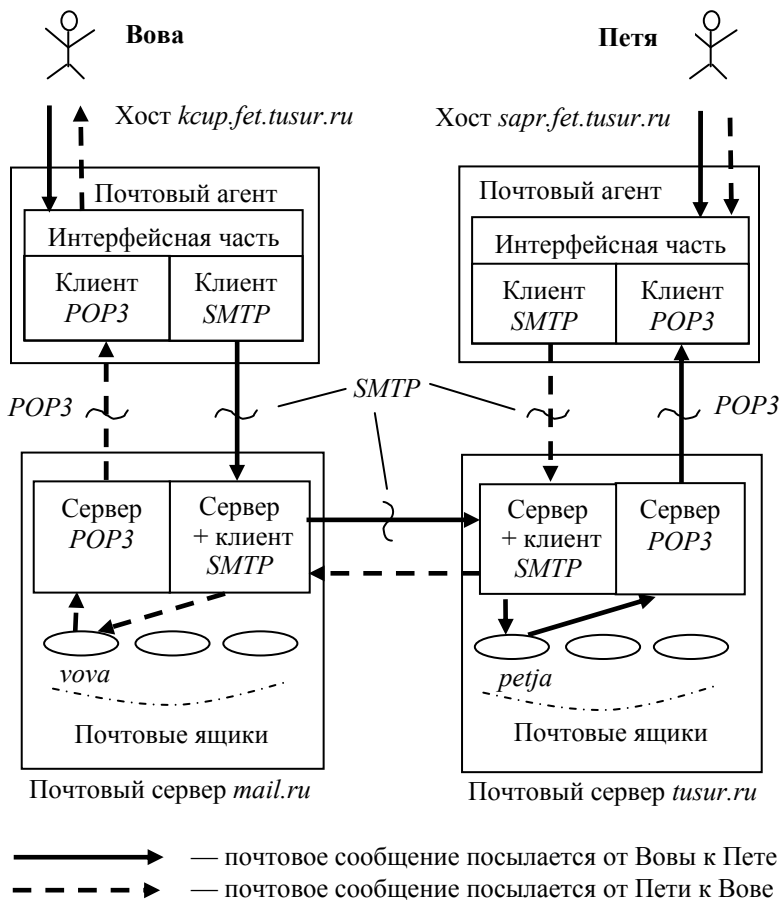


Рис. 66. Пересылка почтовых сообщений между хостами

Для чтения письма Петя запускает почтовый агент на своем хосте, а затем через интерфейсную часть он инициирует клиент *POP3*. *POP3* (*Post Office Protocol*) — почтовый протокол версии 3. В отличие от протокола *SMTP*, который является протоколом отправки сообщений (от клиента к серверу), протокол *POP3* является протоколом приема сообщений (от сервера к клиенту).

Получив запрос от *POP3*-клиента, *POP3*-сервер посылает ему требуемое письмо-сообщение, хранящееся в почтовом ящике клиента.

Следует отметить, что данный протокол позволяет пользователю не только читать сообщения, находящиеся в его почтовом ящике, но и удалять эти сообщения из почтового ящика. Сходные функции выполняет протокол ***IMAP4*** — *Internet Message Access Protocol* (протокол доступа к сообщениям в сети *Internet*). Кроме того, для получения сообщений из почтового ящика может использоваться протокол *HTTP*, основной областью применения которого являются *Web*-приложения (рассматриваются в п. 8.5.3).

Для реализации изложенной схемы работы электронной почты принципиально важное значение имеют протоколы отправки и приема сообщений *SMTP* и *POP3* (или *IMAP4*, *HTTP*), а также стандарт ***RFC 822***, задающий единый формат сообщения-письма. Применение для всей системы электронной почты стандартного формата сообщения-письма очень удобно, так как позволяет различным почтовым приложениям обрабатывать одно и то же электронное письмо, не преобразуя его. Рассмотрение перечисленных вопросов начнем со стандарта *RFC 822*.

Любое электронное письмо включает заголовок и тело письма, разделенные пустой строкой. И заголовок, и тело письма представляют собой текст, набираемый в семибитовом коде *ASCII*. Причем содержание тела письма полностью определяется его автором, а содержание заголовка должно отвечать следующим требованиям:

1) каждая строка заголовка состоит из ключевого слова с символом «:», за которым записаны один или несколько параметров;

2) все строки заголовка делятся на обязательные, необязательные и произвольные. При этом обязательные строки присутствуют в любом сообщении. Необязательные строки записываются в сообщение по необходимости, но если присутствуют, то интерпретируются стандартным образом. Произвольные строки заголовка имеют ключевые слова, не указанные в стандарте *RFC 822*.

Перечислим обязательные строки заголовка:

***From:*** почтовый\_ящик\_отправителя

***To:*** почтовый\_ящик\_получателя

Примеры необязательных строк:

***Subject:*** тема\_сообщения

***Date:*** дата\_время\_отправки\_сообщения

**MIME-Version:** версия\_стандарта\_MIME

**Content-Type:** тип\_данных/подтип\_данных

**Content-Transfer-Encoding:** метод\_кодирования

Последние три строки образуют *MIME*-заголовок, правило записи которого определяет стандарт *MIME* (*Multipurpose Internet Mail Extensions*) — стандарт многоцелевых расширений электронной почты в сети *Internet*. Данный стандарт предназначен для того, чтобы передавать в электронном письме не только текст в семиразрядном коде *ASCII*, но и любые другие данные. *MIME*-заголовок анализируется почтовым агентом на приемном конце с целью правильной перекодировки принятых данных из семиразрядного *ASCII*-кода в нужный код и для последующей интерпретации этого кода одним из локальных приложений.

Строка *MIME*-заголовка **Content-Transfer-Encoding** указывает на метод кодирования, который был использован для представления передаваемых данных в семиразрядном *ASCII*-коде. Например, двоичный файл, содержащий исполняемый код какой-то программы, для включения в тело сообщения должен быть закодирован в семиразрядном *ASCII*-коде. Один из универсальных методов кодирования заключается в использовании шестнадцатеричных цифр. Каждая такая цифра заменяет последовательность из четырех двоичных цифр, независимо от смысла передаваемой информации. Недостаток: из семи битов, отводимых для размещения *ASCII*-кода шестнадцатеричной цифры, фактически используются лишь четыре.

Строка *MIME*-заголовка **Content-Type** указывает, какой тип данных содержится в теле сообщения, а также уточняет подтип этих данных. Перечислим три из существующих семи типов данных:

1) **text** — сообщение содержит текст, например, документ *Microsoft Word*;

2) **image** — тело сообщения является изображением. Два подтипа: **gif** (*graphics interchange format* — формат для обмена графикой) и **jpeg** (*joint photography experts group* — формат, разработанный объединенной группой экспертов в области фотографии);

3) **multipart** — тело сообщения состоит из разнотипных данных. Например, подтип **mixed** указывает, что тело сообщения состоит из нескольких частей, разделяемых пустой строкой, а также символьной строкой, заданной в строке *Content-Type* после ключевого слова **Boundary**. В начале каждой части тела сообщения

после разделительной символьной строки записываются свои строки *Content-Type* и *Content-Transfer-Encoding*, задающие для своей части тела сообщения тип данных и используемый метод кодирования.

Вернемся к примеру, в котором Вова посылает электронное письмо Пете. Возможное содержимое письма:

*From: vova@mail.ru*

*To: petja@tusur.ru*

*Date: Wed, 30 Dec 06 11:18:25 EST*

*Subject: поздравление с Новым годом*

*MIME-Version: 1.0*

*Content-Type: multipart/mixed: Boundary=Start*

*--Start*

*Петя! Поздравляю тебя с Новым годом и высылаю тебе фотографию всей нашей семьи. Вова.*

*--Start*

*Content-Type: image/gif*

*Content-Transfer-Encoding: base64*

*... Изображение семьи Вовы в коде base64 ...*

Перейдем к рассмотрению почтового протокола *SMTP*. Данный протокол задает формат команд, передаваемых *SMTP*-клиентом *SMTP*-серверу, а также определяет формат ответных сообщений от *SMTP*-сервера к *SMTP*-клиенту. По своей форме все команды и ответы на них представляют собой символьные строки в коде *ASCII*. При этом команда клиента есть одно или два ключевых слова, последнее из которых часто заканчивается символом «:». Далее может быть записан параметр команды. Ответное сообщение *SMTP*-сервера всегда представляет собой трехзначное десятичное число, за которым записывается сообщение на английском языке, предназначенное не для *SMTP*-клиента, которому достаточно числа, а для пользователя, читающего запись беседы между *SMTP*-клиентом и *SMTP*-сервером. Любая команда или ответ всегда располагается на отдельной строке и поэтому заканчивается символами *cr* (возврат каретки) и *lf* (перевод строки). Рассмотрим основные команды и ответы на них.

Как всегда, инициатором сетевого взаимодействия является клиентская часть приложения. *SMTP*-клиент начинает с того, что посылает в тот хост, на котором находится требуемый *SMTP*-сервер, запрос на установление *TCP*-соединения, используя в качестве

номера *TCP*-порта число 25. Если *TCP*-соединение успешно установлено, *SMTP*-клиент ожидает поступления от *SMTP*-сервера ответа в виде числа 220, приход которого означает, что сервер готов принимать электронную почту.

Получив ответ 220, *SMTP*-клиент посылает команду *HELO* (сокращение от слова *hello*) с указанием в качестве параметра команды своего *DNS*-имени. Если сервер согласен продолжить диалог с клиентом, то он присылает ответ 250 с указанием своего *DNS*-имени. Получив этот ответ, *SMTP*-клиент посылает команду ***MAIL FROM:*** с заданием в качестве параметра почтового адреса отправителя (копируется из строки *From:* заголовка сообщения-письма). Приход от сервера сообщения 250 означает, что сервер подготовился к приему самого письма. Получив его, *SMTP*-клиент отправляет команду ***RCPT TO:***, задав в качестве ее параметра почтовый адрес получателя письма. Ответное сообщение сервера 250 означает, что *SMTP*-сервер или нашел искомый почтовый ящик у себя, или определил *IP*-адрес того почтового сервера, которому ящик принадлежит. Если от сервера придет ответ 550, то это означает, что почтовый адрес указан неверно.

Получив ответ 250, *SMTP*-клиент посылает без параметров команду ***DATA***, которая означает, что клиент готов отправить полный текст сообщения (заголовок плюс тело). В ответ сервер посылает сообщение 354, приведя в комментарии к нему ту строку символов, которая должна использоваться в качестве признака окончания почтового сообщения. Эта строка состоит из пяти символов: *cr*, *lf*, точка, *cr*, *lf*. Естественно, что внутри почтового сообщения не должно быть строки, содержащей всего один символ — точку.

Прежде чем выдать команду закрытия *TCP*-соединения ***QUIT***, *SMTP*-клиент проверит наличие в своей очереди хотя бы одного переданного письма этому же *SMTP*-серверу. В случае наличия такого письма оно также отправляется по ранее созданному *TCP*-соединению.

Вернемся к нашему примеру, в котором Вова посылает новогоднее поздравление Пете. После того как Вова запустил на своем хосте *kcup.fet.tusur.ru* почтового агента, он воспользовался услугами его интерфейсной части и ввел письмо для Пети, которого в этот день не было на работе, но который имеет обычай работать на хосте *sapr.fet.tusur.ru*. Получив письмо, интерфейсная часть передала его *SMTP*-клиенту, входящему в состав того же самого почтового агента. Далее *SMTP*-клиент устанавливает *TCP*-

соединение с тем *SMTP*-сервером, который имеет *DNS*-имя, указанное в адресе отправления письма, — *mail.ru*. Далее между *SMTP*-клиентом (К) и *SMTP*-сервером (С) происходит следующий диалог:

```
С: 220 mail.ru
К: HELO kcup.fet.tusur.ru
С: 250 Hello kcup.fet.tusur.ru
К: MAIL FROM: vova@mail.ru
С: 250 ok
К: RCPT TO: <petja@tusur.ru >
С: 250 ok
К: DATA
С: 354 Start mail input; end with <cr><lf>.<cr><lf>
К: .... заголовок сообщения
К: .... тело сообщения
К: .
С: 250 ok
К: QUIT
С: 221 mail.ru closing connection
```

Так как в данном примере почтовый ящик получателя *petja@tusur.ru* принадлежит не рассматриваемому почтовому серверу *mail.ru*, а другому почтовому серверу, то *SMTP*-сервер в *mail.ru* передает полученное письмо *SMTP*-клиенту, расположенному также в *mail.ru*, с целью передачи письма *SMTP*-серверу с адресом *tusur.ru*. *SMTP*-клиент начинает с того, что передает в *SMTP*-сервер запрос на установление *TCP*-соединения. Если *SMTP*-сервер не может установить *TCP*-соединение, то через некоторое время *SMTP*-клиент опять повторяет свою попытку. Подобное повторение будет периодически происходить и далее. Что касается диалога рассматриваемых *SMTP*-клиента и *SMTP*-сервера после установления *TCP*-соединения, то он очень похож на приведенный выше. В результате этого диалога *SMTP*-сервер с адресом *tusur.ru* поместит полученное письмо в ящик *petja@tusur.ru*.

Теперь вспомним, что Петя работает на другом хосте, имеющем адрес *sapr.fet.tusur.ru*. Для доступа к своему почтовому ящику Петя запускает на своем хосте почтовый агент и через его интерфейсную часть инициирует *POP3*-клиент. Далее этот программный модуль выполняет доступ к удаленному почтовому ящику, используя почтовый протокол *POP3*. Данный протокол начинает выполняться после того, как *POP3*-клиент установит

*TCP*-соединение с портом 110 *POP3*-сервера. Перечислим основные команды *POP3*-клиента:

1) **user** *имя\_пользователя*. Участвует в аутентификации пользователя, сообщая его имя;

2) **pass** *пароль*. Участвует в аутентификации пользователя, сообщая его пароль;

3) **list** — запрос на перечень почтовых сообщений, хранящихся в ящике. Для каждого сообщения указывается длина;

4) **retr** *имя\_сообщения* — запрос на получение из почтового ящика сообщения с заданным именем;

5) **dele** *имя\_сообщения* — удалить сообщение из почтового ящика;

6) **quit** — закончить работу. После получения команды *quit* *POP3*-сервер выполняет фактическое удаление сообщений, указанных ему *POP3*-клиентом.

Что касается ответных сообщений *POP3*-сервера, то таких сообщений всего два: **+OK**, за которым могут следовать запрошенные данные, а также **ERR** (в случае ошибки).

При реализации почтового агента вместо *POP3*-клиента может быть использован *HTTP*-клиент, основной функцией которого является прием не писем, а другой информации, используемой при воспроизведении *HTML*-документов.

### 8.5.3. WEB-приложения

*WEB*-приложения являются самыми распространенными сетевыми приложениями, выполняемыми в сети *Internet*. **WEB** (*World Wide Web* — всемирная паутина) — огромная распределенная система, включающая миллионы программных модулей — *WEB*-клиентов и *WEB*-серверов. *WEB*-клиент называется **браузером**. Наиболее часто в качестве программ браузеров используются *Netscape Navigator* и *Microsoft Internet Explorer*. Основной функцией браузера является вывод на экран *WEB*-страниц, выбранных пользователем. **WEB-страница** — документ типа гипермедиа (*hypermedia*). В слове **гипермедиа** приставка *hyper* означает, что документ, возможно, содержит указатели на другие гипермедиа-документы. Так как вывод на экран каждого такого документа определяется выбором пользователя, то подобные указатели, содержащиеся в гипермедиа-документе, называются **выбираемыми ссылками**. Корень *медиа* в слове гипермедиа означает, что



документ (*WEB*-страница) может содержать не только текст, но и другие данные, например графические изображения.

Основной функцией *WEB*-сервера является выдача по запросам браузеров содержимого *WEB*-страниц, находящихся в ведении данного *WEB*-сервера. Наиболее часто в качестве программ *WEB*-серверов используются *Apache* и *Microsoft Internet Information Server*. Каждая *WEB*-страница хранится на своем *WEB*-сервере в «разобранном» виде — в виде совокупности файлов, называемых далее ***WEB-объектами***. Один из этих *WEB*-объектов играет для конкретной *WEB*-страницы главную роль. Будучи по форме обычным текстовым *ASCII*-файлом, он содержит не только тот текст, который будет находиться на данной *WEB*-странице, но и требования, которые предъявляются к представлению этого текста на экране. Кроме того, данный текстовый файл содержит перечень нетекстовых *WEB*-объектов, входящих в состав данной страницы, а также перечень размещенных на ней выбираемых ссылок. Для записи этого базового текстового файла используется специальный язык ***HTML*** (*HyperText Markup Language* — язык гипертекстовой разметки). Поэтому данный *WEB*-объект, содержащий «план» *WEB*-страницы, принято называть ***HTML-документом***.

Так как *WEB* является частным случаем сетевого приложения, то укрупненные структуры его клиентской и серверной частей соответствуют рис. 64. При этом интерфейсная часть браузера запрашивает у протокольной части, а затем обрабатывает тот *HTML*-документ, который соответствует *WEB*-странице, указанной пользователем. Если в процессе обработки *HTML*-документа, в нем встретился указатель на какой-то *WEB*-объект, то интерфейсная часть запрашивает этот объект у протокольной части. После того как *WEB*-страница выведена на экран, пользователь может указать на ней с помощью мыши любую выбираемую ссылку. При этом интерфейсная часть опять обращается к протокольной части браузера с просьбой выдать тот *HTML*-документ, который соответствует выбранной ссылке.

В отличие от интерфейсной части протокольная часть браузера не обрабатывает *HTML*-документ, а рассматривает его как обычный файл, принимаемый, как и другие *WEB*-объекты, по сети от *WEB*-сервера. В процессе передачи *WEB*-объектов браузер и *WEB*-сервер «беседуют» друг с другом согласно прикладному протоколу ***HTTP*** (*HyperText Transfer Protocol* — протокол передачи гипертекста). Для того чтобы выполнять заявки браузеров на передачу им

*WEB*-объектов, протокольная часть *WEB*-сервера запрашивает требуемые текстовые и нетекстовые файлы у модуля управления данными, который входит в состав этого же *WEB*-сервера и может рассматриваться как надстройка над файловой подсистемой локальной ОС.

Прежде чем рассматривать стандарты *HTML* и *HTTP*, используемые в *WEB*-приложениях, рассмотрим сетевые имена *WEB*-объектов. Подобное имя используется пользователем при задании *HTML*-документа, соответствующего требуемой *WEB*-странице. Оно нужно интерфейсной части браузера при обращении к его протокольной части с целью получения нужного *WEB*-объекта, а далее используется при общении между протокольными частями браузера и сервера. В качестве такого универсального сетевого имени *WEB*-объекта используется **URL** (*Uniform Resource Locator* — универсальный указатель ресурса). Общая структура *URL*:

*прикл.\_протокол://адрес\_хоста:порт/имя\_путь\_файла*

где *прикл.\_протокол* — имя прикладного протокола, согласно которому взаимодействуют браузер и *WEB*-сервер при получении данного *WEB*-объекта; *адрес\_хоста* — *DNS*-имя хоста, на котором находится сервер, контролирующий требуемый *WEB*-объект; *порт* — номер порта, используемый для связи с сервером. При использовании порта, стандартного для данного прикладного протокола, данный параметр опускается; *имя\_путь\_файла* — локальное абсолютное имя файла, соответствующего запрашиваемому *WEB*-объекту, на хосте сервера.

Например, следующий *URL* указывает на *HTML*-документ, используемый для получения *WEB*-страницы кафедры КСУП:

*http://www.tusur.ru/kcup/10.html*

Данный *URL* сообщает, что для получения *WEB*-объекта в качестве прикладного протокола следует использовать *HTTP*. Сам *WEB*-объект находится на хосте, имеющем *DNS*-имя — *www.tusur.ru*. Локальное имя *WEB*-объекта на этом хосте — */kcup/10.html*. Перейдем теперь к рассмотрению стандарта (языка) *HTML*, используемого для описания структур *WEB*-страниц.

В соответствии с этим стандартом каждый *HTML*-документ содержит выводимый на экран текст в коде *ASCII*, а также записанные в коде *ASCII* команды, называемые **тегами**. С помощью тегов задается структура *WEB*-страницы, соответствующей данному *HTML*-документу, а также указываются требования к формату

выводимых данных. При этом следует заметить, что *HTML*-документ задает лишь основные параметры форматирования данных, оставляя за браузером известную свободу выбора. Поэтому два различных браузера представят на экране одну и ту же *WEB*-страницу по-разному.

В принципе, язык *HTML* можно рассматривать как язык программирования, предназначенный для записи виртуальных программ — *HTML*-документов. Эти виртуальные программы обрабатываются лингвистическими процессорами — браузерами (а точнее — их интерфейсными частями). Подобно любой исходной программе, в *HTML*-документ можно добавлять пробелы и даже целые пустые строки, не влияющие на результат представления соответствующей *WEB*-страницы на экране. Естественно, что здесь речь не идет о самом тексте, выводимом на экран, так как добавляемые в него пробелы выводятся в качестве обычных символов.

Что касается тегов, то каждый из них заключается между символами «<» и «>». Если тег задает требования к участку текста, то этому тегу, предворяющему участок текста, соответствует тег, завершающий этот участок. При этом завершающий тег отличается от предворяющего наличием в начале тега символа «/». Перечислим наиболее распространенные парные и одиночные теги:

- 1) **<HTML>** и **</HTML>** — обрамляют любой *HTML*-документ;
- 2) **<HEAD>** и **</HEAD>** — обрамляют *заголовок HTML*-документа. Этот заголовок не выводится пользователю, а используется лишь самим браузером;
- 3) **<TITLE>** и **</TITLE>** — обрамляют название *HTML*-документа, входящее в состав его заголовка;
- 4) **<BODY>** и **</BODY>** — обрамляют *тело HTML*-документа. Наряду с заголовком тело *HTML*-документа является обязательной его частью. В отличие от заголовка оно выводится на экран пользователю;
- 5) **<H<sub>*i*</sub>>** и **</H<sub>*i*</sub>>** — обрамляют заголовок раздела уровня *i*, где *i* = 1, 2, 3, 4, 5, 6 — номер уровня раздела. Чем меньше *i*, тем уровень раздела выше, а его заголовок крупнее;
- 6) **<UL>** и **</UL>** — обрамляют список, каждый элемент которого должен предваряться одиночным тегом **<LI>**. Браузер выводит каждый элемент списка на новой строке, предваряя его маркером абзаца — специальным символом, принятым в данном браузере;

7) **<BR>** — переход на новую строку. Следующий за данным тегом текст выводится на экран, начиная с новой строки;

8) **<IMG SRC="имя\_файла">** — *WEB*-страница содержит изображение, закодированное в заданном файле. Здесь использована сокращенная форма записи *URL*, содержащая только имя файла. Она допустима в том случае, если и *HTML*-документ, и изображение находятся на одном хосте;

9) **<A HREF="URL">** и **</A>** — обрамляют участок текста, выполняющий роль выбираемой ссылки на *WEB*-объект с заданным *URL*. В зависимости от браузера этот участок текста будет выделен на экране подчеркивом, цветом или тем и другим. Для отображения *WEB*-объекта на экране достаточно, чтобы пользователь щелкнул мышью на любом символе выделенного участка текста. Вместо текста теги **<A>** и **</A>** могут обрамлять тег, задающий изображение. В этом случае для выбора *WEB*-объекта пользователю достаточно щелкнуть мышью на любой точке изображения.

#### Пример:

Следующий *HTML*-документ предназначен для получения простой *WEB*-страницы «Кафедра КСУП»:

```
<HTML>
  <HEAD>
    <TITLE>
      Кафедра КСУП
    </TITLE>
  </HEAD>
  <BODY>
    <H1> Кафедра КСУП </H1>
    <H2> Список сотрудников </H2>
    <UL>
      <LI> Шурыгин Ю.А.
      <LI> Коцубинский В.П.
      <LI> Одинокое В.В.
      <LI> Звонков Д. А.
    </UL>
    <H2> Фотография сотрудников кафедры </H2>
    <IMG SRC="/kcup/photo_100.gif">
    <A HREF="http://www.tusur.ru/kcup/15.html">
      Методическая работа </A>
    <A HREF="http://www.tusur.ru/kcup/16.html">
      Личная жизнь сотрудников </A>
  </BODY>
```

</HTML>

В результате обработки данного *HTML*-документа браузером на экран будет выведено примерно то, что изображено на рис. 67.

*HTML*-документы, изображения и другие *WEB*-объекты рассматриваются протокольными частями браузера и *WEB*-сервера как обычные файлы, предназначенные для передачи по сети. В процессе этой передачи браузер и *WEB*-сервер взаимодействуют между собой согласно прикладному протоколу *HTTP*. Прежде чем начнет выполняться этот протокол, по инициативе браузера создается надежный транспортный канал — *TCP*-соединение с номером порта сервера 80.

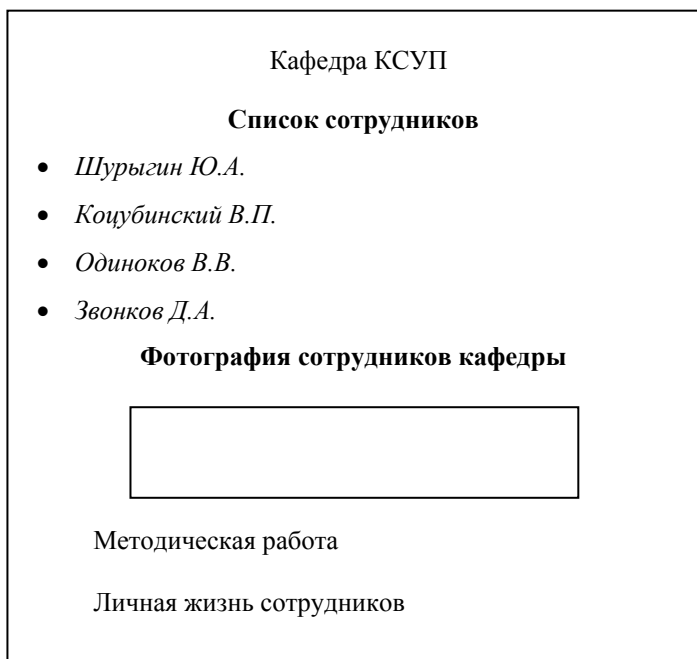


Рис. 67. Пример *WEB*-страницы

Существуют две основные версии протокола *HTTP* — 1.0 и 1.1. Основное различие между этими версиями заключается в том, что *HTTP/1.0* предполагает использование непостоянного *TCP*-соединения, предназначенного для передачи лишь одного *WEB*-объекта. После того как объект послан, передающая сторона (чаще это *WEB*-сервер) закрывает *TCP*-соединение. Например, в приведенном

выше примере для создания *WEB*-страницы браузер должен получить от *WEB*-сервера два *WEB*-объекта (*HTML*-документ и изображение). Для их получения между браузером и *WEB*-сервером создаются два *TCP*-соединения.

Применение протокола *HTTP/1.1* по умолчанию предполагает создание постоянного *TCP*-соединения, по которому передаются все запросы браузера по созданию текущей *WEB*-страницы, а также ответные сообщения *WEB*-сервера на эти запросы. Закрытие постоянного *TCP*-соединения происходит в том случае, если оно не используется в течение некоторого времени. По сравнению с непостоянным соединением применение постоянного *TCP*-соединения позволяет сократить затраты времени на открытие *TCP*-соединений, сократить затраты ОП на хостах браузера и сервера, так как каждому *TCP*-соединению нужны свои буферы, а также позволяет уменьшить нагрузку на сеть за счет сокращения передаваемых служебных сообщений. Следует заметить, что протокол *HTTP/1.1* может использовать и непостоянные *TCP*-соединения, позволяя тем самым успешно взаимодействовать браузеру и *WEB*-серверу, использующим разные версии протокола *HTTP*.

Независимо от версии *HTTP*, каждое передаваемое сообщение состоит из обязательного заголовка, содержащего служебную информацию, и необязательного тела сообщения, содержащего передаваемые данные. Независимо от наличия тела сообщения, после заголовка обязательно размещается пустая строка (аналогично почтовым сообщениям). Что касается заголовков сообщений, то в запросах браузеров и в ответах *WEB*-серверов они разные.

Первая строка заголовка в запросе браузера называется *строкой запроса* и содержит тип запроса, называемый *HTTP-методом*, а также два его параметра: 1) *URL* того *WEB*-объекта, к которому относится запрос; 2) версия протокола *HTTP*. Перечислим *HTTP*-методы:

- 1) ***GET*** — просьба прислать *WEB*-объект с заданным *URL*;
- 2) ***POST*** — просьба к *WEB*-серверу принять данные от браузера. Передача данных от браузера к *WEB*-серверу требуется, например, при заполнении пользователем браузера каких-то форм. Это могут быть, например, ключевые слова для поиска информации, требуемой пользователю, или реквизиты платежа, сделанного пользователем. Заполненные поля форм отсылаются *WEB*-серверу в виде тела сообщения браузера;

3) **HEAD** — отличается от *GET* тем, что ответное сообщение *WEB*-сервера содержит лишь заголовок, но не содержит самого *WEB*-объекта (нет тела сообщения). Данный метод обычно используется при отладке;

4) **PUT** — просьба к *WEB*-серверу поместить в свою базу данных *WEB*-объект с заданным *URL*. При этом сам *WEB*-объект находится в теле сообщения браузера;

5) **DELETE** — удалить заданный *WEB*-объект.

Первые три метода имеются во всех версиях протокола *HTTP*, а два последних отсутствуют в версии 1.0. Каждая строка заголовка запроса браузера, начиная со второй, содержит ключевое слово с двоеточием, а также один параметр. В качестве примера приведем запрос браузера на получение рассмотренного ранее *HTML*-документа:

```
GET /kcup/10.html HTTP/1.1
Host: www.tusur.ru
Connection: close
User-agent: Mozilla/4.0
Accept-language: ru
```

Из строки запроса (первой строки заголовка) видно, что браузер использует метод *GET* для получения *WEB*-объекта с локальным именем (на хосте сервера) — */kcup/10.html*. Причем браузер готов обеспечить выполнение протокола *HTTP/1.1*. Вторая строка заголовка сообщает, что *WEB*-сервер находится на хосте *www.tusur.ru*, а третья строка заголовка информирует сервер о том, что используется непостоянное *TCP*-соединение, которое должно быть закрыто сразу после передачи *WEB*-объекта. Четвертая и пятая строки указывают, соответственно, на тип программы браузера и язык, на котором, желательно, должен быть записан передаваемый текст. Задание типа программы браузера позволяет *WEB*-серверу выбрать из нескольких альтернативных *WEB*-объектов тот, который наиболее подходит для данного браузера.

Перейдем к рассмотрению ответного сообщения *WEB*-сервера. Такое сообщение никак не зависит от предыдущих запросов браузера и ответов на них сервера. Поэтому говорят, что *WEB*-сервер работает без запоминания состояния. Например, если браузер сделает сто запросов на один и тот же *WEB*-объект, то *WEB*-сервер сто раз пошлет ему один и тот же ответ.

Подобно запросу браузера, ответ *WEB*-сервера также состоит из заголовка и тела сообщения, разделенных пустой строкой.

Первая строка заголовка ответа называется *строкой состояния*. Она содержит: 1) версию протокола; 2) код состояния; 3) словесное дополнение к коду состояния. Код состояния представляет собой трехзначное десятичное число. Подобно ранее рассмотренным приложениям *FTP* и *SMTP*, это трехзначное число и содержит основной ответ сервера. Что касается словесного дополнения к коду состояния, то оно нужно не браузеру, а человеку, который хочет разобраться в ответе *WEB*-сервера. Приведем некоторые комбинации кодов состояния и дополнения к нему:

1) 200 *OK* — запрос успешно выполнен;

2) 404 *Not Found* — запрашиваемый *WEB*-объект не найден на сервере;

3) 505 *HTTP Version Not Supported* — запрашиваемая браузером версия *HTTP* не поддерживается сервером.

Каждая строка заголовка ответа *WEB*-сервера, начиная со второй, содержит ключевое слово с двоеточием, а также один или несколько параметров. В качестве примера приведем возможный ответ *WEB*-сервера на получение рассмотренного ранее запроса браузера:

*HTTP/1.1 200 OK*

*Connection: close*

*Date: Sat. 17 Apr 2005 14:15:27 GMT*

*Server: Apache/1.3.0 (Unix)*

*Last-Modified: Thu. 07 Feb 2005 10:21:12 GMT*

*Content-Length: 238*

*Content-Type: text/html*

*...Тело ответа...*

Данный ответ означает, что требуемый *WEB*-объект найден и пересылается браузеру (в качестве тела сообщения). Вторая строка заголовка сообщает, что *TCP*-соединение будет закрыто сразу же после пересылки объекта. Третья строка содержит дату и время по Гринвичу создания ответа. Следующая строка содержит тип *WEB*-сервера. Строка *Last-Modified*: указывает дату и время создания или последней модификации. Строка *Content-Length*: содержит длину пересылаемого объекта в байтах. Последняя строка сообщает о том, что пересылаемый *WEB*-объект представляет собой текст на языке *HTML*.

Для того чтобы повысить скорость обслуживания и одновременно уменьшить нагрузку на сеть, создаваемую *WEB*-приложе-



ниями, широко используется кэширование *WEB*-объектов. Кэширование производится и самими браузерами, и специально предназначенными для этого **прокси-серверами**. При использовании прокси-сервера браузер специально настраивается так, что запрос на получение *WEB*-объекта, содержащий метод *GET*, направляется браузером не на *WEB*-сервер, а на близко расположенный прокси-сервер. При наличии в кэш-памяти прокси-сервера требуемого *WEB*-объекта этот объект пересылается браузеру. Иначе прокси-сервер посылает запрос в *WEB*-сервер. При получении *WEB*-объекта он посылается браузеру, а также помещается в кэш-память прокси-сервера.

Независимо от того, где произведено кэширование *WEB*-объекта, весьма актуален вопрос о пригодности хранящейся копии *WEB*-объекта. Дело в том, что *WEB*-сервер мог внести изменения в *WEB*-объект, что сразу делает ранее сделанные копии непригодными для использования. Решение данного вопроса состоит в том, что прежде чем использовать копию *WEB*-объекта, браузер или прокси-сервер посылает в *WEB*-сервер запрос с методом *GET*, содержащий в качестве одной из строк заголовка следующую:

*If-modified-since: Thu. 07 Feb 2005 10:21:12 GMT*

При получении запроса с такой строкой *WEB*-сервер посылает *WEB*-объект в том случае, если он модифицировался позже момента времени, указанного в данной строке. Иначе отсылается ответ с кодом состояния 304 и с пустым телом сообщения.

## 9. ЛАБОРАТОРНЫЙ КУРС

Данный лабораторный курс предназначен для выполнения в среде реальной *UNIX*, установленной в учебном классе. При этом локальная сеть, объединяющая компьютеры класса, может находиться под управлением любой сетевой ОС, поддерживающей протоколы связи с *UNIX*-системой, выполняемой на одном из компьютеров класса или вне него. При отсутствии такого класса допускается выполнение лабораторных работ в среде локальных *UNIX*-систем.

По тематике данные лабораторные работы можно разбить на две группы, различные между собой по решаемым задачам. В первой группе (лабораторные работы № 1–4) решается задача обучения студентов навыкам использования командного языка *UNIX*, то есть языка *shell*. Во второй группе (лабораторные работы № 5–12) решается задача изучения системных вызовов *UNIX*, а также использования этих вызовов в своих программах. При этом заметим, что, в отличие от теоретической части данного пособия, в данных работах используются не упрощенные формы записи системных вызовов, а реальные формы, используемые при написании программ на языке СИ.

### 9.1. Лабораторная работа № 1.

#### Первоначальное знакомство с *UNIX*

Целью выполнения настоящей лабораторной работы является получение начальных навыков работы в среде *UNIX*: 1) вход в систему; 2) знакомство с текстовым редактором *ed*; 3) применение команд *shell* для работы с файлами; 4) работа в среде *Midnight Commander*.

##### 9.1.1. Подготовка к выполнению работы

В начале выполнения данной работы обязательно следует изучить следующие вопросы из теоретической части пособия:

1) управление доступом пользователя в систему (подразд. 3.1) — понятия имени пользователя и пароля; псевдотерминал;

2) файлы с точки зрения пользователя (подразд. 1.2) — понятие файла; простое имя файла; каталоги; файловая структура

системы; текущий каталог; относительное и абсолютное имена файла;

3) простые команды *shell* (утилиты) для работы с файловой структурой (подразд. 1.3) — *pwd*; *ls*; *cd*; *mkdir*; *rmdir*; *rm* с флагом *-r*; *cp* с флагом *-r*;

4) простые команды *shell* (утилиты) для работы с файлами (подразд. 1.3) — *cat*; *cp*; *mv*; *rm*;

5) упрощение пользовательского интерфейса (подразд. 1.3);

6) справочная команда *man* (подразд. 1.3).

В процессе изучения перечисленных вопросов обязательно следует изучить примеры, приведенные в пособии. Для выполнения заданий в некоторых из этих примеров требуется предварительное создание вспомогательных текстовых файлов и (или) каталогов. Для этого можно использовать команды *cat* и *mkdir*.

### 9.1.2. Вход в систему и выход из нее

При работе с многопользовательской системой, каковой является *UNIX*, каждый пользователь имеет свой уникальный идентификатор — имя пользователя. Кроме того, для избежания использования имени пользователя другим лицом каждый пользователь может иметь пароль пользователя.

Имя пользователя — символьная строка. Например, оно может включать (одновременно) вашу фамилию, имя и отчество. Другой вариант — имя содержит номер студенческой группы и инициалы пользователя. В любом случае имя пользователя должно быть согласовано с администратором системы, который регистрирует его в «журнале» системы.

Работа с ВС, на которой установлена *UNIX*, начинается с включения терминала. В качестве терминала может использоваться совокупность экрана и клавиатуры или даже целая периферийная ЭВМ. В любом случае на экране появляется сообщение *UNIX – login*, в ответ на которое следует ввести имя пользователя. Пример ввода имени, зарегистрированного в системе:

```
UNIX → login:
польз. → vlad <Enter>
UNIX → $
```

Символ *\$* есть приглашение *UNIX* к вводу команды. Его появление на экране говорит об успешном входе в систему. Пример ввода имени, не зарегистрированного в системе:

```
UNIX → login:
```

польз. → *vlid* <Enter>

UNIX → *passwd*:

Слово *passwd* означает просьбу ввести пароль. Может показаться странным, что в ответ на неправильное имя пользователя UNIX выводит такую просьбу. Дело в том, что таким образом система скрывает от лица, использующего неверное имя, сам факт такого использования. Поэтому вместо того чтобы угадывать правильное имя, данное лицо будет заниматься угадыванием пароля, что заведомо безнадежно, так как ввод любого пароля приведет к повторению вывода на экран сообщения «*login*».

Применение пароля — дело добровольное. Но если вы хотите, чтобы никто (или почти никто) в будущем не разрушил результат вашей работы или не воспользовался бы им без вашего разрешения, без пароля не обойтись. Пароль пользователя регистрируется в системе не администратором, а самим пользователем с помощью команды *passwd*. Пример:

UNIX → \$

польз. → *passwd* <Enter>

UNIX → *Changing password for vlad*

*New password:*

польз. → *dracon* <Enter>

UNIX → *Retype new password:*

польз. → *dracon* <Enter>

UNIX → \$

Набор пароля производится пользователем «вслепую» — без отображения набираемых на клавиатуре символов на экране. Это позволяет избежать «подсматривания» пароля.

Требования к паролю: 1) если для написания пароля используются обычные алфавитно-цифровые символы, его длина должна быть не менее 6 символов (в примере пароль пользователя *dracon* отвечает этому требованию); 2) при использовании специальных и управляющих символов допускается меньшая длина пароля.

Если пароль отвечает хотя бы одному из приведенных двух требований, UNIX выводит просьбу повторить ввод нового пароля (с целью избежания ошибки при наборе пароля). Иначе UNIX выведет сообщение «*Please use a longer password*» («Пожалуйста, введите длинный пароль»).

После того как пароль зарегистрирован, он будет запрашиваться всякий раз, когда вы будете входить в систему. Пример такого входа:

UNIX → *login*:

```
польз. → vlad <Enter>
UNIX → passwd:
польз. → dracon <Enter>
UNIX → $
```

Если вы забудете пароль, то не сможете войти в систему без помощи администратора. Он удалит прежний пароль, а затем вы зарегистрируете в системе новый пароль. Замену пароля может выполнить и сам пользователь при условии, что прежний пароль им не забыт. Для этого вновь используется команда *passwd*.

**Пример:**

```
UNIX → $
польз. → passwd <Enter>
UNIX → Changing password for vlad
      Old password:
польз. → dracon <Enter>
UNIX → New password:
польз. → dracon7 <Enter>
UNIX → Retype new password:
польз. → dracon7 <Enter>
UNIX → $
```

**З а р е г и с т р и р у й т е** свое имя пользователя у администратора *UNIX*-системы. Выполните вход в систему и установку пароля.

*Примечание.* Допустим, что после включения терминала вы оказываетесь не в среде *UNIX*, а в среде другой операционной системы, позволяющей связываться с *UNIX*. Тогда прежде чем ввести команды по входу в *UNIX*, необходимо войти в исходную операционную систему, а затем ввести ее команду по запуску «терминала» в среде *UNIX*. При этом под «терминалом» понимается не аппаратное устройство, а псевдотерминал — программный процесс, связанный с *UNIX*.

Например, если ваш терминал находится в локальной сети, управляемой операционной системой *Novell Net Ware*, то вы сначала входите в эту систему под своим логическим именем, которое зарегистрировано в этой локальной сети. После этого следует набрать команду *Kermit*, которая осуществляет запуск программного процесса, имитирующего терминал *UNIX*. При этом в ответ на запрос этой программы следует ввести идентификатор используемой *UNIX*-системы. Далее *Kermit* обратится к *UNIX* так, как будто речь идет о включении реального терминала. В ответ *UNIX*

передаст в *Kermit* версию *UNIX*-системы и номер «терминала» *UNIX*. Далее *Kermit* выводит эти данные на экран.

Ситуация усложняется тем, что сетевая операционная система *Novell Net Ware* представляет собой надстройку над другой системой, например над *MS-DOS*. В этом случае подготовка к запуску *UNIX* может выглядеть следующим образом:

```
MS-DOS → n:\
польз.  → Kermit
Kermit  → [n:]JMS-Kermit>
польз.  → telnet dark.tusur.ru
UNIX    → FreeBSD/:386(dark.tusur.ru)(ftyp0)
UNIX    → login:
```

Вопрос о взаимодействии различных операционных систем весьма интересен, но сейчас нас интересуют лишь его технические аспекты.

**Выход из системы.** При завершении сеанса работы необходимо сообщить об этом *UNIX* командой `<Ctrl>&<D>` (одновременное нажатие клавиш `<Ctrl>` и `<D>`).

**Пример:**

```
UNIX → $
польз. → <Ctrl>&<D>
UNIX → login:
```

Другой способ выхода из *UNIX* — использование команды *exit*.

**В ы й д и т е** из системы, а затем опять войдите в нее.

### 9.1.3. Текстовый редактор *ed*

Редактор *ed* ориентирован на построчное редактирование текстовой информации. При этом под строкой понимается текст, введенный до нажатия клавиши `<Enter>`.

Запуск редактора выполняется командой *ed* с параметром, в качестве которого выступает имя редактируемого текстового файла. Так как *ed* — весьма старый редактор, то простое имя файла не должно иметь длину более 14 символов. При этом, как и в любой команде *UNIX*, команда *ed* отделяется от параметра одним или несколькими пробелами. Пример:

```
UNIX → $
польз. → ed letter<Enter>
ed    → ?letter
```

В данном примере файл *letter* новый, и в нем еще ничего нет. Поэтому в ответном сообщении редактора присутствует символ «?». Если бы мы задали имя уже существующего файла, то *ed* сообщил бы длину редактируемого файла в байтах. При этом последняя строка могла бы выглядеть, например, так:

*ed* → 1234

Допустим пока, что мы создали новый (пустой) файл. Для того чтобы поместить в него текст, необходимо перейти в режим добавления. Это делается командой *a* редактора.

**Пример:**

польз. → *a*<Enter>

*ed* → *ответа нет; переход на новую строку*

польз. → *This is a test to see if I am*<Enter>  
*entering text in the file "letter".*<Enter>  
*Once I have completed it I shall find*<Enter>  
*that I have created 4 new lines of data*<Enter>  
*.* <Enter>

Обратите внимание на точку в последней строке. Она представляет собой команду выхода из режима добавления. Другого способа выхода из этого режима нет, так как только одиночная точка в строке будет воспринята редактором как команда, а не как часть вводимого текста.

После того как текст файла набран, его можно выводить на экран и редактировать. Для вывода файла используется команда *p* редактора. Вывод файла можно выполнять построчно или в виде группы строк. Для построчного вывода редактору передается номер первой требуемой строки, в ответ на что *ed* выводит текст этой строки на экран. Далее вывод каждой следующей строки файла предваряется нажатием клавиши <Enter>.

**Пример:**

польз. → *1p*<Enter>

*ed* → *This is a test to see if I am*

польз. → <Enter>

*ed* → *entering text in the file "letter".*

польз. → <Enter>

*ed* → *Once I have completed it I shall find*

польз. → <Enter>

*ed* → *that I have created 4 new lines of data*

польз. → <Enter>

*ed* → ?

Символ «?» означает, что текст файла закончился.

Для вывода на экран группы строк в редактор передаются одновременно номер первой и номер последней выводимой строки файла, разделенные запятой. При этом для обозначения последней строки файла можно использовать символ *\$*.

**Пример:**

польз. → 1,\$p<Enter>  
ed → *This is a test to see if I am  
entering text in the file "letter".  
Once I have completed it I shall find  
that I have created 4 new lines of data*

Для выполнения редактирования файла, во-первых, устанавливается требуемая *текущая строка*, а во-вторых, выполняется требуемая операция редактирования. Для установки требуемой текущей строки можно использовать построчный вывод текста. Получив на экране требуемую строку, следует ввести одну из команд редактирования: *a* — добавление текста после текущей строки; *i* — добавление текста перед текущей строкой; *c* — замена текущей строки текстом новой строки (строк); *d* — удаление текущей строки. Пример:

польз. → \$p<Enter>  
ed → *that I have created 4 new lines of data*  
польз. → a<Enter>  
*I will now enter two new lines of<Enter>  
text to see if it is accepted.<Enter>  
.<Enter>*

Команда редактора *\$p* требует от него вывода на экран последней строки файла. Поэтому две строки, набранные в примере пользователем, добавляются в конец файла.

**Пример** добавления текста перед текущей строкой:

польз. → 3p<Enter>  
ed → *Once I have completed it I shall find*  
польз. → i<Enter>  
*I am now inserting two lines of<Enter>  
text to demonstrate how it works.<Enter>  
.<Enter>*

**Пример** замещения текста текущей строки текстом другой строки:

польз. → 3p<Enter>  
ed → *Once I have completed it I shall find*  
польз. → c<Enter>  
*After completion, I shall find.<Enter>*



.<Enter>

В данном примере текущая строка заменена текстом одной строки. На самом деле число новых строк может быть любым.

В следующем **примере** производится удаление двух строк:

польз. → 3p<Enter>

ed → I am now inserting two lines of

польз. → d<Enter>

польз. → d<Enter>

Одной командой можно удалить целую группу строк.

В следующем **примере** производится удаление всего содержимого файла:

польз. → 1,\$d<Enter>

польз. → 1,\$p<Enter>

ed → ?

Если текст файла достаточно большой, рекомендуется выполнять промежуточное запоминание файла на диске, используя команду *w*. Выполнив ее, редактор выведет на экран новую длину файла.

Выход из редактора *ed* в *UNIX* выполняет команда редактора — *q*.

**Пример:**

польз. → q<Enter>

UNIX → \$

**В ы п о л н и т е** создание нового текстового файла, а затем выйдите из редактора в *UNIX*. Далее скорректируйте этот файл, добавив новые строки в конец, в начало и в середину файла. Затем удалите некоторые строки и проверьте результаты редактирования с помощью вывода файла на экран.

#### 9.1.4. Работа в среде *Midnight Commander*

**1. Представление на экране файловой структуры.** Утилита *Midnight Commander* предназначена для того, чтобы предоставить пользователю *UNIX* удобный интерфейс для общения с этой системой. Это обеспечивается, во-первых, наглядным выводом на экран информации о файловой структуре системы. Во-вторых, *Midnight Commander* существенно упрощает для пользователя ввод команд *UNIX*.

Для того чтобы запустить *Midnight Commander*, достаточно набрать команду *UNIX* — *mc*. Часто эту команду включают в ко-

мандный файл, выполняемый после загрузки системы, и поэтому она выполняется автоматически.

В любом случае в верхней части экрана появляются две серые панели, каждая из которых содержит перечень файлов, «зарегистрированных» в одном из каталогов файловой структуры системы. При этом в заголовке каждой панели указано имя каталога. Ниже располагается командная строка *UNIX* с обычным ее приглашением и мерцающим курсором. В последней строке экрана находится список клавиш *<F1>* – *<F10>* с кратким обозначением их функций.

Каталоги, отображаемые на левой и правой панелях, могут совпадать или не совпадать, но в любом случае одна из панелей, называемая активной, отображает текущий каталог. Заголовок активной панели выделяется белым цветом. Кроме того, одно из имен файлов на активной панели выделено псевдокурсором. В отличие от обычного курсора (он находится в командной строке *UNIX*), *псевдокурсор* генерируется не аппаратурой, а программой (в графическом режиме экрана). Переключение активной панели производится нажатием клавиши *<Tab>*.

Перемещение псевдокурсора внутри активной панели производится с помощью клавиш управления курсором — вниз, вверх, влево, вправо. Нажатие клавиши *<End>* приводит к установке псевдокурсора на последнюю, а *<Home>* — на первую строку панели. Щелчком левой клавиши мыши можно установить псевдокурсор в любую позицию не только активной, но и соседней панели (происходит смена активной панели).

В общем случае панель содержит строки трех типов:

- 1) строку «...», обозначающую выход в «родительский каталог» данного каталога;
- 2) строки с именами подкаталогов данного каталога;
- 3) строки с именами файлов данного каталога.

В последней строке панели, как правило, записано имя выделенного файла, его длина, дата и время создания или последней модификации.

Для того чтобы перейти на подкаталог текущего каталога, достаточно установить на него псевдокурсор и нажать *<Enter>*. Для возврата в родительский каталог требуется установить псевдокурсор на «...» и нажать *<Enter>*. Перемещаясь подобным образом вверх-вниз по файловой структуре логического диска, можно сделать текущим любой каталог на нем.

Прекращение работы *Midnight Commander* происходит в результате нажатия клавиши <F10>. В ответ на появившийся затем вопрос о намерении прекратить работу следует нажать <Enter>.

**2. Команды для работы с файлами.** *Midnight Commander* предоставляет пользователю команды для работы с файлами, намного более удобные, чем соответствующие команды *UNIX*. Рассмотрим их.

**Создание каталога.** Для этого достаточно установить в заголовке активной панели «родительский» каталог по отношению к вновь создаваемому каталогу, а затем нажать клавишу <F7>. На экране появится диалоговое окно с приглашением набрать имя нового каталога. В ответ следует набрать имя каталога прописными или строчными буквами и нажать клавишу <Enter>. В результате на активной панели появится имя нового каталога.

**Копирование файла.** Для этого требуется установить в заголовке одной из панелей «родительский» каталог по отношению к копируемому файлу, а в заголовке другой панели — «родительский» каталог по отношению к файлу-копии. Далее следует установить псевдокурсор на копируемый файл и нажать клавишу <F5>. На экране появится диалоговое окно с сообщением о готовности выполнить копирование. В ответ достаточно нажать клавишу <Enter>. (Для отмены этой или другой команды следует нажать <Esc>.) В результате на второй панели появится имя скопированного файла.

Для того чтобы создать копию файла в том же каталоге, что и исходный файл, необходимо обеспечить, чтобы старый и новый файл имели разные имена. Для этого следует в диалоговом окне, появившемся после нажатия <F5>, набрать имя файла-копии, а уж затем нажать <Enter>.

Копирование каталога выполняется аналогично копированию обычного файла данных. При этом копируется все поддерево файловой структуры, начинающееся с данного каталога.

Копирование нескольких «дочерних» файлов (каталогов) текущего каталога можно ускорить, используя выделение группы файлов. Для выделения файла необходимо установить на его имя псевдокурсор и нажать клавишу <Ins>. В результате имя файла высвечивается белым цветом и оно включается в группу. (Для исключения файла из группы необходимо повторить эти же два действия — установку псевдокурсора и нажатие <Ins>.) После того как требуемая группа файлов выделена (в нижней строке панели

информация об общем числе выделенных файлов и их общем объеме), ее можно скопировать последовательным нажатием <F5> и <Enter>.

**Уничтожение файла.** Для этого требуется установить псевдокурсor на имя уничтожаемого файла и нажать клавишу <F8>. На экране появится диалоговое окно с просьбой подтвердить намерение удалить файл. В ответ достаточно нажать <Enter> (для отмены — <Esc>).

Выделив группу файлов (аналогично выделению при копировании), ее можно уничтожить нажатием <F8> и последующими нажатиями <Enter> в ответ на вопросы *Midnight Commander*.

**Переименование файла.** Для этого следует установить псевдокурсor на требуемый файл и нажать клавишу <F6>. В появившемся диалоговом окне следует набрать новое имя файла, а затем нажать <Enter>.

**Создание текстового файла.** Для создания нового файла необходимо запустить с командной строки один из текстовых редакторов, например *ed* или *vi* (этот редактор встроен в *Midnight Commander*).

**Редактирование текстового файла.** Для работы с ранее созданным текстовым файлом с помощью редактора *vi* необходимо установить псевдокурсor в каталоге файлов на нужный файл и нажать клавишу <F4>.

*Примечание.* Нажатие функциональной клавиши <Fi> можно заменить последовательным нажатием двух клавиш — <Esc> и <i>, где *i* — номер функциональной клавиши.

**3. Ввод команд UNIX.** В отличие от рассмотренных выше операций с файлами, для выполнения которых *Midnight Commander* предоставляет пользователю свои команды, формат остальных команд UNIX остается без изменений. При этом помощь *Midnight Commander* заключается в ускорении набора этих команд.

Крайний случай — пользователь набирает команду в командной строке UNIX полностью вручную, не пользуясь помощью *Commander*. Такой метод используется для ввода коротких или редко используемых команд.

Другой метод позволяет обойтись вообще без записи в командную строку. Установив псевдокурсor на имени исполняемого файла (такое имя начинается с символа «\*»), следует нажать <Enter>. Данный метод обычно применяется тогда, когда команда не имеет параметров.

Третий метод заключается в том, что команда *UNIX* переписывается в командную строку из протокола команд. Протокол — список последних команд (не более 16), сохраненный в *Commander*. Для вывода на экран протокола команд достаточно нажать последовательность клавиш

**<F9> → Command → Command History → Ok**

Установив псевдокурсор на требуемую команду в протоколе, следует нажать **<Enter>**.

Если программа, запускаемая любым способом из *Commander*, выводит какие-то данные на экран, то чтение их будет невозможно из-за присутствия на экране панелей. Для того чтобы убрать с экрана обе панели, требуется одновременно нажать **<Ctrl>&<O>**. Для восстановления панелей достаточно опять нажать **<Ctrl>&<O>**.

Удаление (восстановление) только левой панели производится нажатием последовательности клавиш

**<F9> → Left → Listing mode → Ok**

Удаление (восстановление) только правой панели:

**<F9> → Right → Listing mode → Ok**

**4. Настройка *Midnight Commander*.** *Commander* предоставляет своим пользователям возможность выполнять настройку формата информации, выводимой на экран.

В результате нажатия клавиши **<F9>** в верхней части экрана появляется главное (горизонтальное) меню из пяти пунктов: **Left** (левая), **Files** (файлы), **Commands** (команды), **Options** (параметры), **Right** (правая). Одна из позиций меню выделена псевдокурсором. Для выбора требуемого пункта меню достаточно установить на него псевдокурсор (с помощью клавиш управления курсором) и нажать **<Enter>**. Это же можно сделать щелчком левой клавиши мыши. В результате подобного выбора на экране появится ниспадающее меню, выбор в котором позволит выполнить требуемое действие.

Пункт горизонтального меню **Files** позволяет с помощью своих ниспадающих меню выдавать команды по работе с файлами. Многие из этих команд могут быть введены другим способом — нажатием клавиш **<F1>** **<F10>**. Примеры других команд: **Chmod** — установка прав доступа к файлу; **Chown** — замена владельца файла.

В пункте **Command** находится большое количество вспомогательных команд *MC*. Примеры таких команд: **Find File** — поиск

файла; *Swap panels* — переключение панелей; *Show directories size* — показ размера каталогов.

Пункты горизонтального меню *Left* и *Right* предназначены для настройки левой и правой панелей соответственно. При этом, установив в ниспадающем меню режим *Listing mode...*, режим *Brief* (краткий), мы обеспечим вывод на соответствующей панели лишь одних имен файлов. Задание режима *Full* (полный) позволяет выводить на экран не только имена файлов, но и их основные характеристики (размер в байтах, дату и время создания или последней модификации). *Name, Extension, Time, Size* — группа полей в ниспадающем меню, позволяющая выполнить сортировку имен файлов, выводимых на панель, соответственно, по имени, расширению имени, времени создания или модификации, размеру файла. Для интеграции в сетевое пространство предусмотрены пункты *Network link* и *FTP link*.

Пункт горизонтального меню *Options* позволяет настраивать интерфейс *MC*. Например, в ниспадающем меню можно установить режимы: *Layout* — задание информации, выводимой вне панелей; *Learn Keys* — задание функциональных клавиш, используемых для работы с *MC*; *Virtual FS* — задание параметров для установления связи. В этом же ниспадающем меню можно выбрать пункт *Configuration* (конфигурация). На экране появится диалоговое окно, в котором можно установить дополнительные параметры настройки *Commander*.

### 9.1.5. Задание

Для успешного выполнения лабораторной работы требуется выполнить наизусть следующие операции:

- 1) войти в *UNIX* с паролем;
- 2) создать трехуровневое поддерево каталогов;
- 3) с помощью *ed* создать в одном из новых каталогов текстовый файл;
- 4) вывести файл на экран;
- 5) выполнить добавление текста в начало, в середину и в конец файла;
- 6) вывести файл на экран;
- 7) произвести переименование файла;
- 8) выполнить копирование файла (исходный файл и файл-копия должны располагаться в разных каталогах);
- 9) удалить созданные файлы и каталоги;

- 10) выполнить операции 2–9 с помощью *Midnight Commander*;
- 11) выйти из *UNIX*.

## 9.2. Лабораторная работа № 2.

### Дальнейшее знакомство с командами *UNIX*

Целью выполнения настоящей лабораторной работы является развитие навыков работы в среде *UNIX*: 1) использование в командах *shell* метасимволов и перенаправление ввода-вывода; 2) запуск конвейеров программ; 3) применение в командах *shell* переменных; 4) построение командных файлов; 5) изменение прав доступа к файлам.

#### 9.2.1. Подготовка к выполнению работы

В начале выполнения данной работы обязательно следует изучить следующие вопросы из теоретической части пособия:

1) команда вывода строки символов на экран *echo* (подразд. 1.3);

2) использование метасимволов и перенаправление ввода-вывода (п. 1.5.2);

3) конвейеры программ (п. 1.5.3);

4) переменные (п. 1.5.4) — понятие переменной; способы задания значений переменных (непосредственное задание строки символов; использование значения другой переменной; использование выходных данных команды *shell*; применение команды ввода *read*); вывод переменных с помощью команды *set*; переменные окружения;

5) командные файлы (п. 1.5.6) — понятие скрипта; способы запуска скрипта; вложенные скрипты; комментарии; позиционные параметры; инициализационный скрипт *.profile*;

6) защита файлов (п. 3.2) — типы пользователей; основные формы доступа к файлу; права доступа; команда *ls* с флагом *-l*; команда *chmod*.

В процессе изучения перечисленных вопросов обязательно следует обратить внимание на примеры, приведенные в пособии.

#### 9.2.2. Задание

Выполнить (желательно наизусть) последовательность действий:

1) создать два каталога и поместить в один из них четыре текстовых файла, два из которых имеют в своем имени одинаковую символьную последовательность, называемую далее «словом»;

2) поместить во второй каталог скрипт, имеющий два входных параметра: имя каталога и набор символов. Скрипт выполняет действия:

- вывод на экран перечня файлов, «дочерних» к заданному каталогу, которые имеют в своем имени заданный набор символов;
- уничтожение всех остальных файлов заданного каталога;
- любые другие действия (по вашему желанию);

3) создать свой инициализационный скрипт, который выполняет следующие действия:

- здороваются;
- «переделывает» приглашения *shell*;
- запускает вложенный скрипт, созданный в (2), задав ему в качестве параметров каталог и «слово» из (1);
- любые другие действия (по вашему желанию);

4) выйти из *UNIX*, а затем войти вновь с целью демонстрации результатов выполнения инициализационного скрипта.

*Примечание 1.* При выполнении задания разрешается использовать те средства *shell*, которые рассмотрены в данной и предыдущей работах. В частности, нельзя применять управляющие операторы (рассматриваются в следующей работе).

*Примечание 2.* Для избирательного удаления файлов в (2) рекомендуется использовать команду *rm* с флагом *-i*, предварительно установив права доступа к файлам. При этом для обеспечения автоматического выполнения *rm* ее стандартный ввод должен быть переключен на вспомогательный файл, содержащий любой символ, кроме «у».

## 9.3. Лабораторная работа № 3.

### Управляющие операторы командного языка

Целью выполнения настоящей лабораторной работы является развитие навыков программирования на языке *shell* путем использования в скриптах управляющих операторов *if*, *case*, *for*, *while*, *until*.

#### 9.3.1. Подготовка к выполнению работы



В начале выполнения данной работы обязательно следует изучить следующие вопросы из теоретической части пособия (п. 1.5.5):

- 1) оператор двухальтернативного выбора *if*;
- 2) оператор многоальтернативного выбора *case*;
- 3) оператор цикла с перечислением *for*;
- 4) оператор цикла с условием *while*;
- 5) оператор цикла с инверсным условием *until*.

В процессе изучения перечисленных вопросов обязательно следует обратить внимание на примеры, приведенные в пособии.

### 9.3.2. Задание

Требуется разработать программу на языке *shell* (без использования команды *find*), выполняющую поиск в заданном поддереве файловой структуры всех файлов, имена которых отвечают заданному шаблону. Результатом работы программы является перечень имен искомых файлов на экране.

*Примечание.* Программа состоит из двух скриптов. Главный скрипт выполняет вывод на экран приглашения ввести с клавиатуры имя-путь начального каталога и шаблон поиска. Далее он выполняет ввод этих данных с клавиатуры и выводит на экран перечень искомых файлов в начальном каталоге поиска (если они там есть). Затем он вызывает для каждого подкаталога вложенный скрипт, передав ему два входных параметра: 1) относительное имя подкаталога; 2) шаблон поиска.

Вложенный скрипт выполняет поиск в заданном каталоге искомых файлов, а для каждого подкаталога вызывает точно такой же скрипт. (При выполнении любого скрипта запускается новый экземпляр *shell*, поэтому рекурсивное выполнение скриптов не приводит к каким-либо трудностям.)

## 9.4. Лабораторная работа № 4. Процессы в *UNIX*

Целью выполнения настоящей лабораторной работы является развитие навыков работы в среде *UNIX*: 1) управление программными процессами при использовании команды *shell*; 2) обмен сообщениями с другими пользователями этой же системы *UNIX*.

### 9.4.1. Подготовка к выполнению работы

В начале выполнения данной работы обязательно следует изучить следующие вопросы из теоретической части пособия:

1) общие сведения о процессах (подразд. 2.2) — понятие процесса; дерево процессов; имя процесса; получение сжатой информации о процессах с помощью команды *ps* без флагов;

2) запуск процессов в фоновом режиме и ожидание завершения дочернего фонового процесса с помощью команды *wait* (п. 1.5.2);

3) задержка выполнения текущего *shell* на заданное число секунд с помощью команды *sleep* (п. 1.5.5);

4) уничтожение процессов с помощью команды *shell – kill* (п. 2.4.2);

5) приоритеты процессов (п. 4.4) — приоритеты *CPU*, *PRI* и *NICE*; команда задания приоритета *nice*; получение подробной информации о процессах с помощью команды *ps* с флагом *-l*.

В процессе изучения перечисленных вопросов обязательно следует обратить внимание на примеры, приведенные в пособии.

### 9.4.2. Сообщения другому пользователю

Рано или поздно у вас могло появиться желание побеседовать с другими пользователями *UNIX*. Для этого, во-первых, желательно ознакомиться со списком пользователей, работающих в данный момент в системе. Это можно сделать с помощью команды *who*. Она выводит список пользователей с указанием для каждого из них имени, терминала и времени входа в систему.

Посылка сообщения на экран другого пользователя может быть выполнена командой *write* с указанием имени получателя сообщения. Например, пусть мы вошли в систему под именем *vlad* и хотим послать сообщение пользователю с именем *sergei*. Для этого набираем команду

```
$ write sergei
```

Ввод этой команды приведет к последствиям как для нашего терминала, так и для терминала другого пользователя. Во-первых, наш *shell* будет ждать ввода сообщения до тех пор, пока мы не введем конец файла, то есть *<Ctrl>&<D>*. Во-вторых, ввод нами этой команды приведет к появлению на другом экране сообщения

```
message from vlad tty1 (сообщение от vlad и терминала  
tty1)
```

Получив эту строку, пользователь *sergei* может поступить двояко. Во-первых, он может перейти к ожиданию собственно сообщения (оно поступит после нажатия нами `<Ctrl>&<D>`). Во-вторых, *sergei* может набрать ответную команду *write vlad*. Если отправка нашего сообщения (момент нажатия `<Ctrl>&<D>`) совпадет по времени с работой другого пользователя на клавиатуре, то на его экране произойдет наложение двух текстов. Аналогично наложение двух текстов может произойти и на нашем терминале, если во время набора нами сообщения начнет поступать ответное сообщение. Для предотвращения подобных неприятных ситуаций необходимо, чтобы диалог двух пользователей соответствовал некоторому протоколу.

Протокол — алгоритм диалога двух удаленных партнеров, предотвращающий порчу информации. Например, простейший протокол применения команд *write* заключается в следующем. Допустим, что мы получили на своем экране сообщение «*message from ...*». Тогда мы будем ждать появления в конце строки одиночной буквы «о», которая означает, что пришла наша очередь набирать сообщение. Набрав его, мы поместим в конец строки букву «о», а уж затем нажмем `<Ctrl>&<D>`. Если мы хотим прекратить диалог, то набираем в конце строки не одну, а две буквы «о».

Вне зависимости от того, применяем ли мы этот или другой подобный протокол, важно помнить, что за его выполнение *UNIX* не несет никакой ответственности. Выполнение подобного «прикладного» протокола полностью основано на добровольном согласии партнеров диалога соблюдать требования протокола.

Возможно, что в данное время вы не хотите вести какие-то диалоги со своими «соседями» по *UNIX*. Тогда вам следует набрать команду *mesg* с параметром *n*. В результате все сообщения, идущие на ваш терминал, будут отменены до тех пор, пока вы не наберете эту же команду *mesg*, но с параметром *y*.

### 9.4.3. Задание

Выполните наизусть последовательность действий:

- 1) запуск из командной строки в фоновом режиме скрипта, выполняющего задержку на 10 секунд;
- 2) автоматическое ожидание завершения запущенного скрипта;
- 3) одновременный запуск в фоновом режиме двух одинаковых скриптов, выполняющих бесконечные циклы, с разными приоритетами;

4) по истечении нескольких минут получение на экране и объяснение информации о затратах времени ЦП двух запущенных ранее бесконечных процессов;

5) уничтожение созданных процессов;

6) обмен двумя-тремя сообщениями с другим пользователем, работающим в системе.

*Примечание.* При выполнении задания 3 можно обеспечить одновременность запуска скриптов, поместив их в общий конвейер (несмотря на отсутствие информационного обмена между скриптами).

## 9.5. Лабораторная работа № 5.

### Операции с файлами в программе на языке СИ

Целью выполнения настоящей лабораторной работы является получение начальных навыков использования системных вызовов *UNIX* в программах на языке СИ. Данные вызовы выполняют основные операции с файлами: открытие и создание, чтение и запись, закрытие и уничтожение.

#### 9.5.1. Подготовка к выполнению работы

В начале выполнения данной работы рекомендуется изучить следующие вопросы из теоретической части пособия:

- 1) логические файлы (п. 6.1.1);
- 2) открытие файла (п. 6.1.2);
- 3) создание файла (п. 6.1.3);
- 4) понятие файлового указателя (п. 6.1.2) и его перемещение (п. 6.1.3);
- 5) чтение из файла и запись в него (п. 6.1.3);
- 6) закрытие и уничтожение файла (п. 6.1.3).

#### 9.5.2. Характеристика языка СИ

До сих пор мы создавали программные процессы, используя для этого уже готовые программы (утилиты). Теперь пришла пора заняться изготовлением своих собственных программ. Основным языком программирования в среде *UNIX* является СИ (или СИ++). Именно на этом языке написано 85 % ядра *UNIX* (остальные 15 % — на Ассемблере). Применение СИ обеспечивает мо-

бильность ОС (переносимость с одной аппаратной базы на другую) при достаточно хорошей эффективности программ (достаточно низкие затраты времени ЦП и низкие затраты ОП).

Разработку программ начнем с программирования операций с файлами. Для этого программа на СИ имеет две основные возможности. Первая из них предполагает непосредственное обращение из программы к ядру ОС с помощью системных вызовов. Вторая возможность реализуется путем вызова функций **стандартной библиотеки ввода-вывода**. Эти функции предоставляют прикладной программе более удобный интерфейс, освобождая ее, в частности, от собственной буферизации файловых операций. Но основную работу с файлами по-прежнему выполняют системные вызовы, которые теперь делаются не из прикладной программы, а из функций стандартной библиотеки. Не допускается использовать оба перечисленных способа для работы с одним и тем же файлом в одной и той же программе.

Одним из различий двух названных подходов, учитываемых при программировании, является способ задания логического имени файла в программе. В отличие от имени физического файла, которое уникально для всей системы (это или имя-путь, или относительное имя — смещение относительно текущего каталога), логическое имя файла обладает уникальностью только в пределах данного процесса. Если для работы с данным файлом в программе применяются системные вызовы, то в качестве логического имени файла используется его номер, уникальный для данного процесса.

Если для работы с файлом в программе применяются функции стандартной библиотеки, то в качестве логического имени файла используется указатель на таблицу (структуру), используемую этими функциями для работы с файлом. Данная структура имеет тип **FILE**. Одно из ее полей содержит номер файла, позволяя тем самым стандартным функциям осуществлять системные вызовы. (Полезно заметить, что аналогичные структуры применяются для работы с файлами и в языке ПАСКАЛЬ.)

### 9.5.3. Открытие существующего файла

Результатом выполнения операции открытия файла является или логическое имя файла, или признак ошибки.

**Системные вызовы.** Открытие файла выполняет системный вызов **open**. Его описание:

```
int open(const char *pathname, int flags, [mode_t mode]);
```

Данный вызов возвращает в программу или номер открытого файла (неотрицательное целое число), или (–1) (признак ошибки). Ошибка может произойти, например, если файл не существует.

Первый параметр (*pathname*) является указателем на символьную строку, содержащую имя физического файла. В качестве этого имени здесь, а также во всех приводимых ниже системных вызовах и в стандартных функциях можно использовать или имя-путь файла, или его относительное имя.

Второй параметр (*flags*) вызова *open* определяет запрашиваемый метод доступа к файлу. Этот параметр принимает одно из значений, заданных константами в заголовочном файле *<fcntl.h>*. (Как и большинство заголовочных файлов, *<fcntl.h>* находится в каталоге */usr/include* и может быть включен в программу с помощью директивы *#include <fcntl.h>*.) Три особо интересных константы:

- 1) *O\_RDONLY* — открыть файл только для чтения;
- 2) *O\_WRONLY* — открыть файл только для записи;
- 3) *O\_RDWR* — открыть файл для чтения и для записи.

Третий параметр (*mode*) необязательный (об этом свидетельствуют квадратные скобки). При открытии существующего файла он не используется.

**Пример программы:**

```
#include <stdlib.h>          /* Для вызова exit */
#include <fcntl.h>
char *namefile="abc";      /* Имя файла */
main()
{
    int fil;
    /* Открыть файл для чтения-записи */
    if ((fil = open(namefile, O_RDWR)) == -1)
    {
        printf("Невозможно открыть %s\n", namefile);
        exit(1);             /* Выход по ошибке */
    }
    exit(0);                 /* Нормальный выход */
}
```

Данная программа не выполняет никакой полезной работы, а просто демонстрирует некоторые важные моменты. Во-первых, всякая файловая операция может завершиться с ошибкой. Поэтому такая возможность должна быть обязательно предусмотрена в программе. Во-вторых, выход из программы (а точнее — завершение

соответствующего программного процесса) выполняет системный вызов *exit*, единственный параметр которого содержит код завершения процесса (0 — успешно; 1 — с ошибкой). Прототип этого системного вызова содержится в заголовочном файле *<stdlib.h>*.

Следующая программа отличается от предыдущей только тем, что имя открываемого файла задается не в тексте программы, а из командной строки *shell* в качестве параметра запускаемого файла. Допустим, что готовая программа помещена в файл *open\_file*. Тогда для ее запуска можно использовать командную строку

```
$ open_file abc
```

**Текст программы:**

```
#include <stdlib.h>                /* Для вызова exit */
#include <fcntl.h>
main(int argc, char *argv[ ])
{
    int fil;
    if (argc !=2)
    {
        printf("Need 1 argument for open\n");
        exit(1);                    /* Выход по ошибке */
    }
    /* Открыть файл для чтения-записи */
    if ((fil = open(argv[1], O_RDWR)) == -1)
    {
        printf("Cannot open file %s\n", argv[1]);
        exit(1);                    /* Выход по ошибке */
    }
    exit(0);                        /* Нормальный выход */
}
```

*Shell* запускает главную подпрограмму *main*, присвоив ее аргументу *argc* значение количества параметров в списке *argv*, а каждому элементу массива *argv* — значение соответствующего параметра из командной строки. В примере *argc* равно 2, элемент *argv[0]* содержит строку символов «*open\_file*» (имя программы условно является нулевым параметром), *argv[1]* — строку символов «*abc*». В начале своего выполнения программа проверяет, правильное ли число параметров ей было передано при запуске.

**Функции стандартной библиотеки.** Открытие файла выполняет стандартная функция *fopen*. Следующая программа выполняет то же самое, что и программа в предыдущем примере, — открывает

файл, имя которого пользователь набирает в командной строке в качестве параметра запускаемого файла:

```
#include <stdlib.h>          /* Для вызова exit */
#include <stdio.h>           /* Содержит определения FILE, */
                             /* fopen и т.д. */

main(int argc, char *argv[])
{
    FILE *stream;
    if (argc != 2)
    {
        printf("Need 1 argument for open\n");
        exit(1);             /* Выход по ошибке */
    }
    /* Открыть файл для чтения-записи */
    if ((stream = fopen(argv[1], "r+")) == NULL)
    {
        printf("Cannot open file %s\n", argv[1]);
        exit(1);             /* Выход по ошибке */
    }
    exit(0);                 /* Нормальный выход */
}
```

Первый параметр функции *fopen* — указатель на имя открываемого физического файла. Второй параметр — символьная строка, которая определяет запрашиваемый метод доступа к файлу: «r» — файл открывается только для чтения; «a» — файл открывается только для записи; «r+» — для чтения и записи. В случае успеха функция *fopen* заполняет для файла структуру *FILE* и возвращает в переменной *stream* ее адрес. В случае ошибки возвращается указатель *NULL*.

**В ы п о л н и т е** приведенные выше программы.

## 9.5.4. Создание файла

Результатом выполнения операции создания файла является или логическое имя файла, или признак ошибки.

**Системные вызовы.** Для создания файла может быть использован один из двух системных вызовов: *open* и *creat*. Вызов *creat* представляет собой традиционный способ создания файла. Его описание:

```
int creat(const char *pathname, mode_t mode);
```



Первый параметр (*pathname*) является указателем на символьную строку, содержащую имя физического файла. Параметр *mode* задает права доступа к файлу. Обычно это 3-значное восьмеричное число, старшая цифра которого определяет права доступа владельца файла, средняя цифра — владельца-группы, а младшая цифра задает права доступа всех остальных пользователей. Например, восьмеричное число (в СИ восьмеричное число начинается с нуля) 0764 задает право доступа *rwx* для владельца, *rw* — для владельца-группы, *r* — для всех остальных пользователей. Пример использования вызова *creat*:

```
fil = creat("/tmp/newfile", 0764);
```

В случае успешного завершения системный вызов *creat* возвращает номер нового файла, открытого на запись. А в случае ошибки возвращает значение  $-1$ . Для того чтобы только что созданный файл был открыт для чтения, его необходимо сначала закрыть, а затем открыть вновь.

Если файл, заданный в качестве первого параметра *creat*, уже существует, то, во-первых, второй параметр вызова *creat* игнорируется, а во-вторых, данный файл усекается до нулевой длины и, следовательно, прежняя информация файла уничтожается.

Рассмотрим применение для создания файла системного вызова *open*, который предоставляет больше возможностей по сравнению с вызовом *creat*. Для этого, во-первых, в состав второго операнда вызова *open* обязательно должна быть включена константа **O\_CREAT**. Во-вторых, данный вызов должен иметь третий операнд — *mode*, аналогичный операнду вызова *creat*.

Одним из отличий создания файла при помощи *open* является то, что новый файл сразу же оказывается открытым заданным способом. Для этого в состав второго операнда кроме **O\_CREAT** включается константа **O\_RDONLY**, **O\_WRONLY** или **O\_RDWR**. Естественно, что запрашиваемый доступ к файлу не должен выходить за рамки прав доступа, задаваемых параметром *mode*. Иначе вызов *open* возвратит признак ошибки. Например, следующий вызов предназначен для создания файла, а также для его открытия на чтение и запись:

```
fil = open("/tmp/newfile", O_RDWR | O_CREAT, 0760);
```

Третья константа, включаемая в состав второго операнда вызова *open*, определяет поведение этого вызова, если файл с заданным именем уже существует. В частности, если данная константа

опущена, то существующий файл будет открыт на запись (если позволяют права доступа к нему), а параметр *mode* игнорируется.

Если в состав второго операнда включена константа **O\_EXCL** (*exclusive* — исключительный), то в случае существования файла вызов *open* завершится с ошибкой. Пример такого вызова:

```
fil = open("abc", O_WRONLY | O_CREAT | O_EXCL, 0760);
```

Если в состав второго операнда включена константа **O\_TRUNC**, то в случае существования файла он будет усечен до нулевого размера (если позволяют права доступа). Пример такого вызова:

```
fil = open("abc", O_WRONLY | O_CREAT | O_TRUNC, 0760);
```

**Функции стандартной библиотеки.** Для создания файла используется описанная ранее функция *fopen*, в качестве второго параметра которой задается строка «и» или «а». Каждая из этих строк приводит к созданию нового файла, открытого на запись. Различие между ними проявляется в том случае, если уже существует файл с заданным именем. Строка «и» приводит к усечению этого файла до нулевой длины, а строка «а» ограничивается открытием существующего файла.

**Работа программы** создания файла, получающую имя файла из командной строки. После выполнения программы проверьте наличие созданного файла.

### 9.5.5. Указатель файла

После того как файл открыт любым из изложенных выше способов, программный процесс может выполнять информационный обмен с ним, то есть выполнять или чтение данных из файла, или запись данных в файл. Для каждого открытого файла ядро содержит отдельную переменную, называемую указателем файла, которая содержит номер того байта файла, начиная с которого будет выполняться последующее чтение файла или запись в него. Поэтому прежде чем выполнять операцию чтения (записи) файла, необходимо четко представлять себе, номер какого байта находится в указателе файла. Например, в результате обычного открытия файла его указатель содержит 0. Если же во второй операнд вызова *open* добавить константу **O\_APPEND**, то указатель файла будет установлен на его конец.

**Системные вызовы.** Системный вызов *lseek* позволяет установить указатель файла на любую позицию. Его описание:

```
#include <sys/types.h>
#include <unistd.h>
off_t lseek(int fil, off_t offset, int whence);
```

При успешном завершении вызов *lseek* возвращает новое значение указателя файла. В случае ошибки: –1. Первый параметр (*fil*) — номер файла, открытого ранее. Второй параметр (*offset*) задает требуемое смещение относительно той позиции указателя файла, которая задается третьим параметром (*whence*), который можно задать в виде одной из следующих трех констант, определенных в файле **<unistd.h>**:

**SEEK\_SET** — смещение прибавляется к началу файла;

**SEEK\_CUR** — смещение прибавляется к текущему значению указателя;

**SEEK\_END** — смещение прибавляется к концу файла.

Несмотря на то что тип смещения и тип значения указателя файла один и тот же — *off\_t* (он определен в файле **<sys/types.h>**), область допустимых значений для каждого из них различна. В отличие от указателя файла, принимающего лишь неотрицательные целые значения, смещение может быть и отрицательным.

Вообще говоря, вызов *lseek* выдаст признак ошибки для существующего файла только в том случае, если мы попытаемся установить указатель файла на позицию, предшествующую началу файла. И наоборот, операция установки указателя на позицию, расположенную далее конца файла, приведет к появлению в файле «дыры», заполненной нулями.

**Пример** использования вызова *lseek*:

```
off_t newpos;
...
newpos = lseek(fil, (off_t)-16, SEEK_END);
```

В данном примере новое значение указателя файла будет на 16 меньше, чем значение, соответствующее концу файла. Обратите внимание, что для числа (–16), задающего смещение, используется явное задание типа (*off\_t*).

Интересно отметить, что вызов *lseek* можно использовать для определения длины файла. Например, пусть переменная *filsize* содержит длину файла, тогда

```
off_t filsize;
int fil;
...
filsize = lseek(fil, (off_t) 0, SEEK_END);
```

**Функции стандартной библиотеки.** Установку указателя файла выполняет стандартная функция *fseek*. Ее описание:

```
#include <stdio.h>
```

```
int fseek(FILE *stream, long offset, int whence);
```

Функция возвращает ненулевое значение только в случае ошибки. Первый параметр (*stream*) идентифицирует файл. Назначение двух других параметров аналогично вызову *lseek*.

**С о з д а й т е** с помощью текстового редактора небольшой файл, а затем с помощью своей программы образуйте в этом файле «дыру», наличие которой проверьте с помощью текстового редактора.

### 9.5.6. Чтение из файла

После того как указатель файла установлен в требуемое место файла, можно выполнить чтение из этого файла требуемого числа байтов.

**Системные вызовы.** Чтение из файла выполняет системный вызов *read*. Его описание:

```
#include <unistd.h>
```

```
ssize_t read(int fil, void *buffer, size_t n);
```

Данный вызов выполняет чтение из файла, номер которого задается первым параметром (*fil*), такого числа байтов, которое задается третьим параметром (*n*). Считанные байты помещаются в буфер, указатель на который задается вторым параметром (*buffer*). Буфер представляет собой массив, элементы которого имеют произвольный тип (*void*). (Чаще всего буфер оформляется как массив символьного типа.)

Если выполнение *read* завершилось успешно, то он возвращает число байтов, считанных из файла. Это число равно *n* или меньше его. Второе выполняется тогда, когда мы выполняем чтение из конечной части файла, содержащей меньше символов, чем запрашивается. Если после этого опять выполнить *read*, то он возвратит число 0, означающее, что достигнут конец файла. В случае ошибки *read* возвращает (−1).

Одним из результатов выполнения *read* является перемещение указателя файла на первый, не считанный байт файла или на его конец.

#### Пример

Следующая программа выполняет подсчет байтов в файле, имя которого передается из командной строки:

```
#include <stdlib.h>                /* Для вызова exit */
#include <fcntl.h>
#include <unistd.h>
#define BUFSIZE 512
main(int argc, char *argv[])
{
    int fil;
    char buffer[BUFSIZE];
    ssize_t nread;
    long total = 0;
    if (argc != 2)
    {
        printf("Need 1 argument for open\n");
        exit(1);                    /* Выход по ошибке */
    }
    /* Открыть файл для чтения */
    if ((fil = open(argv[1], O_RDONLY)) == -1)
    {
        printf("Cannot open file %s\n", argv[1]);
        exit(1);                    /* Выход по ошибке */
    }
    /* Повторять до конца файла */
    while((nread = read(fil, buffer, BUFSIZE)) > 0)
        total = total + nread;
    printf("total = %ld\n", total);
    exit(0);
}
```

Поясним, почему длина буфера взята равной 512 байтов. Дело в том, что каждый системный вызов *read* инициирует считывание с диска блока байтов. Величина этого блока есть число 512, умноженное на степень двойки, и зависит от системы. Если считывать данные с диска небольшими порциями (по несколько байтов), то никаких изменений в результатах работы программы не будет. Единственное — резко возрастет время выполнения программы, так как, во-первых, чтение каждой порции байтов требует считывания в лучшем случае из дискового КЭШа, а в худшем — с диска целого блока. А во-вторых, большое количество системных вызовов приводит к такому же числу переключений ЦП из режима

«задача» в режим «ядро» и обратно, что требует заметных затрат времени ЦП.

**Функции стандартной библиотеки.** Применение для чтения из файла стандартных библиотечных функций позволяет не заботиться об эффективности информационного обмена, так как, во-первых, эти функции занимаются буферизацией обмена с файлом. Поэтому запрос из программы очередной порции байтов не приводит неизбежно к считыванию блока файла. Если эта порция находится в том блоке, который был считан последним с диска, то требуемые байты стандартная функция находит в своем буфере, откуда она и копирует их в буфер программы. Во-вторых, функции стандартной библиотеки выполняются в режиме «задача» и поэтому их вызов не приводит к переключению процессора.

Примером стандартной функции чтения файла является функция чтения символа *getc*. Ее описание:

```
#include <stdio.h>
int getc(FILE *istream);
```

Данная функция возвращает или код символа или число (–1), которое означает «конец файла» (*EOF*). Единственный параметр функции (*istream*) представляет собой указатель на структуру *FILE*. Этот параметр должен быть получен в результате открытия файла в программе (до вызова функции *getc*).

**В ы п о л н и т е** приведенную выше программу подсчета символов.

### 9.5.7. Запись в файл

После того как указатель файла установлен в требуемое место файла, можно выполнить запись в файл требуемого числа байтов. Если указатель файла был установлен на конец файла, то запись будет выполняться на свободное место за счет увеличения длины файла. Иначе — запись выполняется «поверх» прежнего содержимого файла.

**Системные вызовы.** Запись в файл выполняет системный вызов *write*. Его описание:

```
#include <unistd.h>
ssize_t write(int fil, void *buffer, size_t n);
```

Данный вызов выполняет запись в файл, номер которого задается первым параметром (*fil*), такого числа байтов, которое задается третьим параметром (*n*). Запись в файл выполняется из буфера,

указатель на который задается вторым параметром (*buffer*). Буфер представляет собой массив, элементы которого имеют произвольный тип (*void*).

Если выполнение *write* завершилось успешно, то он возвращает число байтов, записанных в файл. Почти всегда это число равно *n*. В случае ошибки *write* возвращает (−1).

Одним из результатов выполнения *write* является перемещение указателя файла на первый байт после записанного участка файла.

В качестве примера приведем фрагмент программы, выполняющей запись содержимого буфера *outbuf*, имеющего длину *OBSIZE*, в файл с именем *abc*:

```
fil = open("abc", O_RDWR | O_APPEND);  
write(fil, outbuf, OBSIZE);
```

**Функции стандартной библиотеки.** Примером стандартной функции записи в файл является функция записи символа *putc*. Ее описание:

```
#include <stdio.h>  
int putc(int c, FILE *ostream);
```

При успешном завершении данная функция возвращает код символа (совпадает с параметром *c*), а в случае ошибки — число (−1). Параметр *ostream* аналогичен параметру *istream* для функции *getc*.

**С о з д а й т е** программу, выполняющую добавление символов в конец текстового файла.

### 9.5.8. Заккрытие и уничтожение файла

После того как программа завершила работу с файлом, этот файл желательно закрыть, а если файл в будущем не понадобится, то его желательно уничтожить. Каждая из этих операций (особенно уничтожение) освобождает ресурсы, использовавшиеся для работы с файлом. При завершении процесса все файлы, открытые им, закрываются автоматически.

**Системные вызовы.** Заккрытие файла выполняет системный вызов *close*. Его описание:

```
#include <unistd.h>  
int close(int fil);
```

В случае успешного завершения вызов *close* возвращает 0. В случае ошибки (−1). Единственный параметр вызова — номер закрываемого файла.

Для уничтожения файла может быть использован вызов ***unlink***. Его описание:

```
#include <unistd.h>
int unlink(const char *pathname);
```

В случае успешного завершения вызов возвращает 0, иначе (−1). Единственный параметр (*pathname*) есть указатель на имя файла.

**Функции стандартной библиотеки.** Заккрытие файла выполняет стандартная функция ***fclose***. Ее описание:

```
#include <stdio.h>
int fclose(FILE *stream);
```

Функция возвращает значение 0 при успешном завершении. В случае ошибки (−1).

Что касается уничтожения файла, то для этого вполне достаточно системного вызова ***unlink*** — и соответствующая стандартная функция не существует.

### 9.5.9. Задание

Требуется разработать одну из следующих программ (по указанию преподавателя):

1) копирование файла (имя старого и нового файлов передается в командной строке) при использовании системных вызовов;

2) копирование файла (имена старого и нового файлов передаются в командной строке) при использовании стандартных функций;

3) вывод на экран содержимого текстового файла, имя которого задается в командной строке, при использовании системных вызовов;

4) вывод на экран содержимого текстового файла, имя которого задается в командной строке, при использовании стандартных функций;

5) ввод с клавиатуры содержимого текстового файла, имя которого задается в командной строке, при использовании системных вызовов;

6) ввод с клавиатуры содержимого текстового файла, имя которого задается в командной строке, при использовании стандартных функций.



## 9.6. Лабораторная работа № 6.

### Системные вызовы для управления процессами

Целью выполнения настоящей лабораторной работы является получение навыков использования в программах на языке СИ системных вызовов *UNIX*, предназначенных для управления процессами *fork*, *exec*, *wait*, а также работа с переменными окружения.

#### 9.6.1. Подготовка к выполнению работы

В начале выполнения данной работы рекомендуется ознакомиться со следующими вопросами из теоретической части пособия:

- 1) состояния процесса (подразд. 4.1);
- 2) создание процесса (подразд. 4.2);
- 3) переменные окружения (п. 1.5.4).

#### 9.6.2. Создание процесса

Любой процесс может порождать дочерние процессы, используя системный вызов *fork*. Его описание:

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```

В случае успешного завершения вызова *fork* создается новый процесс, который является почти полной копией своего процесса-отца. При этом он имеет такую же программу, переменные окружения, а также структуры данных ядра, обслуживающие процесс. Единственное различие между процессами в том, что новый процесс имеет другой *pid*. (Вспомним, что *pid* — номер процесса, уникальный для всей системы.) Таким образом, результатом выполнения *fork* является параллельное (одновременное) существование в данный момент двух процессов, выполняющих одну и ту же программу, причем в одной и той же ее точке. Эта точка — команда, которая следует в программе за командой вызова подпрограммы *fork*.

Дальнейшее выполнение одной и той же программы процессом-отцом и дочерним процессом различно по той причине, что различается код возврата подпрограммы *fork* в эти два процесса.

В процесс-отец возвращается код возврата, который совпадает с *pid* нового процесса, а код возврата в новый процесс равен 0. Поэтому после оператора вызова подпрограммы *fork* программа должна содержать или оператор *if*, или оператор *switch*, выполняющий ветвление программы в зависимости от кода возврата подпрограммы *fork*.

### Пример

Следующая программа при выполнении порождает дочерний процесс. Дальнейшее выполнение обоих процессов различается тем сообщением, которое процесс выводит на экран:

```
#include <unistd.h>
main()
{
    pid_t pid;
    switch(pid = fork()) {
        case -1:
            perror("Error");
            exit(1);
            break;
        case 0:
            /* Выполнение в дочернем процессе */
            printf("Потомок\n");
            exit(0);
            break;
        default:
            /* Выполнение в родительском процессе */
            printf("Родитель\n");
            exit(0);
    }
}
```

Библиотечная процедура ***perror*** выполняет вывод той символической строки, которая задана в качестве ее единственного параметра, в файл стандартного вывода ошибки. После этого она выводит символ «:» и диагностическое сообщение, уточняющее тип произошедшей ошибки. (Для получения диагностического сообщения процедура *perror* использует переменную *errno* из окружения процесса, которая содержит код последней ошибки, возникшей в результате системного вызова.)

**В ы п о л н и т е** приведенную выше программу.

### 9.6.3. Ожидание завершения потомка

В приведенном выше примере создание дочернего процесса приводит к параллельному существованию двух процессов, никак не связанных далее между собой. Простейшим средством синхронизации процессов является системный вызов *wait*. Его описание:

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);
```

Вызов *wait* временно приостанавливает выполнение процесса-отца до тех пор, пока не завершится любой дочерний процесс. (Это завершение является следствием системного вызова *exit*.) После этого *wait* возвращает в процесс-отец *pid* завершившегося процесса. Единственный параметр (*status*) представляет собой указатель на целое число. Если этот указатель равен *NULL*, то данный параметр далее не используется. Иначе в результате завершения *wait* переменная *status* указывает на код завершения дочернего процесса, переданный при помощи вызова *exit*.

#### Пример

Переделаем приведенную в п. 9.6.2 программу так, чтобы процесс-отец ожидал завершения дочернего процесса. Заключительный фрагмент программы:

```
...
default:
/* Выполнение в родительском процессе */
wait((int *) 0);
printf("Завершение потомка\n");
exit(0);
}
}
```

**В ы п о л н и т е** новый вариант программы с использованием вызова *wait*.

Допустим, что процесс породил не один, а несколько дочерних процессов. Так как вызов *wait* сообщает о завершении лишь одного потомка, то для ожидания завершения всех потомков необходимо в программе, выполняемой процессом-отцом, организовать соответствующий цикл. Если процесс-отец должен ожидать завершения не всех, а лишь конкретного потомка, то следует использовать вместо *wait* системный вызов *waitpid*. Его описание:

```
#include <sys/types.h>
```

```
#include <sys/wait.h>
pid_t waitpid(pid_t pid, int *status, int options);
```

Первый параметр (*pid*) есть номер дочернего процесса, завершение которого ожидается. Третий параметр (*options*) может принимать постоянные значения, определенные в файле `<sys/wait.h>`. Наиболее интересная константа — **WNOHANG**. Задание ее в качестве третьего параметра позволяет в цикле вызывать *waitpid* без блокирования процесса-отца. При этом если требуемый дочерний процесс еще не завершился, *waitpid* возвращает значение 0.

### 9.6.4. Загрузка новой программы

Любая программа может заменить себя в адресном пространстве процесса на другую программу, используя одну из шести функций семейства **exec**. Каждая из этих функций преобразуется транслятором в один и тот же системный вызов, различаясь между собой лишь деталями предоставляемого программного интерфейса. Важно подчеркнуть, что любой такой вызов приводит не к появлению нового процесса, а лишь к смене содержимого памяти того процесса, который содержит вызов *exec*.

В дальнейшем изложении будем использовать функцию **execl**. Ее описание:

```
#include <unistd.h>
int execl(const char *path, const char *arg0, ..., const char
*argn, (char *) 0);
```

Все параметры вызова *execl* являются указателями строк. Первый из них (*path*) задает имя-путь того файла, который содержит запускаемую программу или скрипт. Второй параметр (*arg0*) задает простое имя загружаемой программы. Этот параметр и оставшееся переменное число параметров (от *arg1* до *argn*) доступны в вызываемой программе аналогично параметрам командной строки при запуске программы из *shell*. (На самом деле *shell* сам использует вызовы *fork* и *exec* для запуска программ.) Так как список параметров имеет произвольную длину, он должен заканчиваться нулевым указателем для обозначения конца списка.

#### Пример

Следующая программа заменяет сама себя на известную программу *cat* (полное имя файла `/bin/cat`). Допустим, что *cat* должна

вывести на экран содержимое файла *letter*, расположенного в том же каталоге, что и сама следующая программа:

```
#include <unistd.h>
main()
{
    printf("Запуск программы cat\n");
    execl("/bin/cat", "cat", "letter", (char *) 0);
    /* Если execl возвращает значение, значит ошибка */
    perror("Вызов execl не смог запустить программу cat");
    exit(1);
}
```

Следует обратить внимание, что в случае успеха функции *execl* (и любой другой *exec*) она никогда не возвращает управление в старую программу. Поэтому единственным возвращаемым значением является признак ошибки (–1).

**В ы п о л н и т е** запуск из своей программы какой-то известной вам готовой программы, например, *ed*.

### 9.6.5. Совместное применение вызовов *fork* и *exec*

Полученный в результате применения вызова *fork* дочерний процесс может продолжать выполнение копии программы процесса-отца или потребовать от ядра ОС выполнения своей собственной программы, используя вызов *exec*. Первый вариант был рассмотрен ранее, а теперь перейдем ко второму.

#### Пример

Следующая программа *command* выполняет запуск командного интерпретатора *bash*:

```
#include <unistd.h>
main(int argc, char *argv[])
{
    pid_t pid;
    if (argc < 2)
    {
        printf("Отсутствуют параметры в командной строке\n");
        exit(1);
        /* Выход по ошибке */
    }
    switch(pid = fork()) {
        case -1:
```

```

    perror("Error fork");
    exit(1);
    break;
case 0:
/* Выполнение в дочернем процессе */
    printf("Запуск shell\n");
    switch(argc)
    case 2:
/* Только имя команды */
        execl("/bin/bash", "bash", "-c", argv[1], (char *) 0);
        break;
    case 3:

/* У команды один параметр */
        execl("/bin/bash", "bash", "-c", argv[1], argv[2], (char *) 0);
        break;
    case 4:
/* У команды два параметра */
        execl("/bin/bash", "bash", "-c", argv[1], argv[2], argv[3],
(char *) 0);
        break;
/* Если execl возвращает значение, значит ошибка */
    perror("Вызов execl не смог запустить программу
bash");
    exit(1);
    default:
/* Ожидание завершения bash */
        wait((int *) 0);
        exit(0);
    }
}

```

Данный пример иллюстрирует тот факт, что относительно ядра ОС *shell* является обыкновенной прикладной программой и поэтому может быть запущен из любой другой прикладной программы, например, из рассматриваемой программы *command*. Сама программа *command* запускается из командной строки *shell*, получив из нее свои параметры. Первый из этих параметров есть «*command*», второй параметр — имя запускаемой программы, например *cat* или *ed*. Остальные параметры потребуются для передачи запускаемой программе в качестве ее параметров.

Выполнение программы *command* начинается с проверки числа переданных ей параметров. Если это число меньше двух, то про-

грамма завершается с ненулевым кодом завершения. Иначе — с помощью вызова *fork* порождается дочерний процесс, который, в свою очередь, загружает программу *bash*. Оператор *switch* обеспечивает передачу этой программе тех параметров, которые получила ранее *command* (кроме ее собственного имени, которое заменяется на строку «*bash*»), а также параметра *-c*. Этот параметр означает, что *bash* должен брать для себя команды не из стандартного ввода (с клавиатуры), а из последующей символьной строки. Определив из этой команды имя программы, *bash* обеспечивает ее выполнение.

Интересно отметить, что *bash* (как и любой другой *shell*) использует для запуска требуемой программы точно такие же системные вызовы *fork*, *exec* и *wait*, что и рассматриваемая прикладная программа. При этом вызов *wait* используется только для запуска дочерних процессов в оперативном режиме. Отсутствие этого вызова приводит к запуску процесса в фоновом режиме.

**В ы п о л н и т е** программу *command*, задавая для нее различное число параметров.

### 9.6.6. Наследование данных при создании процесса и при загрузке программы

Дочерний процесс, созданный в результате вызова *fork*, наследует точно такое же адресное пространство, что и процесс-отец. Термин «точно такое же» не означает «это же самое». Это означает, что все данные процесса-отца копируются в адресное пространство процесса-потомка в момент его создания. По мере дальнейшего выполнения процессов их данные могут все более различаться.

Применение после вызова *fork* системного вызова *exec* существенно сокращает количество совпадающих данных. Из них остаются лишь переменные окружения, существующие в прикладной части адресного пространства процесса, и переменные ядра, которые относятся к данному процессу и которые существуют в системной части адресного пространства. Относительно переменных окружения прикладная программа может выполнять любые действия, а доступ к переменным ядра возможен лишь при помощи системных вызовов.

Примером переменных ядра являются идентификаторы (номера) открытых файлов. Они всегда наследуются дочерним процессом. Поэтому данному процессу нет никакой необходимости заново открывать стандартный ввод (номер 0), стандартный вывод (1) и стандартный вывод ошибки (2). Кроме того, не нужно открывать те файлы, которые были открыты самим процессом-отцом или достались ему по наследству. Благодаря этому такие файлы могут использоваться для информационного обмена между «родственными» процессами. Этому способствует тот факт, что процессы наследуют также указатели открытых файлов (это также переменные ядра). Более того, так как при открытии файла создается единственный экземпляр такого указателя, то результат установки указателя файла в одном процессе всегда может быть учтен в другом.

Рассмотрим более внимательно, что представляют собой переменные окружения. Каждая переменная окружения есть строка вида

*имя переменной = ее содержание*

Можно напрямую использовать переменные окружения в программе процесса, добавив еще один параметр *envp* в список параметров функции *main*. Но более предпочтительный метод доступа из программы процесса к его переменным окружения заключается в использовании глобальной переменной

*extern char \*\*environ;*

### Пример

Следующая программа выполняет вывод на экран своих переменных окружения:

```
#include <stddef.h>
#include <stdio.h>
extern char **environ;
main(int argc, char *argv[])
{
    int i, ch;
    for(i=0; i<20; i++)
        printf("Переменная окружения %d: %s\n", i, environ[i]);
    ch=getchar();
    for(i=20; i<40; i++) {
        if(environ[i] == NULL) break;
        printf("Переменная окружения %d: %s\n", i, environ[i]);
    }
}
```



```
}
```

Данная программа сначала выводит на экран первые двадцать переменных окружения. Затем она ожидает ввода с клавиатуры любого символа и нажатия *<Enter>*. После этого на экран выводятся оставшиеся переменные окружения.

Следует обратить внимание на то, что переменная окружения не есть обычная переменная программы, к которой можно обращаться по ее имени, а это символьная строка, частью которой имя переменной окружения является.

### 9.6.7. Задание

Разработайте программу, выполняющую:

- 1) создание файла и размещение в нем небольшого текста;
- 2) создание двух дочерних процессов, каждый из которых выполняет запись в файл из п. 1 своих переменных окружения, предварительно внося в некоторые из них любые изменения. После этого дочерний процесс завершается;
- 3) дождавшись завершения обоих потомков, вывод результирующего файла на экран;
- 4) осуществление над результирующим файлом одного из действий (по вашему усмотрению):
  - вывод на экран количества слов в файле;
  - вывод на экран строк с заданным ключевым словом.

## 9.7. Лабораторная работа № 7. Обработка сигналов

Целью выполнения настоящей лабораторной работы является получение навыков программного управления процессами с помощью сигналов.

### 9.7.1. Подготовка к выполнению работы

В начале выполнения данной работы следует ознакомиться со следующими вопросами из теоретической части пособия:

- 1) синхронизация процессов с помощью сигналов (п. 2.4.1) — понятие сигнала; типы сигналов; выдача сигнала процессом;

2) терминальное управление процессами (п. 2.4.2) — управляющий терминал; сеанс; группа процессов; получение информации о процессах с помощью команды *shell – ps* с флагом *-j*; применение команды *shell – kill* с целью посылки сигнала процессу;

3) обработка сигналов (подразд. 4.3) — варианты обработки сигналов; диспозиция сигналов;

4) применение таймера для управления процессами (подразд. 4.5) — таймер; аларм; таймер интервала.

## 9.7.2. Изменение диспозиции сигналов

Сразу же после создания процесса с помощью вызова *fork* диспозиция сигналов нового процесса точно такая же, как и у процесса-отца (наследование диспозиции). Но если далее для загрузки программы выполняется вызов *exec*, то для всех сигналов устанавливается диспозиция «по умолчанию». Это объясняется тем, что все прежние обработчики сигналов уничтожены (как и все другие прикладные подпрограммы). Если такая начальная диспозиция не устраивает, то для ее изменения процесс выполняет системный вызов ***sigaction***:

```
#include <signal.h>
int sigaction(int signo, const struct sigaction *act, struct
sigaction *oact);
```

Первый параметр (*signo*) задает сигнал, диспозицию которого требуется изменить. Второй параметр (*act*) задает новый обработчик сигнала. Третий параметр (*oact*) указывает на структуру, в которой будет сохранено описание прежнего обработчика сигнала. Если в качестве этого параметра задано значение *NULL*, то это описание сохранено не будет.

Структура *sigaction* определена в файле *<signal.h>*:

```
struct sigaction {
    void (*sa_handler)(int); /* Функция обработчика */
    sigset_t sa_mask; /* Сигналы, которые
                        блокируются во время
                        обработки сигнала */
    int sa_flags; /* Флаги, влияющие на поведение
                  сигнала */
                  /* Указатель на обработчик */
    void (*sa_sigaction) (int, siginfo_t *, void *);
};
```

Первое поле (*sa\_handler*) задает обработчик сигнала. Оно может содержать одно из значений:

1) **SIG\_DFL** — восстановить обработку сигнала по умолчанию;

2) **SIG\_IGN** — игнорировать данный сигнал;

3) адрес функции, имеющей аргумент типа *int*. Эта функция используется в качестве обработчика сигнала *signo*, а численный идентификатор этого сигнала будет передан ей в качестве входного параметра.

Второе поле (*sa\_mask*) задает набор сигналов, которые должны блокироваться на время обработки сигнала *signo*. **Блокирование сигнала** означает не его игнорирование, а лишь задержку в его обработке. Благодаря блокированию появляется уверенность, что прикладной обработчик сигнала, выполняясь в режиме «Задача», может полностью выполнить свою обработку. Тип *sigset\_t*, используемый для задания набора сигналов, будет описан позже.

Третье поле (*sa\_flags*) позволяет менять характер реакции на сигнал. Например, поместив в это поле константу **SA\_RESETHAND**, мы обеспечим после завершения своего обработчика изменение диспозиции сигнала так, что она будет задавать обработку сигнала по умолчанию. Другой пример: значение **SA\_SIGINFO** позволяет обработчику сигнала получать дополнительную информацию. В этом случае обработчик сигнала задается не полем *sa\_handler*, а полем *sa\_sigaction*. Передаваемая на вход обработчика структура *siginfo\_t* содержит номер сигнала, номер пославшего сигнала процесса и идентификатор пользователя процесса. Не допускается одновременное использование полей *sa\_handler* и *sa\_sigaction* в программе процесса для одного и того же сигнала.

### Пример

Следующая программа задает диспозицию для двух сигналов (**SIGINT** и **SIGUSR1**), поступающих в процесс.

```
#include <signal.h>
/* Функция – обработчик сигнала SIGINT */
void sig_hndlr (int signo)
{
    printf("Получен сигнал SIGINT\n");
}
main()
{
    static struct sigaction act1, act2;
    /* Изменение диспозиции сигнала SIGINT */
```

```

act1.sa_handler = sig_hndlr;      /* Запись адреса
                                   обработчика */
sigaction(SIGINT, &act1, NULL);
/* Изменение диспозиции сигнала SIGUSR1 */
act2.sa_handler = SIG_IGN;        /* Игнорировать сигнал */
sigaction(SIGUSR1, &act2, NULL);
/* Бесконечный цикл */
for (;;)
    pause ();
}

```

В данном примере структура *act* типа *sigaction* определена как *static*. Поэтому при инициализации структуры все ее поля (в том числе *sa\_flags*) обнуляются. Используемый в программе системный вызов *pause* приостанавливает ее выполнение до прихода в процесс любого сигнала.

Для того чтобы проверить работу этой и других программ, обрабатывающих сигналы, надо уметь эти сигналы генерировать. Используя команду *shell – kill*, можно имитировать любой источник выдачи сигналов. Допустим, что мы запускаем приведенную выше программу (предварительно откомпилировав ее) на выполнение. Тогда, используя для генерации сигналов команду *kill*, мы получим на экране, возможно, следующее:

```

$ ./a.out &
284767                                pid порожденного процесса
$ kill -SIGINT 284767
Получен сигнал SIGINT                сигнал SIGINT перехвачен
$ kill -SIGUSR1 284767
сигнал SIGUSR1                        сигнал SIGUSR1
и игнорируется
$ kill 284767
сигнал SIGTERM вызывает              завершение процесса

```

**Проверьте** работу приведенной выше программы.

### 9.7.3. Наборы сигналов

Каждый такой набор представляет собой список сигналов, который используется для выполнения системных вызовов. Набор сигналов определяется в программе с помощью типа *sigset\_t*, определенного в файле *<signal.h>*. Размер этого типа таков, что в нем может поместиться весь набор сигналов, определенных для данной системы.

Задать требуемый набор сигналов можно с помощью одного из двух противоположных способов. В первом из них сначала с помощью процедуры ***sigfillset*** формируется полный набор, включающий все сигналы, который затем урезается до требуемого набора с помощью процедуры ***sigdelset***. Описания процедур:

```
#include <signal.h>
int sigfillset (sigset_t *set);
int sigdelset (sigset_t *set, int signo);
```

Первый параметр обеих процедур (*set*) представляет собой указатель на требуемый набор сигналов. Вторым параметром процедуры ***sigdelset*** (*signo*) есть номер сигнала, исключаемого из набора *set*. Нетрудно заметить, что многократное применение процедуры ***sigdelset*** позволяет получить из полного набора сигналов любой требуемый набор.

### Пример

В следующем фрагменте программы формируется набор сигналов, отличающийся от полного набора отсутствием сигнала ***SIGCHLD***:

```
#include <signal.h>
sigset_t mask1;
. . . . .
sigfillset(&mask1);
sigdelset(&mask1, SIGCHLD);
. . . . .
```

Во втором подходе сначала с помощью процедуры ***sigemptyset*** формируется пустой набор, не включающий ни одного сигнала, который затем расширяется до требуемого с помощью процедуры ***sigaddset***. Описания процедур:

```
#include <signal.h>
int sigemptyset (sigset_t *set);
int sigaddset (sigset_t *set, int signo);
```

Вторым параметром процедуры ***sigaddset*** есть номер сигнала, включаемого в набор *set*. Многократное применение данной процедуры позволяет получить из пустого набора сигналов любой требуемый набор.

### Пример

В следующем фрагменте программы формируется набор из двух сигналов — ***SIGINT*** и ***SIGQUIT***:

```
#include <signal.h>
sigset_t mask2;
. . . . .
sigemptyset(&mask2);
sigaddset(&mask2, SIGINT);
sigaddset(&mask2, SIGQUIT);
. . . . .
```

Одно из наиболее полезных применений наборов сигналов — блокирование обработки сигналов на время выполнения участка программы. Такое блокирование предназначено для обеспечения целостности данных программы путем временного запрета обработки сигналов, которая может повредить эти данные. (Нетрудно заметить аналогию с запретом прерываний в однопрограммной системе.) Ранее мы рассмотрели блокирование сигналов на время выполнения обработчика сигнала. Теперь рассмотрим более универсальный подход, применимый для защиты любого участка программы (а точнее — обрабатываемых этим участком данных).

Сущность этого подхода заключается в том, что перед началом защищаемого участка, а также сразу после его завершения программа выполняет системный вызов *sigprocmask*, определенный следующим образом:

```
#include <signal.h>
int sigprocmask (int how, const sigset_t *set, sigset_t *oset);
```

Первый параметр (*how*) задает тип действия этого вызова. Например, значение *SIG\_SETMASK* приведет к блокированию с момента выполнения вызова всех сигналов, заданных вторым параметром (*set*). А значение *SIG\_UNBLOCK*, наоборот, отменяет блокирование сигналов, заданных вторым параметром. Третий параметр задает текущую маску блокируемых сигналов. Если эта маска нас не интересует, то в качестве этого параметра помещается *NULL*.

### Пример

В следующей программе выполнено блокирование всех сигналов за исключением *SIGINT* и *SIGQUIT*:

```
#include <signal.h>
main ()
{
    sigset_t set1;
    sigfillset (&set1);    /* Создать полный набор сигналов */
```

```

sigdelset (&set1, SIGINT); /* Удалить из набора SIGINT */
sigdelset (&set1, SIGQUIT); /*      --/--      SIGQUIT */
sigprocmask (SIG_SETMASK, &set1, NULL);
. . . . . /* Критический участок кода */
sigprocmask (SIG_UNBLOCK, &set1, NULL);
}

```

**В ы п о л н и т е** данную программу, заменив точки бесконечным циклом. Проверьте ее реакцию на различные сигналы.

#### 9.7.4. Сеансы и группы процессов

Рассмотрим системные вызовы, позволяющие процессу определять идентификаторы своих сеанса и группы, а также создавать новые их экземпляры. Идентификатор сеанса можно узнать с помощью функции **getsid**:

```

#include <sys/types.h>
#include <unistd.h>
pid_t getsid(pid_t pid);

```

Единственный параметр функции (*pid*) представляет собой номер того процесса, для которого требуется определить номер сеанса. Искомый процесс может не совпадать с процессом, содержащим функцию **getsid**. Единственное требование: оба процесса должны принадлежать к одному сеансу.

Создание нового сеанса выполняет функция **setsid**:

```

#include <sys/types.h>
#include <unistd.h>
pid_t setsid(void);

```

Новый сеанс создается лишь при условии, что процесс уже не является лидером какого-то сеанса. В случае успеха процесс становится лидером сеанса и лидером новой группы.

Для определения идентификатора своей группы процесс должен выполнить системный вызов **getpgrp**:

```

#include <sys/types.h>
#include <unistd.h>
pid_t getpgrp(void);

```

Системный вызов **setpgid** позволяет процессу стать членом существующей группы или создать новую группу:

```

#include <sys/types.h>
#include <unistd.h>

```

```
int setpgid(pid_t pid, pid_t pgid);
```

Данная функция может быть применена процессом по отношению к самому себе или к своему потомку при условии, что потомок не выполнил вызов *exec*, запускающий на выполнение другую программу. В результате выполнения функции *setpgid* процесс *pid* принадлежит к группе с идентификатором *pgid*.

Если значения *pid* и *pgid* равны, то создается новая группа с идентификатором *pgid*, а процесс становится лидером этой группы. При этом так как идентификатор *pid* уникален в пределах всей системы, то будет уникален и идентификатор группы.

**О п р е д е л и т е** идентификаторы сеансов и групп для своих ранее созданных процессов. Запустите процесс, создающий новый сеанс и новую группу.

## 9.7.5. Посылка сигналов другим процессам

Процесс может послать любой сигнал любому процессу или группе процессов, принадлежащих этому же пользователю. Для этого он должен воспользоваться системным вызовом *kill* (не путать с одноименной командой *shell*):

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig);
```

Первый параметр (*pid*) определяет процесс (процессы), которому (которым) будет направлен сигнал *sig*. Основные варианты его задания:

1) если *pid* положителен, то сигнал посылается процессу с идентификатором *pid*;

2) если *pid* равен нулю, то сигнал посылается всем процессам, принадлежащим к той же группе, что и процесс-отправитель (включая и его самого);

3) если *pid* равен (-1), то сигнал посылается всем процессам, принадлежащим тому же пользователю, что и процесс-отправитель (включая и его самого);

4) если *pid* меньше нуля, но не равен (-1), то сигнал посылается всем процессам, принадлежащим к той группе, идентификатор которой равен абсолютному значению *pid*.

Так как при применении *kill* обычно требуется знание *pid* конкретного процесса, то передача сигналов обычно производится



между близкими родственниками, например между процессом-отцом и дочерним процессом. Если процесс пытается послать сигнал процессу, принадлежащему другому пользователю, то вызов *kill* возвратит значение (–1). Несмотря на то что данный вызов позволяет передавать любые сигналы, на практике обычно передаются лишь сигналы *SIGUSR1*, *SIGUSR2*, так как передача с помощью *kill* других сигналов может внести путаницу в обработку таких же сигналов, поступающих от их настоящих источников.

### Пример

Допустим, что мы хотим скоординировать деятельность двух процессов (процесса-отца и дочернего процесса) по выводу своих сообщений на экран так, чтобы на экране эти сообщения чередовались. Для достижения этого процессы будут обмениваться однотипными сигналами *SIGUSR1*. При этом каждый процесс не производит вывод на экран до тех пор, пока не получит сигнал от другого процесса.

Программа:

```
#include <unistd.h>
#include <signal.h>
int ntimes = 0;
/* Функция-обработчик сигнала SIGUSR1 в процессе-отце */
void p_action (int sig)
{
    printf("Процесс-отец получил сигнал #%d\n", ++ntimes);
}
/* Функция-обработчик сигнала SIGUSR1 в дочернем
процессе */
void c_action (int sig)
{
    printf("Дочерний процесс получил сигнал #%d\n", ++
ntimes);
}
main()
{
    pid_t pid, ppid;
    static struct sigaction pact, cact;
    /* Изменение диспозиции сигнала SIGUSR1 в процессе-
отце */
    pact.sa_handler = p_action ; /* Запись адреса обработчика */
    sigfillset(&(pact.sa_mask)); /* Создать полный набор
сигналов */
```

```

sigaction(SIGUSR1, &pact, NULL);
switch (pid = fork()) {
    case -1:
        perror("Ошибка");
        exit(1);
    case 0:
        /* Дочерний процесс */
        /* Изменение диспозиции сигнала SIGUSR1 в дочернем
        процессе */
        cact.sa_handler = c_action; /* Запись адреса
        обработчика */
        sigfillset(&(cact.sa_mask)); /* Создать полный набор
        сигналов */
        sigaction(SIGUSR1, &cact, NULL);
        /* Получение идентификатора процесса-отца */
        ppid = getppid();
        /* Бесконечный цикл */
        for (;;)
        {
            sleep(1);
            kill(ppid, SIGUSR1);
            pause ();
        }
        default:
            /* Процесс-отец */
            /* Бесконечный цикл */
            for (;;)
            {
                pause ();
                sleep(1);
                kill(pid, SIGUSR1);
            }
        }
    }
}

```

Поочередный вывод сообщений процессами на экран будет продолжаться до тех пор, пока на клавиатуре не будет нажата одна из комбинаций клавиш прерывания процесса.

**Проверьте** правильность выполнения приведенной программы.

### 9.7.6. Сигнал таймера

Несмотря на то что вызов *kill* может использоваться процессом для выдачи сигналов самому себе, польза от таких сигналов будет

невелика. Намного полезнее для внутрипроцессного использования системный вызов *alarm*, который устанавливает для процесса *таймер интервала*. Вызов *alarm* выполняет запись в эту переменную первоначального значения, определяющего требуемый интервал времени. По истечении этого времени обработчик прерывания аппаратного таймера (не следует путать с таймером интервала) выдает в процесс сигнал *SIGALRM*. Определение вызова *alarm*:

```
#include <unistd.h>
unsigned int alarm(unsigned int sec);
```

Параметр *sec* задает время в секундах, на которое устанавливается таймер. В отличие от вызова *sleep*, вызов *alarm* не приостанавливает выполнение процесса. До тех пор пока не придет сигнал *SIGALRM*, никакого воздействия на это выполнение не будет. Выключить таймер можно при помощи вызова *alarm* с нулевым параметром.

Особенно полезно применение вызова *alarm* для ограничения времени выполнения какой-то операции. При этом, сделав вызов *alarm*, процесс начинает выполнение операции. Если она завершается вовремя, то таймер сбрасывается. Иначе — с помощью сигнала *SIGALRM* процесс прерывается и выполняет корректирующее действие.

### 9.7.7. Задание

Разработать одну из следующих программ (по указанию преподавателя).

1. Процесс-отец порождает четыре дочерних процесса, каждый из которых выполняет бесконечный цикл. Далее в течение 10 секунд процесс-отец выводит на экран какое-то сообщение. По истечении этого времени он уничтожает два процесса из четырех, используя для этого всего одну команду. Перед завершением процесс-отец выводит на экран перечень процессов данного пользователя.

2. Процесс-отец порождает четыре дочерних процесса, каждый из которых выполняет бесконечный цикл. При этом каждый из дочерних процессов особым образом реагирует на сигнал *SIGINT*:

— процесс 1 при получении сигнала *SIGINT* выводит сообщение на экран и продолжается;

– процесс 2 обрабатывает сигнал *SIGINT* в бесконечном цикле, выдавая свое сообщение на экран. Данный цикл защищен от воздействия сигнала *SIGQUIT*;

– вся программа процесса 3 защищена от воздействия сигнала *SIGINT*;

– перед входом процесса 4 в бесконечный цикл для него меняется идентификатор сеанса.

Сразу после порождения дочерних процессов процесс-отец завершается.

Далее следует проверить реакцию оставшихся процессов на сигналы *SIGINT* и *SIGQUIT*.

3. Процесс-отец открывает существующий текстовый файл, а затем порождает два дочерних процесса, которые по очереди выводят содержимое этого файла фиксированными порциями по 10 символов, предвеля каждый вывод номером своего процесса. Вывод на экран заканчивается или при достижении конца файла, или по истечении интервала времени в 10 секунд.

*Примечание.* Для получения номера своего процесса можно использовать функцию *getpid*:

```
#include <sys/types.h>
pid_t getpid(void);
```

## 9.8. Лабораторная работа № 8.

### Управление терминалом

Целью выполнения настоящей лабораторной работы является получение навыков по программному управлению терминалом.

#### 9.8.1. Функции терминальной линии

Напомним, что терминал — устройство, выполняющее ввод и вывод символьной информации. Чаще всего это совокупность клавиатуры (устройства ввода) и экрана (устройства вывода), но могут использоваться и другие символьные устройства. Подобно другим периферийным устройствам, терминал представлен в файловой структуре системы специальным файлом в каталоге */dev*. Примеры имен таких файлов: */dev/console*, */dev/tty03*, */dev/tty*. Последнее имя является «собираательным», позволяя в любой программе обращаться к своему управляющему терминалу стандартным образом.

Если пользователь использует не терминал системы *UNIX*, а работает на терминале другой ЭВМ, соединенной с *UNIX*-системой каналами связи, то *UNIX* предоставляет в распоряжение такого пользователя псевдотерминал, который с точки зрения пользователя и его программных процессов в *UNIX*-системе выглядит точно так же, как и реальный терминал. Поэтому сказанное далее о терминалах справедливо также и для псевдотерминалов.

**Драйвер терминала** — программный модуль ядра ОС, выполняющий низкоуровневое программное управление терминалом подобно обычному строковому драйверу, обеспечивающему обмен символьной информацией между программным процессом и устройством (а точнее — с его интерфейсным устройством (контроллером)). Перечислим дополнительные функции программного управления терминалом, которые не обеспечивает драйвер терминала:

1) вывод на экран терминала «эха» символов, вводимых с клавиатуры. «Эхо» — изображение на экране той клавиши, которая была только что нажата. Наличие данной функции обусловлено тем, что устройство ввода (клавиатура) и устройство вывода (экран) никак не связаны между собой на аппаратном уровне;

2) предоставление возможности пользователю выполнять простейшее редактирование информации, вводимой с клавиатуры. Примером является удаление символов, расположенных левее или правее позиции, отмеченной курсором;

3) выдача сигналов (см. лабораторную работу № 7) при нажатии пользователем определенных клавиш или их комбинаций.

Для выполнения перечисленных функций между процессом и драйвером терминала включается дополнительный программный модуль, называемый **дисциплиной линии**. Данное название обусловлено тем, что данный модуль совместно с драйвером терминала образует **терминальную линию** — канал связи между процессом и терминалом. На рис. 68 приведена упрощенная логическая схема терминальной линии, отражающая только информационные потоки (управляющие взаимосвязи опущены).

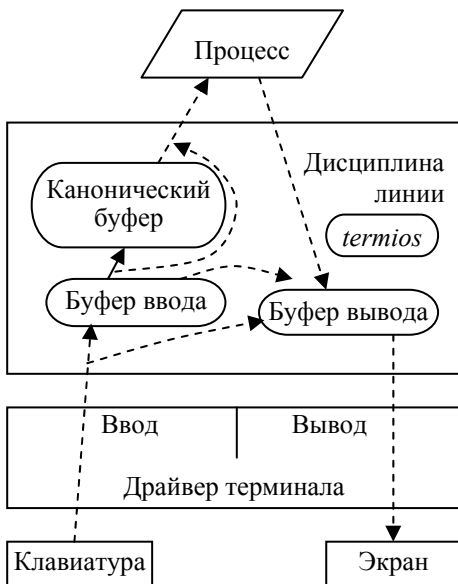


Рис. 68. Информационные потоки в терминальной линии

Дисциплина линии имеет три внутренних буфера, два из которых (буфер ввода и буфер вывода) предназначены для посимвольного обмена с драйвером. Длина буфера ввода достаточна для размещения одной экранной строки. Буфер вывода может быть большего размера, что позволит разместить в нем несколько экранных строк. Этот буфер организован в виде очереди: чем раньше символ помещен в очередь, тем раньше он будет извлечен из нее драйвером для вывода на экран. Что касается третьего буфера (канонического буфера), то он также организован в виде очереди, элементами которой являются не отдельные символы, а целые строки символов.

Дисциплина линии (и вся терминальная линия) может работать в двух режимах: каноническом и неканоническом. В **каноническом режиме**, во-первых, дисциплина линии помещает каждый вводимый с клавиатуры (с помощью драйвера) символ одновременно в буфер ввода и в буфер вывода. При этом запись в очередь вывода обеспечивает вывод «эха» символа. Во-вторых, наблюдая на экране результаты своей работы на клавиатуре, пользователь выполняет редактирование введенной последовательности символов. Остановимся на процессе редактирования более подробно.

На любом экране всегда присутствует изображение курсора. **Курсор** — светящийся прямоугольник, предназначенный для того чтобы указывать на ту позицию экрана, в которой будет показан следующий символ, выбранный драйвером из очереди вывода. Курсор генерируется аппаратурой экрана, а его координаты задаются драйвером. Пользователь перемещает курсор в пределах экранной строки с помощью клавиш <←> и <→>. Чем больше раз пользователь нажмет соответствующую клавишу, тем на большее число позиций переместится курсор. Это происходит благодаря тому, что при каждом нажатии клавиши драйвер передает ее код *ASCII* в дисциплину линии, которая, во-первых, помещает код символа в буфер вывода, а во-вторых, корректирует свой внутренний указатель на элемент (символ) буфера ввода.

Подобно тому как на экране курсор указывает на ту позицию, в которой будет выполняться редактирование (вставка или удаление символа), для буфера ввода эту же роль выполняет внутренний указатель. Если теперь пользователь нажимает какую-то клавишу редактирования, например <Delete>, то код этой клавиши не помещается в буфер вывода (вывода «эха» нет), а используется дисциплиной линии лишь для корректировки своего буфера ввода. После этого содержимое буфера ввода, начиная с корректируемого символа, копируется в буфер вывода для отображения полученных изменений на экране.

Результатом работы дисциплины линии в каноническом режиме при вводе являются отредактированные строки символов, каждая из которых всегда заканчивается символом *nl*. (Здесь и далее без угловых скобок будем записывать обозначения символов, принятые в *UNIX* или в *СИ*.)

Что касается вывода в каноническом режиме, то дисциплина линии ограничивается лишь добавлением к каждому символу *nl* символа *carriage-return* (возврат каретки). В результате каждая выводимая строка будет начинаться с левого края экрана.

Канонический режим терминальной линии используют, например, интерпретатор команд *shell*, а также строковые текстовые редакторы *ed*, *sed*, *vi* и т.д.

В **неканоническом режиме** дисциплина линии передает вводимую с клавиатуры последовательность символов без каких-либо изменений. Такой режим работы с терминалом используют, например, экранные редакторы. Они сами обеспечивают редактирование вводимой информации. Действуя при этом по тем же принципам, что и дисциплина очереди в каноническом режиме, экранные

редакторы выполняют гораздо большее число функций редактирования.

## 9.8.2. Специальные символы редактирования и выдачи сигналов

Следующие символы, вводимые с клавиатуры, используются дисциплиной линии для редактирования введенной строки, а также для выдачи ею сигналов:

***erase*** — символ, приводящий к стиранию предыдущего символа в строке. В качестве данного символа пользователь может задать любой код *ASCII*. Чаще всего для этого используется код клавиши *<Backspace>*. Например, если в командной строке *UNIX* набрать

**\$ whp<Backspace>o<Enter>**

то интерпретатору команд будет передана строка *who*;

***kill*** — стирание всех символов до начала строки. По умолчанию для получения символа *kill* одновременно нажимаются две клавиши *<Ctrl>&<?>*. Другие используемые комбинации: *<Ctrl>&<X>* и *<Ctrl>&<U>*;

***nl*** — обычный разделитель строк. Он всегда имеет значение символа ***line-feed*** (перевод строки). Этот же код *ASCII* имеет символ *CH newline* (новая строка);

***stop*** — используется для временной приостановки вывода на терминал. Это позволяет приостановить вывод прежде, чем выводимый текст исчезнет за границей экрана. Обычно используется комбинация клавиш *<Ctrl>&<S>*. Лишь в некоторых системах может быть изменен пользователем;

***start*** — используется для продолжения вывода, приостановленного символом *stop*. Если *stop* не был введен, то *start* игнорируется. Для получения символа *start* обычно используется *<Ctrl>&<Q>*. Лишь в некоторых системах может быть изменен пользователем;

***eof*** — окончание входного потока с терминала (этот символ должен быть единственным символом в начале новой строки). Стандартным значением является символ *ASCII eot*, получаемый нажатием *<Ctrl>&<D>*;

***intr*** — символ прерывания. Ввод данного символа пользователем приводит к послке всем процессам оперативной группы сеанса, управляемого данным терминалом, сигнала *SIGINT*. Стандартная реакция процесса на такой сигнал — завершение. Обыч-



но символ *intr* соответствует нажатию клавиши *<Delete>* или *<Ctrl>&<C>*;

*quit* — приводит к послыке всем процессам оперативной группы сеанса, управляемого данным терминалом, сигнала *SIGQUIT*. Обычно символ *quit* соответствует нажатию клавиш *<Ctrl>&<Q>*;

*susp* — символ терминального останова. Ввод данного символа пользователем приводит к послыке всем процессам оперативной группы сеанса, управляемого данным терминалом, сигнала *SIGTSTP*. Стандартная реакция процесса на этот сигнал — переход процесса в состояние «Останов», в которое процесс может попасть из состояний «Задача», «Готов» и «Сон». («Останов» — дополнительное состояние процесса, существующее не во всех версиях *UNIX*.) Кроме того, оперативная группа процессов переводится в фоновый режим. Обычно символ *susp* соответствует нажатию клавиш *<Ctrl>&<Z>*.

С помощью утилиты *stty* пользователь может заменять клавиши (и соответствующие им коды), используемые для реализации управляющих символов: *erase*, *kill*, *eof*, *intr*, *quit*, *susp*. Пример:

```
$ stty erase "^f"
```

Здесь в качестве символа *erase* задается комбинация клавиш *<Ctrl>&<f>*. Обратите внимание, что и в других случаях часто вместо нажатия *<Ctrl>&<f>* можно набрать строку *"^f"*.

**З а м е н и т е** с помощью утилиты *stty* некоторые символы редактирования и выдачи сигналов. Проверьте результат такой замены.

### 9.8.3. Управляющая структура *termios*

Утилита *stty*, изменяющая кодировку управляющих символов, вносит изменения в управляющую структуру (дескриптор) дисциплины линии — *termios*. Аналогичные изменения можно выполнять и из программы процесса.

Для запоминания текущего состояния *termios* и для записи ее нового содержимого могут использоваться системные вызовы *tcgetattr* и *tcsetattr*:

```
#include <termios.h>
int tcgetattr(int ttyfd, struct termios *tsaved);
int tcsetattr(int ttyfd, int actions, const struct termios *tnew);
```

Первый вызов сохраняет текущее состояние терминала, которому соответствует номер файла *tyfd*, в структуре *tsaved*. Второй вызов установит новое состояние терминала, заданное структурой *tnv*. При этом второй параметр этого вызова *actions* уточняет момент изменения атрибутов терминала. В файле *<termios.h>* определены возможные варианты этого параметра:

- 1) *TCSANOW* — немедленное изменение атрибутов терминала;
- 2) *TCSADRAIN* — изменение атрибутов сразу же после опустошения очереди вывода;
- 3) *TCSAFLUSH* — изменение атрибутов после опустошения очередей вывода и ввода.

Структура *termios* определена в файле *<termios.h>* и содержит:

```

tcflag_t  c_iflag;          /* Режим ввода */
tcflag_t  c_oflag;          /* Режим вывода */
tcflag_t  c_cflag;          /* Управляющий режим */
tcflag_t  c_lflag;          /* Режим дисциплины линии */
cc_t      c_cc[NCCS];       /* Управляющие символы */

```

*c\_iflag* — поле, которое служит для общего управления вводом с терминала. Каждый бит этого поля представляет собой флаг, задающий какую-то особенность этого ввода.

Три флаговые константы управляют обработкой входного символа *cr* (возврат каретки):

***INLCR*** — преобразовать символ *lf* (перевод строки) в *cr*;

***IGNCR*** — игнорировать *cr*;

***ICRNL*** — преобразовать символ *cr* в символ *lf*.

Заметим, что сама *UNIX* ожидает в качестве последнего символа строки символ *lf*. Поэтому обычно используется константа *ICRNL*.

Следующие три константы позволяют «тормозить» символьный обмен с терминалом в том случае, если пользователь (при выводе) или программный процесс (при вводе) не успевает обработать поступающую информацию:

***IXON*** — разрешить старт-стопное управление выводом. То есть для временной приостановки вывода на терминал пользователь должен ввести символ *stop* (комбинация клавиш *<Ctrl>&<S>*). Для продолжения вывода вводится символ *start* — комбинация клавиш *<Ctrl>&<Q>*;

***IXANY*** — продолжать вывод при нажатии любого символа. Эта константа дополняет предыдущую, позволяя использовать для продолжения вывода вместо символа *start* любой символ;

***IXOFF*** — разрешить старт-стопное управление вводом. Если установлен соответствующий флаг, система сама посылает терминалу символ *stop* в том случае, если заполнен буфер ввода. Когда в этом буфере появится место, терминалу будет передан символ *start*.

Каждая из перечисленных флаговых констант имеет такую же длину, как и поле *c\_iflag*. Причем только один бит (флаг) установлен в 1. Все остальные биты флаговой константы сброшены в 0. Поэтому для установки какого-либо флага в поле *c\_iflag* достаточно выполнить побитовое логическое сложение этого поля с соответствующей флаговой константой. Например, пусть *tdes* — структура типа *termios*. Тогда для игнорирования символа *cr* достаточно записать оператор:

***tdes.c\_iflag |= IGNCR***

Для сброса требуемого флага в поле *c\_iflag* достаточно выполнить побитовое логическое умножение этого поля с инверсией соответствующей флаговой константой. Например, для отмены игнорирования символа *cr* достаточно записать оператор:

***tdes.c\_iflag &= ~IGNCR***

***c\_oflag*** — поле, которое служит для общего управления выводом на терминал. Каждый бит этого поля представляет собой флаг, задающий какую-то особенность этого вывода. Некоторые из флаговых констант:

***OPOST*** — содержит наиболее важный флаг в этом поле. Если он сброшен, то выводимые символы передаются без изменений. Иначе символы подвергаются преобразованию, заданному остальными флагами этого поля;

***ONLCR*** — преобразовать символ перевода строки *lf* в символ возврата каретки *cr* и символ *lf*;

***OCRNL*** — преобразовать символ *cr* в символ *lf*.

***c\_cflag*** — поле, содержащее параметры, управляющие портом терминала: скорость ввода-вывода, контроль четности и т.д. Обычно эти параметры задаются самой *UNIX*.

***c\_lflag*** — поле, задающее режим дисциплины линии. Оно содержит флаги, задаваемые следующими константами:

***ICANON*** — канонический построчный ввод. Если флаг сброшен, то неканонический ввод;

***ISIG*** — разрешить обработку символов прерываний (*intr* и *quit*). Если флаг сброшен, то при получении этих символов сигналы

процессам не посылаются, а сами символы *intr* и *quit* передаются в программу без изменений;

**IEXTEN** — разрешить дополнительную, зависящую от реализации обработку вводимых символов;

**ECHO** — разрешить отображение вводимых символов на экране;

**ECHOE** — отображать символ удаления *erase* как «*erase*–пробел–*erase*». В результате курсор сначала переместится на одну позицию влево, во-вторых, стоящий в этой позиции символ будет затерт пробелом (при этом курсор вернется на соседнюю позицию вправо), а в-третьих, курсор опять переместится на соседнюю позицию слева, указывая таким образом на первую свободную позицию;

**TOSTOP** — посылать сигнал *SIGTTOU* при попытке вывода фонового процесса.

**c\_cc** — массив, содержащий специальные символы редактирования и выдачи сигналов, рассмотренные ранее. Каждый символ занимает в массиве *c\_cc* позицию, задаваемую соответствующей константой из файла *<termios.h>*:

**VERASE** — символ стирания *erase*;

**VKILL** — символ удаления строки *kill*;

**VSTOP** — символ остановки передачи данных *stop*;

**VSTART** — символ продолжения передачи данных *start*;

**VEOF** — символ конца файла *eof*;

**VINTR** — символ прерывания *intr*;

**VQUIT** — символ завершения *quit*;

**VSUSP** — символ временной приостановки выполнения *susp*.

### Пример

Следующий фрагмент программы изменяет значение символа *quit* для терминала, выполняющего стандартный ввод (номер файла — 0):

```
struct termios tdes;
/* Получение настроек терминала */
tcgetattr(0, &tdes);
tdes.c_cc[VQUIT] = "\u001b"; /* CTRL-Y */
/* Изменение настроек терминала */
tcsetattr(0, TCSAFLUSH, &tdes);
```

## 9.8.4. Задание

Требуется разработать программу, которая выполняет следующие действия:

1) задает старт-стопное управление выводом на экран. Причем для продолжения вывода может использоваться любой символ;

2) задает новые значения (по вашему выбору) символов стирания *erase*, удаления строки *kill*, прерывания *intr*, временной приостановки выполнения *susp*.

Проверить произведенные установки: а) с помощью утилиты *cat* выполнить вывод на экран достаточно большого текстового файла; б) выполнить редактирование строки символов, используя символы *erase* и *kill*; в) из командной строки запустить один оперативный и один фоновый процесс, выполняющие бесконечные циклы, а затем проверить на них действие сигналов *intr* и *susp*.

*Примечание.* После выполнения перечисленных действий требуется восстановить прежнее состояние терминала, так как иначе другие процессы «не поймут» новые настройки терминала.

## 9.9. Лабораторная работа № 9. Датаграммные локальные каналы

Целью выполнения настоящей лабораторной работы является получение навыков создания и использования локальных датаграммных информационных каналов между процессами при использовании метода сокетов.

### 9.9.1. Подготовка к выполнению работы

В начале выполнения данной работы следует ознакомиться со следующими вопросами из теоретической части пособия:

1) понятие информационного канала и его характеристики (п. 2.5.1);

2) сокет (п. 4.6.3) — понятие сокета; имена сокетов; системные вызовы для реализации датаграммного канала;

3) понятие протокола (подразд. 3.4).

### 9.9.2. Порядок выдачи системных вызовов

В качестве простейшего примера создания и использования локального датаграммного канала рассмотрим получение двумя процессами «эха» символьной строки. В этом примере процесс-

клиент посылает другому процессу-серверу любую символьную строку. Далее сервер посылает эту строку обратно клиенту. Получив символьную строку, процесс-клиент выводит ее на экран, сопроводив этот вывод пояснительной надписью. На рис. 69 приведен возможный порядок выдачи системных вызовов во времени.

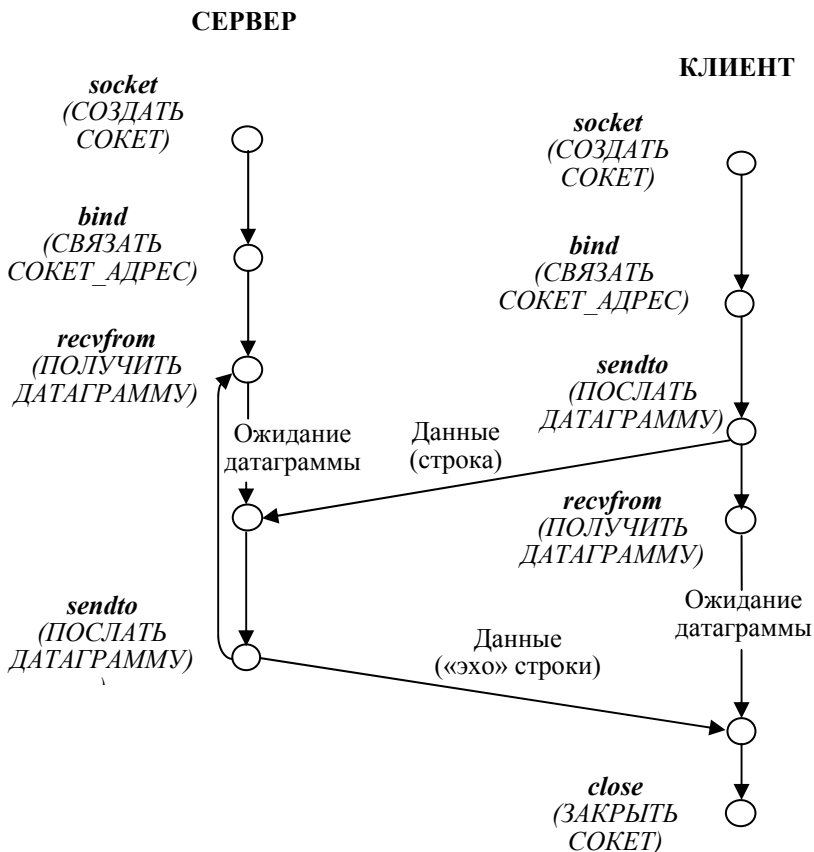


Рис. 69. Пример создания и использования датаграммного канала

### 9.9.3. Создание сокета

Создание сокета выполняет процесс-«владелец», используя системный вызов *socket*. Его определение:

```
#include <sys/types.h>
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
```

Все три параметра этого системного вызова определяют характеристики будущего информационного канала, при образовании которого будет использоваться создаваемый сокет:

*domain* — определяет область применения создаваемого канала: **AF\_UNIX** — процессы, соединяемые каналом, выполняются на одной ЭВМ; **AF\_INET** — соединяемые процессы выполняются на удаленных ЭВМ, подключенных к сети *Internet*;

*type* — определяет устойчивость создаваемого канала: **SOCK\_STREAM** — виртуальное соединение; **SOCK\_DGRAM** — передача датаграмм;

*protocol* — тип протокола, которому должны следовать подпрограммы ядра, обслуживающие сокет, расположенные по обоим концам информационного канала. Обычно в качестве данного параметра задают 0, что эквивалентно выбору протокола самим ядром.

Если системный вызов *socket* завершился с ошибкой, то он возвращает значение (−1). Иначе в результате своего выполнения *socket* возвращает номер нового сокета, который может использоваться далее процессом-«владельцем» во всех операциях с данным сокетом (и с соответствующим информационным каналом).

**Пример** создания сокета:

```
int sock;
if ((sock = socket(AF_UNIX, SOCK_DGRAM, 0)) < 0)
{
    printf("Невозможно создать сокет\n"); exit(1);
}
```

#### 9.9.4. Связывание сокета со своим адресом

Только что созданный сокет еще не имеет адреса (внешнего имени). Для получения такого адреса необходимо выполнить системный вызов *bind*:

```
#include <sys/types.h>
#include <sys/socket.h>
int bind(int sockfd, struct sockaddr *address, int add_len);
```

где *sockfd* — номер сокета, созданного с помощью вызова *socket*; *address* — адрес (указатель) структуры типа *sockaddr*; *add\_len* — длина структуры типа *sockaddr*.

В случае успешного завершения вызова *bind* он возвращает значение 0, иначе (−1).

Структура *sockaddr* — структура обобщенного адреса сокета, которая определяется в файле *<sys/socket.h>*:



```

struct sockaddr {
    sa_family_t sa_family; /* Область применения канала */
    char sa_data[];        /* Внешнее имя сокета */
};

```

При выполнении программой процесса конкретного вызова *bind* структура *sockaddr* заменяется или структурой *sockaddr\_un*, используемой при внутримашинном обмене, или структурой *sockaddr\_in*, используемой при взаимодействии удаленных процессов. Структура *sockaddr\_un* определяется в файле *<sys/un.h>*:

```

struct sockaddr_un {
    sa_family_t sun_family; /* ==AF_UNIX */
    char sun_path[108];     /* Имя-путь файла-сокета */
};

```

Применение вызова *bind* в программе зависит от роли, которую играет данная программа среди взаимодействующих программ. Адрес сокета, принадлежащего программе-серверу, всегда известен заранее, для того чтобы программы-клиенты могли посылать свои запросы по этому адресу. Пример выполнения вызова *bind* в сервере:

```

#define LEN sizeof(struct sockaddr_un)
struct sockaddr_un server;
int sock, s_len;
. . . . . /* Создание сокета с номером
                                sock*/
/* Задание адреса своего сокета */
unlink("/home/vlad/abc.server");
bzero(&server, LEN);
server.sun_family = AF_UNIX;
strcpy(server.sun_path, "/home/vlad/abc.server");
s_len=sizeof(server.sun_family)+strlen(server.sun_path);
/* Связывание сокета сервера с его адресом */
if (bind(sock, (struct sockaddr *) &server, s_len) < 0)
{
    printf("Ошибка связывания сокета с адресом\n");
    exit(1);
}

```

В приведенном фрагменте программы производится связывание адреса (имени-пути */home/vlad/abc.server*) сокета с номером этого сокета *sock*. Возможно, ранее это имя-путь уже использовалось для создания файла или сокета. Поэтому в начале данного фрагмента производится уничтожение файла (сокета), созданного

прежде. Далее структура *server* адреса сокета сначала обнуляется, а затем заполняется. После этого определяется длина этой структуры *s\_len* и выполняется вызов *bind*.

В отличие от сервера, имя-путь сокета клиента может быть заранее не известно и может быть выбрано клиентом в известной степени произвольно. Единственное требование — в пределах системы имя-путь сокета должно быть уникальным. Для получения уникального внешнего имени сокета удобно использовать библиотечную функцию *mktemp*, которая, получая на входе шаблон имени-пути файла, получает уникальное имя-путь на основе номера текущего процесса. Например, для шаблона */tmp/clnt.XXX* производится замена трех символов «X». Соответствующий фрагмент программы-клиента:

```
#define LEN sizeof(struct sockaddr_un)
struct sockaddr_un client;
int sock, c_len;

. . . . .
/* Создание сокета с номером sock */
/* Задание адреса своего сокета */
bzero(&client, LEN);
client.sun_family = AF_UNIX;
strcpy(client.sun_path, "/tmp/clnt.XXX");
mktemp(client.sun_path);
c_len= sizeof(client.sun_family) + strlen(client.sun_path);
/* Связывание сокета клиента со своим адресом */
if (bind(sock, (struct sockaddr *) &client, c_len) < 0)
{
    printf("Ошибка связывания сокета с адресом\n");
    exit(1);
}
```

### 9.9.5. Прием и передача датаграмм

Выполнение вызовов *socket* и *bind* каждым из двух взаимодействующих процессов приведет к созданию двух сокетов, каждый из которых имеет как локальное, так и внешнее имя (адрес). Этого достаточно для создания датаграммного информационного канала. Пользуясь этим каналом, процесс может отправлять свои датаграммы другому процессу, а также принимать ответные датаграммы.

Для того чтобы получить датаграмму из входного буфера своего сокета, программа процесса должна выполнить системный вызов *recvfrom*:

```
#include <sys/types.h>
#include <sys/socket.h>
int recvfrom(int sockfd, void *buf, int len, int flags,
struct sockaddr *fromaddr, int *fromlen);
```

где *sockfd* — номер своего сокета; *buf* — указатель на прикладной буфер, в который ядро должно записать информацию, принятую от другого процесса; *len* — длина прикладного приемного буфера (в байтах); *flags* — флаги, задающие особые условия приема датаграммы. Если такие условия не требуются, то задается 0; *fromaddr* — указатель на структуру, содержащую адрес сокета-источника датаграммы; *fromlen* — указатель на целочисленную переменную, содержащую длину структуры, содержащей адрес сокета-источника датаграммы.

В результате выполнения вызова *recvfrom* из входного буфера сокета с номером *sockfd* извлекается датаграмма и содержащаяся в ней полезная информация помещается в прикладной буфер, а адрес сокета-отправителя датаграммы помещается в структуру, на которую указывает параметр *fromaddr*. При этом вызов *recvfrom* возвращает в программу процесса число фактически полученных байтов полезной информации, а в случае ошибки — число (–1). Таким образом, в результате успешного выполнения вызова программа получает не только полезную информацию, содержащуюся в датаграмме, но и «знает» адрес сокета-источника этой датаграммы. Используя этот адрес в качестве адреса назначения, процесс может отправить ответную датаграмму. Если в момент выполнения *recvfrom* во входном буфере сокета датаграмма отсутствует, то процесс переводится в состояние «Сон» и находится в нем до появления датаграммы.

*Примечание.* Если адрес сокета-отправителя не нужен, в качестве предпоследнего параметра вызова следует записать *NULL*, а в качестве последнего параметра — 0.

Для отправления своей датаграммы программа процесса должна выполнить системный вызов *sendto*:

```
#include <sys/types.h>
#include <sys/socket.h>
int sendto(int sockfd, const void *msg, int len, int flags,
const struct sockaddr *toaddr, int tolen);
```

где *sockfd* — номер своего сокета; *msg* — указатель на прикладной буфер, содержащий передаваемую информацию; *len* — длина передаваемой информации (в байтах); *flags* — флаги, задающие особые условия отправления датаграммы. Если такие условия не требуются, то задается 0; *toaddr* — указатель на структуру, содержащую адрес сокета назначения датаграммы; *tolen* — длина структуры, содержащей адрес сокета назначения.

В результате выполнения вызова *sendto* в выходной буфер сокета-источника помещается датаграмма, содержащая полезную информацию (копируется из прикладного буфера), а также заголовок датаграммы, состоящий из адреса назначения и адреса отправления. После этого данный вызов возвращает управление программе процесса, передав ей длину (в байтах) переданной полезной информации. (В случае ошибки возвращается (–1).) Что касается самой передаваемой датаграммы, то она перемещается ядром во входной буфер сокета назначения и находится там до тех пор, пока не будет взята оттуда процессом-потребителем датаграммы.

Из перечня входных параметров системного вызова *sendto* видно, что прежде чем его можно использовать, данная часть приложения (клиент или сервер) должна получить адрес того удаленного сокета, которому отправляется датаграмма. Существуют два способа задания такого адреса:

1) адрес сокета сервера известен всем клиентам заранее. Поэтому для задания такого адреса в программе клиента выполняются действия, очень похожие на действия по заданию адреса своего сокета, рассмотренные нами в предыдущем пункте;

2) адрес сокета клиента серверу заранее не известен. Поэтому для получения такого адреса программа сервера использует датаграмму, полученную от клиента. Напомним, что наряду с полезной прикладной информацией принятая датаграмма содержит служебную информацию — адрес отправителя, то есть адрес сокета клиента.

После того как датаграммный канал стал не нужен, клиент закрывает свой сокет, используя тот же самый системный вызов *close*, который используется для закрытия файлов:

```
int close(int sockfd);
```

где *sockfd* — номер сокета.

### 9.9.6. Программы взаимодействующих процессов

Предоставление взаимодействующим процессам датаграммного информационного канала на основе сокетов никак не регламентирует порядок выдачи датаграмм клиентом и сервером. Этот порядок полностью определяется алгоритмом совместной работы клиента и сервера, то есть прикладным протоколом.

Согласно любому прикладному протоколу первым посылает свою датаграмму клиент. Эта датаграмма содержит запрос на обслуживание и посылается на адрес сокета сервера, заранее известный клиенту. С учетом того что сервер обычно предназначен для обслуживания не одного, а многих клиентов, его алгоритм должен содержать бесконечный цикл опроса входного буфера своего сокета. Получив датаграмму, сервер может в общем случае или отвергнуть пришедший запрос, или запросить дополнительную информацию, или приступить к выполнению этого запроса. Для простоты допустим, что возможна только последняя реакция сервера.

В некоторых прикладных протоколах каждая новая датаграмма клиента обслуживается сервером независимо от ранее принятых датаграмм. В этом случае можно ограничиться однопроцессной моделью сервера, которая предполагает, что всю работу по приему датаграмм, их обслуживанию и отправлению ответных датаграмм выполняет единственный процесс-сервер. Реализация многопроцессной модели сервера целесообразна для более сложных прикладных протоколов и в настоящей работе не рассматривается. Вернемся к примеру с выводом «эха» и запишем программы однопроцессного сервера и клиента.

#### **Сервер:**

```
/* Подсоединение заголовочных файлов*/
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
/* Объявления констант и глобальных переменных*/
#define LBUF 100
#define LEN sizeof(struct sockaddr_un)
char bufer[LBUF];
main ()
{
/* Объявления локальных переменных и структур данных*/
```

```

    struct sockaddr_un server, client;
    int n, sock, s_len;
    int c_len = LEN;
/* Создание сокета сервера */
    if ((sock = socket(AF_UNIX, SOCK_DGRAM, 0)) < 0)
    {
        printf("Сервер: невозможно создать сокет\n"); exit(1);
    }
/* Задание адреса своего сокета */
    unlink("/home/vlad/abc.server");
    bzero(&server, LEN);
    server.sun_family = AF_UNIX;
    strcpy(server.sun_path, "/home/vlad/abc.server");
    s_len = sizeof(server.sun_family) + strlen(server.sun_path);
/* Связывание сокета сервера с его адресом */
    if (bind(sock, (struct sockaddr *) &server, s_len) < 0)
    {
        printf("Сервер: ошибка связывания сокета с адресом\n");
        exit(1);
    }
/* Бесконечный цикл приема и обработки датаграмм
клиентов */
    for ( ; ; ){
        n = recvfrom(sock, bufer, LBUF, 0, (struct sockaddr *)&client,
        &c_len);
        if (n < 0) { printf("Сервер: ошибка приема\n"); continue;}
/* В результате системного вызова recvfrom был получен
адрес клиента. На него и возвращается сообщение
клиента */
        if (sendto(sock, bufer, n, 0, (struct sockaddr *)&client,
        c_len) != n) {
            printf("Сервер: ошибка передачи\n"); continue;}
    }
}

```

### Клиент:

```

/* Включение заголовочных файлов */
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
/* Объявления констант и глобальных переменных */
char *msg = "С новым годом!\n";
#define LBUF 100

```

```
#define LEN sizeof(struct sockaddr_un)
char bufer[LBUF];
main ()
{
    /* Объявления локальных переменных и структур данных*/
    struct sockaddr_un server, client;
    int n, sock, s_len, c_len, msglen;
    /* Создание сокета клиента*/
    if ((sock = socket(AF_UNIX, SOCK_DGRAM, 0)) < 0)
    {
        printf("Клиент: невозможно создать сокет\n"); exit(1);
    }
    /* Заполнение адреса сокета клиента */
    bzero(&client, LEN);
    client.sun_family = AF_UNIX;
    strcpy(client.sun_path, "/tmp/clnt.XXX");
    mktemp(client.sun_path);
    c_len= sizeof(client.sun_family) + strlen(client.sun_path);
    /* Связывание сокета клиента со своим адресом */
    if (bind(sock, (struct sockaddr *) &client, c_len) < 0)
    {
        printf("Клиент: ошибка связывания сокета с адресом\n");
        exit(1);
    }
    /* Заполнение адреса сокета сервера */
    bzero(&server, LEN);
    server.sun_family = AF_UNIX;
    strcpy(server.sun_path, "/home/vlad/abc.server");
    s_len=sizeof(server.sun_family)+strlen(server.sun_path);
    /* Отправление запроса на обслуживание к серверу*/
    msglen = strlen(msg);
    if (sendto(sock, msg, msglen, 0,
        (struct sockaddr *)&server, s_len) != msglen) {
        printf("Клиент: ошибка передачи\n"); exit(1);}
    /* Прием от сервера «эха» сообщения*/
    if((n = recvfrom(sock, bufer, LBUF, 0, NULL, 0)) < 0)
    { printf("Клиент: ошибка приема\n"); exit(1);}
    /* Вывод «эха» на экран*/
    printf("Эхо: %s\n", bufer);
    /* Завершение программы */
    close(sock);
    unlink(client.sun_path);
    exit(0);
}
```

}

### 9.9.7. Задание

Требуется разработать три процесса, запускаемые из командной строки *UNIX*: процесс-сервер, запускаемый в оперативном режиме, и два процесса-клиента, запускаемые в фоновом режиме. Между этими процессами должно существовать датаграммное информационное взаимодействие, обеспечивающее решение одной из следующих задач, указываемой преподавателем:

1) сервер экрана. Каждый клиент выводит содержимое своего текстового файла на экран, находящийся под управлением процесса-сервера. Размер каждого файла соответствует нескольким экранным строкам, каждая из которых передается в виде одной датаграммы. Причем вывод каждой такой строки на экран сервер предваряет выводом номера клиента;

2) сервер клавиатуры. Каждый клиент вводит содержимое своего текстового файла с клавиатуры, находящейся под управлением процесса-сервера. При этом каждая датаграмма сервера соответствует одной введенной строке текста. Размер каждого клиентского файла многократно превосходит размер одной строки-датаграммы;

3) сервер файла для записи. Каждый клиент выводит содержимое своего текстового файла в файл на диске, находящийся под управлением процесса-сервера. При этом каждая датаграмма клиента соответствует одной строке файла. Размер каждого клиентского файла превосходит размер одной строки-датаграммы. Запись каждой такой датаграммы в свой файл сервер предваряет записью номера клиента;

4) сервер файла для чтения. Каждый клиент выполняет запись в свой текстовый файл на диске информации из файла, находящегося под управлением процесса-сервера. При этом каждая датаграмма сервера соответствует одной строке файла. Размер каждого файла многократно превосходит размер одной строки-датаграммы. Заметим, что в данном задании производится не копирование серверного файла, а распределение его копии между клиентами.

*Примечание 1.* Первый байт каждой датаграммы клиента должен содержать номер этого клиента. (Датаграммы в заданиях 2 и 4 не содержат другой полезной информации.)

*Примечание 2.* Ответная датаграмма сервера должна содержать или полезную информацию (задания 2 и 4), или является лишь



подтверждением со стороны сервера правильности выполненной им операции (задания 1 и 3).

## 9.10. Лабораторная работа № 10.

### Сетевые датаграммные каналы

Целью выполнения настоящей лабораторной работы является получение навыков создания на основе сокетов датаграммных *UDP*-каналов, используемых для связи между удаленными процессами.

В начале выполнения данной работы следует ознакомиться с п. 8.3.2 из теоретической части пособия.

#### 9.10.1. Состав и порядок выдачи системных вызовов

При создании и использовании сетевого *UDP*-канала порядок выдачи системных вызовов удаленными частями сетевого приложения (клиентской и серверной) не отличается от порядка выдачи системных вызовов при создании и использовании локального датаграммного канала. Например, если рассмотренную в предыдущей лабораторной работе задачу передачи «эха» реализовать в виде распределенного приложения, то сохранится порядок выдачи системных вызовов, приведенный на рис. 69. Единственное отличие, не влияющее на логику работы приложения, заключается в том, что клиентская и серверная части приложения выдают системные вызовы разным ОС, так как выполняются на разных хостах.

Несмотря на то что состав и порядок выдачи системных вызовов для *UDP*-канала и для локального канала совпадают, некоторые параметры этих вызовов в обоих случаях различны. Далее рассмотрим эти отличия.

#### 9.10.2. Создание сокета

Создание сокета выполняет системный вызов *socket*. Напомним его определение:

```
#include <sys/types.h>
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
```

При создании протяженного канала в качестве параметра *domain* задается константа *AF\_INET*. Ее наличие сообщает ядру ОС

о том, что соединяемые информационным каналом процессы выполняются на удаленных ЭВМ, подключенных к сети *Internet*.

В качестве параметра *type* при создании датаграммного канала задается значение *SOCK\_DGRAM*.

Задание в качестве параметра *protocol* значения 0 означает, что выбор типа транспортного протокола поручается самой ОС. При задании в качестве параметра *type* значения *SOCK\_DGRAM* таким протоколом является *UDP*.

### 9.10.3. Связывание сокета со своим адресом

Связывание сокета со своим адресом выполняет системный вызов *bind*. Его определение:

```
#include <sys/types.h>
#include <sys/socket.h>
int bind(int sockfd, struct sockaddr *address, int add_len)
```

где *sockfd* — номер сокета, созданного ранее с помощью вызова *socket*; *address* — адрес (указатель) структуры типа *sockaddr*; *add\_len* — длина структуры типа *sockaddr*.

В случае успешного завершения вызова *bind* он возвращает значение 0, иначе (–1).

Напомним, что структура *sockaddr* — структура обобщенного адреса сокета, которая определяется в файле *<sys/socket.h>*:

```
struct sockaddr {
    sa_family_t sa_family; /* Область применения канала */
    char sa_data[];        /* Внешнее имя сокета */
};
```

В отличие от локального случая, при создании *UDP*-канала выполнение вызова *bind* приводит к замене структуры *sockaddr* не на структуру внутреннего адреса *sockaddr\_un*, а на структуру сетевого адреса *sockaddr\_in*, которая определена в файле *<netinet/in.h>*:

```
struct sockaddr_in {
    sa_family_t sin_family; /* == AF_INET */
    in_port_t sin_port;     /* Номер порта */
    struct in_addr sin_addr; /* IP-адрес */
    unsigned char sin_zero[8]; /* Поле выравнивания */
};
```

Прежде чем выполнить в программе вызов *bind*, необходимо заполнить ту структуру сетевого адреса, с которой производится

связывание. Пример заполнения данной структуры серверной частью приложения:

```
/* Задание в сервере адреса своего сокета */  
struct sockaddr_in server = (AF_INET, 5000, INADDR_ANY);
```

В данном примере в качестве номера порта сервера взято число 5000. (Напомним, что этот номер должен быть заранее известен клиентским частям приложения.) Для задания IP-адреса в примере используется константа *INADDR\_ANY*. Эта константа содержит IP-адрес «своего» хоста. Причем этот адрес хранится не в «человеческой» форме, представляющей собой 4 десятичных числа, разделенных точками, а в формате, ориентированном на системное использование.

**Пример** заполнения структуры сетевого адреса клиентской частью приложения:

```
/* Задание в клиенте адреса своего сокета */  
struct sockaddr_in client = (AF_INET, INADDR_ANY,  
INADDR_ANY);
```

Обратим внимание, что вместо конкретного номера порта задана константа *INADDR\_ANY*. Ее наличие сообщает ядру ОС о том, что в качестве номера порта ядро может назначить любой свободный номер.

**Пример** выполнения сервером операции связывания своего сокета с его адресом (связывание сокета клиентом производится аналогично):

```
#include <sys/types.h>  
#include <ctype.h>  
#include <sys/socket.h>  
#include <netinet/in.h>  
#define LEN sizeof(struct sockaddr_in)  
main()  
{  
    int sock;  
    /* Задание в сервере адреса своего сокета */  
    struct sockaddr_in server = (AF_INET, 5000, INADDR_ANY);  
    . . . . .  
    /* Связывание сокета сервера с его адресом */  
    if (bind(sock, (struct sockaddr *) &server, LEN) < 0)  
    {  
        printf("ошибка связывания сокета с адресом\n");  
        exit(1);  
    }
```

```
}
. . . . .
```

#### 9.10.4. Прием и передача датаграмм

Напомним, что для того чтобы принять датаграмму из входного буфера своего сокета, программа процесса должна выполнить системный вызов *recvfrom*:

```
#include <sys/types.h>
#include <sys/socket.h>
int recvfrom(int sockfd, void *msg, int len, int flags,
             struct sockaddr *address, int *ad_len)
```

где *sockfd* — номер своего сокета; *msg* — указатель на прикладной буфер, в который должна быть записана принимаемая информация; *len* — длина прикладного приемного буфера (в байтах); *flags* — флаги, задающие особые условия приема датаграммы. Если такие условия не требуются, то задается 0; *address* — указатель на структуру, содержащую адрес сокета-источника датаграммы; *ad\_len* — указатель на целочисленную переменную, содержащую длину структуры, содержащей адрес сокета-источника датаграммы.

При правильном завершении вызов *recvfrom* возвращает в программу процесса число фактически полученных байтов полезной информации, а в случае ошибки — число (–1).

*Примечание.* Если адрес сокета-отправителя не нужен, в качестве предпоследнего параметра вызова следует записать *NULL*, а в качестве последнего параметра — 0.

Напомним, что для отправления своей датаграммы программа процесса должна выполнить системный вызов *sendto*:

```
#include <sys/types.h>
#include <sys/socket.h>
int sendto(int sockfd, const void *msg, int len, int flags,
           const struct sockaddr *toaddr, int tolen);
```

где *sockfd* — номер своего сокета; *msg* — указатель на прикладной буфер, содержащий передаваемую информацию; *len* — длина передаваемой информации (в байтах); *flags* — флаги, задающие особые условия отправления датаграммы. Если такие условия не требуются, то задается 0; *toaddr* — указатель на структуру, содержащую адрес сокета назначения датаграммы; *tolen* — длина структуры, содержащей адрес сокета назначения.

**Пример** отправления датаграммы клиентом на сокет сервера:

```
#include <sys/types.h>
```

```

#include <ctype.h>
#include <sys/socket.h>
#include <netinet/in.h>
char *msg = "С новым годом!\n";
#define LEN sizeof(struct sockaddr_in)
main()
{
    . . . . .
    /* Задание адреса сокета сервера */
    struct sockaddr_in server = (AF_INET, 5000);
    server.sin_addr.s_addr = inet_addr("225.80.13.7");
    /* Отправление запроса на обслуживание к серверу */
    msglen = strlen(msg);
    if (sendto(sock, msg, msglen, 0, (struct sockaddr *)&server,
    LEN) != msglen)
        { printf("Клиент: ошибка передачи\n"); exit(1);}
    . . . . .
}

```

В данном примере используется функция *inet\_addr*, выполняющая преобразование IP-адреса сервера из «человеческой» формы (225.80.13.7) в программную форму, требуемую для передачи ядру ОС.

В следующем примере сервер принимает датаграмму от клиента и возвращает «эхо» принятого сообщения обратно клиенту:

```

#include <sys/types.h>
#include <ctype.h>
#include <sys/socket.h>
#include <netinet/in.h>
#define LBUF 100
#define LEN sizeof(struct sockaddr_un)
char bufer[LBUF];
main ()
{
    /* Объявления локальных переменных и структур данных */
    struct sockaddr_in server, client;
    int n, sock, s_len;
    int c_len = LEN;
    . . . . .
    /* Бесконечный цикл приема и обработки датаграмм
    клиентов */
    for ( ; ; ){
        n= recvfrom(sock, bufer, LBUF, 0, (struct sockaddr *)&client,

```

```
&c_len);

    if (n < 0) { printf("Сервер: ошибка приема\n"); continue;}
    /* В результате системного вызова recvfrom был получен
    адрес клиента. На него и возвращается "эхо" сообщения */
    if (sendto(sock, bufer, n, 0, (struct sockaddr *)&client, c_len)
        !=n) { printf("Сервер: ошибка передачи\n"); continue;}
    }
}
```

### 9.10.5. Задание

Требуется разработать клиентскую и серверную части, принадлежащие разным приложениям, использующим сетевой *UDP*-канал. При этом серверная часть предназначена для обслуживания запросов, поступающих от клиентских частей, принадлежащих другим студентам, а клиентская часть выполняет выдачу запросов к какому-то чужому серверу.

Функции серверной части:

- 1) запись датаграммы, поступившей от какого-то клиента, а также текущего времени в качестве строки текстового файла, расположенного на своем хосте;

- 2) отправка клиенту ответной датаграммы, имеющей длину не более 100 байтов, которая содержит имя и фамилию владельца сервера, а также, возможно, другую информацию по желанию владельца.

Функции клиентской части:

- 1) отправка одному из «известных» серверов (список серверов на доске) датаграммы-запроса, имеющей длину не более 40 байтов, содержащей имя и фамилию владельца клиентской части;

- 2) прием ответной датаграммы от сервера и вывод ее содержания на экран.

Порядок выполнения работы:

- 1) выяснение у преподавателя *IP*-адреса своего хоста;
- 2) написание программ клиентской и серверной частей;
- 3) совместная отладка своих клиентской и серверной частей в локальном режиме. В процессе данной отладки клиентская и серверная части располагаются на одном и том же хосте, хотя и используют сетевой *UDP*-канал. При этом *UDP*-модуль (входит в состав ОС), обнаружив в качестве адреса назначения свой собственный

IP-адрес, никакой отправки датаграммы по сети не выполняет, а направляет ее в указанный локальный сокет;

4) отлаженный в локальном режиме сервер запускается в бесконечном цикле до конца занятия в фоновом режиме. При этом владелец сервера записывает на доске адрес сокета сервера в виде пары чисел: IP-адрес, номер порта;

5) клиентская часть запускается несколько раз в оперативном режиме с целью получения обслуживания от соответствующего числа «известных» серверов.

В процессе защиты лабораторной работы студент должен, в частности, продемонстрировать обслуживание своей клиентской части со стороны чужого сервера, а также вывести на экран (например, с помощью команды *shell – cat*) результирующий файл своей серверной части.

## 9.11. Лабораторная работа № 11.

### Локальные виртуальные соединения

Целью выполнения настоящей лабораторной работы является получение навыков создания и использования локальных виртуальных соединений между процессами, используя метод сокетов.

В начале выполнения данной работы следует ознакомиться с п. 4.6.3 из теоретической части пособия, где рассматривается вопрос построения локальных виртуальных соединений.

#### 9.11.1. Порядок выдачи системных вызовов

В качестве простейшего примера создания и использования локального виртуального соединения опять рассмотрим получение двумя процессами «эха» символьной строки. В этом примере процесс-клиент посылает другому процессу-серверу любую символьную строку. Далее сервер посылает эту строку обратно клиенту. Получив символьную строку, процесс-клиент выводит ее на экран, сопроводив этот вывод пояснительной надписью. На рис. 70 приведен возможный порядок выдачи системных вызовов во времени.

При выполнении сервером системных вызовов *socket* (создать сокет) и *bind* (связать сокет с адресом), а также при выполнении клиентом вызова *socket* не существует отличий от применения этих вызовов при работе с локальным датаграммным каналом, за исключением того, что в вызове *socket* в качестве параметра *type*

(определяет устойчивость создаваемого канала) задается значение *SOCK\_STREAM* — виртуальное соединение.

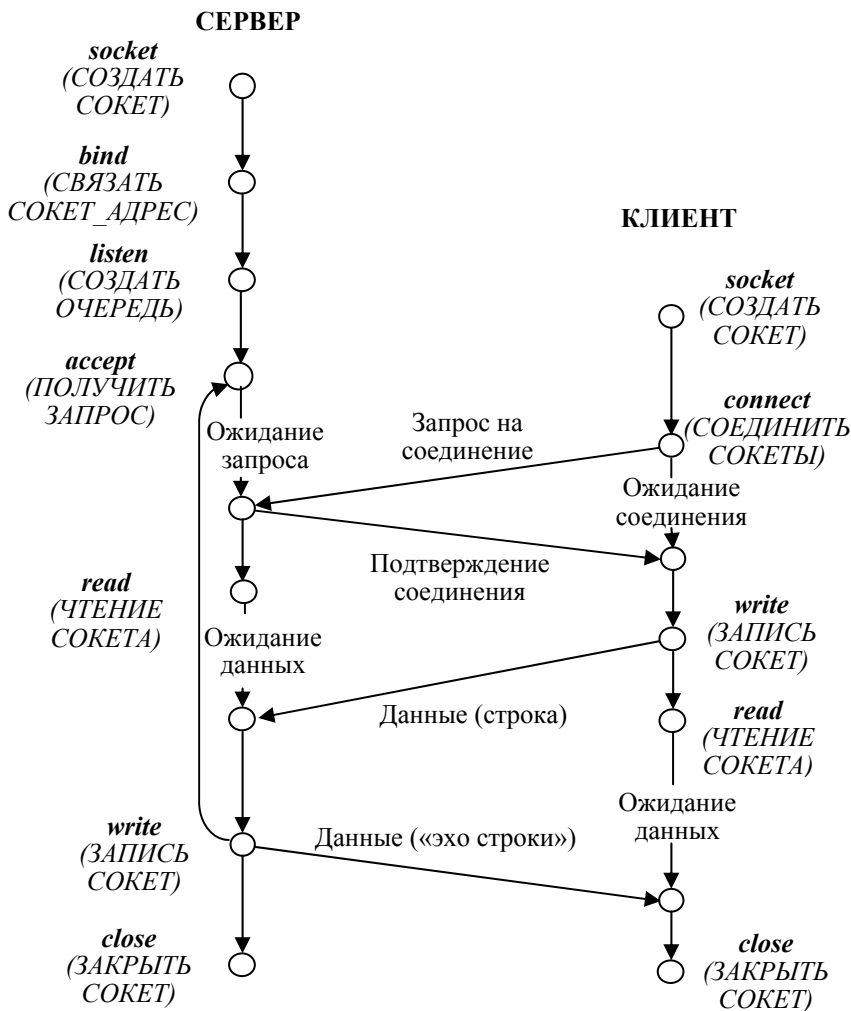


Рис. 70. Пример создания и использования виртуального соединения

### 9.11.2. Создание очереди запросов на соединение и работа с ней



В результате выполнения серверной частью приложения вызовов *socket* и *bind* создается сокет, имеющий адрес (имя-путь сокета). Для того чтобы данный сокет стал главным сокетом сервера, необходимо снабдить данный сокет очередью запросов на создание виртуальных соединений. Это делает системный вызов *listen*:

```
#include <sys/socket.h>
int listen (int sockfd, int queue_size)
```

где *sockfd* — номер главного сокета; *queue\_size* — максимальное число запросов на соединение, которые могут находиться в очереди (это число не должно быть менее пяти).

После того как очередь запросов на создание виртуальных соединений создана, сервер может ее опрашивать с целью выборки того запроса, который стоит в очереди первым. Для этого используется системный вызов *accept*:

```
#include <sys/types.h>
#include <sys/socket.h>
int accept (int sockfd, struct sockaddr *address, int *add_len)
```

где *sockfd* — номер главного сокета; *address* — указатель на структуру, в которую системный вызов *accept* поместит адрес сокета того клиента, который выдал запрос на соединение. Если серверу не нужен адрес сокета клиента (сервер будет использовать для связи виртуальное соединение), то в качестве данного параметра следует записать *NULL*; *add\_len* — длина адресной структуры, заданной предыдущим параметром. Если эта структура не используется, то записывается *NULL*.

Если системный вызов *accept* завершился с ошибкой, то он возвращает значение (−1). Иначе в результате своего выполнения *accept* возвращает номер нового сокета сервера, который является окончанием нового виртуального соединения со стороны сервера. Этот номер сокета используется далее серверной частью приложения подобно номеру открытого файла для выполнения операций чтения и записи.

Что касается клиента, то для того чтобы его запрос на создание виртуального соединения поступил к серверу в очередь главного сокета, клиент должен выдать системный вызов *connect* (*соединить сокеты*):

```
#include <sys/types.h>
#include <sys/socket.h>
```

```
int connect (int csockfd, const struct sockaddr *address, int  
*add_len)
```

где *csockfd* — номер сокета клиента, который предполагается использовать в качестве клиентского окончания создаваемого виртуального соединения; *address* — указатель на структуру, которая содержит адрес главного сокета сервера; *add\_len* — длина адресной структуры, заданной предыдущим параметром.

Если вызов *connect* завершился с ошибкой, то он возвращает значение (−1).

### 9.11.3. Работа с виртуальным соединением

После того как виртуальное соединение создано, клиентская и серверная части приложения могут передавать по нему свои данные. Для этого на каждом конце виртуального соединения известен номер своего сокета, который используется при передаче данных точно так же, как используется номер открытого файла при выполнении операций чтения и записи. При этом для чтения из виртуального соединения может использоваться системный вызов *read*:

```
ssize_t read(int sockfd, void *buffer, size_t n);
```

где *sockfd* — номер сокета, являющегося окончанием виртуального соединения; *buffer* — указатель на буфер в ОП, в который считываются байты из виртуального соединения; *n* — длина буфера.

В случае ошибки *read* возвращает (−1). Если выполнение *read* завершилось успешно, то он возвращает число байтов, считанных из виртуального соединения. Это число равно *n* или меньше его. Второе выполняется тогда, когда заканчиваются байты, записанные в виртуальное соединение с другого его конца. Возможно, что через некоторое время требуемые байты появятся в виртуальном соединении, а возможно, что и нет. Вопрос о том, повторять ли в подобном случае выдачу вызова *read*, находится исключительно в ведении прикладного протокола. При этом конец передаваемых данных может быть определен по специальному байту (байтам), предусмотренному в прикладном протоколе.

Для записи в виртуальное соединение может использоваться системный вызов *write*:

```
ssize_t write(int sockfd, void *buffer, size_t n);
```

где *sockfd* — номер сокета, являющегося окончанием виртуального соединения; *buffer* — указатель на буфер в ОП, из которого

записываются байты в виртуальное соединение;  $n$  — число байтов, записываемых в виртуальное соединение.

Если выполнение *write* завершилось успешно, то он возвращает число байтов, записанных в виртуальное соединение. Почти всегда это число равно  $n$ . В случае ошибки *write* возвращает  $(-1)$ .

#### 9.11.4. Закрытие виртуального соединения

Созданное виртуальное соединение существует до тех пор, пока один из партнеров (клиент или сервер) не уничтожит его, используя тот же самый системный вызов *close*, который используется для закрытия файлов:

```
int close(int sockfd);
```

где *sockfd* — номер сокета, являющегося окончанием виртуального соединения.

Интересно отметить, что если закрываемый сокет содержит данные, предназначенные для передачи по виртуальному соединению, то операция закрытия будет заблокирована (то есть выдавший ее процесс переводится в состояние «Сон») до тех пор, пока данные не будут переданы.

Логично предположить, что после того как один из партнеров закроет виртуальное соединение, второй партнер также должен завершить использование этого информационного канала. При этом возможны следующие варианты:

1) второй партнер выполняет «плановое» закрытие виртуального соединения, предусмотренное прикладным протоколом. В этом случае процессу достаточно выдать системный вызов *close*;

2) второй партнер выполняет чтение данных из виртуального соединения (например, с помощью вызова *read*) в такой момент времени, когда это соединение уже не существует. В этом случае *read* возвратит 0 в качестве числа считанных байтов. Поэтому для избежания заикливания в программе процесса результат, возвращаемый вызовом *read*, всегда следует сравнивать с нулем;

3) второй партнер выполняет запись в виртуальное соединение (например, с помощью вызова *write*) в такой момент времени, когда это соединение уже не существует. В этом случае процесс получит сигнал *SIGPIPE*. Стандартный обработчик этого сигнала выполняет завершение процесса. Во избежание такого завершения программа процесса должна содержать свой обработчик сигнала *SIGPIPE*.

### 9.11.5. Программы взаимодействующих процессов

Предоставление взаимодействующим процессам виртуального соединения на основе сокетов никак не регламентирует порядок передачи данных по этому каналу клиентом и сервером. Этот порядок полностью определяется алгоритмом совместной работы клиента и сервера, то есть прикладным протоколом.

Реализация прикладного протокола в серверной части приложения почти всегда является многопроцессной. При этом процесс-отец создает главный сокет, снабжает его адресом, а затем использует для приема запросов на создание виртуальных соединений. Для каждого принятого запроса он создает виртуальное соединение, а также дочерний процесс, предназначенный для «эксплуатации» этого информационного канала. Далее дочерний процесс и клиент выполняют обмен данными по виртуальному соединению согласно прикладному протоколу.

Обычно серверная часть приложения разрабатывается гораздо более тщательно, чем клиентские части. В частности, серверная часть приложения должна обрабатывать различные варианты обрыва виртуального соединения со стороны клиентской части. В клиентской части подобные меры могут и отсутствовать.

Вернемся к примеру с выводом «эха» и запишем программы многопроцессного сервера и клиента.

#### **Сервер:**

```
/* Подсоединение заголовочных файлов */
#include <sys/types.h>
#include <ctype.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <signal.h>
/* Объявления констант и глобальных переменных */
#define LBUF 100
#define LEN sizeof(struct sockaddr_un)
char bufer[LBUF];
int nsock;
/* Функция – обработчик сигнала SIGPIPE в сервере */
void s_action (int sig)
{
    close(nsock);
    printf("Сервер: вирт. соединение оказалось оборвано при
```

```
    записи в него\n");
    exit(0);
}
main ()
{
    /* Объявления локальных переменных и структур данных */
    struct sockaddr_un server;
    int n, sock, s_len;
    static struct sigaction sact;
    /* Изменение диспозиции сигнала SIGPIPE во всем сервере */
    sact.sa_handler = s_action;
    sigfillset(&(sact.sa_mask));
    sigaction(SIGPIPE, &sact, NULL);
    /* Создание главного сокета сервера */
    if ((sock = socket(AF_UNIX, SOCK_STREAM, 0)) < 0)
    {
        printf("Сервер: невозможно создать сокет\n"); exit(1);
    }
    /* Задание адреса главного сокета */
    unlink("/home/vlad/abc.server");
    bzero(&server, LEN);
    server.sun_family = AF_UNIX;
    strcpy(server.sun_path, "/home/vlad/abc.server");
    s_len = sizeof(server.sun_family) + strlen(server.sun_path);
    /* Связывание главного сокета сервера с его адресом */
    if (bind(sock, (struct sockaddr *) &server, s_len) < 0)
    {
        printf("Сервер: ошибка связывания сокета с адресом\n");
        exit(1);
    }
    /* Создание очереди запросов на создание соединений */
    if (listen (sock, 5) < 0)
    {
        printf("Сервер: ошибка создания очереди запросов\n");
        exit(1);
    }
    /* Бесконечный цикл приема запросов клиентов на создание
    соединений */
    for ( ; ; ){
        /* Выборка из очереди запроса на создание соединения */
        if (nsock = accept(sock, NULL, NULL)) < 0)
        {
```

```

    printf("Сервер: ошибка выборки запроса из очереди
    запросов\n");
    continue;
}
/* Создание дочернего процесса для обслуживания
соединения */
if (fork() == 0)
{
    n = read(nsock, bufer, LBUF);
    if (n < 0) { printf("Сервер: ошибка приема\n") }
    if (n = 0) { printf("Сервер: вирт. соедин. оборвано
    при чтении\n") }
    if (n > 0) {
        if (write(nsock, bufer, n) != n) {
            printf("Сервер: ошибка передачи\n")
        }
    }
}
/* Завершение дочернего процесса */
close(nsock);
exit(0);
}
/* В главном процессе новый сокет не нужен */
close(nsock);
}
}

```

### **Клиент:**

```

/* Включение заголовочных файлов*/
#include <sys/types.h>
#include <ctype.h>
#include <sys/socket.h>
#include <sys/un.h>
/* Объявления констант и глобальных переменных*/
char *msg = "С новым годом!\n";
#define LBUF 100
#define LEN sizeof(struct sockaddr_un)
char bufer[LBUF];
main ()
{
    /* Объявления локальных переменных и структур данных*/
    struct sockaddr_un server;
    int n, sock, s_len, msglen;
    /* Создание сокета клиента*/

```

```
if ((sock = socket(AF_UNIX, SOCK_STREAM, 0)) < 0)
{
    printf("Клиент: невозможно создать сокет\n"); exit(1);
}
/* Задание адреса главного сокета сервера */
bzero(&server, LEN);
server.sun_family = AF_UNIX;
strcpy(server.sun_path, "/home/vlad/abc.server");
s_len=sizeof(server.sun_family)+strlen(server.sun_path);
/* Отправление запроса к серверу на создание вирт.
соединения */
if ( connect (sock, (struct sockaddr *)&server, LEN) < 0)
{
    printf("Клиент: невозможно создать соединение\n");
    exit(1);
}
/* Отправка сообщения к серверу */
msglen = strlen(msg);
if (write(sock, msg, msglen) != msglen) {
    printf("Клиент: ошибка передачи\n"); exit(1);}
/* Прием от сервера «эха» сообщения*/
if((n = read (sock, bufer, LBUF)) < 0)
{ printf("Клиент: ошибка приема\n"); exit(1);}
/* Вывод «эха» на экран*/
printf("Эхо: %s\n", bufer);
/* Завершение программы */
close(sock);
exit(0);
}
```

### 9.11.6. Задание

Требуется разработать процессы, запускаемые из командной строки *UNIX*: процесс-сервер, запускаемый в оперативном режиме, и два процесса-клиента, запускаемые в фоновом режиме. Между этими процессами должны существовать информационные взаимодействия в виде виртуальных соединений, предназначенные для решения той же самой задачи, что и в лабораторной работе № 9.

1. Сервер экрана. Каждый клиент выводит содержимое своего текстового файла на экран, находящийся под управлением процесса-сервера. Размер каждого файла соответствует нескольким экранным строкам, каждая из которых передается клиентом в виде

одного сообщения. Причем вывод каждой такой строки на экран сервер предваряет выводом номера клиента.

2. Сервер клавиатуры. Каждый клиент вводит содержимое своего текстового файла с клавиатуры, находящейся под управлением процесса-сервера. При этом каждое сообщение сервера соответствует одной введенной строке текста. Размер каждого клиентского файла многократно превосходит размер одной строки-сообщения.

3. Сервер файла для записи. Каждый клиент выводит содержимое своего текстового файла в файл на диске, находящийся под управлением процесса-сервера. При этом каждое сообщение клиента соответствует одной строке файла. Размер каждого клиентского файла превосходит размер одной строки-сообщения. Запись каждого такого сообщения в свой файл сервер предваряет записью номера клиента.

4. Сервер файла для чтения. Каждый клиент выполняет запись в свой текстовый файл на диске информации из файла, находящегося под управлением процесса-сервера. При этом каждое сообщение сервера соответствует одной строке файла. Размер каждого файла многократно превосходит размер одной строки-сообщения. Заметим, что в данном задании производится не копирование серверного файла, а распределение его копии между клиентами.

*Примечание 1.* Первый байт каждого сообщения клиента должен содержать номер этого клиента. (Сообщения клиентов в заданиях 2 и 4 не содержат другой полезной информации.)

*Примечание 2.* Ответное сообщение сервера должно или содержать полезную информацию (задания 2 и 4), или являться лишь подтверждением со стороны сервера правильности выполненной им операции (задания 1 и 3).

## **9.12. Лабораторная работа № 12.**

### **Сетевые виртуальные соединения**

Целью выполнения настоящей лабораторной работы является получение навыков создания на основе сокетов виртуальных *TCP*-соединений, используемых для связи между удаленными процессами.

В начале выполнения данной работы следует ознакомиться с п. 8.3.3 из теоретической части пособия.



### 9.12.1. Состав и порядок выдачи системных вызовов

При создании и использовании сетевого *TCP*-канала порядок выдачи системных вызовов удаленными частями сетевого приложения (клиентской и серверной) не отличается от порядка выдачи системных вызовов при создании и использовании локального виртуального канала. Например, если рассмотренную в предыдущей лабораторной работе задачу передачи «эха» реализовать в виде распределенного приложения, то сохранится порядок выдачи системных вызовов, приведенный на рис. 70, несмотря на то что клиентская и серверная части приложения выполняются на разных хостах.

Хотя состав и порядок выдачи системных вызовов для *TCP*-канала и для локального канала совпадают, некоторые параметры этих вызовов в обоих случаях различны. Далее рассмотрим эти отличия.

Во-первых, вызов *socket*, выполняющий в серверной части приложения создание главного сокета сервера, а в клиентской части — создание сокета клиента, имеет единственное отличие — в качестве первого параметра вызова (*domain*) задается константа *AF\_INET*. Ее наличие сообщает ядру ОС о том, что соединяемые информационным каналом процессы выполняются на удаленных ЭВМ, подключенных к сети *Internet*.

### 9.12.2. Задание адреса главного сокета

Задание адреса главного сокета сервера используется и в серверной части (обеспечение выдачи вызова *bind*), и в клиентской части (для вызова *connect*). В отличие от локального случая, при создании *TCP*-канала структура *sockaddr* заменяется не на структуру внутреннего адреса *sockaddr\_un*, а на структуру сетевого адреса *sockaddr\_in*, которая определена в файле *<netinet/in.h>*:

```
struct sockaddr_in {
    sa_family_t    sin_family;           /* == AF_INET */
    in_port_t      sin_port;             /* Номер порта */
    struct in_addr  sin_addr;             /* IP-адрес */
    unsigned char  sin_zero[8];          /* Поле выравнивания */
};
```

**Пример** заполнения данной структуры в серверной части приложения:

```
/* Задание в сервере адреса главного сокета */
struct sockaddr_in server = (AF_INET, 5000, INADDR_ANY);
```

В данном примере в качестве номера порта сервера взято число 5000. Для задания IP-адреса в примере используется константа `INADDR_ANY`, которая содержит IP-адрес «своего» хоста.

**Пример** заполнения структуры сетевого адреса главного сокета сервера в клиентской части приложения:

```
/* Задание адреса главного сокета сервера */
struct sockaddr_in server = (AF_INET, 5000);
server.sin_addr.s_addr = inet_addr("225.80.13.7");
```

В данном примере используется функция `inet_addr`, выполняющая преобразование IP-адреса сервера из «человеческой» формы (225.80.13.7) в программную форму, требуемую для передачи ядру ОС.

### 9.12.3. Использование адреса главного сокета

Пример выполнения сервером операции связывания главного сокета с его адресом:

```
#include <sys/types.h>
#include <ctype.h>
#include <sys/socket.h>
#include <netinet/in.h>
#define LEN sizeof(struct sockaddr_in)
main()
{
    int sock;
    /* Задание в сервере адреса своего главного сокета */
    struct sockaddr_in server = (AF_INET, 5000, INADDR_ANY);
    . . . . .
    /* Связывание сокета сервера с его адресом */
    if (bind(sock, (struct sockaddr *) &server, LEN) < 0)
    {
        printf("ошибка связывания сокета с адресом\n");
        exit(1);
    }
    . . . . .
```

**Пример** выполнения клиентом операции выдачи запроса на создание виртуального соединения:

```
#include <sys/types.h>
```

```

#include <ctype.h>
#include <sys/socket.h>
#include <netinet/in.h>
#define LEN sizeof(struct sockaddr_in)
main()
{
    int sock;
    /* Задание адреса главного сокета сервера */
    struct sockaddr_in server = (AF_INET, 5000);
    server.sin_addr.s_addr = inet_addr("225.80.13.7");
    . . . . .
    /* Отправление запроса к серверу на создание вирт.
    соединения */
    if ( connect (sock, (struct sockaddr *)&server, LEN) < 0)
    {
        printf("Клиент: невозможно создать соединение\n");
        exit(1);
    }
    . . . . .

```

Выполнение остальных системных вызовов (*listen*, *accept*, *write*, *read*, *close*) не имеет каких-либо отличий от использования этих вызовов для построения и использования локальных виртуальных соединений.

### 9.12.4. Задание

Требуется разработать клиентскую и серверную части, принадлежащие разным приложениям, использующим сетевое виртуальное соединение на основе транспортного протокола *TCP*. При этом серверная часть предназначена для обслуживания запросов, поступающих от клиентских частей, принадлежащих другим студентам, а клиентская часть выполняет выдачу запросов к какому-то чужому серверу.

Функции серверной части:

- 1) выполнение запросов, поступающих от клиентов, на создание виртуальных соединений;
- 2) выполнение запросов на обслуживание, поступающих от клиентов по виртуальным соединениям. Содержимое каждого сообщения-запроса клиента добавляется к содержимому учетного текстового файла сервера. В ответ сервер отправляет клиенту по виртуальному соединению содержимое другого своего текстового

файла, имеющего длину не менее 100 байтов и содержащего среди прочих данных обязательно имя и фамилию владельца сервера. Длина сообщений сервера — 20 байтов.

Функции клиентской части:

1) отправка одному из «известных» серверов (список серверов на доске) запроса на создание виртуального соединения;

2) отправка по созданному виртуальному соединению серверу сообщения-запроса на обслуживание. Это сообщение имеет длину не более 40 байтов и содержит имя и фамилию владельца клиентской части, а также любую прочую информацию;

3) прием от сервера текстового файла и вывод его содержимого на экран.

Порядок выполнения работы:

1) выяснение у преподавателя *IP*-адреса своего хоста;

2) написание программ клиентской и серверной частей;

3) совместная отладка своих клиентской и серверной частей в локальном режиме. В процессе данной отладки клиентская и серверная части располагаются на одном и том же хосте, хотя и используют сетевой *TCP*-канал. При этом *TCP*-модуль (входит в состав ОС), обнаружив в качестве адреса назначения свой собственный *IP*-адрес, никакой отправки сообщения по сети не выполняет, а направляет его в указанный локальный сокет;

4) отлаженный в локальном режиме сервер запускается в бесконечном цикле до конца занятия в фоновом режиме. При этом владелец сервера записывает на доске адрес главного сокета сервера в виде пары чисел: *IP*-адрес, номер порта;

5) клиентская часть запускается несколько раз в оперативном режиме с целью получения обслуживания от соответствующего числа «известных» серверов.

В процессе защиты лабораторной работы студент должен, в частности, продемонстрировать обслуживание своей клиентской части со стороны чужого сервера, а также вывести на экран (например, с помощью команды *shell – cat*) учетный файл своей серверной части.

## Литература

1. Робачевский А. Операционная система *UNIX* / А. Робачевский. – СПб. : БХВ – Санкт-Петербург, 1999. – 528 с.
2. Дунаев С. UNIX SYSTEM V. Release 4.2: общее руководство / С. Дунаев. – М. : ДИАЛОГ-МИФИ, 1995. – 287 с.
3. Рейчард К. Unix: справочник / К. Рейчард, Э. Фостер Джонсон. – СПб. : Питер, 2000. – 384 с.
4. Столлингс В. Операционные системы. Внутреннее устройство и принципы проектирования / В. Столлингс. – М. : Вильямс, 2002. – 848 с.
5. Олифер В.Г. Сетевые операционные системы / В.Г. Олифер, Н.А. Олифер. – СПб. : Питер, 2001. – 544 с.
6. Соловьев А.А. Программирование на shell / А.А. Соловьев. – М. : Финансы и статистика, 1999. – 318 с.

Учебное издание

**Одинок** Владимир Викторович  
**Коцубинский** Владислав Петрович

## ОПЕРАЦИОННЫЕ СИСТЕМЫ И СЕТИ

Учебное пособие

Корректор О.В. Полещук  
Компьютерная верстка Г.В. Черновой

Подписано в печать 29.11.2008. Формат 60×84/16.  
Усл. печ. л. 22,79. Тираж 100. Заказ 1239.

Томский государственный университет  
систем управления и радиоэлектроники.  
634050, Томск, пр. Ленина, 40.  
Тел. (3822) 533018.