Министерство образования и науки Российской Федерации

ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ СИСТЕМ УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ (ТУСУР)

ФАКУЛЬТЕТ ДИСТАНЦИОННОГО ОБУЧЕНИЯ (ФДО)

В. П. Коцубинский, А. А. Изюмов

ОПЕРАЦИОННЫЕ СИСТЕМЫ

Учебно-методическое пособие

Томск
2016

УДК 004.451(075.8)
ББК 32.973.2-018.2я73
К 754

Рецензенты:

Н. Ю. Хабибулина, канд. техн. наук, доцент кафедры компьютерных систем в управлении и проектировании ТУСУР;

Г. Н. Решетникова, канд. техн. наук, доцент кафедры прикладной математики Национального исследовательского Томского государственного университета

Коцубинский В. П.

К 754 Операционные системы : учебно-методическое пособие / В. П. Коцубинский, А. А. Изюмов. – Томск : ФДО, ТУСУР, 2016. – 244 с.

В пособии рассмотрены история становления и развития операционных систем, принципы реализации пользовательских интерфейсов операционной системы; основы языка управления операционной системой; принципы многопользовательской работы и структура системы, а также подсистемы UNIX – подсистема управления процессами, подсистема управления памятью, файловая подсистема. Содержится лабораторный курс, предназначенный для закрепления теоретических знаний и формирования практических навыков программирования на командном языке операционной системы UNIX.

Для студентов направлений подготовки «Управление в технических системах» и «Информатика и вычислительная техника», обучающихся с применением дистанционных образовательных технологий.

© Коцубинский В. П.,
Изюмов А. А., 2016
© Оформление.
ФДО, ТУСУР, 2016

Оглавление

Введение 6
1 История операционных систем 9
1.1 Предыстория 9
1.2 Первое поколение (1945–1955): электронные лампы 9
1.3 Второе поколение (1955–1965): транзисторы и системы пакетной обработки 10
1.4 Третье поколение (1965–1980): интегральные схемы и многозадачность 13
1.5 Четвертое поколение (с 1980 г. по наши дни): персональные компьютеры 22
2 Интерфейсы пользователя системы 29
2.1 Функции системных программ 29
2.2 Файлы 35
2.3 Утилиты 41
2.4 Трансляторы 49
2.5 Язык управления операционной системой 57
2.5.1 Общие сведения 57
2.5.2 Простые команды 58
2.5.3 Составные команды 62
2.5.4 Переменные и выражения 65
2.5.5 Управляющие операторы 73
2.5.6 Командные файлы 80
3 Системная поддержка мультипрограммирования 87
3.1 Общие сведения 87
3.2 Процессы 88
3.3 Ресурсы 92
3.4 Синхронизация параллельных процессов 95
3.4.1 Синхронизация с помощью сигналов 96
3.4.2 Терминальное управление процессами 98
3.4.3 Синхронизация конкурирующих процессов 103
3.4.4 Синхронизация кооперирующихся процессов 108
3.5 Информационные взаимодействия между процессами 109
4 Поддержка многопользовательской работы и структура системы 115
4.1 Управление доступом пользователя в систему 115

4.2 Защита файлов 120
4.3 Укрупненная структура операционной системы 125
4.4 Структура сетевой операционной системы 128
5 Подсистема управления процессами 136
5.1 Состояния процесса 136
5.2 Создание процесса 140
5.3 Обработка сигналов 143
5.4 Диспетчеризация процессов 145
5.5 Использование таймера для управления процессами 151
6 Управление оперативной памятью 154
6.1 Основные положения 154
6.2 Сегментная виртуальная память 158
6.2.1 Преобразование адресов 158
6.2.2 Распределение памяти 164
6.2.3 Защита информации в оперативной памяти 166
6.3 Линейная виртуальная память 171
6.3.1 Преобразование адресов 171
6.3.2 Распределение памяти 174
7 Управление файлами 178
7.1 Виртуальная файловая система 178
7.1.1 Логические файлы 178
7.1.2 Открытие файла 180
7.1.3 Другие операции с файлами 185
7.2 Реальные файловые системы 190
7.2.1 Критерии оценки файловых систем 190
7.2.2 Физическое размещение информации на носителе 193
7.2.3 Каталоги 199
7.2.4 Управляющие структуры данных 201
7.3 Объединение реальных файловых систем 204
8 Лабораторный курс 212
8.1 Лабораторная работа № 1. Первоначальное знакомство с UNIX 212
8.1.1 Цель работы 212
8.1.2 Подготовка к выполнению работы 213
8.1.3 Текстовый редактор ed 214
8.1.4 Задание 219

8.2 Лабораторная работа № 2. Управляющие операторы командного языка ... 220
8.2.1 Цель работы ... 220
8.2.2 Подготовка к выполнению работы ... 220
8.2.3 Задание ... 220
8.3 Лабораторная работа № 3. Операции с файлами в программе на языке Си ... 221
8.3.1 Цель работы ... 221
8.3.2 Подготовка к выполнению работы ... 221
8.3.3 Введение ... 221
8.3.4 Открытие существующего файла ... 222
8.3.5 Создание файла ... 225
8.3.6 Указатель файла ... 227
8.3.7 Чтение из файла ... 229
8.3.8 Запись в файл ... 231
8.3.9 Закрытие и уничтожение файла ... 233
8.3.10 Задание ... 233
8.4 Лабораторная работа № 4. Обработка сигналов ... 234
8.4.1 Цель работы ... 234
8.4.2 Подготовка к выполнению работы ... 234
8.4.3 Изменение диспозиции сигналов ... 235
8.4.4 Задание ... 235
Заключение ... 239
Литература ... 240
Глоссарий ... 241

Введение

Основной целью данного курса 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 основными подсистемами UNIX – подсистемой управления процессами, подсистемой управления памятью, файловой подсистемой. При этом рассматриваются основные программные интерфейсы и системные структуры данных.

Изучение данной дисциплины предполагает предварительное изучение студентами следующих дисциплин:

1) информатика;
2) алгоритмические языки и программирование.

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

1) история операционных систем и пользовательские интерфейсы – гл. 1 и 2;
2) принципы построения мультипрограммных многопользовательских систем – гл. 3 и 4;
3) программные интерфейсы и системные структуры данных основных подсистем ОС – гл. 5, 6 и 7;
4) глава 8 содержит лабораторный практикум.

Существующие подходы к изложению курса «Операционные системы» можно разделить на два класса:

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 является, 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 неприхотлива к используемой аппаратной базе и может выполняться на различных конфигурациях аппаратуры и 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.

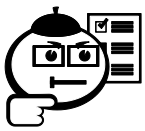
Следует отметить, что при описании системных программных вызовов в пособии приводятся не реальные системные вызовы UNIX, записанные на языке программирования этой системы Си, а упрощенные их варианты, записанные на псевдоязыке. Его применение преследует цель сделать материал пособия доступным для читателя, не знакомого с Си, а также позволяет избавиться от деталей, связанных с применением этого 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.

Соглашения, принятые в учебном пособии

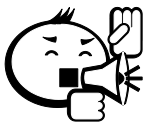
Для улучшения восприятия материала в данном пособии используются следующие пиктограммы и специальное выделение важной информации.



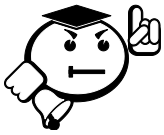
Эта пиктограмма означает определение или новое понятие.



Этот блок означает задание.



Этот блок означает «Внимание!». Здесь выделена важная информация, требующая акцента на ней. Автор здесь может поделиться с читателем опытом, чтобы помочь избежать некоторых ошибок.



В блоке «На заметку» автор может указать дополнительные сведения или другой взгляд на изучаемый предмет, чтобы помочь читателю лучше понять основные идеи.



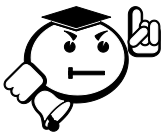
Контрольные вопросы по главе

1 История операционных систем

1.1 Предыстория

История развития операционных систем насчитывает уже много лет. Мы кратко рассмотрим некоторые основные моменты этого развития. Так как операционные системы появились и развивались в процессе конструирования компьютеров, то эти события исторически тесно связаны. Поэтому чтобы представить, как выглядели операционные системы, мы рассмотрим несколько следующих друг за другом поколений компьютеров. Такая схема взаимосвязи поколений операционных систем и компьютеров носит довольно приблизительный характер, но она обеспечивает некоторую структуру, без которой невозможно было бы что-либо понять.

Первый настоящий цифровой компьютер был изобретен английским математиком Чарльзом Бэббиджем (Charles Babbage, 1792–1871). Хотя большую часть жизни Бэббидж посвятил попыткам создания своей «аналитической машины», он так и не смог заставить ее работать должным образом. Это была чисто механическая машина, а технологии того времени не были достаточно развиты для изготовления многих деталей и механизмов высокой точности. Не стоит и говорить, что его аналитическая машина не имела операционной системы.



Интересный исторический факт: Бэббидж понимал, что для аналитической машины ему необходимо программное обеспечение, поэтому он нанял молодую женщину по имени Ада Лавлейс (Ada Lovelace), дочь знаменитого британского поэта лорда Байрона. Она и стала первым в мире программистом, а язык программирования Ada® был назван именно в ее честь.

1.2 Первое поколение (1945–1955): электронные лампы

После безуспешных усилий Бэббиджа прогресс в конструировании цифровых компьютеров практически не наблюдался вплоть до Второй мировой войны, которая стимулировала взрывную активизацию работ над ними. Профессор Джон Атанасов (John Atanasoff) и его аспирант Клиффорд Берри (Clifford Berry) создали в Университете штата Айова конструкцию, которая

сейчас считается первым действующим цифровым компьютером. В ней использовалось 300 электронных ламп. Примерно в то же время Конрад Цузе (Konrad Zuse) в Берлине построил Z3, компьютер, основанный на использовании механических реле. В 1944 г. группой в Блетшли Парк, Великобритания, был построен «Колоссус», в Гарварде Говардом Айкеном (Howard Aiken) был построен «Марк I», а в Университете штата Пенсильвания Вильям Мочли (William Mauchley) и его аспирант Джон Преспер Эккерт (J. Presper Eckert) построили ЭНИАК. Некоторые из этих машин были цифровыми, в некоторых использовались электронные лампы, а работу некоторых из них можно было программировать, но все они были весьма примитивно устроены и тратили многие секунды на производство даже простейших вычислений.

На заре компьютерной эры каждую машину проектировала, создавала, программировала, эксплуатировала и обслуживала одна и та же группа людей (как правило, инженеров). Все программирование велось исключительно на машинном языке или, и того хуже, за счет сборки электрических схем, а для управления основными функциями машины приходилось подключать к коммутационным панелям тысячи проводов. О языках программирования (даже об ассемблере) тогда еще ничего не было известно. Об операционных системах вообще никто ничего не слышал. Режим работы программиста заключался в том, чтобы записаться на определенное машинное время на специальном стенде, затем спуститься в машинный зал, вставить свою коммутационную панель в компьютер и провести следующие несколько часов в надежде, что в процессе работы не выйдет из строя ни одна из примерно 20 тысяч электронных ламп. В сущности, все решаемые задачи сводились к простым числовым расчетам, таким как уточнение таблиц синусов, косинусов и логарифмов.

Когда в начале 1950-х гг. появились перфокарты, положение несколько улучшилось. Появилась возможность вместо использования коммутационных панелей записывать программы на картах и считывать с них, но во всем остальном процедура работы не претерпела изменений.

1.3 Второе поколение (1955–1965): транзисторы и системы пакетной обработки

В середине 1950-х гг. изобретение и применение транзисторов радикально изменило всю картину. Компьютеры стали достаточно надежными, появилась высокая вероятность того, что машины будут работать довольно долго, выполняя при этом полезные функции. Впервые сложилось четкое разделение

между проектировщиками, сборщиками, операторами, программистами и обслуживающим персоналом.

Машины, называемые теперь *мэйнфреймами*, располагались в специальных комнатах с кондиционированием воздуха, где ими управлял целый штат профессиональных операторов. Только большие корпорации, правительственные учреждения или университеты могли позволить себе технику, цена которой исчислялась миллионами долларов. Чтобы выполнить *задание* (то есть программу или комплект программ), программист сначала должен был записать его на бумаге (на Фортране или ассемблере), а затем перенести на перфокарты. После этого он должен был принести колоду перфокарт в комнату ввода данных, передать одному из операторов и идти пить кофе в ожидании, когда будет готов результат. Когда компьютер заканчивал выполнение какого-либо из текущих заданий, оператор подходил к принтеру, отрывал лист с полученными данными и относил его в комнату для распечаток, где программист позже мог его забрать. Затем оператор брал одну из колод перфокарт, принесенных из комнаты ввода данных, и запускал ее на считывание. Если в процессе расчетов был необходим компилятор языка Фортран, то оператору приходилось брать его из картотечного шкафа и загружать в машину отдельно. На одно только хождение операторов по машинному залу терялась впустую масса драгоценного компьютерного времени.

Если учесть высокую стоимость оборудования, неудивительно, что люди довольно скоро занялись поиском способа повышения эффективности использования машинного времени. Общепринятым решением стала *система пакетной обработки*. Первоначально замысел состоял в том, чтобы собрать полный поднос заданий (колод перфокарт) в комнате входных данных и затем переписать их на магнитную ленту, используя небольшой и (относительно) недорогой компьютер, например IBM 1401, который был очень хорош для считывания карт, копирования лент и печати выходных данных, но не подходил для числовых вычислений.

Другие, более дорогостоящие машины, такие как IBM 7094, использовались для настоящих вычислений. Этот процесс изображен на рисунке 1.1.

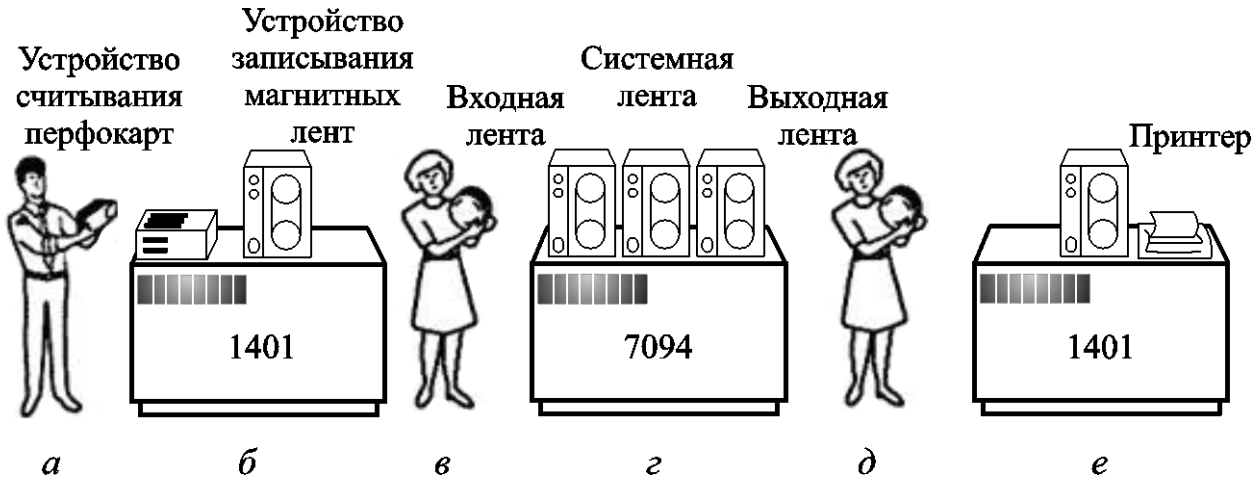


Рис. 1.1 – Ранняя система пакетной обработки: программист приносит карты для IBM 1401 (*а*); IBM 1401 записывает пакет заданий на магнитную ленту (*б*); оператор переносит входные данные на ленте к IBM 7094 (*в*); IBM 7094 выполняет вычисления (*г*); оператор переносит ленту с выходными данными на IBM 1401 (*д*); IBM 1401 печатает выходные данные (*е*)

Примерно после часа сбора пакета карты считывались на магнитную ленту, которую относили в машинный зал, где устанавливали на лентопротяжном устройстве. Затем оператор загружал специальную программу (прообраз сегодняшней операционной системы), которая считывала первое задание с ленты и запускала его. Выходные данные, вместо того чтобы идти на печать, записывались на вторую ленту. Завершив очередное задание, операционная система автоматически считывала с ленты следующее и начинала его обработку. После обработки всего пакета оператор снимал ленты с входной и выходной информацией, ставил новую ленту со следующим заданием, а готовые данные помещал на IBM 1401 для печати в автономном режиме (то есть без связи с главным компьютером).

Структура типичного входного задания показана на рисунке 1.2. Оно начиналось с карты $JOB, на которой указывалось максимальное время выполнения задания в минутах, загружаемый учетный номер и имя программиста. Затем поступала карта $FORTRAN, дающая операционной системе указание загрузить компилятор языка Фортран с системной магнитной ленты. За этой картой следовала программа, которую нужно было компилировать, а после нее шла карта $LOAD, указывающая операционной системе загрузить только что скомпилированную объектную программу. (Скомпилированные программы часто записывались на рабочих лентах, данные с которых могли стираться сразу после использования, и их загрузка должна была выполняться в явном виде.)

Следом шла карта $RUN, дающая операционной системе команду на выполнение программы с использованием данных, следующих за ней. Наконец, карта завершения $SEND отмечала конец задания. Эти примитивные управляющие перфокарты были предшественниками современных оболочек и интерпретаторов командной строки.



Рис. 1.2 – Структура типичного задания для операционной системы FMS

Большие компьютеры второго поколения использовались главным образом для научных и технических вычислений, таких как решение дифференциальных уравнений в частных производных, часто встречающихся в физике и инженерных задачах. В основном программы для них составлялись на языке Фортран и ассемблере, а типичными операционными системами были FMS (Fortran Monitor System) и IBSYS (операционная система, созданная корпорацией IBM для компьютера IBM 7094).

1.4 Третье поколение (1965–1980): интегральные схемы и многозадачность

К началу 1960-х гг. были созданы два различных, несовместимых друг с другом семейства компьютеров. С одной стороны, это были огромные научные компьютеры с пословной обработкой данных типа IBM 7094, которые использовались для числовых расчетов в науке и технике, с другой – коммерческие

компьютеры с посимвольной обработкой данных, такие как IBM 1401, широко используемые банками и страховыми компаниями для задач сортировки и распечатки данных.

Развитие и поддержка двух совершенно разных семейств была для производителей весьма расточительным делом. Кроме того, многим новым покупателям компьютеров сначала нужна была небольшая машина, однако позже ее возможностей становилось недостаточно и требовался более мощный компьютер, который работал бы с теми же самыми программами, но значительно быстрее.

Фирма IBM попыталась решить эти проблемы разом, выпустив серию машин IBM System/360. Это была серия программно-совместимых машин, в которой компьютеры варьировались от машин, сопоставимых с IBM 1401, до машин, значительно более мощных, чем IBM 7094. Эти компьютеры различались только ценой и производительностью (максимальным объемом памяти, быстродействием процессора, количеством возможных устройств ввода-вывода и т. д.). Так как все машины имели одинаковую структуру и набор команд, программы, написанные для одного компьютера, могли работать на всех других (по крайней мере, в теории). Кроме того, 360-я серия была разработана для поддержки как научных (то есть численных), так и коммерческих вычислений. Одно семейство машин могло удовлетворить нужды всех потребителей. В последующие годы, используя более современные технологии, корпорация IBM выпустила компьютеры, совместимые с 360-й серией машин, эти серии известны под номерами 370, 4300, 3080 и 3090. Самыми последними потомками этого семейства стали машины zSeries, хотя они уже значительно отличаются от оригинала.

Семейство компьютеров IBM/360 стало первой основной серией, использующей малые *интегральные схемы*, дававшие преимущество в цене и качестве по сравнению с машинами второго поколения, собранными на отдельных транзисторах. Корпорация IBM добилась моментального успеха, а идею семейства совместимых компьютеров скоро приняли на вооружение и все остальные основные производители. В компьютерных центрах до сих пор можно встретить потомков этих машин. В настоящее время они часто используются для управления огромными базами данных (например, для систем бронирования и продажи билетов на авиалиниях) или в качестве серверов узлов Интернета, которые должны обрабатывать тысячи запросов в секунду.

Основное преимущество «единого семейства» оказалось одновременно и величайшей его слабостью. По замыслу его создателей, все программное обеспечение, включая операционную систему OS/360, должно было одинаково хорошо работать на всех моделях компьютеров: и в небольших системах, которые часто заменяли машины IBM 1401 и применялись для копирования перфокарт на магнитные ленты, и на огромных системах, заменявших машины IBM 7094 и использовавшихся для расчета прогноза погоды и других сложных вычислений. Операционная система должна была хорошо работать как на машинах с небольшим количеством внешних устройств, так и на системах, применяющих эти устройства в большом количестве. Она должна была работать как в коммерческих, так и в научных областях. Более того, она должна была работать эффективно во всех этих различных сферах применения.

Но ни IBM, ни кому-либо еще так и не удалось создать программное обеспечение, удовлетворяющее всем этим противоречивым требованиям. В результате появилась громоздкая и чрезвычайно сложная операционная система, примерно на два или три порядка превышающая по объему FMS. Она состояла из миллионов строк, написанных на ассемблере тысячами программистов, содержала тысячи и тысячи ошибок, что повлекло за собой непрерывный поток новых версий, в которых предпринимались попытки исправления этих ошибок. В каждой новой версии устранялась только часть ошибок, вместо них появлялись новые, так что общее их количество, скорее всего, оставалось постоянным.

Один из разработчиков OS/360, Фред Брукс (Fred Brooks), впоследствии написал остроумную и язвительную книгу с описанием своего опыта работы с OS/360. Мы не можем здесь дать полную оценку этой книги, но достаточно будет сказать, что на ее обложке изображено стадо доисторических животных, увязших в яме с битумом.

Несмотря на свой огромный объем и имеющиеся недостатки, OS/360 и подобные ей операционные системы третьего поколения, созданные другими производителями компьютеров, достаточно неплохо отвечали запросам большинства клиентов. Они даже сделали популярными несколько ключевых технических приемов, отсутствовавших в операционных системах второго поколения. Самым важным достижением явилась *многозадачность*. На компьютере IBM 7094, когда текущая работа приостанавливалась в ожидании операций ввода-вывода с магнитной ленты или других устройств, центральный процессор просто бездействовал до окончания операции ввода-вывода. При сложных научных вычислениях устройства ввода-вывода задействовались менее интен-

сивно, поэтому время, впустую потраченное на ожидание, было не столь существенным. Но при коммерческой обработке данных время ожидания устройства ввода-вывода могло занимать 80 или 90% всего рабочего времени, поэтому для устранения длительных простоев весьма дорогостоящего процессора нужно было что-то предпринимать.

Решение этой проблемы заключалось в разбиении памяти на несколько частей, называемых разделами, в каждом из которых выполнялось отдельное задание, как показано на рисунке 1.3. Пока одно задание ожидало завершения работы устройства ввода-вывода, другое могло использовать центральный процессор. Если в оперативной памяти содержалось достаточное количество заданий, центральный процессор мог быть загружен почти на все 100% времени. Множество одновременно хранящихся в памяти заданий требовало наличия специального оборудования для защиты каждого задания от возможного незаконного присваивания областей памяти и нанесения вреда со стороны других заданий. Компьютеры 360-й серии и другие системы третьего поколения были оборудованы для этих целей специальными аппаратными средствами.

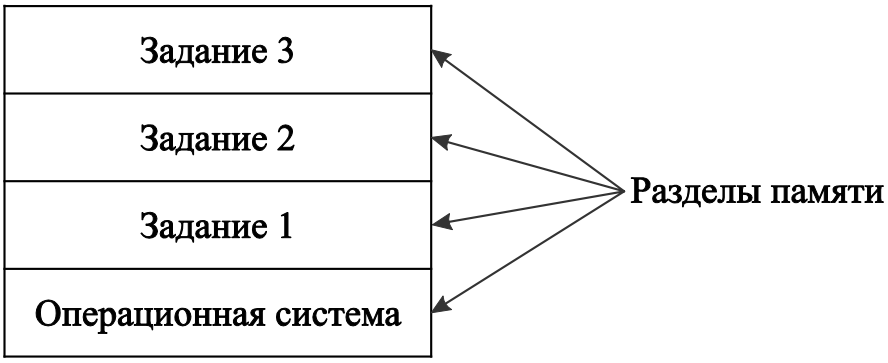


Рис. 1.3 – Многозадачная система с тремя заданиями в памяти

Другим важным плюсом операционных систем третьего поколения стала способность считывать задание с перфокарт на диск по мере того, как их приносили в машинный зал. При окончании выполнения каждого текущего задания операционная система могла загружать новое задание с диска в освободившийся раздел памяти и запускать это задание. Этот технический прием называется «подкачкой» данных, или спулингом (*spooling* – это английское слово, которое произошло от аббревиатуры Simultaneous Peripheral Operation On Line, то есть совместная периферийная операция в интерактивном режиме), и его также используют для выдачи полученных данных. С появлением подкачки отпала

надобность как в 1401-х машинах, так и в многочисленных переносах магнитных лент.

Хотя операционные системы третьего поколения неплохо справлялись с большинством научных вычислений и крупных коммерческих задач по обработке данных, но по своей сути они были все еще разновидностью систем пакетной обработки. Многие программисты тосковали по первому поколению машин, когда они могли распоряжаться всей машиной в течение нескольких часов и имели возможность быстро производить отладку своих программ. При системах третьего поколения промежуток времени между передачей задания и возвращением результатов часто составлял несколько часов, так что единственная поставленная не в том месте запятая могла стать причиной сбоя при компиляции, и получалось, что программист тратил половину дня впустую.

Желание сократить время ожидания ответа привело к разработке режима *разделения времени* – варианту многозадачности, при котором у каждого пользователя есть свой диалоговый терминал. Если двадцать пользователей зарегистрированы в системе, работающей в режиме разделения времени, и семнадцать из них думают, беседуют или пьют кофе, то центральный процессор по очереди предоставляется трем пользователям, желающим работать на машине. Так как люди, отлаживая программы, обычно выдают короткие команды (например, компилировать процедуру на пяти страницах) чаще, чем длинные (например, упорядочить файл с миллионами записей), то компьютер может обеспечивать быстрое интерактивное обслуживание нескольких пользователей. При этом он может работать над большими пакетами в фоновом режиме, когда центральный процессор не занят другими заданиями. Первая универсальная система с режимом разделения времени *CTSS* (Compatible Time Sharing System) была разработана в Массачусетском технологическом институте (M.I.T.) на специально переделанном компьютере IBM 7094. Однако режим разделения времени не стал действительно популярным до тех пор, пока на машинах третьего поколения не получили широкого распространения необходимые технические средства защиты.

После успеха системы CTSS Массачусетский технологический институт, исследовательские лаборатории Bell Labs и корпорация General Electric (главный на то время изготовитель компьютеров) решили начать разработку универсальной общей компьютерной системы – машины, которая должна была поддерживать одновременную работу сотен пользователей в режиме разделения времени. За основу была взята система распределения электроэнергии. Когда

вам нужна электроэнергия, вы просто вставляете штепсель в розетку и получаете энергии столько, сколько вам нужно. Проектировщики этой системы, известной как *MULTICS* (MULTiplexed Information and Computing Service – мультиплексная информационная и вычислительная служба), представляли себе одну огромную вычислительную машину, воспользоваться услугами которой мог любой проживающий в окрестностях Бостона человек. Мысль о том, что машины в десять тысяч раз мощнее, чем их мэйнфрейм GEE645, будут продаваться миллионами по тысяче долларов за штуку всего лишь через каких-нибудь 40 лет, показалась бы им чистейшей научной фантастикой. Это было бы сродни мысли о сверхзвуковых трансатлантических подводных поездах для современного человека.

Успех системы MULTICS был весьма неоднозначен. Она разрабатывалась с целью обеспечения сотен пользователей машиной, немногим более мощной, чем персональный компьютер с процессором Intel 386, но имеющей более широкие возможности по вводу-выводу данных. По тем временам это была не такая уж безумная идея, поскольку тогда люди умели создавать небольшие, эффективные программы, то есть обладали мастерством, которое впоследствии было утрачено. Существовало много причин, по которым система MULTICS не получила глобального распространения. Не последнюю роль здесь сыграл и тот факт, что эта система была написана на языке PL/I, а компилятор языка PL/I появился лишь через несколько лет, к тому же первую версию этого компилятора можно было назвать работоспособной лишь с большой натяжкой. Кроме того, система MULTICS была слишком претенциозна для своего времени, что делало ее во многом похожей на аналитическую машину Чарльза Бэббиджа в XIX столетии.

Итак, MULTICS внесла множество оригинальных идей в компьютерную литературу, но превратить ее в серьезный продукт и добиться весомого коммерческого успеха оказалось намного труднее, чем ожидалось. Исследовательские лаборатории Bell Labs вышли из проекта, а компания General Electric вообще ушла с рынка компьютерных технологий. Однако Массачусетский технологический институт проявил упорство и со временем довел систему до работоспособного состояния. В конце концов она была продана компании Honeywell, выкупившей компьютерный бизнес General Electric, и установлена примерно в 80 больших компаниях и университетах по всему миру. Несмотря на свою малочисленность, пользователи MULTICS продемонстрировали приверженность этой системе. Компании General Motors, Ford и Агентство нацио-

нальной безопасности США отказались от системы MULTICS только в конце 1990-х гг., спустя 30 лет после ее выпуска и после многолетних попыток заставить компанию Honeywell обновить оборудование.

К настоящему времени идея создания такого «общего» компьютера уже выдохлась, но она может возродиться в виде крупных централизованных интернет-серверов, выполняющих основную часть работы, к которым будут подключены относительно примитивные пользовательские машины. Скорее всего, существование такой системы будет мотивировано тем, что большинство пользователей не захочет заниматься администрированием все более и более сложных и привередливых компьютерных систем и предпочтет доверить эту работу команде профессионалов, работающих на компанию, обслуживающую сервер. Электронная коммерция уже развивается в этом направлении, компании, занимающиеся электронной коммерцией, используют мультипроцессорные серверы, с которыми соединяются простые машины клиентов. Все это очень напоминает замысел системы MULTICS.

Несмотря на коммерческую неудачу, система MULTICS оказала существенное влияние на последующие операционные системы. Этот факт описан во многих статьях и книгах Системе MULTICS также посвящен до сих пор активный веб-сайт www.multicians.org, содержащий большой объем информации о системе, ее проектировщиках и пользователях.

Еще одной важной разработкой времен третьего поколения были мини-компьютеры, невероятный взлет популярности которых начался с выпуска корпорацией DEC машины PDP-1. Компьютеры PDP-1 обладали оперативной памятью, состоящей всего лишь из 4К 18-битовых слов, но стоили всего 120 тысяч долларов за одну машину (это меньше 5% от цены IBM 7094) и поэтому расхватывались как горячие пирожки. Некоторые виды нечисловой работы они выполняли так же быстро, как и машины IBM 7094, в результате чего родилась новая отрасль производства. За этой машиной быстро последовала целая серия компьютеров PDP других моделей (в отличие от семейства IBM, полностью несовместимых), и как кульминация появилась модель PDP-11.

Кен Томпсон (Ken Thompson), один из ведущих специалистов Bell Labs, работавший над проектом MULTICS, чуть позже нашел мини-компьютер PDP-7, которым никто не пользовался, и решил написать упрощенную однопользовательскую версию системы MULTICS. Эта система позже была переписана и на другие платформы и названа *UNICS* – универсальная компьютерная система. И при очередном копировании две последние буквы «CS» были со-

кращены до «X», так работа переросла в операционную систему UNIX®, ставшую популярной в академических кругах, в правительственных учреждениях и во многих компаниях. Из-за широкой доступности исходного кода различные организации создали свои собственные (несовместимые) версии, что привело к полному хаосу. Были разработаны две основные версии: System V корпорации AT&T и BSD (Berkeley Software Distribution) Калифорнийского университета Беркли. У них также были последующие варианты. Чтобы появилась возможность писать программы, работающие в любой UNIX-системе, Институт инженеров по электротехнике и электронике (IEEE) разработал стандарт системы UNIX, названный POSIX, который в настоящее время поддерживается большинством версий UNIX. Стандарт POSIX определяет минимальный интерфейс системных вызовов, который должны поддерживать совместимые с ним системы UNIX. Фактически на данный момент POSIX-интерфейс поддерживается также рядом других операционных систем.

Между прочим стоит упомянуть, что в 1987 г. автор многих книг по операционным системам Эндрю Таненбаум выпустил в образовательных целях небольшой клон системы UNIX, так называемую систему *MINIX*. Функционально система MINIX очень похожа на UNIX, включая поддержку стандарта POSIX. С тех пор исходная версия превратилась в MINIX 3, систему, обладающую высокой модульностью и ориентированную на достижение очень высокой надежности. Она способна «на лету» определять и заменять дефектные и даже поврежденные модули (например, драйверы устройств ввода-вывода) без перезагрузки и нарушения хода выполнения запущенных программ. Также доступна книга, в которой описывается ее внутренняя работа, а в приложении приводится исходный код. Система MINIX 3 имеется в свободном доступе (включая весь исходный код) в Интернете по адресу www.minix3.org.

Желание получить свободно распространяемую версию MINIX (в отличие от образовательной) привело к тому, что финский студент Линус Торвальдс (Linus Torvalds) создал систему *Linux*. Система MINIX стала непосредственным вдохновляющим фактором и основой для этой разработки, которая первоначально поддерживала различные особенности MINIX (например, ее файловую систему). С тех пор система Linux во многом была существенно расширена, но все еще сохраняет исходную структуру, общую для MINIX и UNIX. Генеалогическое древо операционных систем семейства UNIX приведено на рисунке 1.4.

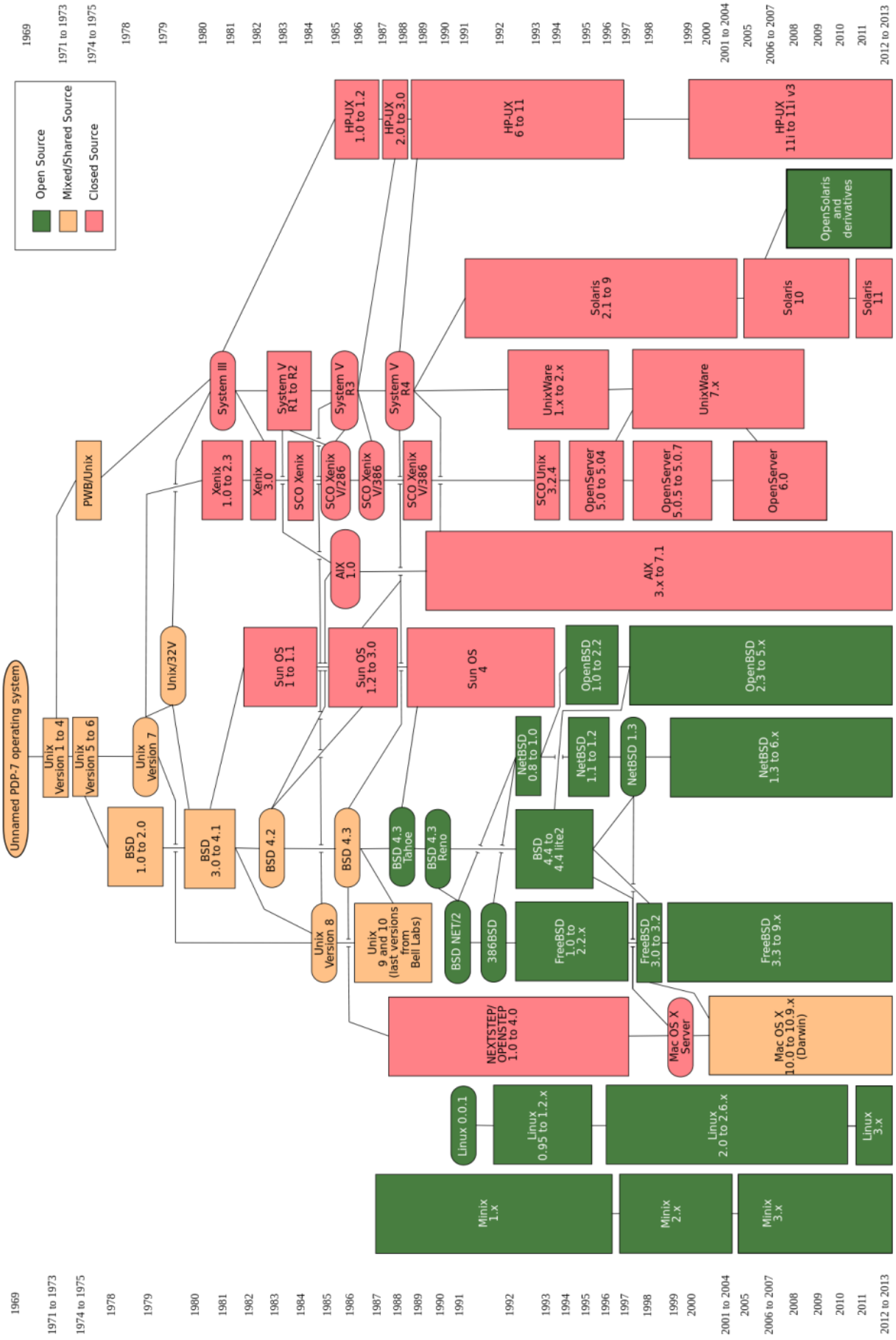


Рис. 1.4 – Генеалогическое древо UNIX-систем

1.5 Четвертое поколение (с 1980 г. по наши дни): персональные компьютеры

Следующий период эволюции операционных систем связан с появлением БИС – больших интегральных схем (LSI, Large Scale Integration) – кремниевых микросхем, содержащих тысячи транзисторов на одном квадратном сантиметре. С точки зрения архитектуры персональные компьютеры (первоначально называемые *микрокомпьютерами*) были во многом похожи на мини-компьютеры класса PDP-11, но, конечно же, отличались по цене. Если появление мини-компьютеров позволило отделам компаний и факультетам университетов иметь собственный компьютер, то с появлением микропроцессоров каждый человек получил возможность купить свой собственный персональный компьютер.

В 1974 г., когда корпорация Intel выпустила Intel 8080 – первый универсальный 8-разрядный центральный процессор, – для него потребовалась операционная система, с помощью которой можно было бы протестировать новинку. Корпорация Intel привлекла к разработкам и написанию нужной операционной системы одного из своих консультантов Гэри Килдэлла (Gary Kildall). Сначала Килдэлл с другом сконструировали контроллер для 8-дюймового гибкого диска, недавно выпущенного компанией Shugart Associates, и подключили этот диск к процессору Intel 8080. Таким образом появился первый микрокомпьютер с диском. Затем Килдэлл создал дисковую операционную систему, названную CP/M (Control Program for Microcomputers – управляющая программа для микрокомпьютеров). Когда Килдэлл заявил о своих правах на CP/M, корпорация Intel удовлетворила его просьбу, поскольку не думала, что у микрокомпьютеров с диском есть будущее. Позже Килдэлл создал свою компанию Digital Research для дальнейшего развития и продажи CP/M.

В 1977 г. компания Digital Research переработала CP/M, чтобы сделать ее пригодной для работы на микрокомпьютерах с процессорами Intel 8080 или Zilog Z80, а также с другими процессорами. Затем было написано множество прикладных программ, работающих в CP/M, что позволило этой системе занимать высшую позицию в мире микрокомпьютеров целую пятилетку.

В начале 1980-х гг. корпорация IBM разработала IBM PC (Personal Computer персональный компьютер) и начала искать для него программное обеспечение. Сотрудники IBM связались с Биллом Гейтсом, чтобы получить лицензию на право использования его интерпретатора языка Бейсик. Они также поинтересовались, не знает ли он операционную систему, которая работала бы

на IBM PC. Гейтс посоветовал обратиться к Digital Research, тогда главенствующей компании в области операционных систем. Но Килдэлл отказался встречаться с IBM, послав вместо себя своего подчиненного. Что еще хуже, его адвокат даже отказался подписывать соглашение о неразглашении, касающееся еще не выпущенного IBM PC, чем полностью испортил дело. Корпорация IBM снова обратилась к Гейтсу с просьбой обеспечить ее операционной системой. После этого повторного обращения Гейтс выяснил, что у местного изготовителя компьютеров, Seattle Computer Products, есть подходящая операционная система DOS (Disk Operating System – дисковая операционная система). Он направился в эту компанию с предложением выкупить DOS (предположительно за $50 000), которое компания Seattle Computer Products с готовностью приняла. Затем Гейтс создал пакет программ DOS/BASIC, и пакет был куплен IBM. Когда корпорация IBM захотела внести в операционную систему ряд усовершенствований, Билл Гейтс пригласил для этой работы Тима Патерсона (Tim Paterson), человека, написавшего DOS, ставшего первым служащим Microsoft – еще не оперившейся компании Гейтса. Видоизмененная система была переименована в MS-DOS (MicroSoft Disk Operating System) и быстро заняла доминирующее положение на рынке IBM PC. Самым важным оказалось решение Гейтса (как оказалось, чрезвычайно мудрое) продавать MS-DOS компьютерным компаниям для установки вместе с их оборудованием, в отличие от попыток Килдэлла продавать CP/M конечным пользователям (по крайней мере, на начальной стадии).

Когда в 1983 г. появился компьютер IBM PC/AT (являющийся дальнейшим развитием семейства IBM PC) с центральным процессором Intel 80286, система MS-DOS уже прочно стояла на ногах, а CP/M доживала свои последние дни. Позже система MS-DOS широко использовалась на компьютерах с процессорами 80386 и 80486. Хотя первоначальная версия MS-DOS была довольно примитивна, последующие версии системы включали в себя расширенные функции, многие из которых были позаимствованы у UNIX. (Корпорация Microsoft была хорошо знакома с системой UNIX и в первые годы своего существования даже продавала ее микрокомпьютерную версию XENIX.)

CP/M, MS-DOS и другие операционные системы для первых микрокомпьютеров полностью основывались на командах, вводимых пользователем с клавиатуры. Со временем, благодаря исследованиям, проведенным в 1960-е гг. Дагом Энгельбартом (Doug Engelbart) в научно-исследовательском институте Стэнфорда (Stanford Research Institute), ситуация изменилась. Энгельбарт изобрел *графический интерфейс пользователя* (GUI, Graphical User

Interface), вкупе с окнами, значками, системами меню и мышью. Эту идею переняли исследователи из Xerox PARC и воспользовались ею в создаваемых ими машинах.

Однажды Стив Джобс (Steve Jobs), один из авторов компьютера Apple, созданного в его гараже, посетил PARC, где увидел GUI и сразу понял уровень заложенного в него потенциала, недооцененного руководством компании Xerox. Затем Джобс приступил к созданию компьютера Apple, оснащенного графическим пользовательским интерфейсом. Этот проект привел к созданию компьютера Lisa, который оказался слишком дорогим и не имел коммерческого успеха. Вторая попытка Джобса, компьютер Apple Macintosh имел огромный успех не только потому, что он был значительно дешевле, чем Lisa, но и потому, что обладал более дружественным пользовательским интерфейсом, предназначенным для пользователей, не разбиравшихся в компьютерах и к тому же совершенно не стремившихся чему-то обучаться. Компьютеры Macintosh нашли широкое применение у представителей творческих профессий – художников-дизайнеров, профессиональных цифровых фотографов и профессиональных производителей цифровой видеопродукции, которые приняли их с восторгом.

Когда корпорация Microsoft решила создать преемника MS-DOS, она была под большим впечатлением от успеха Macintosh. В результате появилась основанная на применении графического интерфейса пользователя система под названием Windows, первоначально являвшаяся надстройкой над MS-DOS (то есть она больше была похожа на оболочку, чем на настоящую операционную систему). На протяжении примерно 10 лет, с 1985 по 1995 г., Windows была просто графической оболочкой, работавшей поверх MS-DOS. Однако в 1995 г. была выпущена самостоятельная версия Windows – Windows 95. Она непосредственно выполняла большинство функций операционной системы, используя входящую в свой состав систему MS-DOS только для загрузки, а также при выполнении старых программ, разработанных для MS-DOS. В 1998 г. была выпущена слегка модифицированная версия этой системы, получившая название Windows 98. Тем не менее обе эти системы – и Windows 95, и Windows 98 – все еще содержали изрядное количество кода, написанного на ассемблере для 16-разрядных процессоров Intel.

Другой операционной системой Microsoft была Windows NT (NT означает New Technology – новая технология), которая на определенном уровне совместима с Windows 95. Однако она была написана заново и представляла собой

полноценную 32-разрядную систему. Ведущим разработчиком Windows NT был Дэвид Катлер (David Cutler), который также был одним из разработчиков операционной системы VAX VMS, поэтому некоторые идеи из VMS присутствуют и в NT (причем столько, что владелец VMS, компания DEC, предъявила иск корпорации Microsoft. Конфликт был улажен во внесудебном порядке за приличную сумму). Microsoft ожидала, что первая же версия вытеснит MS-DOS и все другие версии Windows, поскольку она намного превосходила их, но надежды не оправдались. Только операционной системе Windows NT 4.0 наконец-то удалось завоевать высокую популярность, особенно в корпоративных сетях. Пятая версия Windows NT в начале 1999 г. была переименована в Windows 2000. Она предназначалась для замены обеих версий – Windows 98 и Windows NT 4.0.

Но полностью этим планам также не суждено было сбыться, поэтому Microsoft выпустила еще одну версию Windows 98 под названием Windows Me (Millennium edition – выпуск тысячелетия). В 2001 г. была выпущена слегка обновленная версия Windows 2000, названная Windows XP. Эта версия выпускалась намного дольше, по существу заменяя все предыдущие версии Windows. Затем, в январе 2007 г., Microsoft выпустила окончательную версию преемника Windows XP под названием Vista. У нее был новый графический интерфейс, Aero, и множество новых или обновленных пользовательских программ. Microsoft надеялась, что она полностью заменит Windows XP, но этот процесс мог бы затянуться на значительную часть десятилетия. Новая система вызвала массу недовольства со стороны пользователей, которые не только не активно переходили на систему Vista, но даже иногда требовали установить на новые компьютеры устаревающую уже систему Windows XP.

Были критики, назвавшие Windows Vista самым большим за всю историю провалом Microsoft. Тогда Microsoft была вынуждена коренным образом переделать все недостатки и выпустить более комфортную для пользователей систему. Такой системой стала Windows 7. Название происходит от порядкового номера ядра операционных систем семейства Windows. Для того чтобы избежать большинства ошибок прошлых реализаций, Microsoft пошла на беспрецедентный шаг: перед официальным выходом в свет в Интернет была выложена бета-версия для свободного скачивания. Каждый желающий мог скачать систему, установить, опробовать в действии и высказать все свои замечания. Именно этот шаг позволил, более ли менее, объективно учесть мнения тысяч пользова-

телей и заметно повысить удобство, а следовательно, и популярность операционной системы Windows 7.

Выход операционных систем Windows 8/8.1, а также 10, ознаменовал переход к новой концепции пользовательской среды – online работа с глубокой интеграцией облачных технологий. Концепция интерфейса была изменена для облегчения работы на планшетных компьютерах, интеграция облачных хранилищ позволила обеспечить пользователя доступом к его файлам в любой точке мира. Временная диаграмма появления операционных систем Microsoft приведена на рисунке 1.5.

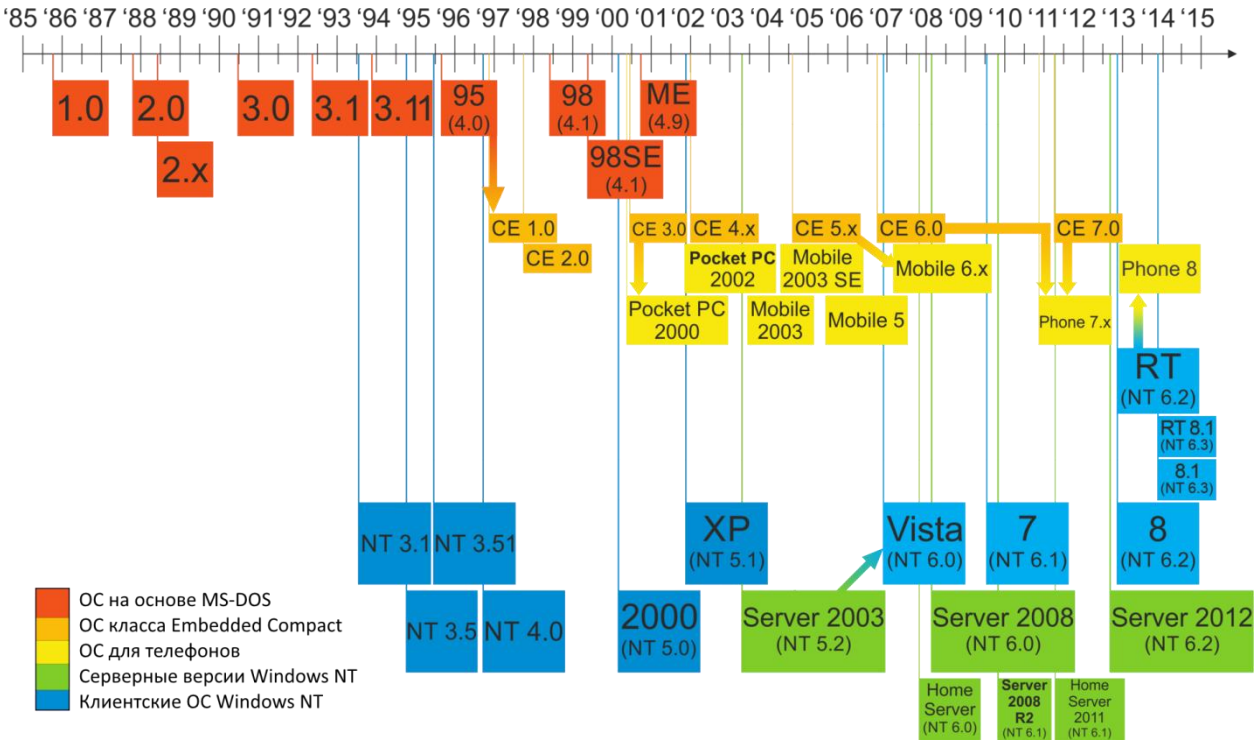


Рис. 1.5 – Временная диаграмма появления операционных систем Microsoft

Другим основным конкурентом в мире персональных компьютеров является операционная система UNIX (и различные производные от этой системы). UNIX имеет более сильные позиции на сетевых и промышленных серверах, также она находит все более широкое распространение и на настольных компьютерах, особенно в быстро развивающихся странах, таких как Индия и Китай. На компьютерах с процессором Pentium популярной альтернативой Windows для студентов и постоянно растущего числа корпоративных пользователей становится операционная система Linux.

Операционная система FreeBSD также является популярной производной от UNIX, порожденной проектом BSD в Беркли. Все современные компьютеры Macintosh работают на модифицированной версии *FreeBSD*. UNIX также является стандартом на рабочих станциях, оснащенных высокопроизводительными RISC-процессорами, среди которых машины, продающиеся компаниями Hewlett-Packard и Sun Microsystems. Хотя многие пользователи UNIX, особенно опытные программисты, отдают предпочтение интерфейсу на основе командной строки, практически все UNIX-системы поддерживают систему управления окнами *X Window System* (или *X11*), созданную в Массачусетском технологическом институте. Эта система выполняет основные операции по управлению окнами, позволяя пользователям создавать, удалять, перемещать окна и изменять их размеры, используя мышь. Зачастую в качестве надстройки над X11 можно использовать полноценный графический пользовательский интерфейс, например *Gnome, KDE, XFCE*, придавая UNIX внешний вид и поведение, чем-то напоминающее Macintosh или Microsoft Windows.

В середине 1980-х гг. начало развиваться интересное явление – рост сетей персональных компьютеров, работающих под управлением *сетевых операционных систем и распределенных операционных систем*. В сетевых операционных системах пользователи знают о существовании множества компьютеров и могут войти в систему удаленной машины и скопировать файлы с одной машины на другую. На каждой машине работает своя локальная операционная система и имеется свой собственный локальный пользователь (или пользователи).

Сетевые операционные системы не имеют существенных отличий от однопроцессорных операционных систем. Ясно, что им нужен контроллер сетевого интерфейса и определенное низкоуровневое программное обеспечение, чтобы управлять этим контроллером, а также программы для осуществления входа в систему удаленной машины и удаленного доступа к файлам, но эти дополнения не изменяют основную структуру операционной системы.

В отличие от этого, распределенная операционная система представляется своим пользователям как традиционная однопроцессорная система, хотя на самом деле в ее составе работает множество процессоров. Пользователям совершенно не обязательно знать, где именно выполняются их программы или где размещены их файлы; все это должно автоматически и эффективно управляться самой операционной системой.

Настоящим распределенным операционным системам требуется намного больше изменений, не ограничивающихся простым добавлением незначитель-

ного объема кода к однопроцессорной операционной системе, поскольку распределенные и централизованные системы существенно отличаются друг от друга. Например, распределенные системы часто позволяют приложениям работать одновременно на нескольких процессорах, для чего требуются более сложные алгоритмы распределения работы процессоров, чтобы оптимизировать степень параллельной обработки данных. Наличие задержек при передаче данных по сети часто подразумевают, что эти (и другие) алгоритмы должны работать в условиях неполной, устаревшей или даже неверной информации. Такая ситуация в корне отличается от работы однопроцессорной системы, где последняя обладает полной информацией о своем состоянии [1].



Контрольные вопросы по главе 1

1. В честь кого назван первый язык программирования?
2. На модифицированной версии какой ОС работают все современные компьютеры Apple?
3. Какая ОС возникла раньше других?
4. Какую ОС разработал Кен Томпсон?
5. Какая ОС, работавшая на семействе Intel 8080, была в течение более пяти лет наиболее популярной?

2 Интерфейсы пользователя системы

2.1 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 рисунке 2.1 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:

1) шины управления;
2) шины адреса;
3) шины данных.

Рассмотрим кратко назначение других аппаратных блоков.

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 рисунке 2.2 приведена структура наиболее типичной машинной команды. Здесь *КОП* – код операции [2].

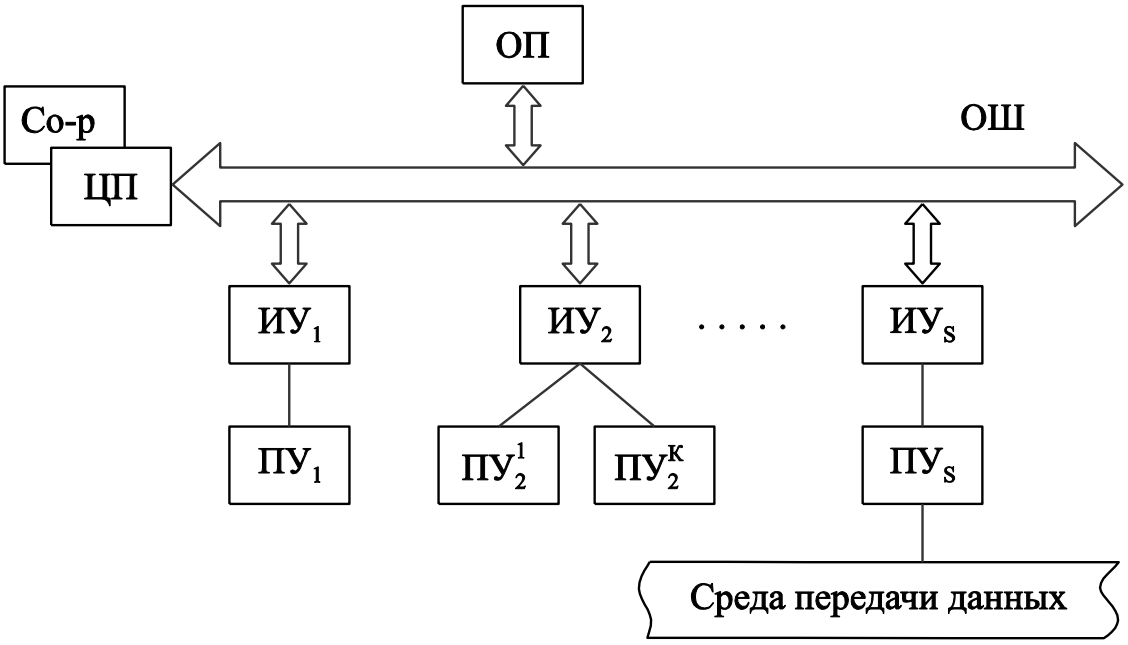


Рис. 2.1 – Однопроцессорная ЭВМ с общей шиной

КОП	Операнд 1	Операнд 2

Рис. 2.2 – Структура машинной команды



КОП *– 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.

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. 2.3). 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.

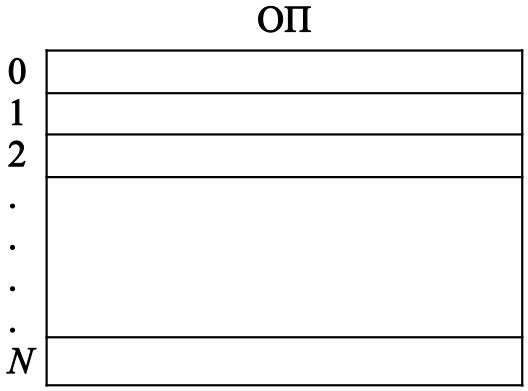


Рис. 2.3 – Логическая структура ОП



. . . . . .

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. [3].

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.



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 рисунке 2.4 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.

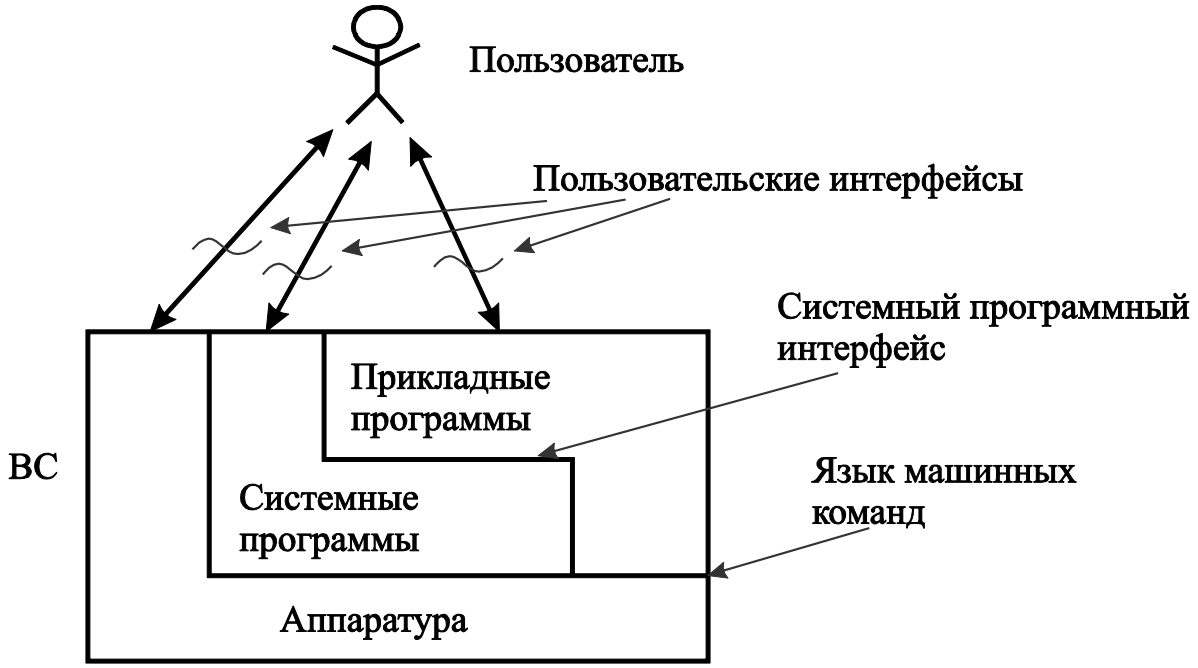


Рис. 2.4 – Укрупненное представление вычислительной системы



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

На рисунке 2.5 приведена классификация системных программ.

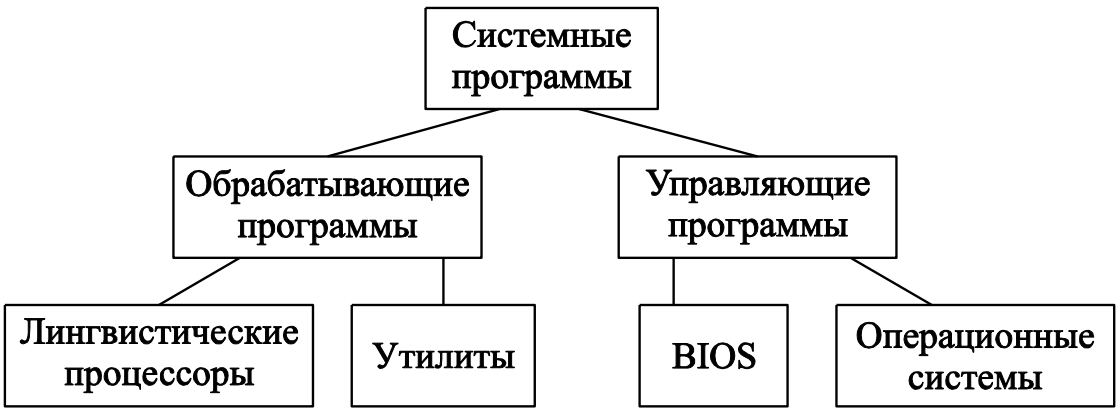


Рис. 2.5 – Классификация системных программ

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. 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 (ОС).



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 [2].

2.2 Файлы

Вся информация, обрабатываемая ВС, содержится в ней на устройствах: ОП, ВП, устройства 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 Гбит (1 Г = $(1024)^3$). Так как работать с такой длинной строкой чрезвы-

чайно 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, то оно будет рассмотрено нами в другом месте.

Прежде чем обсуждать имена файлов, рассмотрим небольшую классификацию имен объектов, находящихся под управлением ОС (рис. 2.6). В качестве первого признака классификации использовано назначение имени.

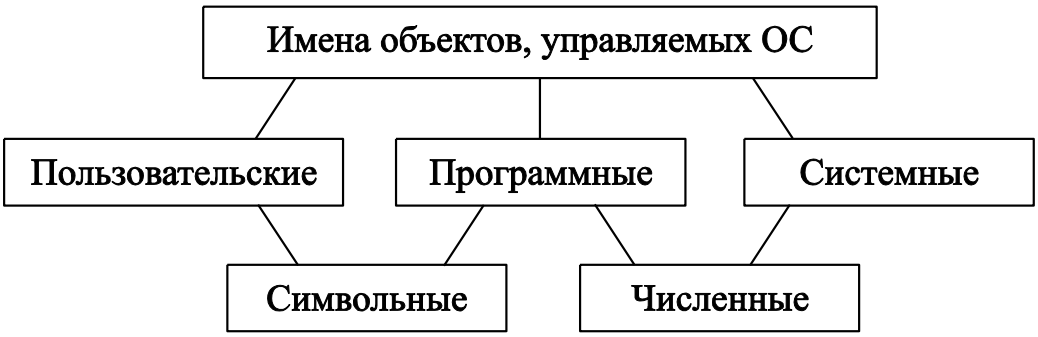


Рис. 2.6 – Классификация имен объектов, управляемых ОС



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 символьными и численными. В дальнейшем мы будем часто использовать данную классификацию имен.

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 с 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, то 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 рисунке 2.7 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 рисунке 2.7 два файла имеют одина-

ковое простое имя `a.txt`, но абсолютные имена у них разные: `/home/vlad/a.txt` и `/home/andrei/a.txt`.

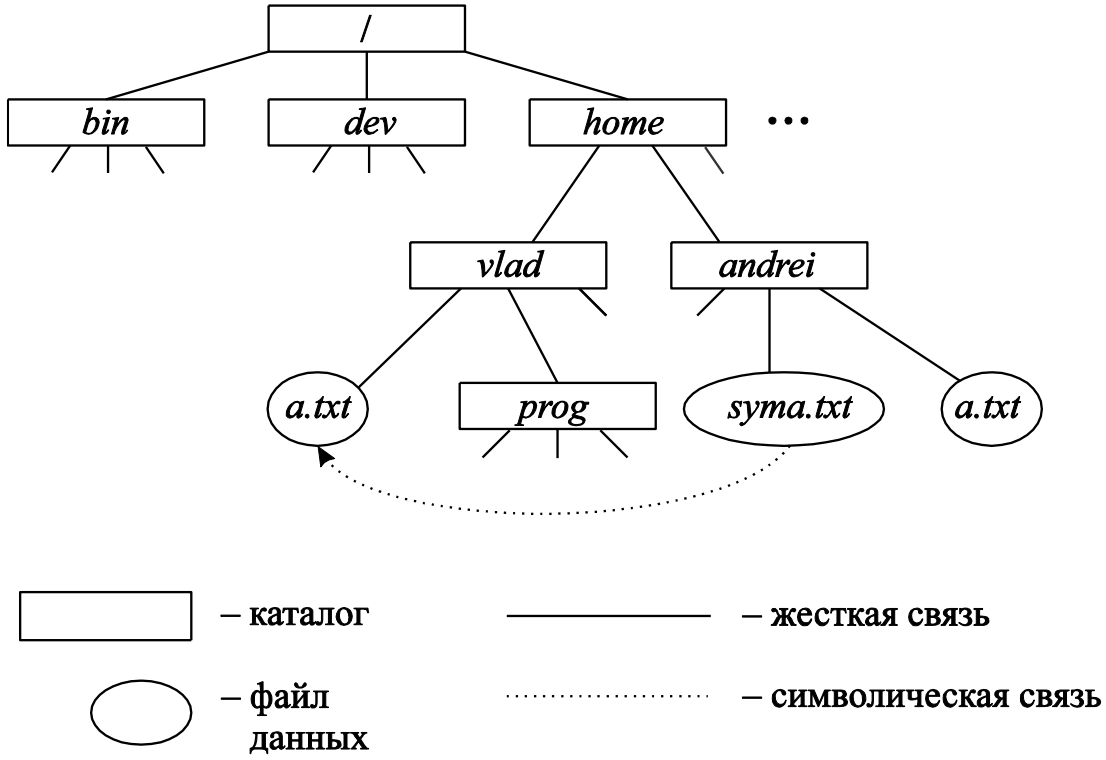


Рис. 2.7 – Фрагмент файловой структуры UNIX

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.

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 рисунке 2.7 файл syma.txt является символической связью с файлом a.txt. То есть содержимым файла syma.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» не допустима. 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 *COM*1;
- /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 [3].

2.3 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 выглядит так:

```
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 используют 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 каталог*:

а) 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;

б) 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 рисунке 2.7).

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`, вообще говоря, 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*:

а) 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 п. 4.2;

б) 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, если 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 имена, начинающиеся с символа «`.`» не выводятся).

3. *Копирование, переименование и перенос файлов*:

а) копирование файла (в MS-DOS – `copy`):

`cp` `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 содержи-

мым, после чего новый файл регистрируется в конечном каталоге. Для копирования из каталога в каталог требуется, чтобы был записан флаг `-r`;

б) 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;

в) создание жестких и символических связей (в MS-DOS – нет):

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*:

а) 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;

б) 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*:

а) создание новых текстовых файлов и корректировка существующих. Данную функцию выполняют утилиты, называемые *текстовыми редакторами*. Примеры текстовых редакторов: `ed, ee, 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;

б) 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»);

в) 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 лексографическом порядке. По умолчанию результат выводится на экран.

Два флага этой команды:

- `-u` – при наличии нескольких одинаковых строк результат содержит только одну строку;
- `-o файл` – вывод результата делается не на экран, а в заданный файл;

г) вывод текста, вводимого с клавиатуры, на экран и одновременное копирование этого текста в заданный файл (файлы):

tee файлы.

Один из флагов этой команды:

- `-a` – запись текста не в начало файла (при этом файл создается заново), а его добавление в конец существующего файла (файлов);

д) 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*:

а) 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 – именованный 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;

б) 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 [2].

7. *Выдача справочной информации*:

а) 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;

б) 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` – минута;
- `yy` – последние две цифры года (необязательная часть параметра команды).

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

в) 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;

г) 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 си-

стемных 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 [1].

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 (*псевдокурсор* – 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. Вспомним, что целью применения любой ВС является выполнение прикладных машинных программ. В следующем разделе рассмотрим применение лингвистических процессоров для получения таких программ.

2.4 Трансляторы

Программисты – не самая многочисленная, но очень важная часть пользователей ВС. Конечной задачей любого программирования является получе-

ние 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 выдаёт 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 параграфе, 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 рисунке 2.8. 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.

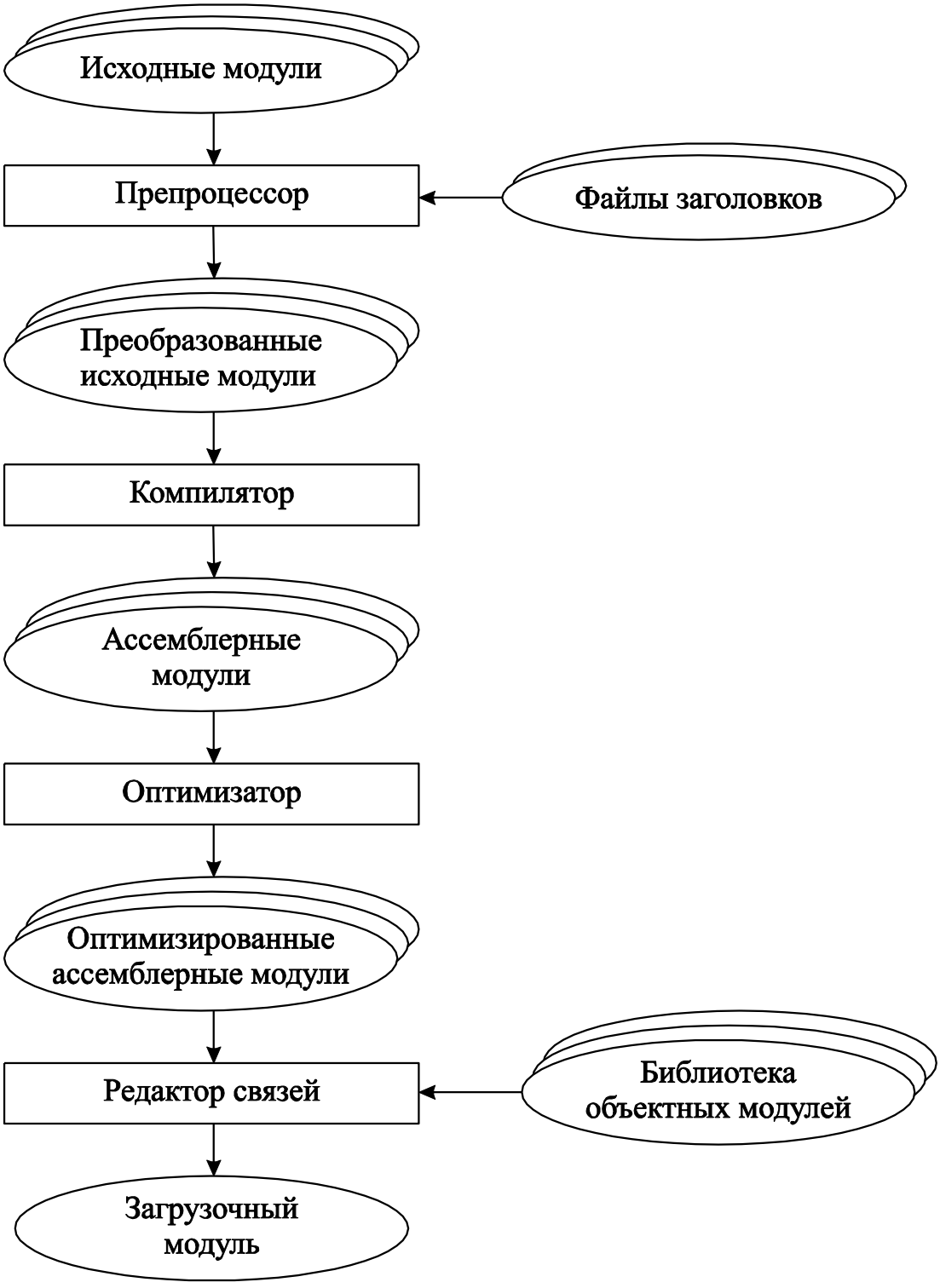


Рис. 2.8 – Преобразование исходной программы в загрузочный модуль

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 `.c`. Любой исходный модуль состоит из операторов двух типов – псевдооператоров и исполнительных операторов. *Исполнительный оператор* – оператор исходной программы, преобразуемый в резуль-

тате 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, выполняющий 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. 2.8) 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 рисунке 2.8 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 – *оптимизатор*.

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 `.o`. При этом частично решается задача по-

лучения 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 его. *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 его.

При получении объектного модуля транслятор-ассемблер проставляет 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: `cc`, `cpp`, `gcc`, `c++`, `g++`. Команды для запуска этих систем программирования похожи на команды для запуска утилит (см. п. 2.3). Точно так же в качестве параметров команды задаются обрабатываемые файлы, а ее функции уточняются с помощью флагов. Вот некоторые флаги для программы `cc`:

1) `-o` – требуется дать исполняемому файлу программы имя, отличное от `a.out`;
2) `-c` – требуется получить не загрузочный, а объектный модуль;
3) `-lимя` – при получении загрузочного модуля использовать требуемую библиотеку объектных модулей.

Благодаря наличию системы программирования программист работает на *виртуальной машине пользователя системы программирования* (рис. 2.9). 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 программирования 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 обрабатывающих программ.

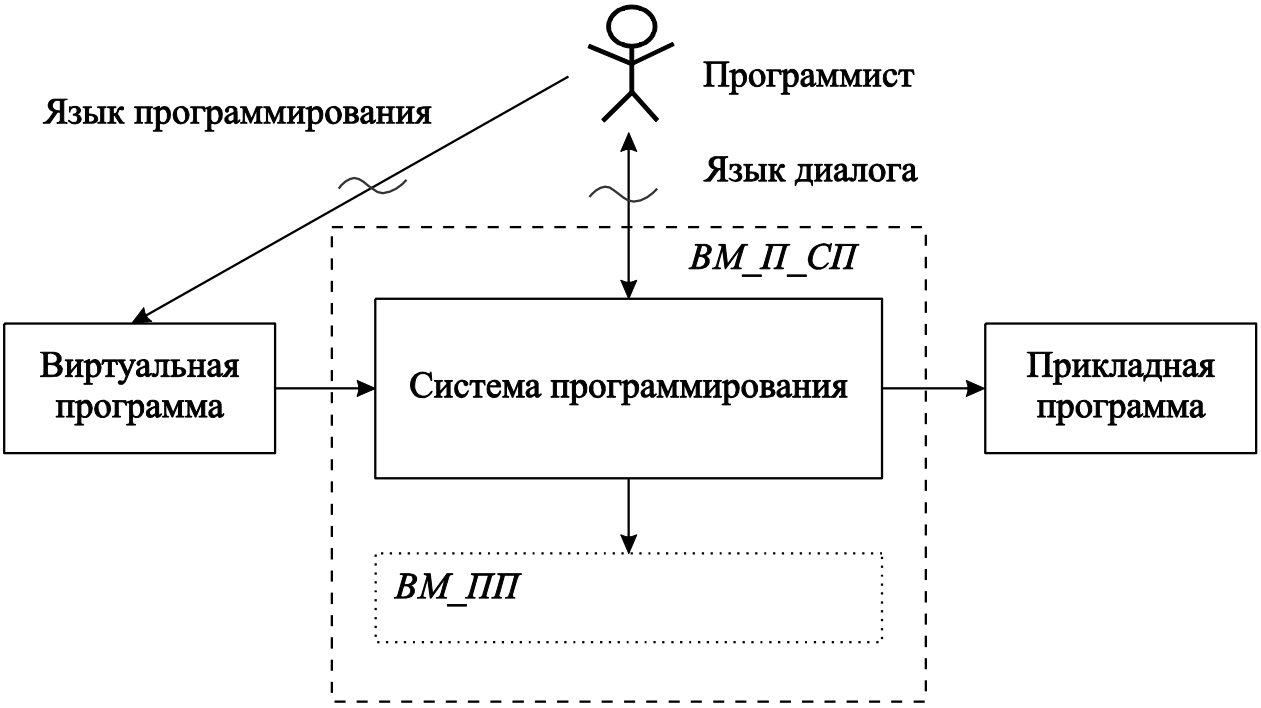


Рис. 2.9 – Виртуальная машина пользователя системы программирования:
ВМ_П_СП – виртуальная машина пользователя системы программирования;
ВМ_ПП – виртуальная машина прикладной программы

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, то мы их рассмотрим позже. А сейчас перейдем к рассмотрению программы, позволяющей пользователю ВС запускать на выполнение прикладные и системные обра-

батывающие программы, зная лишь имя соответствующего исполняемого файла. Речь идет об интерпретаторе команд ОС.

2.5 Язык управления операционной системой

2.5.1 Общие сведения

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 приступать 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 обрабатывающая 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 п. 4.1) 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;
5) командный файл.

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

2.5.2 Простые команды

Простые команды shell делятся на команды: а) запуска программ; б) команды вызова функций shell; в) вспомогательные команды. Последняя из перечисленных групп команд будет рассмотрена нами в пп. 2.5.4 при рассмотрении 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 – запу-

щенная 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 «незаметны», 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 про-

грамме. 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 запущенной 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 поддерева файловой структуры с корнем `dir1` в поддерево с корнем `dir2`:

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

2.5.3 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 эти программы запускаются `shell` одновременно (асинхронно). После 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. 2.10). 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.
```

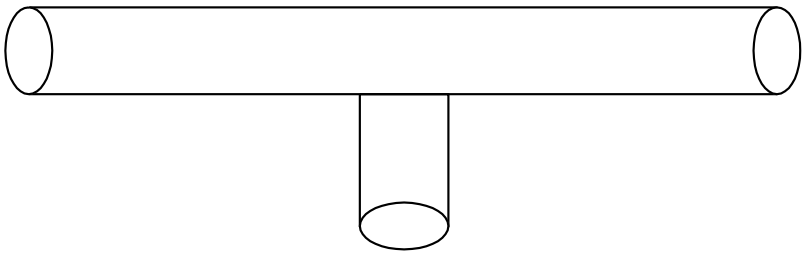


Рис. 2.10 – Наглядное изображение команды `tee`

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).

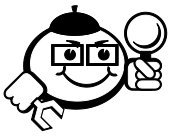
Примеры командных списков:

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.



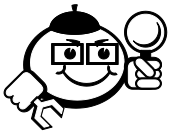
Пример

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
```

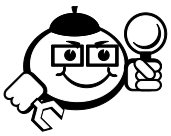
2.5.4 Переменные и выражения

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.

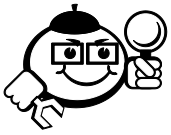


Пример

```
$ var1=/home/vlad/a.txt
$ 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 «`$`».



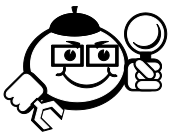
Пример

```
$ 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. пп. 2.5.3). При этом имя любой программы, которая выдает свой результат в виде одной или нескольких строк символов, может быть записано справа от оператора присваивания, заключен-

ным 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 в введенной строке, то все оставшиеся слова записываются в последнюю переменную. Если, наоборот, число переменных больше, то последние переменные получат пустое значение. Если с клавиатуры вместо обычных символов будет введен символ конца файла (<Ctrl>&<D>), то данная команда завершится с ненулевым кодом завершения.

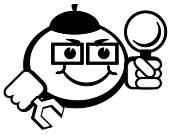
Команду read удобно использовать для того, чтобы присваивать переменным значения слов из некоторого текстового файла. Для этого достаточно перенаправить стандартный ввод с клавиатуры на ввод из требуемого файла. Например, следующая команда присваивает переменным x, y, z значения слов из первой строки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

ной». 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. Второй оператор `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. 3 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.

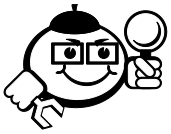
Обыкновенные переменные можно добавить к переменным окружения, сделав их «наследуемыми». Для этого достаточно перечислить имена этих переменных в качестве параметров команды `export`.

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`.



Пример

```
$ expr 5 + 3
8
```

```
$ expr 5 "*" 3
15
$ x=10
$ expr $x + 1
11
$ 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, 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:

`-s file`	размер файла `file` больше `0`
`-r file`	для файла `file` разрешен доступ на чтение
`-w file`	для файла `file` разрешен доступ на запись
`-x file`	для файла `file` разрешено выполнение
`-f file`	файл `file` существует и является обычным файлом
`-d file`	файл `file` существует и является каталогом
`-z string`	строка `string` имеет нулевую длину
`-n string`	строка `string` имеет ненулевую длину
`string1 = string2`	две строки идентичны
`string1 !=`	две строки различны

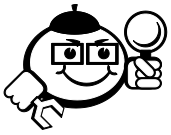
string2	
i1 -eq i2	число i1 равно числу i2
i1 -ne i2	число i1 не равно числу i2
i1 -lt i2	число i1 строго меньше числа i2
i1 -le i2	число i1 меньше или равно числу i2
i1 -gt i2	число i1 строго больше числа i2
i1 -ge i2	число i1 больше или равно числу i2

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.



Пример

```
$ 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`.

2.5.5 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.

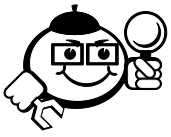
1. *Условный оператор* `if` позволяет выполнить одну из нескольких взаимоисключающих последовательностей команд `shell`. Данный оператор имеет несколько форм записи, наиболее простая из которых следующая:

```
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 слу-

чае, 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 `fil1`. 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`.

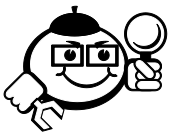
В отличие от распространенных языков программирования условный оператор `if` для `shell` обеспечивает не двух-, а многоальтернативный выбор. Это приближает выразительные возможности данного управляющего оператора к возможностям оператора `case`.

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 «*». Обратим внимание, что любой шаблон отделяется от соответствующей последовательности команд символом «)», а каждая последовательность команд заканчивается двумя символами «;;».

В следующем 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):



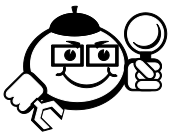
Пример

```
$ 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

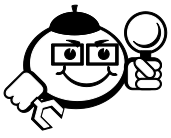
В следующем примере все файлы с суффиксом `txt`, расположенные в поддереве данного пользователя, копируются в каталог `k2`:

```
$ 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

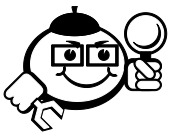
```
$ while read var1 var2
> do
>   case $var1 in
>       1) echo $var2 >>file1 ;;
```

```
>        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:



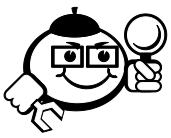
Пример

```
$ 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.

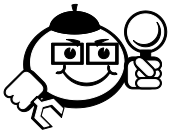


Пример

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 «!!»:

```
$ until [ `echo $var1` = '!!' ]
> do
>   read var1 var2
>   case $var1 in
>       1) echo $var2 >>file1 ;;
>       2) echo $var2 >>file2 ;;
>       *) echo $var2 >>file3 ;;
>   esac
> done
```

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



Пример

Запустив в первой половине дня следующую последовательность команд, мы получим на экране напоминание о начале времени 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 дождаться завершения предыдущей команды, а затем в зависимости от ее результатов выполнить набор следующей. Областью применения управляющих операторов являются командные файлы.

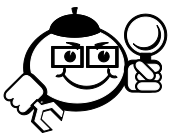
2.5.6 Командные файлы



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`.


Пример

Например, получим скрипт `file`, выполняющий задачу из пп. 2.5.5, 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 (примеры 1 и 2) требует наличия права пользователя на выполнение файла-скрипта. (Вопрос о правах доступа к файлу будет рассмотрен нами в гл. 4.)

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}:./`. В примере 2 такого определения текущего каталога в переменной `PATH` не требуется.

В примерах 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 порождать 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. Больший 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 ***ко****мментарии* – 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:

- `# ! /bin/sh` запускается `Bourne shell`,
- `# ! /bin/csh` запускается `C shell`,
- `# ! /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 «$». При выполнении скрипта каждое значение его позиционного параметра заменяется его значением, полученным из командной строки. Само это значение в командном файле изменено может быть только с помощью команды `set`, и поэтому позиционный параметр никогда не записывается в левой части операции присваивания и, как следствие, его имя всегда предваряется символом «$».

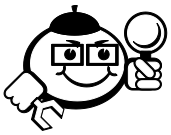
С учетом сделанных замечаний выполним запись рассматриваемого скрипта:

```
$ 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*:

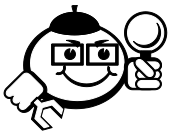
```
$./k1 7 8
7
8
a
b
```

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 опера-

тора `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, выполненными 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.



Контрольные вопросы по главе 2

1. Как при помощи команды `cat` создать файл?
2. Какие текстовые редакторы описаны в главе?
3. Какую служебную информацию о текстовом файле выводит команда `wc`?
4. Для чего используются метасимволы и приведите их?
5. Для чего в UNIX служит команда `cp`?

3 Системная поддержка мультипрограммирования

3.1 Общие сведения

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 удается. Это объясняется прин-

ципиальной 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 данной главе рассматриваются постановки задач, решение которых обеспечивает наличие мультипрограммирования в системе, а также изучаются методы решения этих задач. Реализация данных методов в системе UNIX будет рассмотрена в гл. 5 и 6.

3.2 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. пп. 2.5.3). 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 в начале обработки данной команды shell дерево процессов принимает вид, приведенный на рисунке 3.1. 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 приме-

нение без флагов позволяет вывести на экран минимум информации о процессах.

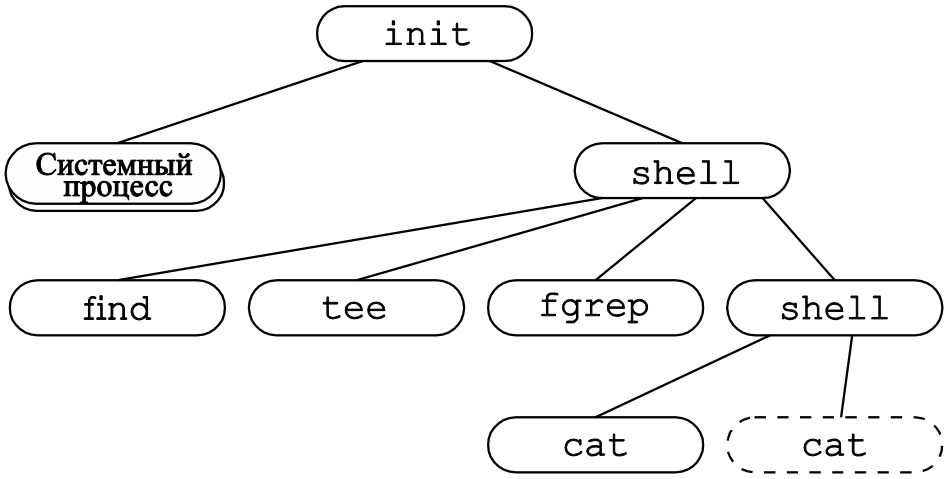
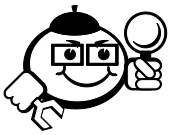


Рис. 3.1 – Пример дерева процессов



Пример

```
$ ps
 PID TTY TIME CMD
 145 ttyp4 0:01 /user/bin/sh
 313 ttyp4 0:03 /user/bin/ed
 431 ttyp4 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 пп. 3.4.2;
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, то необходимо отделить все изменяемые данные в отдельную область. Неизменную 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 использует 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. п. 2.4). 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 компонуемые 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 в управляющие и информационные взаимодействия с другими процессами. 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.

3.3 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 [4].



***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 п. 5.1, а пока лишь заметим, что в данном состоянии процесс не может выполняться на ЦП до тех пор, пока причина блокирования не будет устранена.

На рисунке 3.2 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 то-

го, как данный ресурс стал не нужен тому процессу, которому он был выделен, он может быть распределен какому-то другому процессу.

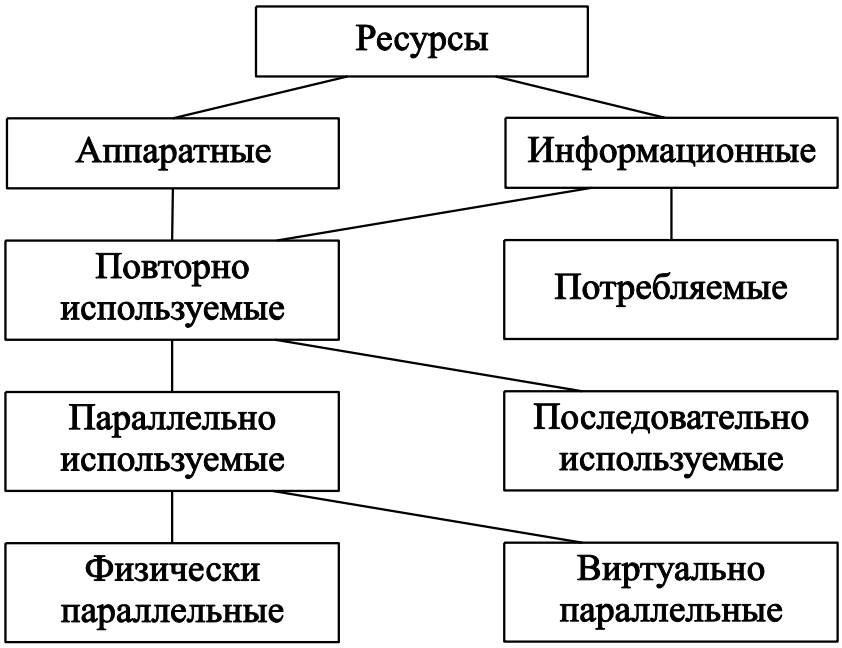


Рис. 3.2 – Классификация ресурсов



*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.

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 п. 4.3).*

***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. 5, 6 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. 3.4.3, 3.4.4 и п. 3.5;
3) оказание помощи процессам, при совместном использовании ими повторно используемых ресурсов, по устранению такого неприятного явления, как тупики;
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 пп. 6.2.3.

3.4 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.

3.4.1 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, 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, источниками сигналов могут быть не только обработ-

чики перечисленных прерываний, но и сами процессы, использующие для этого специально предназначенные системные вызовы.

Современная UNIX-система различает примерно 30 различных 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 «трассировка». 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 посылки 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 ниже.

3.4.2 Терминальное управление процессами

Любая интерактивная ВС имеет среди своих ПУ по крайней мере один терминал.



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 груп-*

пы процессов. Для создания новой группы или для включения процесса в уже существующую группу shell обращается к ядру, используя системный вызов:

УСТАНОВИТЬ_ГРУППУ_ПРОЦЕССОВ (i_g, i_p ||), (на Си – setpgid),

где i_g – номер группы процессов;

i_p – 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 [5].

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 недосягаем 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 п. 4.1, 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

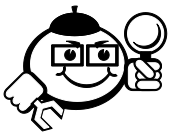
выполнения общесистемных функций, поддерживающих работоспособность системы. Например, демоном является процесс `init`.

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 п. 3.2). 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.

3.4.3 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 п. 6.2). Сейчас нам важно уяснить, что без дополнительной синхронизации процессов наличия разделяемого сегмента явно недостаточно для их информационного взаимодействия. Для того чтобы показать это, рассмотрим следующую задачу.

Допустим, что процесс-сервер выполняет запросы процессов-клиентов по распечатке текстовых файлов на принтере. При этом информационное взаимодействие клиентов и сервера осуществляется через разделяемый сегмент памяти (рис. 3.3). 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.

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*.

Например, 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. По-

кажем, что конкурирующие процессы обязательно должны быть синхронизированы.

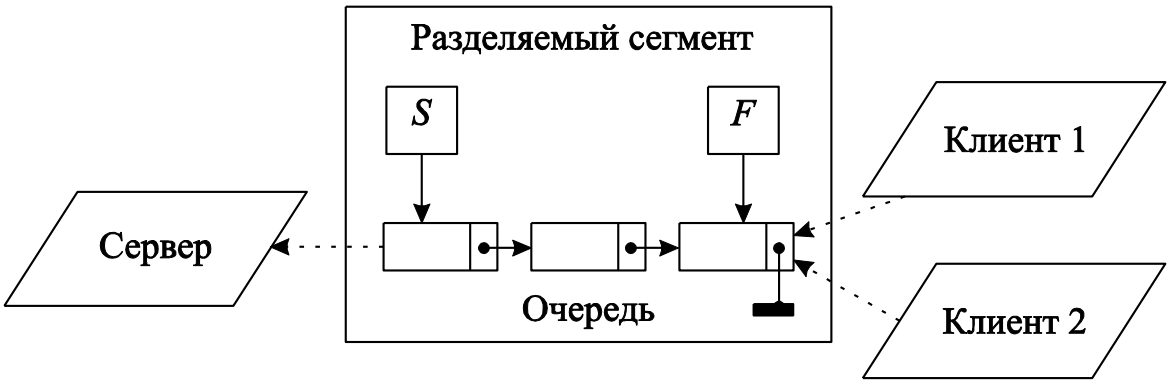


Рис. 3.3 – Три процесса разделяют сегмент памяти

Допустим, что в какой-то момент времени очередь имеет состояние, приведенное на рисунке 3.4, *а*. Пусть клиент *1* хочет поместить в очередь элемент *y*, а клиент *2* – элемент *z*. Очередь, измененная в результате правильного включения, приведена на рисунке 3.4, *б*.

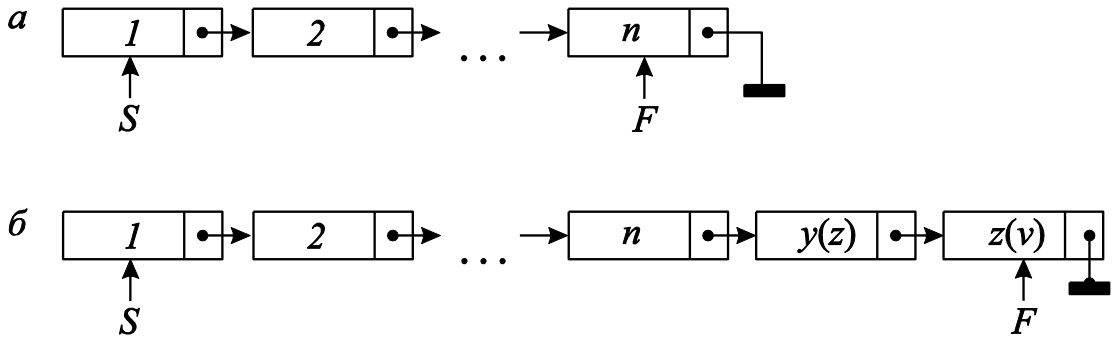


Рис. 3.4 – Состояния очереди:
а – исходное; *б* – правильное включение элементов *y* и *z*

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.

Допустим, что клиент *1* успел выполнить операции *1* и *2*. Потом он был прерван на некоторое время, в течение которого клиент *2* выполнил все три операции. Потом клиент *1* смог выполнить операцию *3*. Соответствующие изменения очереди приведены на рисунке 3.5. В результате единая очередь оказа-

лась разорванной на две несвязанные части, что недопустимо. Рассмотрим, как этого можно избежать.

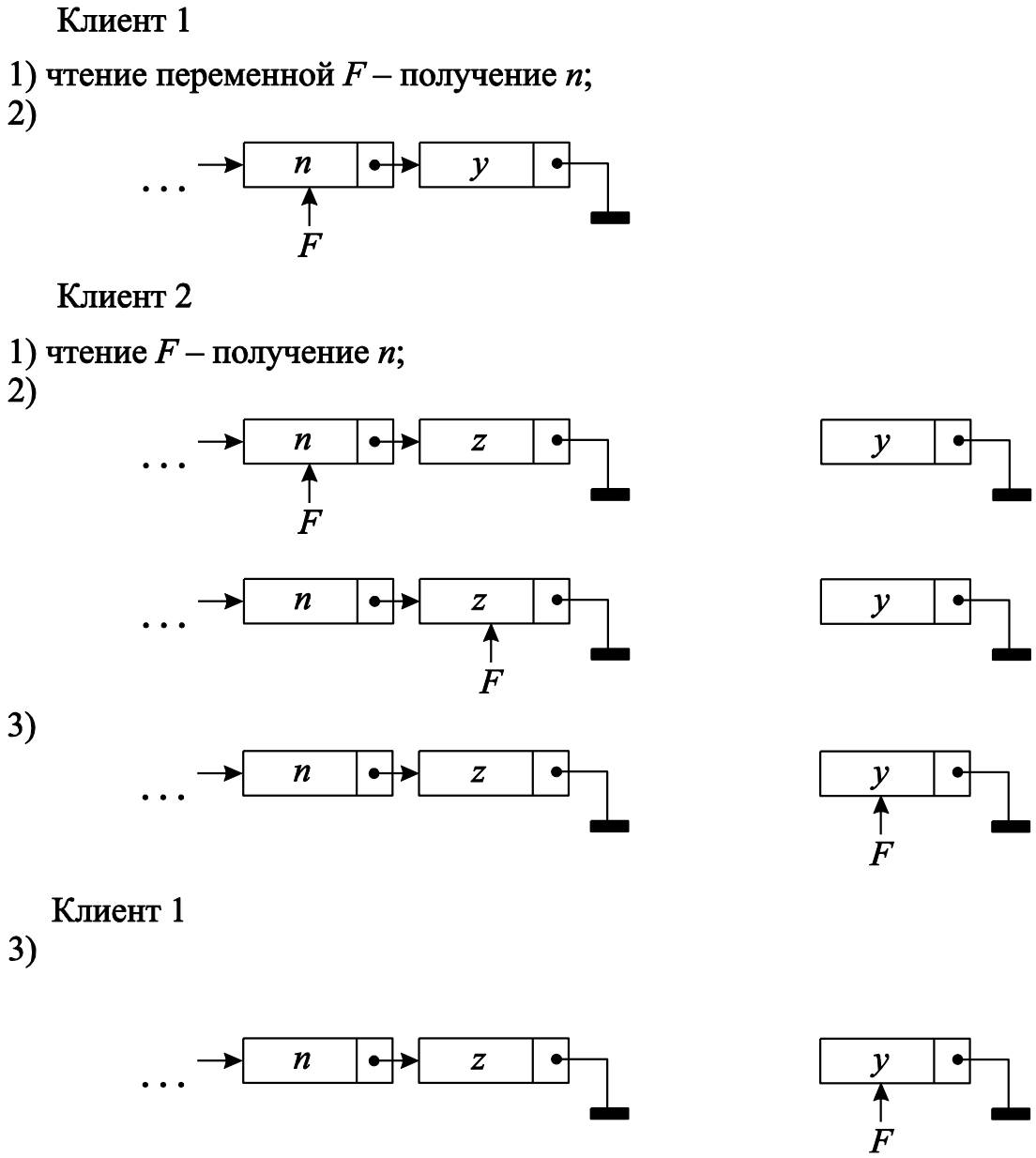


Рис. 3.5 – Возможное преобразование очереди
при отсутствии синхронизации

Пусть два процесса *p1* и *p2* 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 одновременно 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. Недостаток 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 до-

ждется, и время ЦП будет истрачено 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...`*, 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 ← S – 1;
                 продолжение выполнения процесса
        ИНАЧЕ    блокирование процесса по S
                 вызов диспетчера ЦП

v(S):   ЕСЛИ (очередь процессов, ожидающих S, непустая)
        ТО  деблокирование процесса, блокированного по S
        ИНАЧЕ    S ← 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 переменная.) 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) выделить в программе каждого процесса критические секции, каждая из которых выполняет некоторое действие над `x`;
2) задать начальное значение двоичного семафора `S`, равное `1`;
3) записать перед каждой критической секцией оператор `p(S)`, а после нее – `v(S)`.

3.4.4 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 – `S1` и `S2`. Семафор `S1` управляет работой «писателей». Его значение

есть число пустых позиций в буфере. Семафор S2 управляет работой «читателей». Его значение есть число занятых позиций в буфере. Первоначальные значения: S1 = n, S2 = 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
        ..............
```

3.5 Информационные взаимодействия между процессами

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 недоступные непосредственно процессам. Иными словами, ОС предоставляет в распоряжение процессов *информационные каналы* (рис. 3.6).

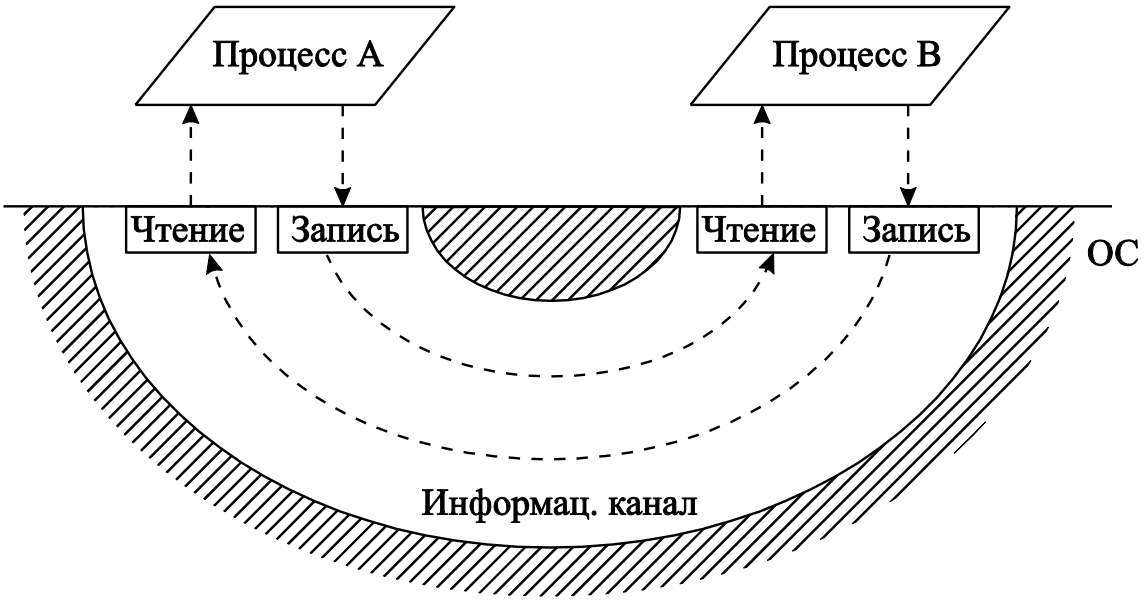


Рис. 3.6 – Взаимодействие процессов через информационный канал

Любой информационный канал позволяет подключенным к нему процессам использовать два системных вызова:

ЧТЕНИЕ_ИНФ.КАНАЛА (…||…), ЗАПИСЬ_ИНФ.КАНАЛ (…||…).

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

1) *направление передачи данных*. Информационные каналы делятся на симплексные, полудуплексные и дуплексные. *Симплексный канал* позволяет передавать данные только в одном направлении, *полудуплексные* – в обоих направлениях неодновременно, а *дуплексный* – в обоих направлениях одновременно;
2) *структурированность передаваемых данных*. Различают информационные каналы, передающие потоки данных и передающие сообщения;
3) *поток данных* – последовательность байтов, в состав которой не входят какие-то особые байты, используемые информационным каналом;
4) *сообщение* – последовательность байтов, предваряемая специальными служебными байтами;
5) *устойчивость информационного канала*. Будем различать *виртуальные соединения* – устойчивые каналы, предназначенные для передачи потока данных или для передачи последовательности сообщений, а также *датаграммные каналы*, предназначенные для передачи отдельных, не связанных между собой сообщений, называемых *датаграммами*;
6) *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.

В качестве примера рассмотрим одно из средств межпроцессного информационного взаимодействия – *каналы*. Наглядно 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 (|| 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 открыт 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 ||), (на Си – read),

ЗАПИСЬ_ФАЙЛ (i_1, A_b, n_b ||), (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 должен быть 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 потока данных.

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 обработать ситуацию, когда получено меньше, чем заказано;
3) если канал пуст, и он 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 команд-

ной строке `shell` (см. пп. 2.5.3). В данном случае каналы создает `shell`, дочерними процессами которого и являются процессы, образующие конвейер.

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 п. 4.2).

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.



Контрольные вопросы по главе 3

1. Какие аппаратные и программные ресурсы используются процессами реально параллельно?
2. Какие ресурсы присущи как мультипрограммной, так и однопрограммной операционной системе?
3. Какие программы и подпрограммы возможно выполнять в двух режимах ЦП – привилегированном и непривилегированном?
4. В чем отличие процессов-демонов от процессов-зомби?

4 Поддержка многопользовательской работы и структура системы

4.1 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.

Как воспринимает своего пользователя ОС? Будучи «бюрократом», любая ОС рассматривает пользователя не учитывая его многочисленные достоинства и недостатки, 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 пп. 3.4.2). 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. 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. В данном пособии 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 в файловой структуре системы, принадлежащего данному пользователю;
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 п. 3.2 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 п. 5.2). Кроме того, от-

метим, 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 [5].

Для обеспечения такого доступа изложенная выше схема должна быть скорректирована (рис. 4.1).

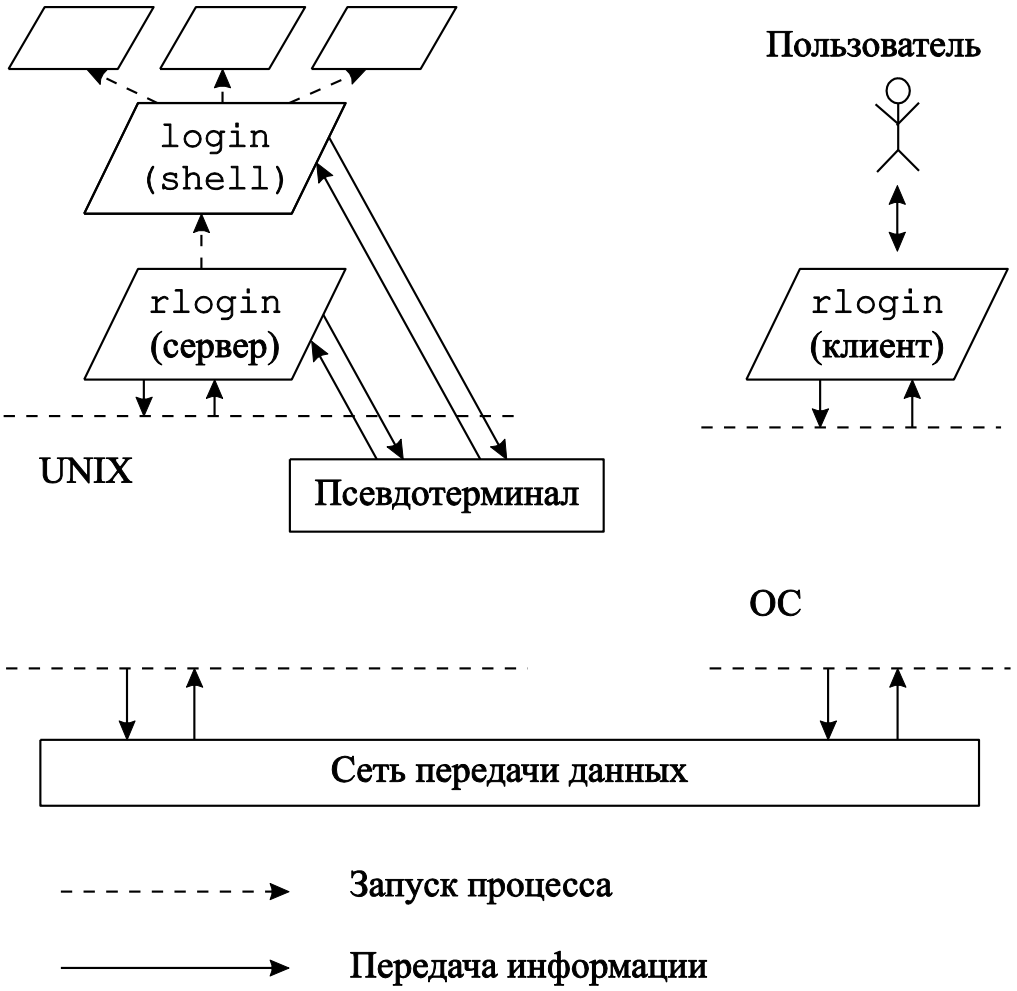


Рис. 4.1 – Доступ удаленного пользователя в UNIX-систему

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 пп. 3.4.2. Получив от rlogin (сервер) код этой комбинации, псевдотерминал распознает ее и выдает тот сигнал, который соответствует этой комбинации клавиш. Таким образом, удален-

ный 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.

4.2 Защита файлов

Каждый пользователь, зарегистрированный в системе, имеет в ней «информационный след» – совокупность информации, связанной с данным пользователем. Этот информационный след схематично показан на рисунке 4.2. На этом рисунке пунктиром показана краткосрочная информация, связанная с выполнением процессов. Долгосрочная информация хранится на устройствах ВП в виде файлов.

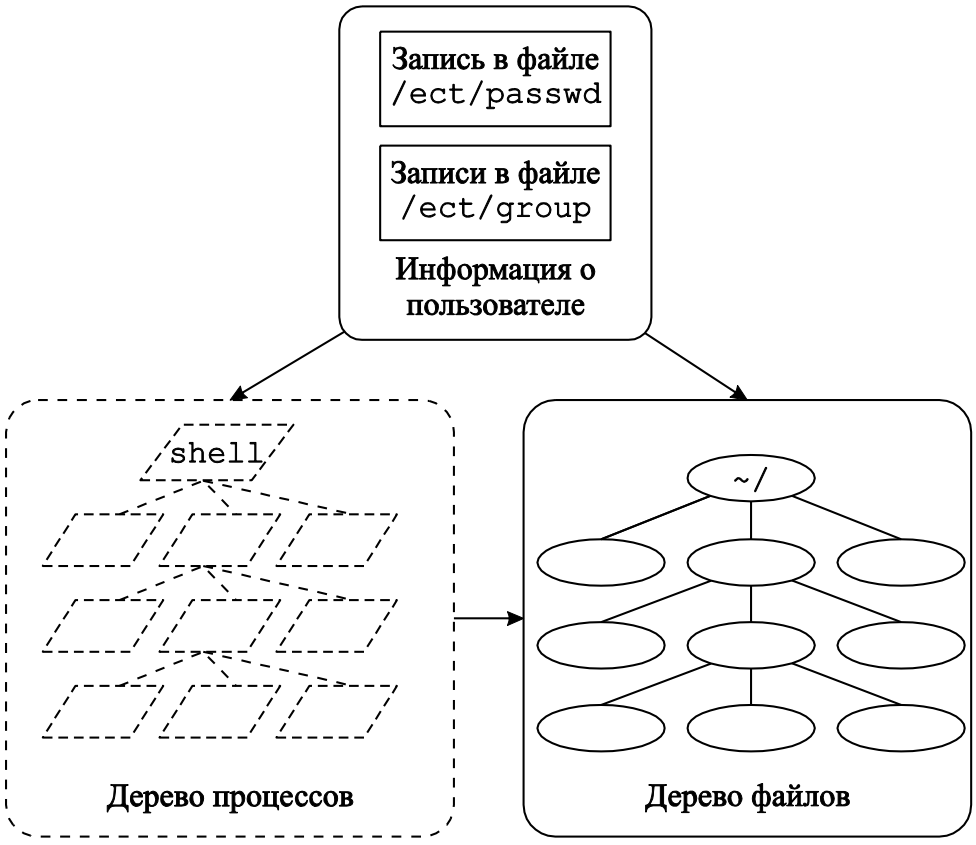


Рис. 4.2 – Информационный след пользователя в системе

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 его записи `/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 записываются, 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 записываются 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 исполняемым файлом программы и может быть выполнен по-

сле загрузки в ОП как машинная программа или же файл является командным (скриптом).

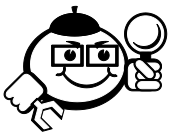
Например, суперпользователь имеет по отношению к любому файлу неограниченные права доступа – `rwx`.

Применительно к каталогу перечисленные формы доступа имеют особенный смысл. Операция «чтение каталога» означает получение перечня имен файлов, записанных в данном каталоге. Получения 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- 1  vlad  gro   1120   Dec15 15:03 abc.txt
    - r- -
    d  rwxr- 2  vlad  gro   11500 Aug28 17:38 mac
    xr- -
$
```

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 прочих пользователей [6].

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 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:

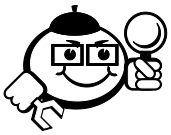
1) число жестких связей – количество каталогов, в которых «зарегистрирован» данный файл;
2) имя пользователя-владельца файла;
3) имя пользователя-группы;
4) длина файла в байтах;
5) дата последнего изменения файла;
6) время последнего изменения;
7) имя файла.

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:

```
| u | | + | | r |
```

```
chmod | 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 принадлежность 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 [7].

Кроме основных форм доступа (и соответствующих прав) к файлам, 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`. Владельцем исполняемого файла этой утилиты является суперпользователь, 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 ее;
3) **t** – в настоящее время используется только для каталогов, разрешая пользователю удалять только те файлы, для которых он является владельцем, или в которых ему разрешена 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.

4.3 Укрупненная структура операционной системы

На рисунке 4.3 приведена укрупненная структура операционной системы UNIX. 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.



Рис. 4.3 – Укрупненная структура ОС

Интерфейс системных вызовов содержит процедуры, доступные для вызова программами, расположенными вне ядра, а также подпрограммы, расположенные внутри ядра и недоступные 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 *асинхронном 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. п. 4.1). 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.

4.4 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 п. 4.1 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. п. 3.5).

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 ин-

терфейсу, 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. 4.4). Реализация данного модуля выходит за рамки данного курса, и для нас важен лишь тот факт, что модуль транспорта обеспечивает передачу сообщения требуемому удаленному программному процессу.

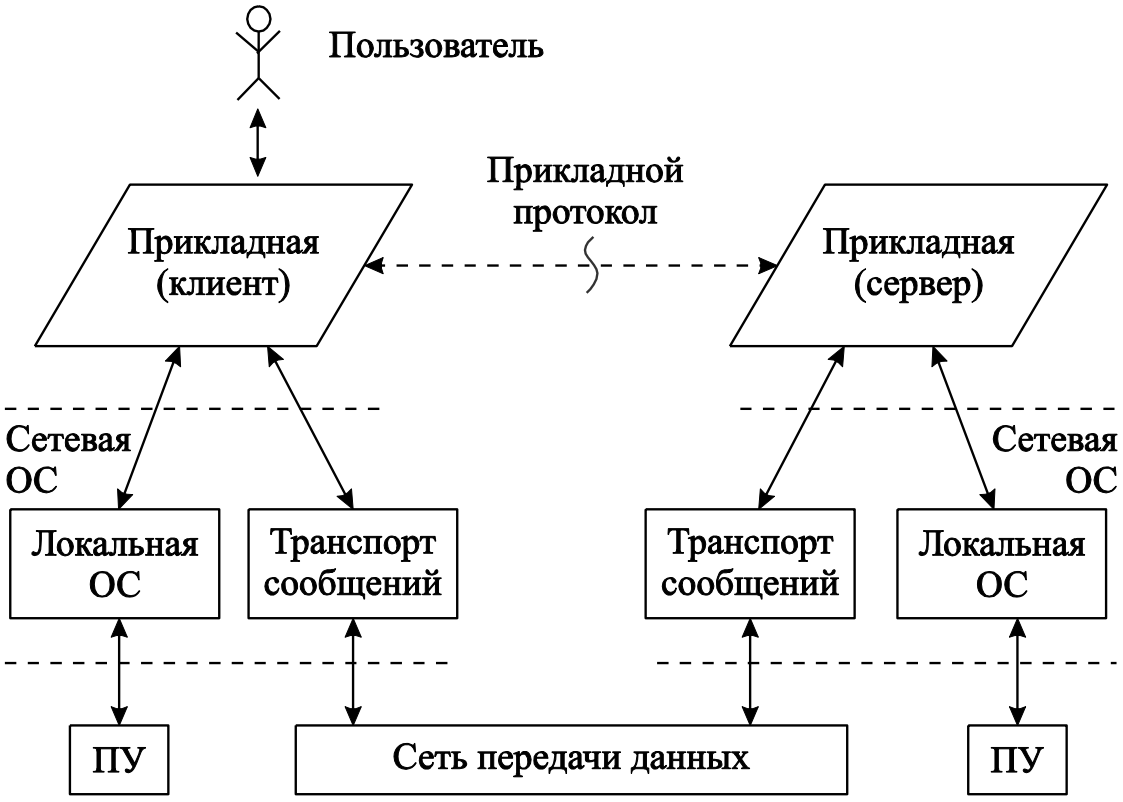


Рис. 4.4 – Выполнение распределенной программы в сети

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. 4.5).

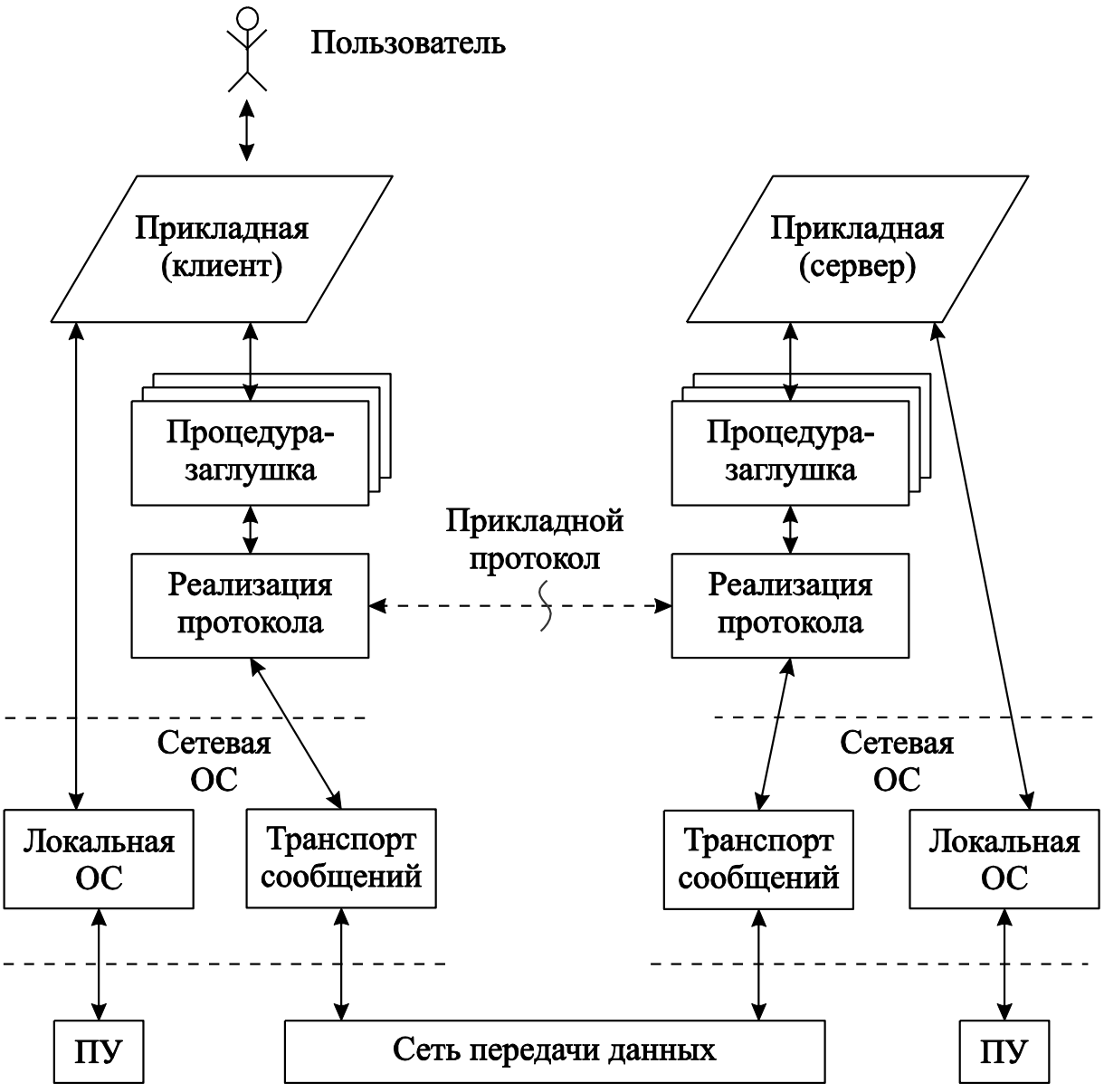


Рис. 4.5 – Распределенная программа с удаленным вызовом процедур

При проектировании распределенной программы производится распределение ее подпрограмм между клиентом и сервером. Все процедуры сервера, ко-

торые 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.

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 инициирует 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 п. 7.3.

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. 4.6).



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).*

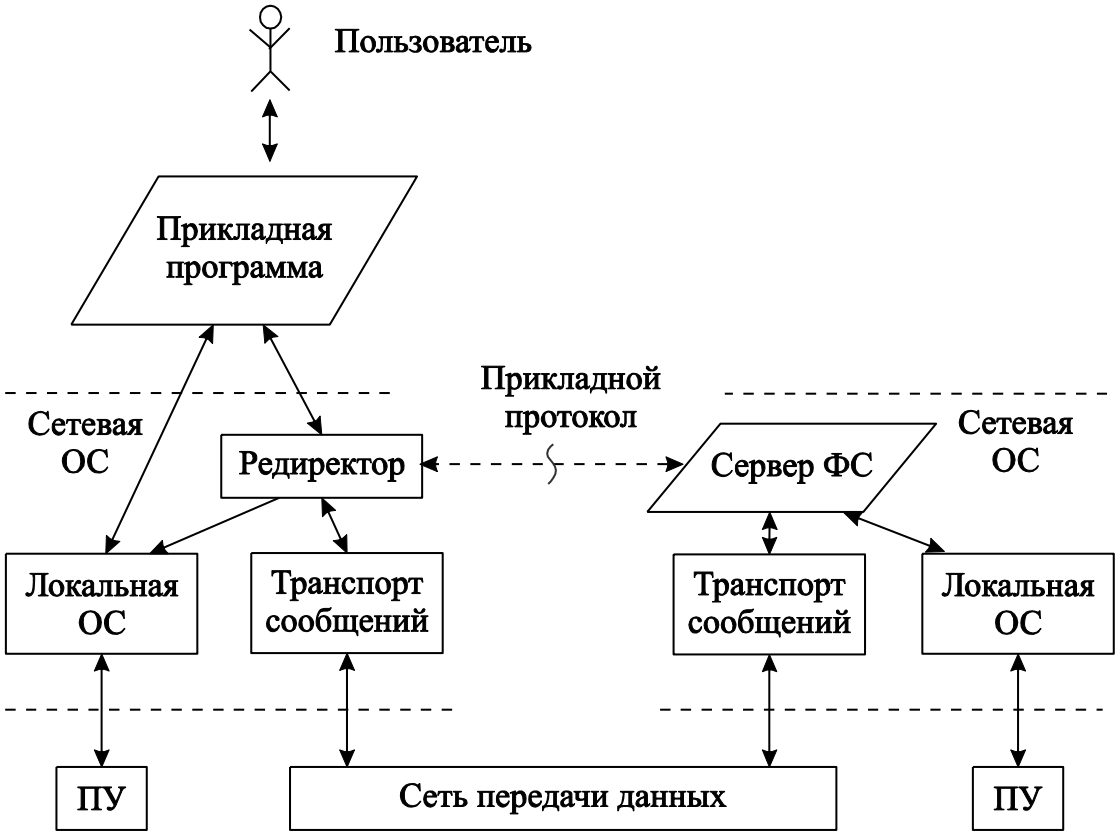


Рис. 4.6 – Сетевое обслуживание локальной прикладной программы

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.

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», а это 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 выполняя 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.



Контрольные вопросы по главе 4

1. Какие из перечисленных программ и подпрограмм выполняются только в привилегированном режиме работы?
   а. Ядро ОС.
   б. Файловая система.
   в. Подсистема управления процессами.
   г. Подсистема управления памятью.
   д. Интерфейс системных вызовов.
   е. Подсистема ввода/вывода.
   ж. Виртуальная файловая система.
   з. Диспетчер ЦП.
2. Зачем нужна процедура «заглушка»?
3. Приведите схему удаленного доступа пользователя к системе UNIX.
4. В чем особенность микроядра?

5 Подсистема управления процессами

Данная подсистема занимает центральное место среди подсистем ОС, выполняя различные функции по обеспечению существования процессов. В п. 3.2 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.

5.1 Состояния процесса

Являясь важнейшим объектом управления со стороны ОС, процесс может находиться в состояниях, изображенных на рисунке 5.1. При этом дуги обозначают переходы между состояниями, а на дугах проставлены причины переходов. Эти причины бывают двух типов – системные вызовы, выдаваемые или 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 (|| ip)`, (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.



Рис. 5.1 – Состояния процесса

В результате второй фазы выполнения системного вызова процесс оказывается в состоянии «Готов». В этом состоянии он обладает всеми ресурсами, за исключением ЦП. В результате выполнения подпрограммы ядра, называемой *диспетчером*, программа процесса начинает выполняться на ЦП.

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 начинает 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 (|| ip, k)`, (на Си – `wait`),

где i_p – номер завершенного дочернего процесса;

`k` – код завершения процесса.

Для выполнения этого системного вызова будет инициирована подпрограмма ядра `wait`, которая выдаст номер завершенного 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`.

5.2 Создание процесса

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

1) управляющая структура `proc`;
2) управляющая структура `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 LTD (рассматривается в п. 6.2);
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. пп. 2.5.4).

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 запускаемую 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. пп. 2.5.6). 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. 6.

5.3 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, 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 завершения процесса, то после выполнения этого обработчика выполнение процесса продолжится с той точки, в которой оно было прервано для обработки сигнала.

5.4 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 оперативном 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. Заметим, что реальные процедуры диспетчеризации достаточно громоздки. Поэтому далее приводится упрощенная такая процедура.

В любой UNIX-системе процессу соответствует не один, а три различных приоритета:

1) пользовательский приоритет;
2) текущий прикладной приоритет;
3) текущий системный приоритет.



Пользовательский приоритет (NICE) *– приоритет, заявленный пользователем при создании процесса.*

Этот приоритет используется диспетчером для определения начальных значений текущих приоритетов процесса. UNIX предоставляет своим пользователям команды, позволяющие изменять NICE у требуемого процесса. 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 системы.



Текущий прикладной приоритет *– динамически изменяющийся приоритет процесса в состоянии «Задача».*

Диспетчер корректирует этот приоритет и использует его при размещении процесса в СГ. Для расчета текущего прикладного приоритета *i*-го процесса используется формула:

$$\mathrm{PRI}_{i,t} = \mathrm{B}_i + \mathrm{NICE}_i + \mathrm{M}_{i,t},$$

где $\mathrm{PRI}_{i,t}$ – текущий прикладной приоритет *i*-го процесса на *t*-й секунде времени;

B_i – базовый прикладной приоритет. Для рассматриваемой системы $\mathrm{B}_i = 40$;

$\mathrm{M}_{i,t}$ – мера использования ЦП *i*-м процессом на *t*-м интервале времени.

$\mathrm{M}_{i,t}$ корректируется следующим образом. Если процесс выполняется на ЦП (в состоянии «Задача»), то через каждый тик (10 мс) его $\mathrm{M}_{i,t}$ увеличивается на небольшую фиксированную величину. И наоборот, если процесс находится в СГ (находится в состоянии «Готов»), то через каждую секунду его $\mathrm{M}_{i,t}$ корректируется в сторону уменьшения по формуле:

$$\mathrm{M}_{i,t} := \mathrm{M}_{i,t-1} \cdot \left(2n_{t-1}\right) / \left(2n_{t-1} + 1\right),$$

где 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.

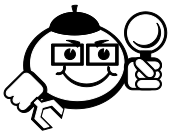


Текущий системный приоритет *– приоритет процесса в состоянии «Ядро».*

Диапазон системных приоритетов никогда не перекрывается с диапазоном прикладных приоритетов. Поэтому самый худший системный приоритет лучше самого лучшего прикладного приоритета. Допустим далее, что диапазон системных приоритетов от 0 до 40. Что касается конкретной величины текущего системного приоритета, то он корректируется динамически так же, как и прикладной приоритет.

Интересно отметить, что системный приоритет присущ процессу не только в состоянии «Ядро», но и в состоянии «Сон». В этом состоянии процесс ожидает получения от системы какого-то ресурса и чем дефицитнее для системы этот ресурс, тем системный приоритет сна лучше (для избежания «простаивания» этого ресурса). Как только процесс переходит из состояния «Сон» в состояние «Готов», его приоритет сна сразу же становится текущим системным приоритетом.

В п. 3.2 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

```
$ ps -l
F S UID  PID PPID CPU PRI NICE ADDR SZ WCHAN TTY TIME CMD
1 S vlad145 1    0   60  10   362  21 4356  2   0:01 s
1 Z vlad313 145  4   56  10   127  19 2125  2   0:03 e
1 S vlad431 145  5   70  20   245  11 12454 2   0:01 ps
```

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 про-

цессом). Например, если F = 3, то это означает, что процесс системный и его программа находится в ОП;

2) S – состояние процесса (O – выполняется на ЦП; S – находится в состоянии «Сон»; R – «Готов»; I – создается; Z – «Зомби»);
3) UID – имя пользователя – владельца процесса;
4) PPID – номер процесса-родителя;
5) CPU – текущий системный приоритет процесса;
6) PRI – текущий прикладной приоритет процесса;
7) NICE – пользовательский приоритет процесса;
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 байт);
10) WCHAN – номер события, которого ожидает процесс в состоянии сна.

Рассмотрим динамику выполнения процессов на ЦП. Допустим, что наш процесс находится в СГ и готов выполняться в состоянии «Задача». Так как со временем его текущий приоритет улучшается, а другие процессы не могут долго находиться в состоянии «Ядро», то процесс неизбежно окажется в начале СГ и в случае освобождения процессора будет загружен в него диспетчером путем загрузки в регистры ЦП аппаратного контекста процесса.

Выполнение процесса в состоянии «Задача» может завершиться в результате одного из следующих событий:

1) закончился квант времени ЦП;
2) переход в состояние «Ядро»;
3) появление в системе более приоритетного процесса.

Квант – небольшая порция времени ЦП, выделяемая очередному процессу. По истечении ее процесс снимается с ЦП, если имеется процесс с таким же (или почти таким же) текущим прикладным приоритетом. Появление более приоритетного процесса приводит к досрочному завершению кванта. Переход процесса в состояние «Ядро» является или результатом системного вызова, или результатом исключения, или результатом внешнего аппаратного прерывания. В результате процесс не снимается с ЦП, а переходит в другое состояние выполнения.

Особую опасность для целостности данных ядра имеет обработка внешних аппаратных прерываний во время нахождения процесса в состоянии «Ядро». В отличие от исключений, происходящих вследствие выполнения про-

граммы 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 этого приоритета 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. пп. 3.4.3) и его прерывать нежелательно. Во время выполнения критической секции приоритет процесса улучшается настолько, чтобы он был лучше приоритетов опасных обработчиков прерываний. Затем восстанавливается прежнее значение приоритета.

Изложенная схема диспетчеризации учитывает интересы как интерактивных, так и 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 вычислительных процессов, то с нахождением 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, худшая – для остальных процессов в состоянии «Ядро», самая худшая – в состоянии «Задача». 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 инициируется 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.

5.5 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 мс, 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) проверка 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) обработка отложенных вызовов. Заявки на отложенные вызовы поступают от подпрограмм ядра;
5) обработка алармов. Заявки на алармы могут поступать от любых процессов;
6) пробуждение тех процессов ядра, которые должны выполняться периодически, через заданный интервал времени.



Отложенный вызов *– выполнение требуемой подпрограммы ядра через требуемое время.*

Аларм *– отправка сигнала процессу по истечении заданного интервала 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 че-

рез 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, осуществляющей задержку программы на заданное число секунд (см. пп. 2.5.5).



Контрольные вопросы по главе 5

1. Что значит процесс в состоянии «Ядро»?
2. Какими ресурсами процесс не обладает в режиме «Задача»?
3. Какими ресурсами процесс не обладает в режиме «Сон»?

6 Управление оперативной памятью

6.1 Основные положения

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 главах 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.

Перейдем теперь к управлению 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. 4.3), 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 роль аппаратуры управления памятью различны. 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. 6.1). Благодаря такому 2-этапному преобразованию адресов каждая из трех перечисленных задач управления ОП также разбивается на две подзадачи.

Дальнейшее изложение решения задач по управлению ОП 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 рисунке 6.2.

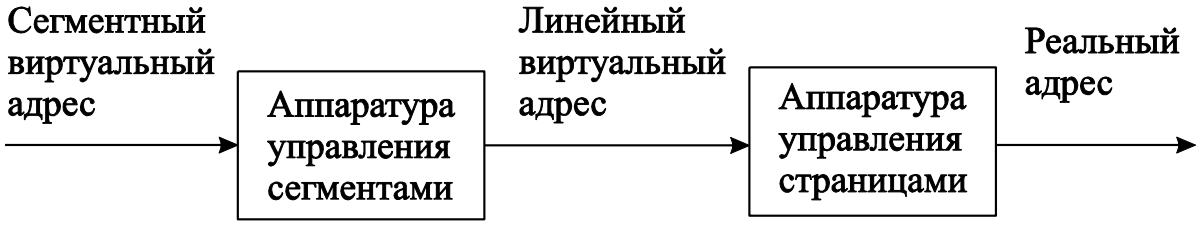


Рис. 6.1 – Состав аппаратуры управления памятью

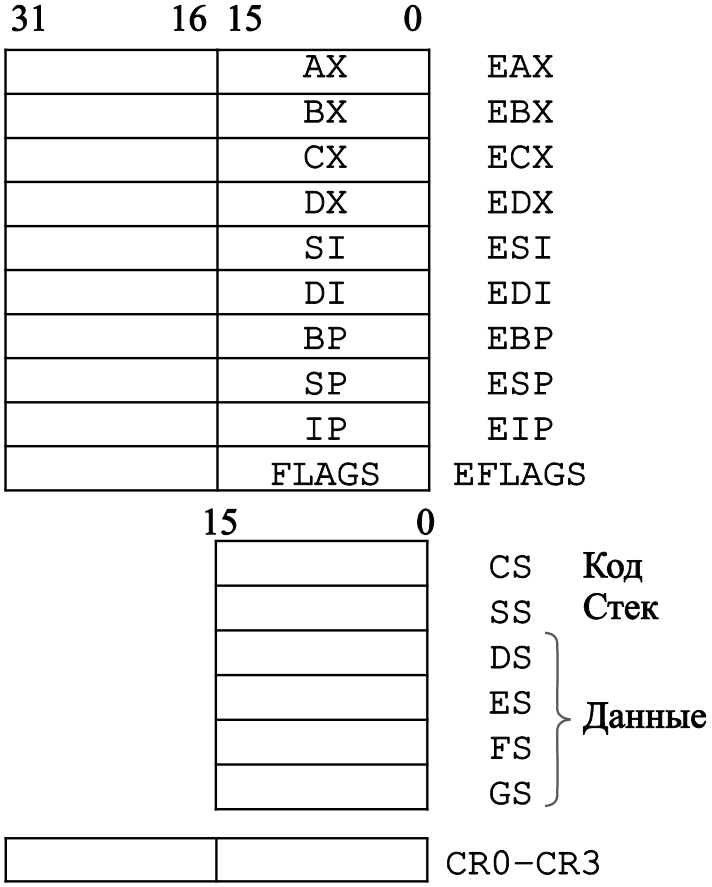


Рис. 6.2 – Регистры процессора *i80386*

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: увеличение числа команд, размера и количества регистров, емкости буферов, тактовой частоты.

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 бит.

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 Г = 1 К^3, 1 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.

6.2 Сегментная виртуальная память

6.2.1 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 *вентилями*. 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 рисунке 6.3:

$$R=B+L,$$

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.

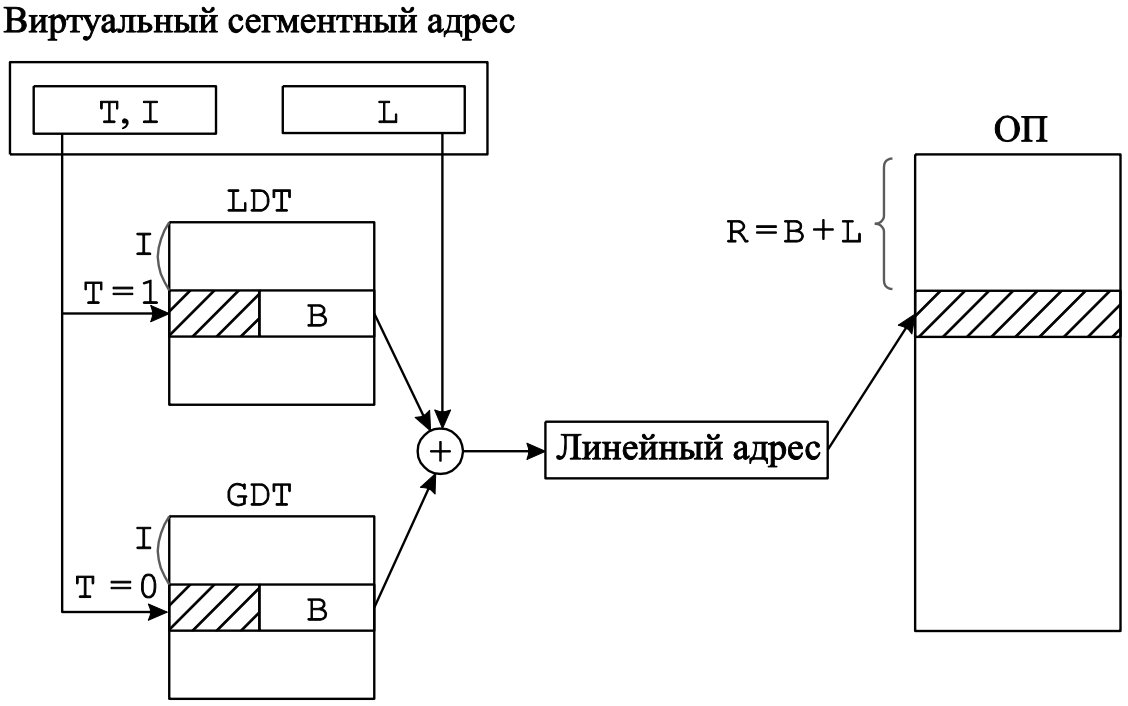


Рис. 6.3 – Преобразование виртуального сегментного адреса в реальный при отсутствии аппаратуры управления страницами.
T – тип таблицы дескрипторов (0 – GDT, 1 – LDT),
I – индекс дескриптора сегмента в GDT или LDT,
L – смещение относительно начала сегмента,
B – базовый линейный адрес сегмента,
R – реальный адрес первой ячейки сегмента

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. 6.4). Поле RPL используется для аппаратной защиты информации в ОП и будет рассмотрено нами в пп. 6.2.3.

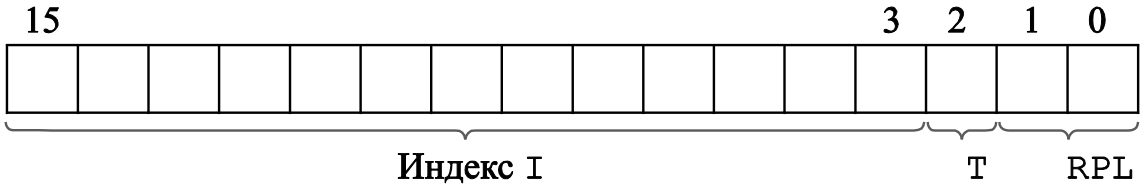


Рис. 6.4 – Структура сегментного регистра в защищенном режиме.
I – индекс дескриптора сегмента в таблице GDT или LDT,
T – тип таблицы дескрипторов (0 – GDT, 1 – LDT),
RPL – уровень запрашиваемых привилегий

Обратим внимание, что размер поля индексов I в сегментном регистре составляет 13 бит. Поэтому максимальная длина (в байтах) GDT или LDT равна 2^{13} = 8 192. Так как длина одного дескриптора равна 8 байтам, то максимальное число дескрипторов в таблице GDT или LDT будет: 8 192/8 = 1 024 = 1 K. Такое максимальное число сегментов памяти может иметь конкретная прикладная программа или ядро ОС.

На рисунке 6.5 приведена структура дескриптора сегмента (строка GDT или LDT). Его длина 8 байт, из которых базовый линейный адрес сегмента (B) занимает 4 байта, или 32 бита. Так как 2^{32} = 4 Г, то общий объем линейного виртуального адресного пространства составляет 4 Гбайт.

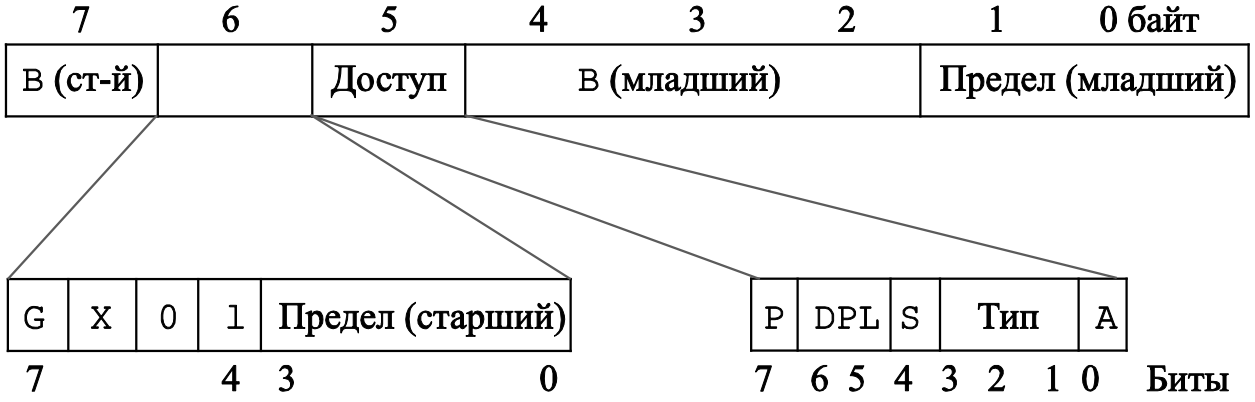


Рис. 6.5 – Структура дескриптора сегмента

Поле *предел* содержит размер сегмента в байтах, уменьшенный на 1. В реальном режиме размер сегмента ОП составляет 64 К. В защищенном режиме можно задавать любую длину сегмента от 1 байта до 2^{20} = 1 М байтов или страниц. Бит `G` – *бит гранулярности*. Если `G=1`, то поле предела задает предельный размер сегмента в страницах (по 4 096 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 `1` 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 пп. 6.2.3. Биты `P` и `A` используются для организации сегментного свопинга. Эти биты будут рассмотрены в пп. 6.2.2. Содержание поля `тип` определяется назначением сегмента. На рисунке 6.6 приведен байт доступа для сегмента кода, содержащего машинные команды программы. Здесь бит `C` – бит подчинения. `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).

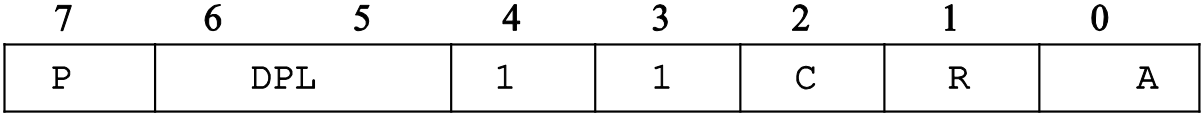


Рис. 6.6 – Байт доступа для сегмента кода

На рисунке 6.7 приведен байт доступа для сегмента данных. Здесь бит `D` задает направление расширения сегмента. Обычный сегмент данных расширя-

ется 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.

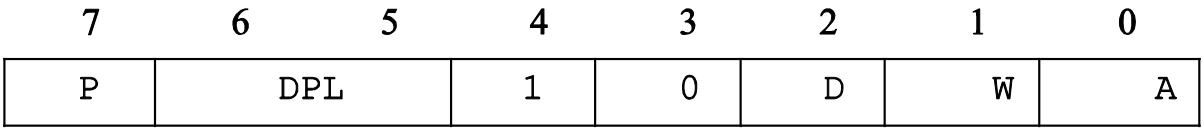


Рис. 6.7 – Байт доступа для сегмента данных

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  0,0,0,0,0,0,0,0
; Дескриптор сегмента кода (выполнение, чтение, 32-
  битные команды)
    db  0FFh, 0FFh, 0, 0, 0, 0FAh, 4Fh, 0
; Дескриптор сегмента данных (рост вверх, чтение, за-
пись)
    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 рисунке 6.1, 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 внутрен-

него регистра позволяет ЦП достаточно быстро вычислять линейные адреса ячеек в сегментах, определенных в текущей LDT.

6.2.2 Распределение памяти

В отличие от процессора i8086 наличие аппаратуры управления сегментами в процессорах, начиная с i80386, обусловлено не стремлением расширить объем адресуемой памяти, а 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 пп. 6.2.3.

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 рисунке 6.8.



Рис. 6.8 – 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 процесс 4 имеет 2 сегмента длиной по 2 Мбайта. Тогда без перемещения про-

граммы 2 в сторону больших или меньших адресов программа процесса 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 $\big((\mathtt{T},\mathtt{I}),\mathtt{L}\big)$, 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 откачан.



Перекачка сегментов между ОП и ВП называется ***сегментным свопингом***.

Распределение ОП, основанное на свопинге, обладает тем существенным недостатком: вследствие неодинаковой длины сегментов размещение одного сегмента может потребовать откачку и (или) перемещение в ОП не одного, а нескольких других сегментов [8].

6.2.3 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 с номером 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, *где* 0 – *самый привилегированный уровень,* 3 – *самый непривилегированный уровень.*

Эти уровни можно наглядно изобразить в виде ***колец защиты*** (рис. 6.9). 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».

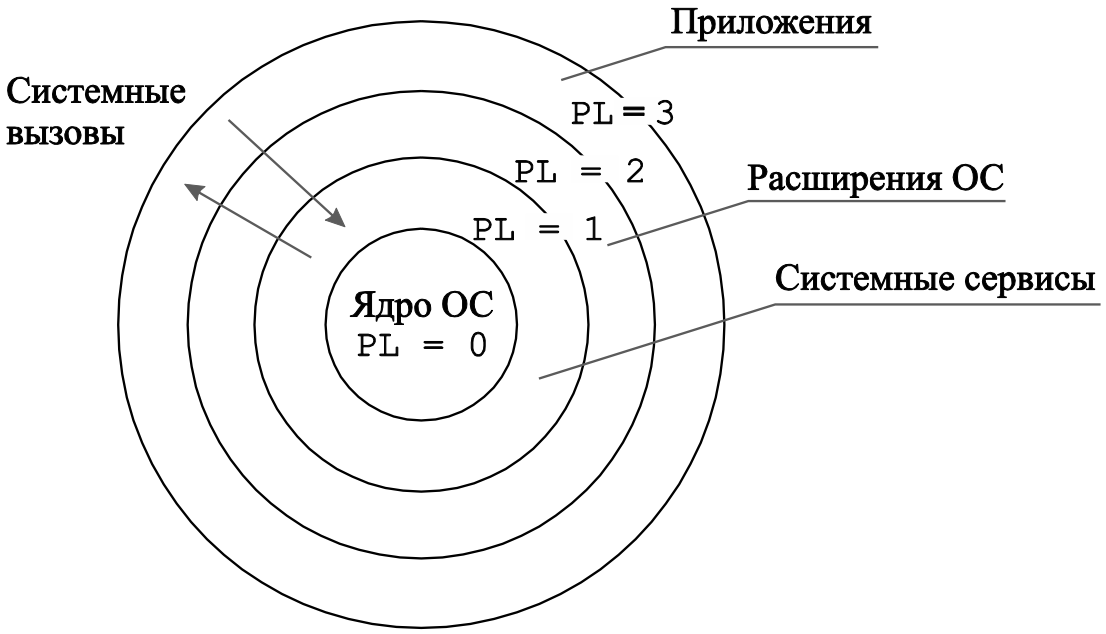


Рис. 6.9 – Возможное распределение программ ВС по кольцам защиты

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.

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. 6.1) 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. 5.1) 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. 5.1) 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:

$$(\mathrm{DPL})_{\text{в дескрипторе сегмента данных}} \geq \max\,[(\mathrm{CPL})_{\text{в CS}}, (\mathrm{RPL})_{\text{в DS}}].$$

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 прикладных программ.

С другой стороны, полный запрет вызова подпрограмм ядра прикладной программой также не допустим, 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 рисунке 6.1, одно из полей дескриптора сегмента содержит *предел* – размер соответствующего сегмента. Попытка использовать в те-

кущей 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. 6.2) бит `R` разрешает (`R=1`) или запрещает (`R=0`) чтение из сегмента кода. В дескрипторе сегмента данных (см. рис. 6.3) 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 некоторые из их:

- `lgdt` – загрузка `GDTR`;
- `lldt` – загрузка `LDTR`;
- `lmsw` – загрузка регистра состояния машины (это 16 младших битов регистра `CR0`);
- `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 зациклившейся 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: $\texttt{CPL} \le \texttt{IOPL}$, где `IOPL` – число в битах 12 и 13 регистра флагов `EFLAGS`. Программы, рабо-

тающие 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 для защиты не только информации, расположенной на ПУ (например, на диске), но и 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 ≤ 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) [9].

6.3 Линейная виртуальная память

6.3.1 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 4 096 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 рисунке 6.10. Так как длина линейного виртуального адреса 32 бита, то общий объем линейного виртуального адресного пространства составляет 2^{32} = 4 Г (байт).

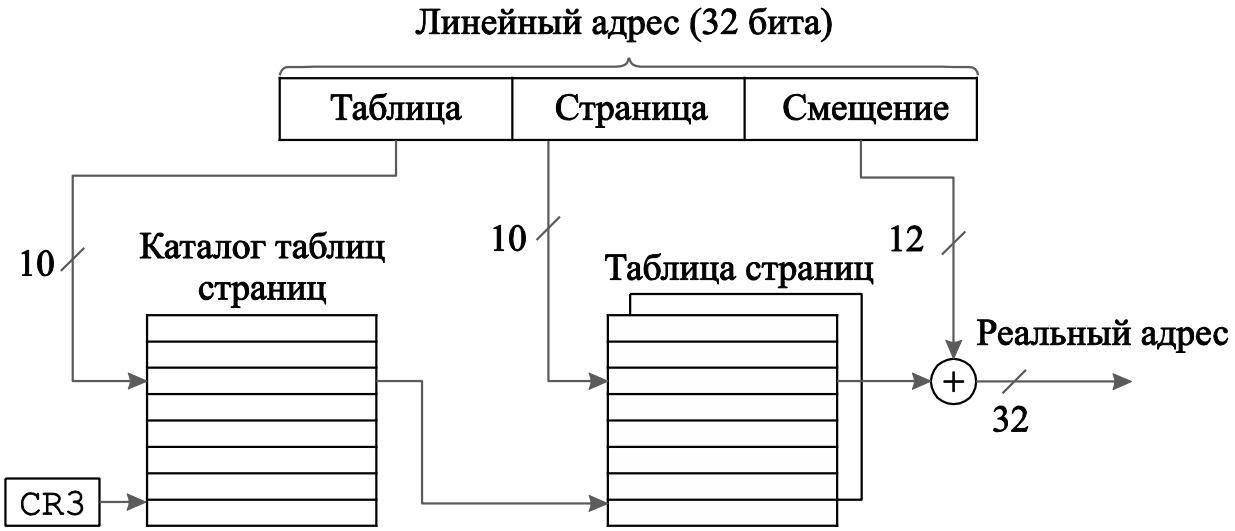


Рис. 6.10 – Преобразование линейного виртуального адреса в реальный

Как видно из рисунков 6.6, 6.7, 32 бита линейного адреса аппаратно разделяются на три части. Старшие 10 бит 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 – 1 024 (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 реги-

стре CR3. Смена содержимого этого регистра приводит к смене текущего каталога.

На рисунке 6.11 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 4 096 (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).



Рис. 6.11 – 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 1 024 *дескрипторов страниц*, каждый из которых описывает одну логическую страницу. Дескриптор страницы имеет тот же состав полей, что и дескриптор таблицы страниц (см. рис. 6.7). 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. 6.6) содержат смещение адресуемой ячейки ОП относительно начала страницы. В результате аппаратного суммирования этого смещения с начальным реальным адресом со-

ответствующей физической страницы получается искомый реальный адрес ячейки ОП.

6.3.2 Распределение памяти

Как следует из п. 6.2, 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 рисунках 6.3–6.5). Эти адреса выбираются ОС следующим образом.

Во-первых, программа каждого процесса «загружается» в собственную линейную виртуальную память размером 4 Гбайта.



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 рисунке 6.12. При этом 3 Гбайта 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.

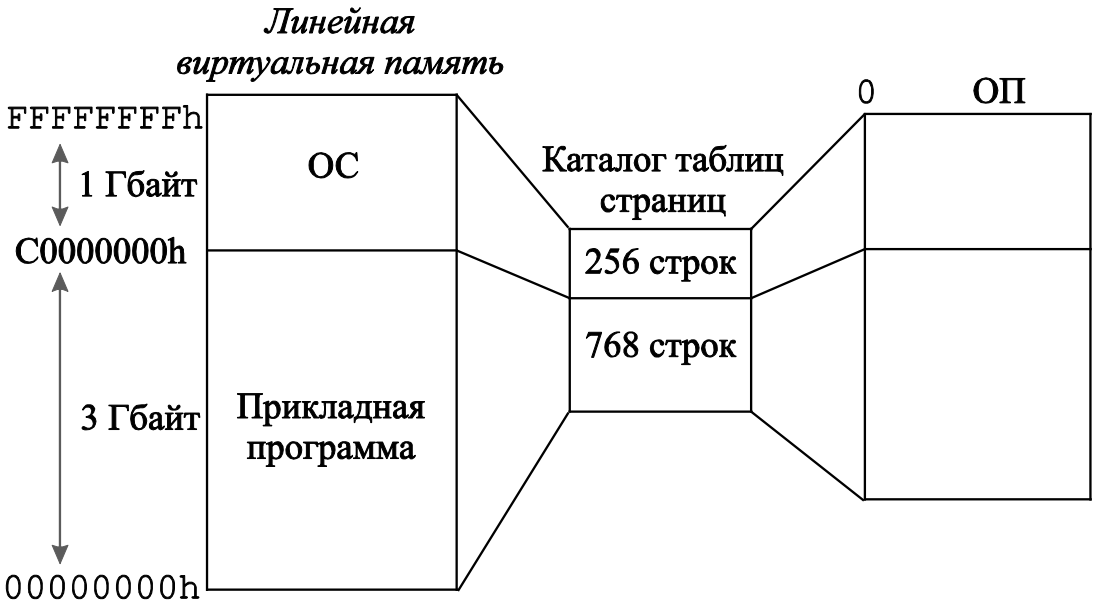


Рис. 6.12 – Отображение линейной виртуальной памяти на реальную ОП

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.

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 1 024 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 1 024 строк, каждая из которых содержит дескриптор одной логической страницы программы. Так как объем одной страницы составляет 4 096 байтов, то максимальный объем ЛВП, соответствующий одной таблице страниц, составляет 4 Мбайта. 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, ко-

торые 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. 6.7):

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 выполне-

нии 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. пп. 6.2.3).



Контрольные вопросы по главе 6

1. Используется ли регистр CS для защиты информации в ОП?
2. Используется ли дескриптор сегмента процесса для защиты информации в ОП?
3. Используется ли регистр EAX для защиты информации в ОП?
4. Как определить реальный адрес первой ячейки сегмента ОП?
5. Используется ли дескриптор таблицы страниц при страничном управлении ОП?

7 Управление файлами

7.1 Виртуальная файловая система

7.1.1 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. п. 4.3) и имеет укрупненную структуру, приведенную на рисунке 7.1. Как видно из данного рисунка, файловая подсистема включает виртуальную файловую систему, программные части реальных файловых систем, а также кэш.

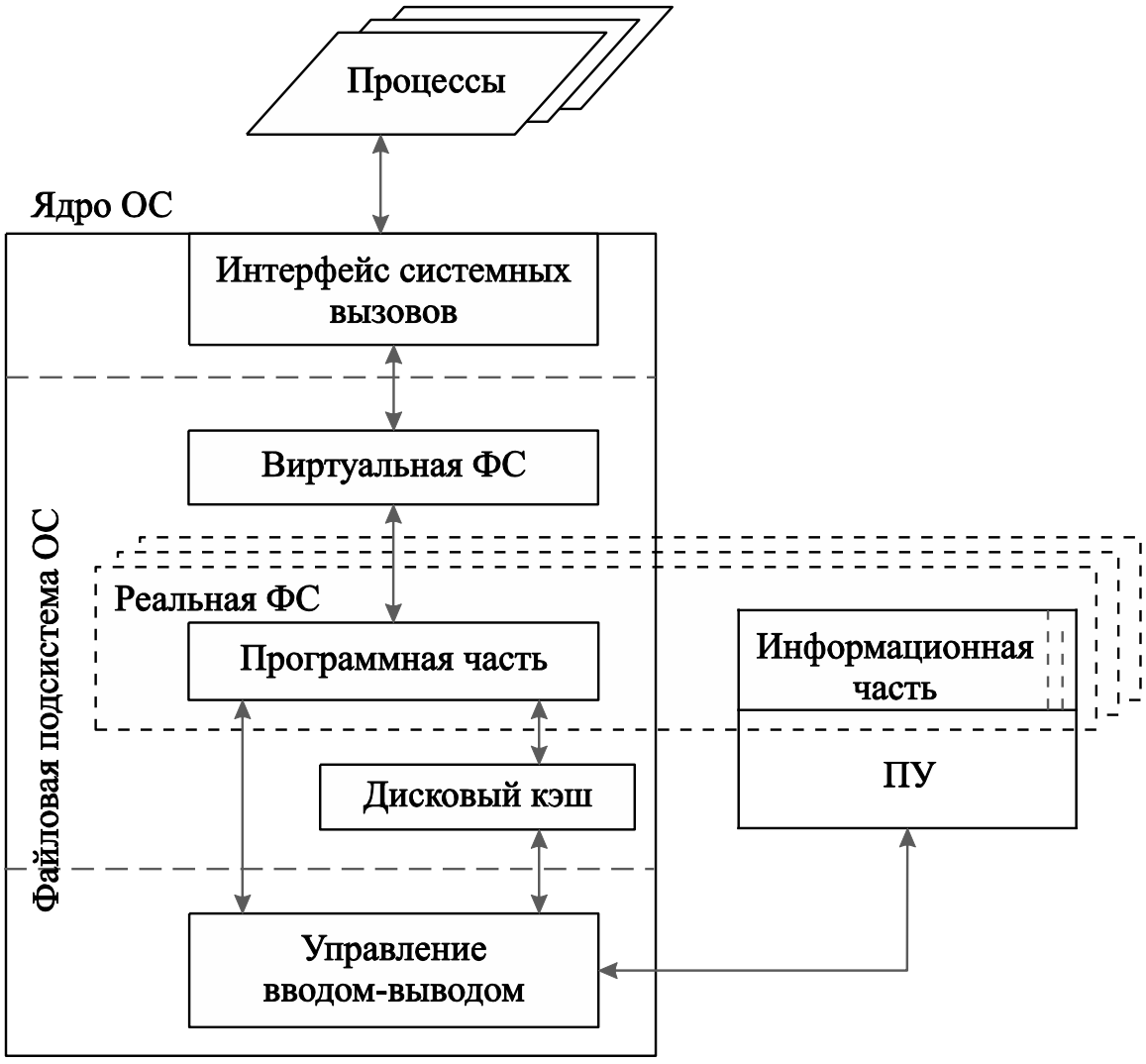


Рис. 7.1 – Структура файловой подсистемы ОС

В п. 2.2 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.

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 не известно, и ее информационное общение с программой сводится к обмену между ними цепочками байтов, которые ОС или помещает в файл, или считывает из него.

В п. 2.2 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, вы-

полняющих обработку соответствующих файлов. Во избежание путаницы будем называть ту совокупность подпрограмм, которая выполняет обработку логических файлов, *виртуальной файловой системой* (ФС).

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, не зависимую ни от физической реализации файлов, ни от физической реализации соответствующей файловой системы.

7.1.2 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 || i), (на Си – open),

где I_f – символьное имя файла одного из трех типов (простое, относительное или абсолютное), рассмотренных в п. 2.2. 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 п. 7.2, а пока лишь заметим, что основной функцией такой системы является выполнение операций с физическими файлами. Кроме того, в данном разделе мы не будем обращать особого внимания на тот факт, что в любой ФС существует не одна, а несколько реальных файловых систем. К этому вопросу мы вернемся в п. 7.3.

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. п. 5.2). 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 п. 7.3), 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 п. 7.2, 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, 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. 7.2). 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 символьное имя искомого файла в качестве входных параметров при вызове процедуры реальной ФС. Эта процедура выполняет *трансляцию имени файла* – получение на основе символьного 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 мини-

мально необходимым требованиям (см. п. 4.2). Так как каталог является разновидностью файла, то для его просмотра также требуется выполнить операцию «открытие файла». Но в отличие от открытия обычного файла открытие каталога выполняется в рамках ядра и скрыто от прикладной программы, выдавшей системный запрос.

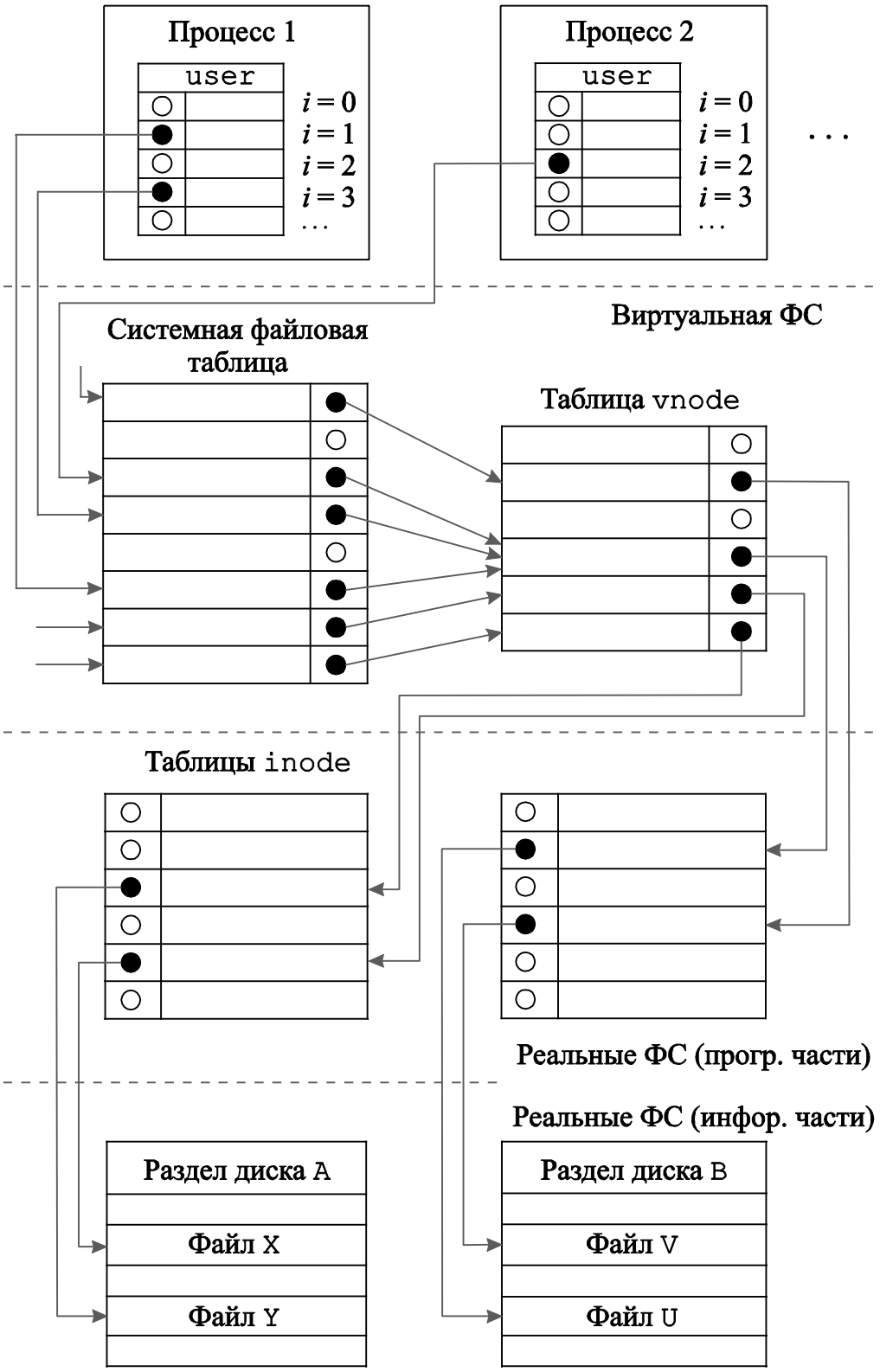


Рис. 7.2 – Системные структуры данных для доступа процессов к файлам

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 рисунке 7.2 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`.

7.1.3 Другие операции с файлами

1. *Создание файла*. Для создания нового файла любой процесс использует системный вызов:

`СОЗДАТЬ_ФАЙЛ (`I_f`, D || i)`, (на Си – `creat`),

где I_f – символьное имя файла;

`D` – задаваемые права доступа к файлу (права доступа были рассмотрены нами в п. 4.2);

`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 файлов 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 || 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 || 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 инициировании 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. *Закрытие 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 – mmap),

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.

7.2 Реальные файловые системы

7.2.1 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 уточнения 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 рассмотрены в п. 7.3, 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 такое наличие;
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 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 п. 4.4, в п. 7.3 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, используемые при создании таких систем, позволяющие существенно улучшить их оценки по основным критериям из перечисленных выше.

При изложении принципов проектирования реальных ФС далее рассматриваются 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`.

7.2.2 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 подпараграфе рассмотрим общие свойства таких носителей, а далее рассмотрим влияние этих свойств на реальную ФС.

Упрощенное представление магнитного диска приведено на рисунке 7.3. 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. 7.3).

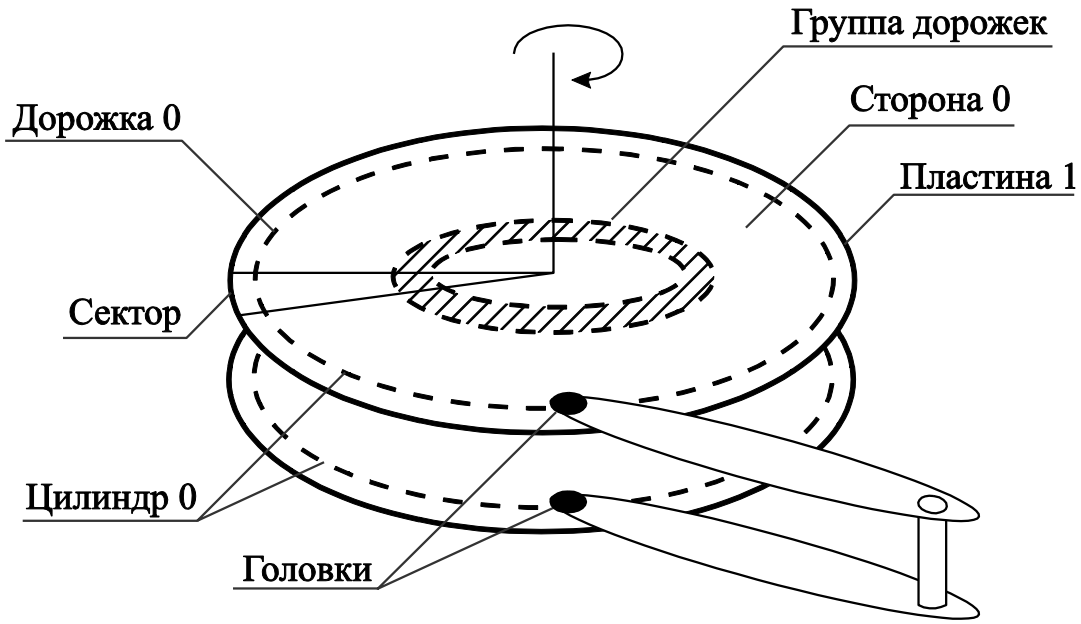


Рис. 7.3 – Упрощенное представление магнитного диска

Для чтения (записи) информации с пластины (на пластину) используется один или несколько элементов, называемых *головками*. Количество головок зависит от используемого варианта конструкции дисковода. 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 дисковода 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.

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 (рис. 7.3). 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 рисунка 7.3, один угловой сектор «вырезает» секторы одинаковой длины (измерение в битах) не только на одной поверхности одной пластины, но и на всех поверхностях всех пла-

стин. 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/с). Гибкие магнитные диски имеют скорость вращения 5–10 об/с, а жесткие – 9–170 об/с. Например, при скорости вращения 170 об/с 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: $T_r = 1/(n \cdot L)$, где L – число секторов на дорожке. Например, если диск имеет на каждой дорожке 320 секторов, а его скорость вращения n = 170 об/с, то для него $T_r = 1/(170 \cdot 320) = 0{,}018$ 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 × N, где N = 1, 2, 4, ... 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 4 K (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:

$$T_b = T_d + T_v + 8 \cdot T_r = 10\,\text{мс} + 3\,\text{мс} + 8 \cdot 0{,}018\,\text{мс} = 13{,}144\,\text{мс}\,.$$

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 1 000 блоков файла. Если эти блоки разбросаны (в результате распределения ВП файлу) по всему разделу, то общее время чтения будет очень велико:

$$T_1 = 1000 \cdot T_b = 1000 \cdot 13{,}144\,\text{мс} = 13{,}144\,\text{с}\,.$$

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 1 000 блоков нашего файла расположены на 4 дорожках одного и того же цилиндра, то среднее время их чтения или записи составит:

$$T_2 = T_d + 4 \cdot T_v + 1000 \cdot 8 \cdot T_r =$$
$$= 10\,\text{мс} + 4 \cdot 3\,\text{мс} + 1000 \cdot 8 \cdot 0{,}018\,\text{мс} = 0{,}116\,\text{с}.$$

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. Очень большим недостатком статического распределения является очень низкая используемость ВП для многих типов файлов. Причиной этого является необходимость одновременного выделения такой памяти файлу, 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. 7.3). 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 подпараграфа 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 использует-

ся. 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 байт). В результате такой фрагментации единицей распределения ВП становится фрагмент. Что касается блока, то он по-прежнему остается единицей информационного обмена между диском и ОП.

7.2.3 Каталоги

Напомним (см. п. 2.2), 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: $2^{16} – 1 = 65\,536 – 1 = 65\,535$;
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 этого 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 байт, 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 байт);
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 программной 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 `rwx rwx` --- (см. п. 4.2).

7.2.4 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 65 535. 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 файлов. 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 `s5fs`, 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 Кбайта дисковой памяти, принадлежащей данной группе цилиндров. *Битовая карта блоков* – последовательность битов, длина которой (в битах) определяется общим числом фрагметов 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 байт);
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.

7.3 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. 7.4). 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 инициирована 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`, соответствующий тому каталогу «старой» реальной ФС, который выбран в качестве точки монтирования. Далее находится тот элемент коммутатора файловых систем, который соответствует заданному типу `t` реальной ФС.

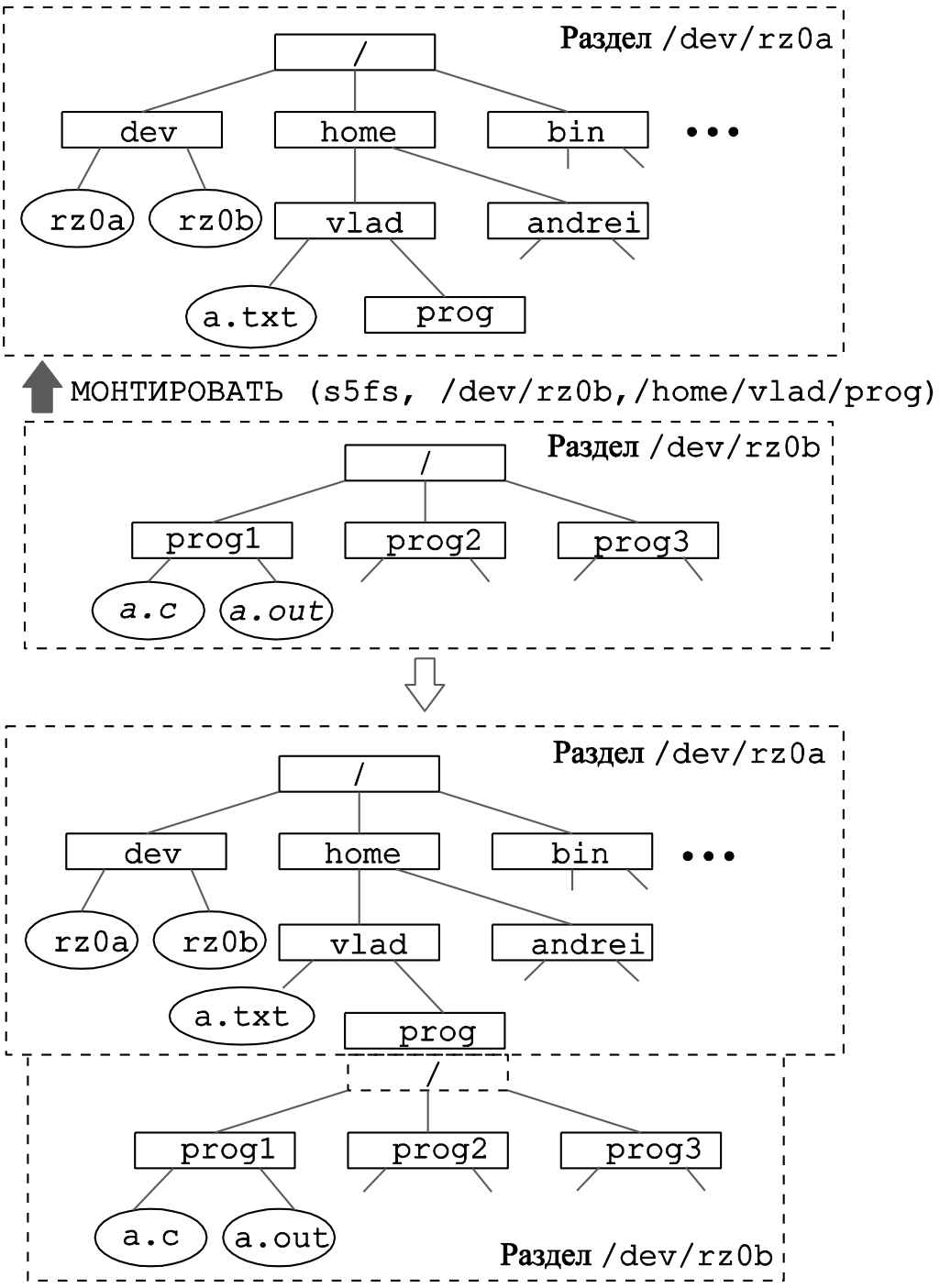


Рис. 7.4 – Пример монтирования реальной файловой системы



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, 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 мон-

тирования. Далее виртуальная ФС должна инициировать 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 поддеревья 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. 7.5). 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 вынуж-

ден 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.

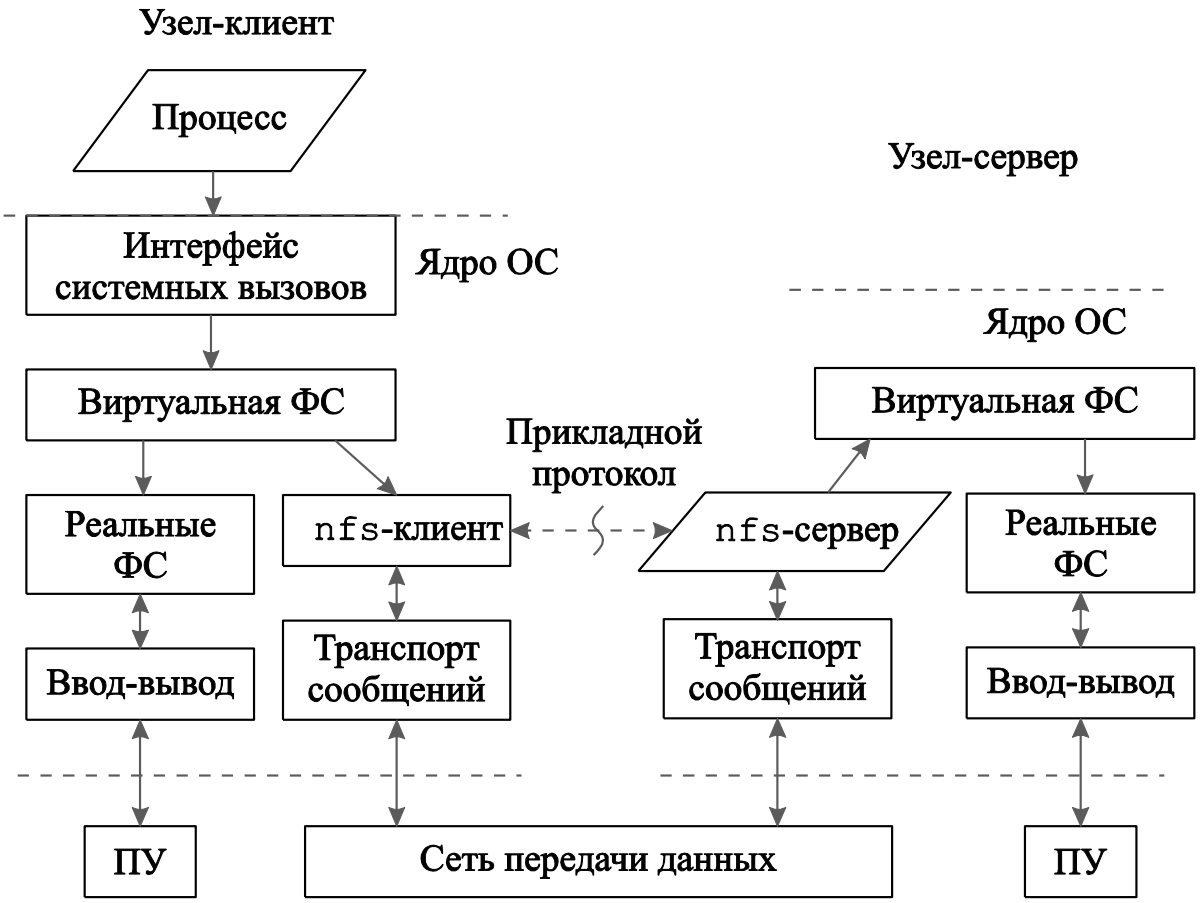


Рис. 7.5 – Распределенная реальная файловая система nfs

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.

В заключение рассмотрим назначение дискового кэша (см. рис. 7.4). Как видно из этого рисунка, 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.



Контрольные вопросы по главе 7

1. В чем отличия `fat16` от `fat32`?
2. Какими основными особенностями обладает файловая система `s5fs`?
3. Опишите типовую логическую структуру жесткого диска.
4. Для чего служит дисковый кэш?
5. В чем отличие высокоуровневого и низкоуровневого форматирования?

8 Лабораторный курс

Целью выполнения данного лабораторного курса является обучение студентов использованию командного языка UNIX, то есть языка `shell`.

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 его. При отсутствии такого класса допускается выполнение лабораторных работ в среде локальных UNIX-систем.

Конечной отчетной формой для всех лабораторных работ является отчет, выполненный в текстовом редакторе в формате *.doc, *.docx или *.odt, с указанием на титульном листе выходных данных студента, описанием пунктов выполненной работы и выводами по итогам лабораторной работы. Отчет сдается на проверку в формате *.pdf.

Для лабораторных работ № 1 и 2 разбиения по вариантам нет, в лабораторных работах № 3 и 4 выбор осуществляется по общим правилам с использованием следующей формулы:

$$V = (N \times K) \text{ div } 100,$$

где V – искомый номер варианта,

N – общее количество вариантов,

div – целочисленное деление,

при $V = 0$ выбирается максимальный вариант,

K – код варианта.

8.1 Лабораторная работа № 1. Первоначальное знакомство с UNIX

8.1.1 Цель работы

Целью выполнения настоящей лабораторной работы является получение начальных навыков работы в среде UNIX: 1) знакомство с текстовым редактором `ed`; 2) применение команд `shell` для работы с файлами; 3) использование в командах `shell` метасимволов и перенаправление ввода-вывода; 4) запуск

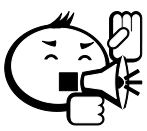
конвейеров программ; 5) применение в командах shell переменных; 6) построение командных файлов; 7) изменение прав доступа к файлам.

8.1.2 Подготовка к выполнению работы

Перед началом выполнения данной работы обязательно следует изучить следующие вопросы из теоретической части пособия:

1) файлы с точки зрения пользователя (п. 2.2) – понятие файла; простое имя файла; каталоги; файловая структура системы; текущий каталог; относительное и абсолютное имена файла;
2) простые команды shell (утилиты) для работы с файловой структурой (пп. 2.3.2) – pwd; ls; cd; mkdir; rmdir; rm с флагом -r; cp с флагом -r;
3) простые команды shell (утилиты) для работы с файлами (пп. 2.3.3, пп. 2.3.4) – cat; cp; mv; rm;
4) команда вывода строки символов на экран echo (пп. 2.3.3);
5) использование метасимволов и перенаправление ввода-вывода (пп. 2.5.2);
6) конвейеры программ (пп. 2.5.3);
7) переменные (пп. 2.5.4) – 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;
8) командные файлы (пп. 2.5.6) – понятие скрипта; способы запуска скрипта; вложенные скрипты; комментарии; позиционные параметры; инициализационный скрипт.profile;
9) защита файлов (п. 4.2) – типы пользователей; основные формы доступа к файлу; права доступа; команда ls с флагом -l; команда chmod.

В процессе изучения перечисленных вопросов обязательно следует выполнить примеры, приведенные в пособии. Некоторые из этих примеров требуют предварительного создания вспомогательных текстовых файлов и (или) каталогов. Для этого можно использовать команды cat и mkdir.



Примечание. Допустим, что после включения терминала вы оказываетесь не в среде UNIX, а в среде другой операционной си-

стемы, 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 WindowsXP/7/8 *(либо старше)*, то вы сначала входите в эту систему под своим логическим именем, которое зарегистрировано в этой локальной сети (студентам ФДО имя пользователя выдается по электронной почте по запросу, отправленному в диспетчерский отдел ФДО на адрес problems@fdo.tusur.ru). После этого следует запустить программу putty (ее можно найти в свободном доступе в сети Internet [10]), которая осуществляет запуск программного процесса, имитирующего терминал UNIX. Необходимо заполнить следующие поля: *Host name* (*or IP address*)*:* x2.kcup.tusur.ru, *Port:* 2222, *Connection type:* SSH, в настройках `putty` указать кодировку UTF-8 и нажать кнопку Open. В дальнейшем действовать в терминале как изложено выше и помнить о том, что при вводе пароля введенные символы не отображаются на экране.

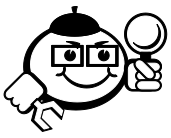
Возможно также выполнение лабораторных работ на локальной системе – для этого достаточно записать образ любого популярного дистрибутива Linux, например Ubuntu, на флеш-диск или CD, и либо загрузиться в режиме LiveCD, либо установить дополнительную ОС на свой жесткий диск [11].

. . . . . .

8.1.3 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:



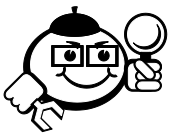
Пример

```
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:



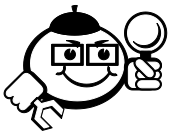
Пример

```
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:

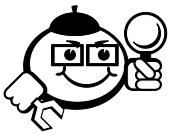


Пример

```
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
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:

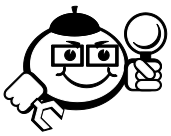


Пример

```
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;

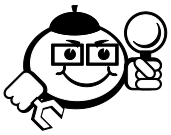
i – добавление текста перед текущей строкой; **c** – замена текущей строки текстом новой строки (строк); **d** – удаление текущей строки. Пример:



Пример

```
польз.   → $p<Enter>
ed       → that I have created 4 new lines
польз.   → a<Enter>
I will now enter two new lines of<Enter>
text to see if it is accepted.<Enter>
.<Enter>
```

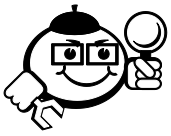
Команда редактора `$p` требует от него вывода на экран последней строки файла. Поэтому две строки, набранные в примере пользователем, добавляются в конец файла. Пример добавления текста перед текущей строкой:



Пример

```
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>
```

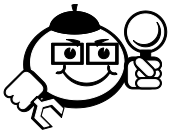
Пример замещения текста текущей строки текстом другой строки:



Пример

```
польз.   → 3p<Enter>
ed       → Once I have completed it I shall find
польз.   → c<Enter>
After completion, I shall find.<Enter>
.<Enter>
```

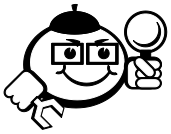
В данном примере текущая строка заменена текстом одной строки. На самом деле число новых строк может быть любым. В следующем примере производится удаление двух строк:



Пример

```
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:

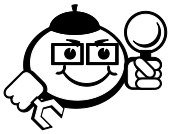


Пример

```
польз.   → 1,$d<Enter>
польз.   → 1,$p<Enter>
ed       → ?
```

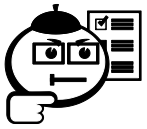
Если текст файла достаточно большой, рекомендуется выполнять промежуточное запоминание файла на диске, используя команду `w`. Выполнив ее, редактор выведет на экран новую длину файла.

Выход из редактора `ed` в UNIX выполняет команда редактора – `q`. Пример:



Пример

```
польз.   → q<Enter>
UNIX     → $
```



Выполните 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.

8.1.4 Задание

Для успешной сдачи работы требуется выполнить наизусть следующие операции:

1) создать два трехуровневых поддерева каталогов и поместить в один из каталогов четыре текстовых файла, два из которых имеют в своем имени одинаковую символьную последовательность, называемую далее «словом»;
2) с помощью `ed` отредактировать текстовый файл;
3) вывести файл на экран;
4) выполнить добавление текста в начало, в середину и в конец файла;
5) вывести файл на экран;
6) произвести переименование файла;
7) выполнить копирование файла (исходный файл и файл-копия должны располагаться в разных каталогах);
8) поместить в любой другой 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);
9) создать свой инициализационный скрипт, выполняющий действия:
   - здоровается;
   - «переделывает» приглашения `shell`;
   - запускает вложенный скрипт, созданный в (8), задав ему в качестве параметров каталог и «слово» из (1);
   - любые другие действия (по вашему желанию);
10) 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 работе. В частности нельзя применять управляющие операторы (рассматриваются в следующей работе).

Примечание 2. Для избирательного удаления файлов в (8) рекомендуется использовать команду `rm` с флагом `-i`, предварительно установив права доступа к файлам. При этом для обеспече-

ния автоматического выполнения `rm` ее стандартный ввод должен быть переключен на вспомогательный файл, содержащий любой символ кроме «y».

. . . . . .

8.2 Лабораторная работа № 2. Управляющие операторы командного языка

8.2.1 Цель работы

Целью выполнения настоящей лабораторной работы является развитие навыков программирования на языке `shell` путем использования в скриптах управляющих операторов `if`, `case`, `for`, `while`, `until`.

8.2.2 Подготовка к выполнению работы

В начале выполнения данной работы обязательно следует изучить следующие вопросы из теоретической части пособия (пп. 2.5.5):

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 выполнить примеры, приведенные в пособии.

8.2.3 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.)

8.3 Лабораторная работа № 3. Операции с файлами в программе на языке Си

8.3.1 Цель работы

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.

8.3.2 Подготовка к выполнению работы

В начале выполнения данной работы рекомендуется изучить следующие вопросы из теоретической части пособия:

1) логические файлы (пп. 7.1.1) [1, 2];
2) открытие файла (пп. 7.1.2) [1, 2];
3) создание файла (пп. 7.1.3) [1, 2];
4) понятие файлового указателя (пп. 7.1.2) [1, 2] и его перемещение (пп. 7.1.3) [1, 2];
5) чтение из файла и запись в него (пп. 7.1.3) [1, 2];
6) закрытие и уничтожение файла (пп. 7.1.3) [1, 2].

8.3.3 Введение

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 при-

кладной 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.)

8.3.4 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:

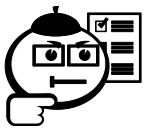
```
#include <stdlib.h> /* Для вызова exit */
#include <fcntl.h>
main(int argc, char *argv[ ])
{
 int fil;
 if (argc !=2)
 {
 printf("Need 1 argument for open\n");
 exit(1); /* Выход по ошибке */
 }
 /* Открыть файл для чтения-записи */
 if ((fil = open(argv[1], O_RDWR)) = = -1)
 {
 printf("Cannot open 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`.



Выполните приведенные выше программы.

8.3.5 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.

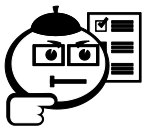
Если в состав второго операнда включена константа O_EXCL (*exclusive* – исключительный), то в случае существования файла вызов open завершится с ошибкой. Пример такого вызова:

```
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 w или a. 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 w приводит к усечению этого файла до нулевой длины, а строка a ограничивается открытием существующего файла.



Разработайте программу создания файла, получающую имя файла из командной строки. После выполнения программы проверьте наличие созданного файла.

8.3.6 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:

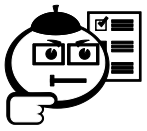
```
off_t filsize;
```

```
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 `seek`.



Создайте с помощью текстового редактора небольшой файл, а затем с помощью своей программы образуйте в этом файле «дыру», наличие которой проверьте с помощью текстового редактора

8.3.7 Чтение из файла

После того как указатель файла установлен в требуемое место файла, можно выполнить чтение из этого файла требуемого числа байтов.

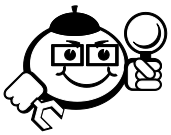
Системные вызовы. Чтение из файла выполняет системный вызов **`read`**. Его описание:

```
#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, несчитанный 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 байтам. Дело в том, что каждый системный вызов `read` инициирует считывание с диска блока байтов.

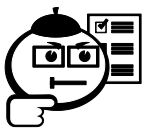
Величина этого блока есть число 512, умноженное на степень двойки, и зависит от системы. Если считывать данные с диска небольшими порциями (по несколько байтов), то никаких изменений в результатах работы программы не будет. Единственное – резко возрастет время выполнения программы, так как, во-первых, 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`).



Выполните приведенную выше программу подсчета символов.

8.3.8 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.

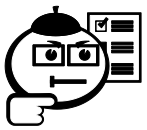
В качестве примера приведем фрагмент программы, выполняющий запись содержимого буфера `outbuf`, имеющего длину `OBSIZE`, в файл с именем `abc`:

```
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`.



Создайте программу, выполняющую добавление символов в конец текстового файла.

8.3.9 Закрытие 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. Закрытие 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. Закрытие 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`.

Для уничтожения файла вполне достаточно системного вызова `unlink`, и соответствующая стандартная функция не существует.

8.3.10 Задание

Требуется разработать одну из следующих программ:

1) копирование файла (имя старого и нового файла передаются в командной строке), используя системные вызовы;
2) копирование файла (имя старого и нового файла передаются в командной строке), используя стандартные функции;

3) вывод на экран содержимого текстового файла, имя которого задается в командной строке, используя системные вызовы;
4) вывод на экран содержимого текстового файла, имя которого задается в командной строке, используя стандартные функции;
5) ввод с клавиатуры содержимого текстового файла, имя которого задается в командной строке, используя системные вызовы;
6) ввод с клавиатуры содержимого текстового файла, имя которого задается в командной строке, используя стандартные функции;
7) ввод с клавиатуры содержимого текстового файла, имя которого задается в командной строке, используя системные вызовы и вывод его на экран, используя системные вызовы;
8) ввод с клавиатуры содержимого текстового файла, имя которого задается в командной строке, используя стандартные функции и вывод его на экран, используя системные вызовы;
9) ввод с клавиатуры содержимого текстового файла, имя которого задается в командной строке, используя системные вызовы и вывод его на экран, используя стандартные функции;
10) ввод с клавиатуры содержимого текстового файла, имя которого задается в командной строке, используя стандартные функции и вывод его на экран, используя стандартные функции.

8.4 Лабораторная работа № 4. Обработка сигналов

8.4.1 Цель работы

Целью выполнения настоящей лабораторной работы является получение навыков программного управления процессами с помощью сигналов.

8.4.2 Подготовка к выполнению работы

В начале выполнения данной работы следует ознакомиться со следующими вопросами из теоретической части пособия:

1) синхронизация процессов с помощью сигналов (пп. 3.4.1) [1, 2] – понятие сигнала; типы сигналов; выдача сигнала процессом;
2) терминальное управление процессами (пп. 3.4.2) [1, 2] – управляющий терминал; сеанс; группа процессов; получение информации о процес-

сах с помощью команды `shell – ps` с флагом `-j`; применение команды `shell - kill` с целью посылки сигнала процессу;

3) обработка сигналов (п. 5.3) [1, 2] – варианты обработки сигналов; диспозиция сигналов;
4) применение таймера для управления процессами (п. 5.5) [1, 2] – таймер; аларм; таймер интервала.

8.4.3 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.

8.4.4 Задание



Примечание. Для получения номера своего процесса можно использовать функцию `getpid`:

```
#include <sys/types.h>
pid_t getpid(void);
```

Разработать одну из следующих программ:

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 из четырех процессов, 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. Далее в течение 15 секунд процесс-отец выводит на экран какое-то сообщение. По истечении этого времени он уничтожает три из четырех процессов, используя для этого всего одну команду. Перед завершением процесс-отец выводит на экран перечень процессов данного пользователя, предваряя их текстовой строкой `hi!:`.
3. 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`.
4. 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 обрабатывает сигнал `SIGINT` в бесконечном цикле, выдавая свое сообщение на экран, добавляя к нему текущую дату. Данный цикл защищен от воздействия сигнала `SIGQUIT`;
   - процесс 2 при получении сигнала `SIGINT` выводит сообщение на экран и продолжается;

- 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.

5. Процесс-отец порождает три дочерних процесса, каждый из которых выполняет бесконечный цикл. При этом каждый из дочерних процессов особым образом реагирует на сигнал SIGINT:
   - процесс 1 обрабатывает сигнал SIGINT в бесконечном цикле, выдавая свое сообщение на экран, добавляя к нему текущую дату Данный цикл защищен от воздействия сигнала SIGQUIT;
   - процесс 2 при получении сигнала SIGINT выводит сообщение на экран и продолжается;
   - вся программа процесса 3 защищена от воздействия сигнала SIGINT.

   Сразу после порождения дочерних процессов процесс-отец завершается.

   Далее следует проверить реакцию оставшихся процессов на сигналы SIGINT и SIGQUIT.
6. Процесс-отец открывает существующий текстовый файл, а затем порождает два дочерних процесса, которые по очереди выводят содержимое этого файла фиксированными порциями по 10 символов, предваряя каждый вывод номером своего процесса. Вывод на экран заканчивается или при достижении конца файла, или по истечении интервала времени в 10 секунд.
7. Процесс-отец открывает существующий текстовый файл, а затем порождает два дочерних процесса, которые по очереди выводят содержимое этого файла фиксированными порциями по 20 символов, предваряя каждый вывод номером своего процесса. Вывод на экран заканчивается или при достижении середины файла, или по истечении интервала времени в 10 секунд.

8. Процесс-отец открывает существующий текстовый файл, а затем порождает два дочерних процесса, которые по очереди выводят содержимое этого файла фиксированными порциями по 15 символов, предваряя каждый вывод номером своего процесса. Вывод на экран заканчивается или при достижении середины файла, или по истечении интервала времени в 5 секунд.
9. Процесс-отец открывает существующий текстовый файл, а затем порождает два дочерних процесса, которые по очереди выводят содержимое этого файла фиксированными порциями по 30 символов, предваряя каждый вывод номером своего процесса. Вывод на экран заканчивается или при достижении конца файла, или по истечении интервала времени в 2 секунды.
10. Процесс-отец открывает существующий текстовый файл, а затем порождает два дочерних процесса, которые по очереди выводят содержимое этого файла фиксированными порциями по 22 символа, предваряя каждый вывод номером своего процесса. Вывод на экран заканчивается или при достижении конца файла, или по истечении интервала времени в 12 секунд.

Заключение

Ограниченный объем данного пособия не позволил рассмотреть все вопросы с одинаковой степенью подробности. При этом некоторые вопросы рассмотрены в сокращенном варианте и при наличии времени и желания могут быть рассмотрены более подробно. Перечислим эти вопросы и возможные темы дальнейшего их изучения:

1) системная поддержка мультипрограммирования – *тупики* (причины тупиков, методы предотвращения тупиков, методы ликвидации тупиков);
2) подсистема управления процессами – *реализация информационного взаимодействия между процессами* (разделяемая память, очереди сообщений, сокеты);
3) управление файлами – *реализация дискового кэша* (традиционный подход, использование подсистемы управления памятью).

Кроме того, по указанной выше причине некоторые вопросы не рассматривались в пособии совсем:

1) подсистема ввода-вывода;
2) управление передачей информации по сети;
3) переключения ЦП;
4) настройка и начальная загрузка операционной системы.

Литература

1. Таненбаум Э. С. Современные операционные системы / Э. С. Таненбаум. – 3-е изд. – СПб. : Питер, 2010. – 1040 с.
2. Одиноков В. В. Операционные системы / В. В. Одиноков, В. П. Коцубинский. – Томск : ТУСУР, 2006. – 342 с.
3. Одиноков В. В. Операционные системы и сети / В. В. Одиноков, В. П. Коцубинский. – Томск : ТУСУР, 2008. – 395 с.
4. Дунаев С. UNIX SYSTEM V. Releas 4.2 : Общее руководство / С. Дунаев. – М. : ДИАЛОГ-МИФИ, 1995. – 287 с.
5. Рейчард К., Фостер-Джонсон Э. UNIX : справочник / К. Рейчард, Э. Фостер-Джонсон. – СПб. : Питер, 2000. – 384 с.
6. Робачевский А. Операционная система UNIX / А. Робачевский. – СПб. : БХВ – Санкт-Петербург, 1999. – 528 с.
7. Соловьев А. А. Программирование на shell / А. А. Соловьев. – М. : Финансы и статистика, 1999. – 318 с.
8. Столлингс В. Операционные системы. Внутреннее устройство и принципы проектирования / В. Столлингс. – М. : Вильямс, 2002. – 848 с.
9. Олифер В. Г. Сетевые операционные системы / В. Г. Олифер, Н. А. Олифер. – СПб. : Питер, 2001. – 544 с.
10. PuTTY Download Page [Электронный ресурс]. – Режим доступа: http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html (дата обращения: 05.05.2016).
11. Установка Ubuntu Linux : инструкция для пользователя Windows [Электронный ресурс]. – Режим доступа: http://help.ubuntu.ru/wiki/ubuntu_install (дата обращения: 11.01.2016).

Глоссарий

BIOS – базовая система ввода-вывода.

Адрес – место в ОП, которое займет соответствующий программный объект – команда или данное.

Аларм – отправка сигнала процессу по истечении заданного интервала времени.

Алгоритм – правило, определяющее последовательность действий над исходными данными, приводящую к получению искомых результатов.

Аппаратура – совокупность аппаратных средств ВС, предназначенных для выполнения машинных программ.

Ассемблер – язык низкого уровня, пригодный для написания виртуальных программ.

Выражение – совокупность нескольких переменных и (или) констант, соединенных знаками операций. Различают арифметические и логические выражения.

Интерфейс – граница между двумя взаимодействующими системами.

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

Канал – специальный файл, запись в который возможна только с одного, а чтение – с другого конца.

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.

Коммутатор файловых систем – массив, элементами которого являются описатели точек входа в программные части реальных ФС.

Компилятор – транслятор, выполняющий преобразование текста программы на языке высокого уровня в программу на языке низкого уровня.

КОП – это комбинация битов, кодирующая тип операции, которую следует выполнить над операндами (например, суммирование).

Линейная виртуальная память (*ЛВП*) – абстракция, используемая не самой программой (она работает с сегментной виртуальной памятью), а операционной системой.

Логический ресурс (*ресурс*) – объект, нехватка которого приводит к блокированию процесса и переводу его в состояние «Сон».

Логическое выражение – выражение, имеющее только два значения – 0 (истина) и 1 (ложь).

Монтирование – операция подсоединения одной информационной части реальной ФС к файловой структуре системы.

Оперативная память (*ОП*) – память, предназначенная для кратковременного хранения программ и обрабатываемых ими данных.

Операционная система (*ОС*) – множество управляющих программ, предназначенных для выполнения трех функций: 1) оказание помощи прикладным и системным обрабатывающим программам в использовании ими ресурсов ВС; 2) обеспечение однопользовательской мультипрограммности; 3) обеспечение многопользовательской мультипрограммности.

Отложенный вызов – выполнение требуемой подпрограммы ядра через требуемое время.

Периферийные устройства (*ПУ*) – устройства ввода-вывода, устройства внешней памяти, а также устройства сопряжения со средой передачи данных.

Пользовательский приоритет (*NICE*) – приоритет, заявленный пользователем при создании процесса.

Поток данных – последовательность байтов, в состав которой не входят какие-то особые байты, используемые информационным каналом.

Права доступа к файлу – формы доступа к файлу, разрешенные для этого пользователя.

Препроцессор – транслятор, выполняющий обработку исходных модулей программы, подсоединяя к ним содержимое файлов заголовков и выполняя подстановки, заданные в этих файлах.

Прикладная программа – форма представления алгоритма решения задачи, ориентированная на машинную реализацию.

Протокол – алгоритм взаимодействия модулей, удаленных друг от друга.

Прототип функции – правильно записанный ее виртуальный интерфейс, с указанием типов всех ее параметров, включая значение, возвращаемое функцией.

Процесс – последовательная программа, располагающая окружением, достаточным для своего выполнения.

Псевдотерминал – программный модуль ОС, предназначенный для передачи информации между двумя процессами, а также для некоторого преобразо-

вания этой информации. Это преобразование нужно для того, чтобы один из процессов, соединяемых псевдотерминалом, выполнялся бы так, как будто он взаимодействует с настоящим терминалом.

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).

Сегментный свопинг – перекачка сегментов между ОП и ВП.

Сервер файловой системы – процесс (демон), предназначенный для выполнения удаленных системных вызовов.

Сеть передачи данных – совокупность ЭВМ, связанных каналами передачи данных.

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

Синхронизация – согласование этапов выполнения двух или более параллельных процессов путем обмена ими инициирующими (командными) воздействиями.

Системная функция – подпрограмма, объектный модуль которой находится в одной из системных библиотек.

Сообщение – последовательность байтов, предваряемая специальными служебными байтами.

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).

Статическое распределение ВП – ситуация распределения, когда вся память назначается файлу при его создании.

Таймер – аппаратное устройство, выдающее сигналы прерывания в ЦП через фиксированный промежуток времени, называемый тиком. В UNIX тик обычно равен 10 мс, но может иметь и другое значение. Прерывания таймера имеют наивысший приоритет среди внешних маскируемых прерываний.

Текущий прикладной приоритет – динамически изменяющийся приоритет процесса в состоянии «Задача».

Текущий системный приоритет – приоритет процесса в состоянии «Ядро».

Терминал – совокупность устройства ввода и устройства вывода.

Уровень привилегий PL – целое число от 0 до 3, где 0 – самый привилегированный уровень, 3 – самый непривилегированный уровень.

Файл – часть пространства носителя ВП (разрывная или непрерывная), которой присвоено имя, уникальное для данной ВС.

Функция – так при программировании на Си называется любая подпрограмма, в том числе и системная.