

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования

**«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ТОМСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»**

В.П. Коцубинский

ОПЕРАЦИОННЫЕ СИСТЕМЫ И СЕТИ

*Рекомендовано в качестве учебного пособия
Редакционно-издательским советом
Томского политехнического университета*

Издательство
Томского политехнического университета
2013

УДК 681.3.066(075.8)
ББК 32.973.2-018.73
К-42

К-42

Коцубинский В.П.

Операционные системы и сети: учебное пособие/
В.П. Коцубинский; Томский политехнический университет. - 2-е изд. - Томск: Изд-во Томского политехнического университета, 2013. – 197 с.

В авторской редакции

Учебное пособие состоит из семи глав, предусмотренных программой курса. Приведено содержание основных тем дисциплины, указан перечень лабораторных работ.

Пособие подготовлено на кафедре и предназначено для работы магистрантов, обучающихся по направлению 2310000 «Программная инженерия».

УДК 681.3.066(075.8)
ББК 32.973.2-018.73

Рецензенты

Кандидат технических наук, доцент ТУСУР
М.В. Черкашин

Кандидат технических наук, доцент ТГУ
Г.Н. Решетникова

© ФГБОУ ВПО НИ ТПУ, 2013
© Коцубинский В.П., 2013
© Обложка. Издательство Томского
политехнического университета, 2013

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ	6
1. ИНТЕРФЕЙСЫ ПОЛЬЗОВАТЕЛЯ СИСТЕМЫ	8
1.1. Функции системных программ	8
1.2. Файлы.....	13
Программные	14
символическая связь	17
1.3. Утилиты	18
1.4. Трансляторы	26
СИСТЕМА ПРОГРАММИРОВАНИЯ	32
1.5. Язык управления операционной системой	33
1.5.1. Типы языков управления	33
1.5.2. Простые команды	34
1.5.3. Составные команды.....	38
1.5.4. Переменные и выражения.....	41
1.5.5. Управляющие операторы.....	47
1.5.6. Командные файлы	53
2. СИСТЕМНАЯ ПОДДЕРЖКА МУЛЬТИПРОГРАММИРОВАНИЯ	60
2.1. Мультипрограммирование.....	60
2.2. Процессы	61
2.3. Ресурсы	65
2.4. Синхронизация параллельных процессов	68
2.4.1. Синхронизация с помощью сигналов	68
2.4.2. Терминальное управление процессами	71
2.4.3. Синхронизация конкурирующих процессов.....	75
2.4.4. Синхронизация кооперирующихся процессов	80
2.5. Информационные взаимодействия между процессами	81
2.5.1. Разделяемая память	81
2.5.2. Понятие информационного канала	84
2.5.3. Обыкновенные программные каналы.....	86
2.5.4. Именованные программные каналы	88
2.5.5. Очереди сообщений.....	89
3. ПОДДЕРЖКА МНОГОПОЛЬЗОВАТЕЛЬСКОЙ РАБОТЫ И СТРУКТУРА СИСТЕМЫ	93
3.1. Управление доступом пользователя в систему	93
3.2. Защита файлов.....	97
3.3. Укрупненная структура операционной системы.....	102
3.4. Структура сетевой операционной системы	105
4. ПОДСИСТЕМА УПРАВЛЕНИЯ ПРОЦЕССАМИ	113
4.1. Состояния процесса	113
4.2. Создание процесса	117
4.3. Обработка сигналов.....	120
4.4. Диспетчеризация процессов	122
4.5. Использование таймера для управления процессами	129
5. УПРАВЛЕНИЕ ФАЙЛАМИ	131
5.1. Виртуальная файловая система	131

5.1.1. Логические файлы	131
5.1.2. Открытие файла	133
5.1.3. Другие операции с файлами	138
5.2. Реальные файловые системы	142
5.2.1. Критерии оценки файловых систем.....	142
5.2.2. Физическое размещение информации на носителе.....	145
5.2.3. Каталоги.....	151
5.2.4. Управляющие структуры данных	153
5.3. Объединение реальных файловых систем	156
5.4. Кэширование блоков данных	162
6. ЛАБОРАТОРНЫЙ КУРС.....	163
6.1 Лабораторная работа №1	163
6.1.1 Цель работы.....	163
6.1.2 Подготовка к выполнению работы	163
6.1.3 Вход в систему и выход из нее	164
6.1.4 Текстовый редактор <i>ed</i>	167
6.1.5 Работа в среде <i>Midnight Commander</i>	170
6.1.6 Задание.....	175
6.2 Лабораторная работа №2	175
6.2.1 Цель работы.....	175
6.2.2 Подготовка к выполнению работы	176
6.2.3 Задание.....	176
6.3 Лабораторная работа №3	177
6.3.1 Цель работы.....	177
6.3.2 Подготовка к выполнению работы	177
6.3.3 Задание.....	177
6.4 Лабораторная работа №4	178
6.4.1 Цель работы.....	178
6.4.2 Подготовка к выполнению работы	178
6.4.3 Сообщения другому пользователю	179
6.4.4 Задание.....	180
6.5 Лабораторная работа № 5	180
6.5.1 Цель работы.....	181
6.5.2 Подготовка к выполнению работы	181
6.5.3 Задание.....	181
6.6 Лабораторная работа № 6	181
6.6.1 Цель работы.....	182
6.6.2 Подготовка к выполнению работы	182
6.6.3 Создание процесса.....	182
6.6.5 Задание.....	183
6.7 Лабораторная работа № 7	183
6.7.1 Цель работы.....	183
6.7.2 Подготовка к выполнению работы	183
6.7.3 Изменение диспозиции сигналов	183
6.7.4 Задание.....	184
6.8 Лабораторная работа № 8	185
6.8.1 Цель работы.....	185

6.8.2	Функции терминальной линии	185
6.8.3	Задание.....	186
6.9	Лабораторная работа № 9	187
6.9.1	Цель работы.....	187
6.9.2	Подготовка к выполнению работы	187
6.9.3	Задание.....	187
6.10	Лабораторная работа № 10	188
6.10.1	Цель работы.....	188
6.10.2	Состав и порядок выдачи системных вызовов	188
6.10.3	Задание.....	189
6.11	Лабораторная работа № 11	190
6.11.1	Цель работы.....	190
6.11.2	Порядок выдачи системных вызовов.....	190
6.11.3	Задание.....	190
6.12	Лабораторная работа № 12	191
6.12.1	Цель работы.....	191
6.12.2	Состав и порядок выдачи системных вызовов	191
6.12.3	Задание.....	192
7.	ИНДИВИДУАЛЬНОЕ ТВОРЧЕСКОЕ ЗАДАНИЕ	193
7.1.	Методические указания по выполнению индивидуального задания	193
7.2.	Задания на творческую работу	194
	ЛИТЕРАТУРА	196

ВВЕДЕНИЕ

Данное пособие предназначено для обучения студентов-бакалавров по направлению 231000 «Программная инженерия» кафедры Оптимальных систем управления, по дисциплине «Операционные системы», изучаемой в течение одного семестра. Изучение дисциплины завершается сдачей экзамена.

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) применение полученных знаний и навыков для анализа одной или нескольких операционных или файловых систем.

Изучение данной дисциплины предполагает предварительное изучение студентами следующих дисциплин: 1) информатика; 2) алгоритмические языки и программирование. Изучение курса «Операционные системы» заканчивается получением зачета и сдачей экзамена.

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

- 1) пользовательские интерфейсы и основные принципы построения операционных систем – главы 1–3;
- 2) реализация основных программных интерфейсов – главы 4 и 5;
- 3) лабораторные работы и индивидуальное задание – главы 6 и 7.

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

Следует отметить, что при описании системных программных вызовов в теоретической части пособия приводятся не реальные системные вызовы *UNIX*, записанные на языке программирования этой системы СИ, а упрощенные их варианты, записанные на псевдоязыке. Его применение преследует цель сделать материал пособия доступным для читателя, не знакомого с СИ, а также позволяет избавиться от деталей, связанных с применением данного языка и не относящихся к ОС.

Особо хотелось бы выразить благодарность доценту кафедры КСУП ТУСУР Одинокovu Владимиру Викторовичу, который разработал этот курс и преподавал его долгие годы в стенах вуза в 1962 года отделившегося от тогда просто Томского Политехнического Института, который сейчас является Национально Исследовательским Томским государственным политехническим университетом.

1. ИНТЕРФЕЙСЫ ПОЛЬЗОВАТЕЛЯ СИСТЕМЫ

1.1. Функции системных программ

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

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

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

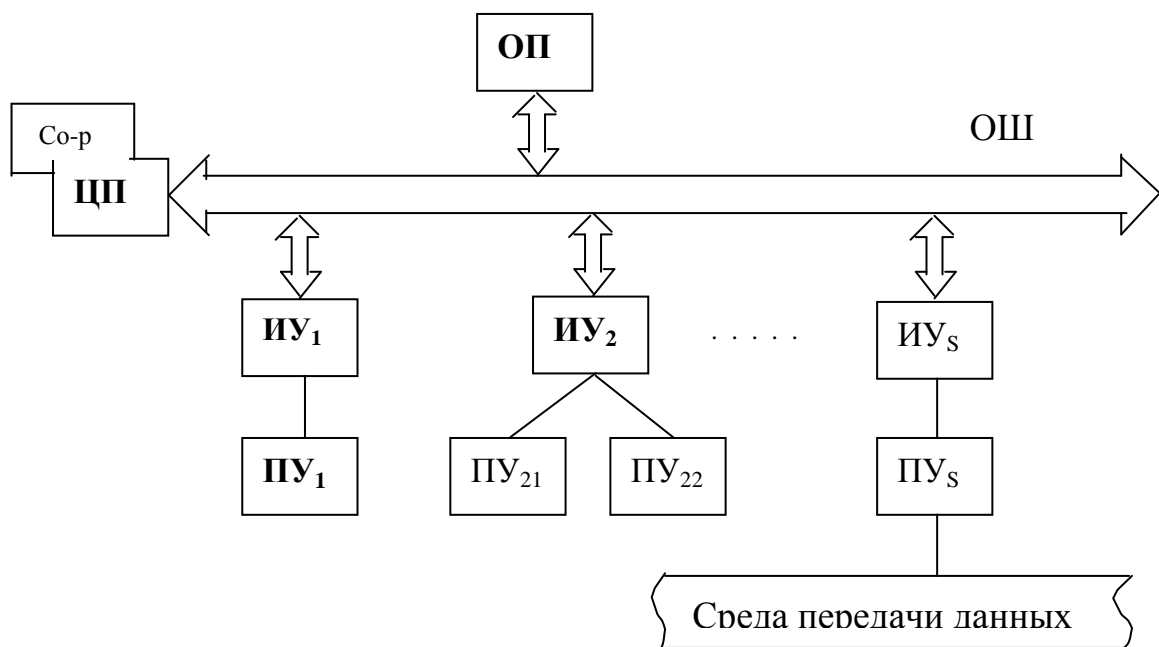


Рис.1. Однопроцессорная ЭВМ с общей шиной

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

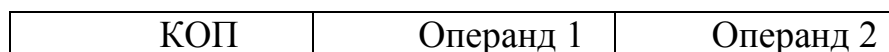


Рис.2. Структура машинной команды

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

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

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

передачи данных. Посредством *устройств ввода-вывода* ЭВМ «разговаривает» с человеком-пользователем. Сюда относятся: клавиатура, экран (дисплей), мышь, принтер и т.д.

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

- 1) обмен информацией между ЦП и ВП выполняется во много раз медленнее, чем между ЦП и ОП;
- 2) ЦП не может выполнять команды, записанные в ВП. Для выполнения этих команд их необходимо предварительно переписать в ОП;
- 3) информация на носителе ВП сохраняется и после выключения питания.

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

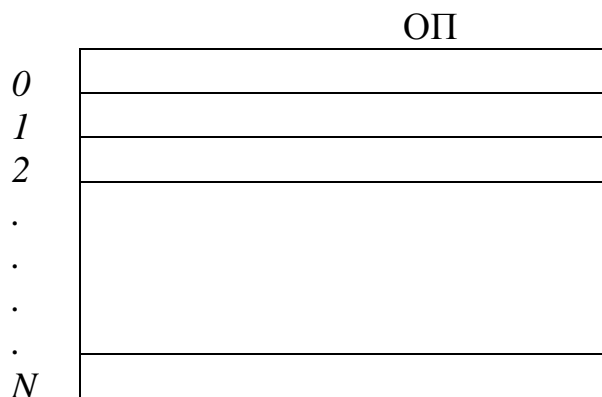


Рис. 3. Логическая структура ОП

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

Интерфейсное устройство (ИУ) предназначено для того, чтобы согласовать стандартную для данной ЭВМ структуру ОШ с конкретным типом ПУ, которых существует очень много. Поясним смысл слова: *интерфейс* – граница между двумя взаимодействующими системами. В данном случае речь идет о взаимодействии ОШ (с одной стороны) и ПУ (с другой).

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

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

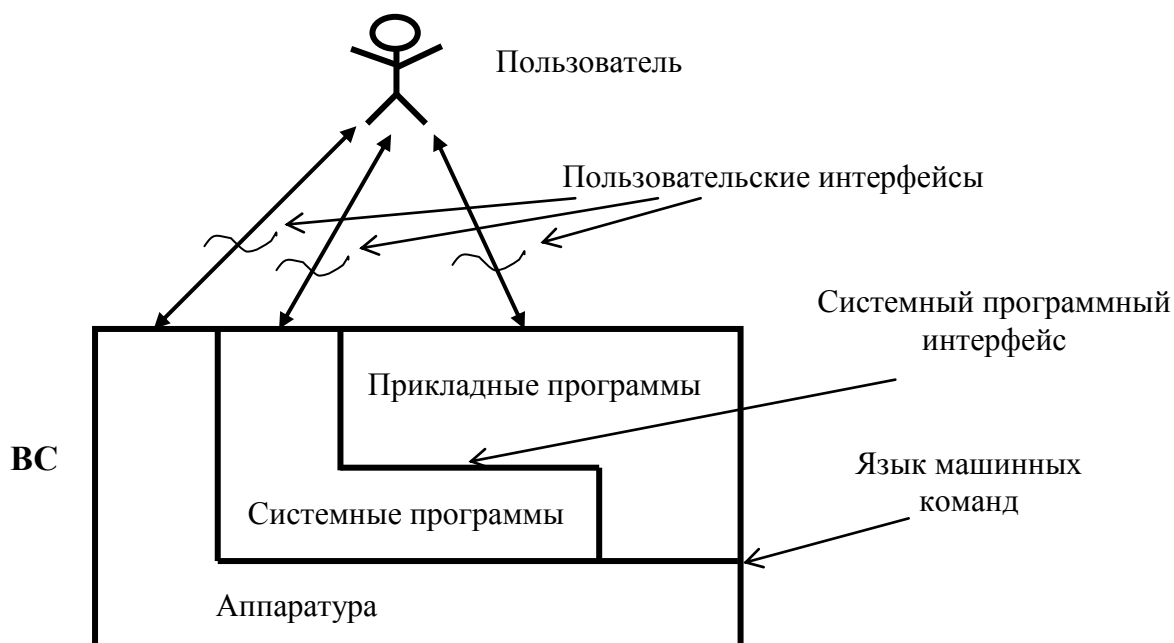


Рис.4. Укрупненное представление вычислительной системы

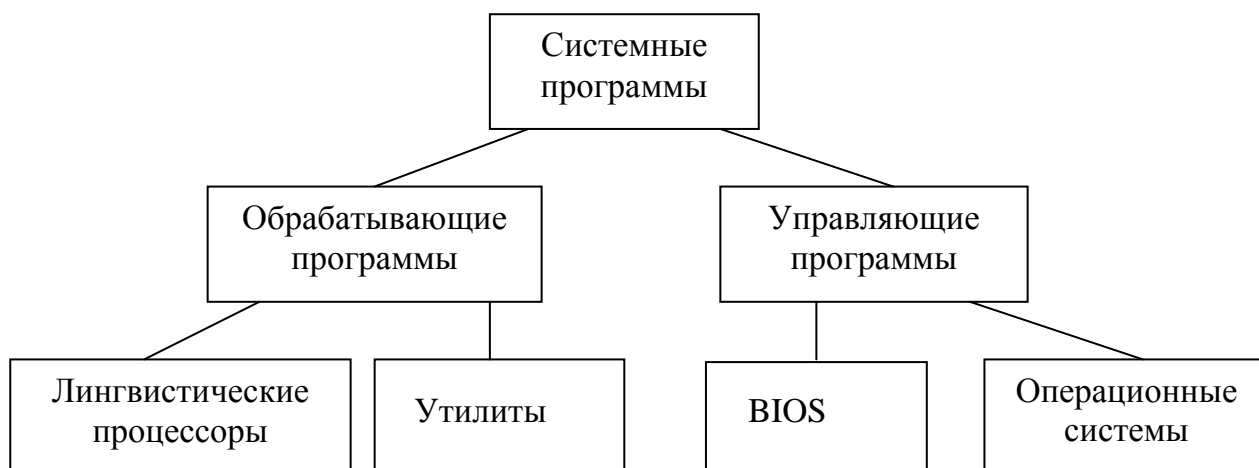


Рис. 5. Классификация системных программ

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

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

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

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

Запуск обрабатывающей системной программы аналогичен запуску прикладной программы.

Основные функции **управляющих программ**:

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

2) обеспечение **однопользовательской мультипрограммности** - одновременное выполнение нескольких прикладных и (или) системных обрабатывающих программ в интересах одного пользова-

теля. 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.

1.2. 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 ($1Г = (1024)^3$). Так как работать с такой длинной строкой чрезвычайно неудобно, то она разделяется на поименованные части разной длины, называемые файлами. Более точное определение: **файл** – часть пространства носителя ВП (разрывная или непрерывная), которой присвоено имя, уникальное для данной ВС.

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

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



Рис. 6. Классификация имен объектов, управляемых ОС

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

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

Файл является достаточно крупной единицей информации. Для удобства работы с ним битовая строка, образующая файл, делится на части, называемые **записями**. Так как при взаимодействии с ВС пользователь «не видит» записи файла, 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 рис.7 приведен фрагмент этого дерева. Корневой каталог имеет имя “/” (прямой слеш). Простое имя любого другого каталога ничем не отличается от имени файла данных. В отличие от *MS-DOS* и *WINDOWS* файловая структура в *UNIX* представляет собой дерево с пересечениями. Наличие таких пересечений обусловлено тем, что один и тот же файл может быть одновременно «зарегистрирован» не в одном, а в нескольких каталогах. При этом все жесткие связи файла полностью равноправны. Следствием этого является то, что для уничтожения файла требуется уничтожение записей о нем во всех его родительских каталогах.

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

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

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*.

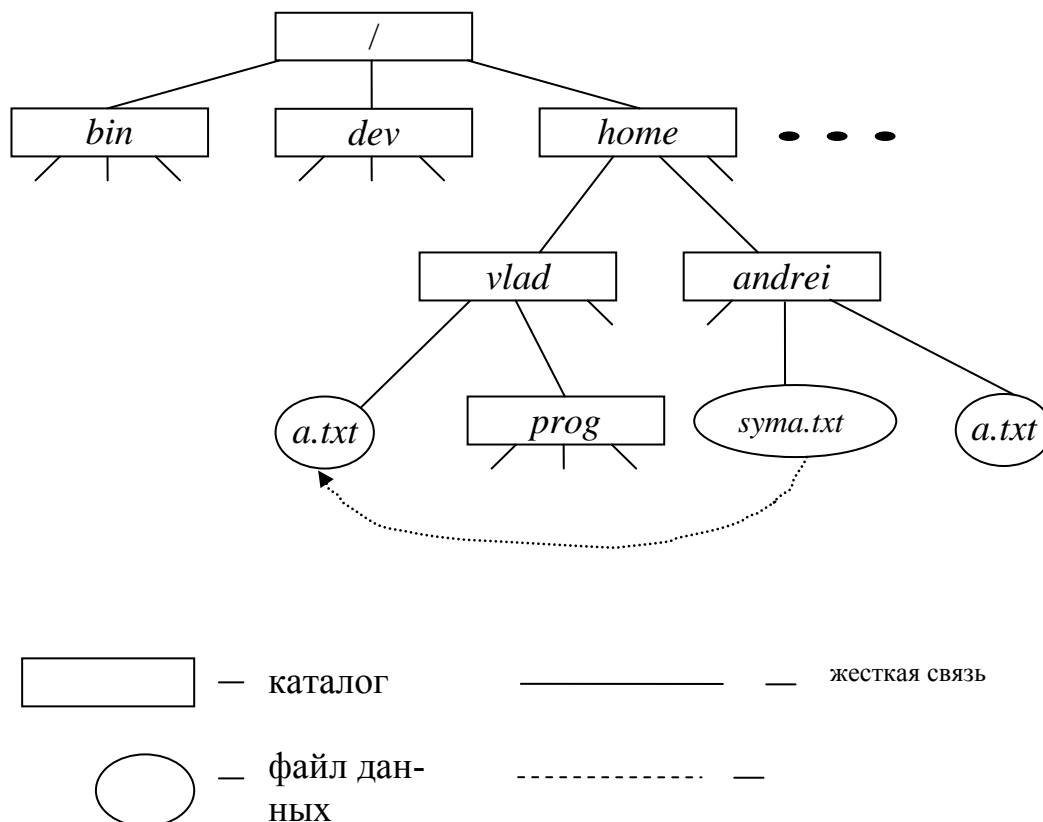


Рис.7. Фрагмент файловой структуры UNIX

Особенностью файловой структуры в мультипрограммной системе является наличие символических связей. **Символическая связь** (в *WINDOWS* – **ярлык**) – специальный файл, содержащий пользовательское имя другого – целевого файла. Например, на рис.7 файл *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 *COM1*;

/dev/rz0a – первый (*a*) раздел первого (*0*) жесткого диска;

/dev/kmem – линейная виртуальная ОП ядра ОС.

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

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

1.3. Утилиты

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

имя [**флаги**] [**файлы**],

где:

- 1) квадратные скобки заключают необязательную часть команды;
- 2) имя – пользовательское имя исполняемого файла, содержащего загрузочный модуль (машинный код) утилиты;
- 3) файлы – имена файлов, над которыми утилита выполняет свои действия. Различают входные файлы, информация из которых (или информация о которых) используется утилитой в качестве ее исходных данных, а также выходные файлы, в которые утилита помещает результаты своей работы. По умолчанию большинство системных утилит использует в качестве входного файла клавиатуру, а в качестве выходного файла – экран. Эти устройства (и соответствующие им файлы) часто называют соответственно стандартным вводом и стандартным выводом;
- 4) флаги – двоичные параметры команды, уточняющие действие, которое должна выполнить запускаемая утилита. Флаг задается своим именем из одной буквы, которой предшествует символ «-». Некоторые флаги уточняются своими параметрами, которые отделяются от имени флага пробелами.

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

1. Идентификация и установка текущего каталога.

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 рис.7). Имя каталога может быть как абсолютным, так и относительным. Задание абсолютного имени позволяет сделать текущим любой ка-

талог, 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 то 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.3.2.

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 создается 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 содержимым, 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. Уничтожения 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*, *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 лексикографическом 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.

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 системных обрабатывающих программ (утилит и лингвистических процессоров), имена системных программных вызовов, а также имена библиотечных функций. Задав имя интересующей вас системной программы, вы можете получить подробные сведения об ее использовании (правда, на английском языке). Например, можно спросить утилиту *man* о ней самой.

8. Упрощение пользовательского интерфейса.

Эту функцию выполняют достаточно сложные утилиты, в названии которых часто присутствует слово *commander*. Примером такой утилиты для *MS-DOS* является *Norton Commander*. Аналогичная утилита для

UNIX называется *Midnight Commander*. (Для того, чтобы запустить *Midnight Commander*, достаточно набрать команду *UNIX – mc.*)

Любая подобная программа предназначена для того, чтобы предоставить пользователю ВС удобный интерфейс для общения с этой системой. Это обеспечивается, во-первых, наглядным выводом на экран информации о файловой структуре системы. Для этого по запросу пользователя утилита переносит с диска на экран информацию, содержащуюся в любом каталоге файловой структуры системы. Во-вторых, любой *commander* существенно упрощает для пользователя ввод команд ОС за счет того, что он переносит имя исполняемого файла программы из позиции экрана, отмеченной пользователем с помощью псевдокурсора (*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.

1.4. Трансляторы

Программисты – не самая многочисленная, но очень важная часть пользователей ВС. Конечной задачей любого программирования является получение реальной программы, записанной на машинном языке. Только такая программа может быть понята и выполнена центральным процессором. К сожалению, трудоемкость программирования на машинном языке очень велика и не позволяет записывать на нем сколько-нибудь сложные (по решаемым задачам) программы. Выходом из этого 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 обрабаты-

вающая 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 разделе, а сейчас обратимся к трансляторам.

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

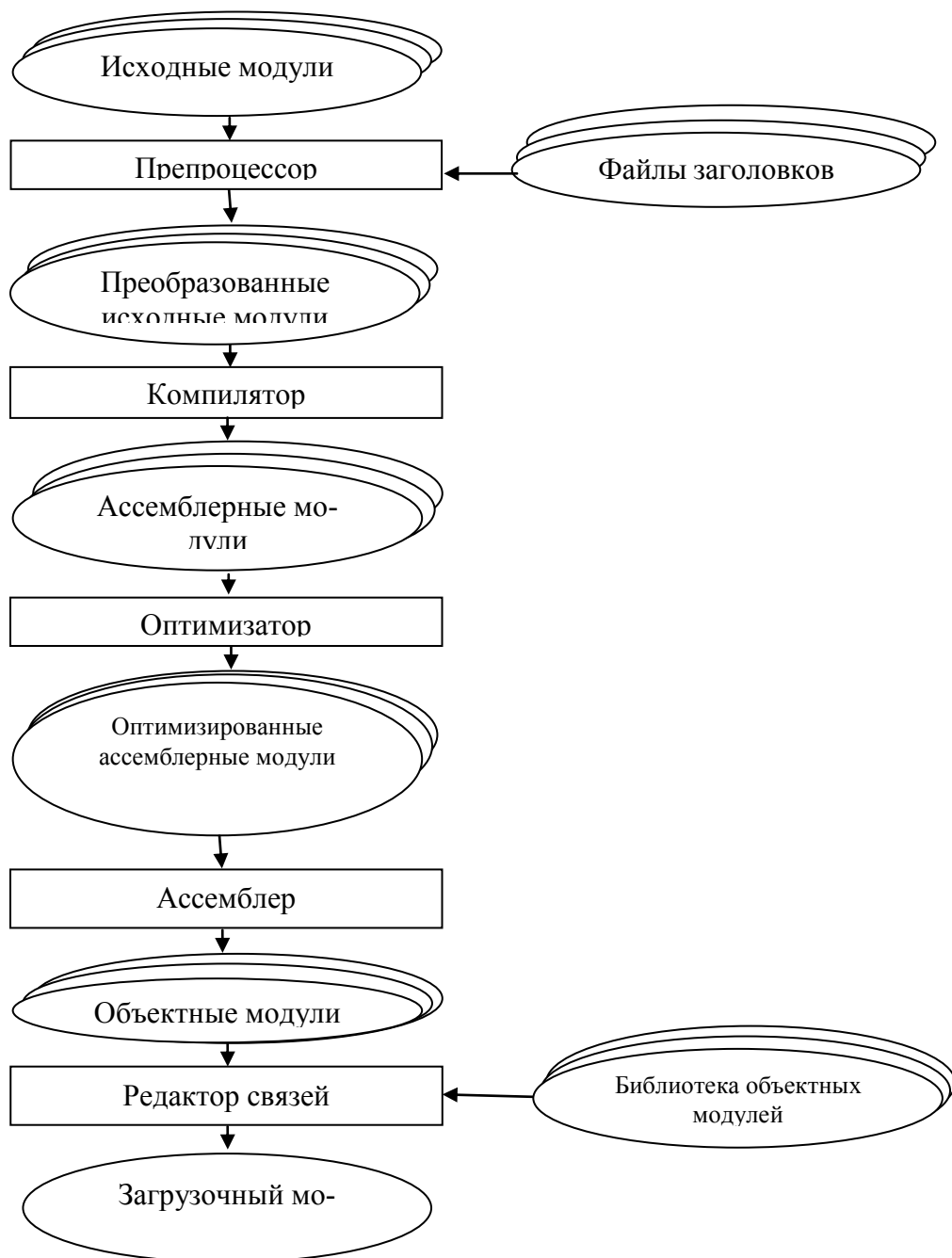


Рис.8. Преобразование исходной программы в загрузочный модуль

преобразование совокупности исходных модулей программы в соответствующий загрузочный модуль.

Исходный модуль программы – текстовый файл, содержащий всю виртуальную программу или ее часть. Если речь идет о программе на СИ, то данный файл имеет имя с суффиксом “.c”. Любой исходный модуль состоит из операторов двух типов – псевдооператоров и исполни-

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

Препроцессор – транслятор, 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.8) языком низкого уровня является ассемблер.

Ассемблер – язык низкого уровня, пригодный для написания виртуальных программ. Трудоемкость разработки таких программ весьма велика. Поэтому на ассемблере пишутся исходные тексты программ только в двух случаях: 1) программа выполняет непосредственное

управление аппаратурой ВС; 2) если предъявляются повышенные требования к эффективности программы. Поэтому программированием на ассемблере для мультипрограммных систем, например, для *WINDOWS* или *UNIX* занимается лишь небольшая часть системных программистов. Это не умаляет роль ассемблера для программирования в однопрограммных системах, а также в качестве учебного языка.

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

В отличие от программиста компилятор «программирует» на ассемблере не очень эффективно. При этом под **эффективностью программы** понимаются два критерия: затраты времени ЦП на выполнение программы и затраты ОП для ее размещения. Для улучшения оценок программы по этим двум критериям ее ассемблерные модули, выданные компилятором, подаются на вход следующего транслятора – **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 “.о”. При этом частично решается задача получения последовательности машинных команд, отображающих алгоритм модуля. Транслятор окончательно записывает коды операций машинных команд, а также проставляет в эти команды номера используемых регистров. Что касается символьных имен, то они обрабатываются транслятором-ассемблером по-разному.

Рассмотрим преобразование адресов транслятором. **Адрес** – место в ОП, которое займет соответствующий программный объект – команда или данное. Любой язык программирования (включая *СИ* и ассемблер) позволяет программисту использовать в программе не адреса, а их заменители – **символьные имена (метки)**. Основные типы меток: 1) метки операторов; 2) имена переменных; 3) имена подпрограмм (имя подпрограммы заменяет для программиста адрес, по которому первая команда подпрограммы находится в ОП).

Все символьные имена в исходном (в том числе, ассемблерном) модуле делятся на внешние и внутренние. Символьное имя называется **внутренним**, если выполняются два условия: 1) соответствующий программный объект (оператор, данное или процедура) находится в

этом же модуле; 2) данный программный объект используется (вызывается) только внутри данного исходного модуля.

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

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

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

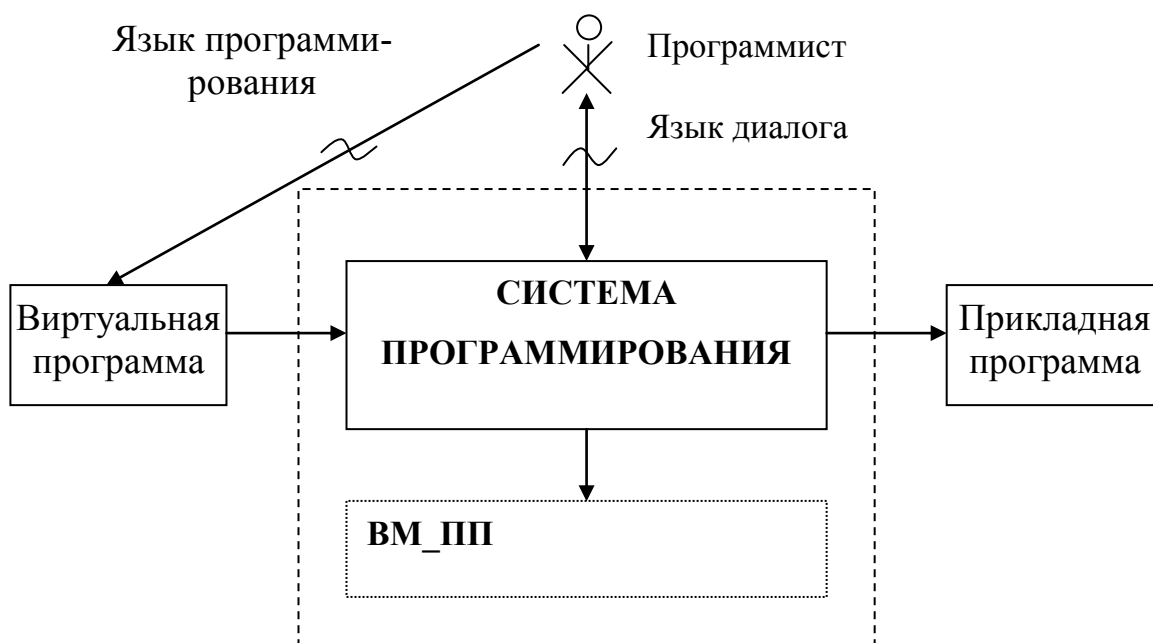
При получении загрузочного модуля редактор связей записывает в ОП один за другим объектные модули. Поэтому он “знает”, где расположен в памяти каждый программный объект. Следовательно, он может заменить все оставленные транслятором-ассемблером внешние метки на соответствующие численные адреса. В конце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

Рассмотренные выше трансляторы обычно реализуются в качестве подпрограмм более крупной программы, называемой **системой программирования**. Кроме них в систему программирования входит также подпрограмма-оболочка, выполняющая диалог с пользователем-программистом, а также утилиты: 1) текстовый редактор, предназна-

ченный 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.1.3). Точно также 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) *-l**имя* – при получении загрузочного модуля использовать требуемую библиотеку объектных модулей.



ВМ_П_СП – виртуальная машина пользователя системы программирования

ВМ_ПП – виртуальная машина прикладной программы

Рис.9. Виртуальная машина пользователя системы программирования
 Благодаря наличию системы программирования программист работает на **виртуальной машине пользователя системы программирования** (рис.9). Эта ВМ “понимает” операторы используемого языка про-

граммирования, 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_III*). 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.

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

1.5. Язык управления операционной системой

1.5.1. Типы языков управления

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

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

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

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

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

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

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

1.5.2. Простые команды

Простые команды *shell* делятся: а) команды запуска программ; б) команды вызова функций *shell*; в) вспомогательные команды. Послед-

няя из перечисленных групп команд будет рассмотрена нами в п.1.5.4 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 обрабатываемой программы (прикладной, утилиты или лингвистического процессора), а команда вызова функции *shell* запускает внутреннюю подпрограмму самого *shell*. С точки зрения пользователя ВС эти два типа команд почти не различимы. Единственное различие: команда запуска программы имеет, а команда вызова функции *shell* не имеет кода завершения. **Код завершения** – целое неотрицательное число: 0 – запущенная программа завершилась успешно; >0 – программа завершилась с ошибкой.

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

```
$cat abc.txt
```

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

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

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

2) ? – соответствует любому одиночному символу;

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

Для самих запускаемых программ метасимволы *shell* «не заметны», так как *shell* подставляет вместо имен, использующих их, обычные имена файлов. Например, пусть пользователь набрал последователь-

ность 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 также, как относится подпрограмма к

запускающей ее программе. То есть, так как ИК не может выполняться без экрана и клавиатуры, которые существуют для конкретного пользователя в единственном экземпляре, то до завершения запущенной программы ИК будет «без движения». При этом говорят, что программа запускается в *оперативном режиме*.

Если программе не нужны ни клавиатура, ни экран, то ее можно запустить в фоновом режиме. Это означает, что после запуска программы она и ИК выполняются асинхронно (не зависимо). (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 &
```

1.5.3. Составные команды

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

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

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

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

Программы, образующие конвейер, не конкурируют между собой из-за экрана и клавиатуры, так как клавиатура может быть нужна только первой, а экран – только последней программе конвейера. Поэтому эти программы запускаются *shell* одновременно (асинхронно). После своего запуска, программы, расположенные по соседству в конвейере, взаимодействуют между собой через промежуточный файл следующим образом. Программа, для которой этот файл является выходным, помещает в

него свои данные построчно (в приведенном примере каждая строка содержит имя файла). Другая программа считывает эти данные также построчно, не дожидаясь завершения работы первой программы. Заметим, что несмотря на то, что промежуточный файл реально существует на диске, его имя неизвестно для пользователя, которому, впрочем, это имя и не нужно.

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

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

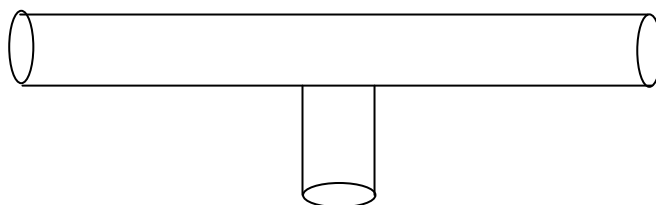


Рис.10. Наглядное изображение команды *tee*

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

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

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

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

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

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

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

Примеры командных списков:

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

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

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

Пример. Следующая двухуровневая команда выполняет уничтожение всех файлов и каталогов, простые имена которых начинаются с буквы *d* и которые расположены в поддереве файловой структуры, принадлежащем данному пользователю: `$ rm -r `find ~/ -name 'd*'``

Обратите внимание, что при задании в команде *find* имени файла (или каталога) с помощью метасимволов, это имя обязательно должно быть заключено в кавычки (одиночные или двойные). Это объясняется тем, что *shell* имеет свои метасимволы, одноименные метасимволам утилиты *find*. Кавычки играют роль “экранирующих” символов, сообщ-

щая *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-х уровневую структуру:

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

1.5.4. Переменные и выражения

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

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

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

```
$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. п. 1.5.3). При этом имя любой программы, которая выдает свой результат в виде одной или нескольких строк символов, может быть записано справа от оператора присваивания, заключенным в обратные апострофы “`”, или может быть заключено в круглые скобки, которым предшествует символ “\$”. *shell* сначала запускает указанную программу на выполнение, а затем подставляет ее выходные данные в качестве значения заданной переменной.

Пример:

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

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

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

го имени переменной должен предшествовать символ “\$”, смысл которого в данном случае есть «значение переменной». Выше приведены примеры использования значения переменной в правой части оператора присваивания, а также в качестве параметра команды *echo*.

Если имя переменной следует отделить от символов, записанных сразу за ним, то это имя следует заключить в символы “{” и “}”. Пример:

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

В данном примере в качестве параметра первого оператора *echo* записано значение неопределенной ранее переменной *varxy*. В подобных случаях *shell* подставляет вместо неопределенной переменной пустое место. 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 п.2 будет показано, что любая обрабатываемая программа имеет «родительскую программу».) Рассмотрим некоторые переменные окружения:

- 1) **HOME** – содержит полное имя корневого каталога пользователя;
- 2) **PATH** – содержит перечень абсолютных имен каталогов, в которых *shell* выполняет поиск исполняемого файла в том случае, если для его задания в своей команде пользователь использовал простое имя файла. Имена-пути каталогов, записанные в данной переменной, разделяются символом «:». Следует отметить, что *shell* не производит поиск исполняемого файла в текущем каталоге по умолчанию, и для этого требуется явное задание текущего каталога в переменной **PATH** (для этого используются символы «./»). Кроме того заметим, что *shell* не использует данную переменную для поиска неисполняемых, например, текстовых файлов. Поэтому для задания таких файлов в команде пользователя требуется или использовать их абсолютные имена, или обеспечить перед выполнением такой команды переход в родительский каталог файла;
- 3) **PS1** – приглашение *shell* в командной строке (обычно – \$);

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

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

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

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

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

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

```
$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 <i>file</i>	размер файла <i>file</i> больше 0
-r <i>file</i>	для файла <i>file</i> разрешен доступ на чтение
-w <i>file</i>	для файла <i>file</i> разрешен доступ на запись
-x <i>file</i>	для файла <i>file</i> разрешено выполнение
-f <i>file</i>	файл <i>file</i> существует и является обычным файлом
-d <i>file</i>	файл <i>file</i> существует и является каталогом
-z <i>string</i>	строка <i>string</i> имеет нулевую длину
-n <i>string</i>	строка <i>string</i> имеет ненулевую длину
<i>string1</i> = <i>string2</i>	две строки идентичны
<i>string1</i> != <i>string2</i>	две строки различны
<i>i1</i> -eq <i>i2</i>	число <i>i1</i> равно числу <i>i2</i>
<i>i1</i> -ne <i>i2</i>	число <i>i1</i> не равно числу <i>i2</i>
<i>i1</i> -lt <i>i2</i>	число <i>i1</i> строго меньше числа <i>i2</i>
<i>i1</i> -le <i>i2</i>	число <i>i1</i> меньше или равно числу <i>i2</i>
<i>i1</i> -gt <i>i2</i>	число <i>i1</i> строго больше числа <i>i2</i>
<i>i1</i> -ge <i>i2</i>	число <i>i1</i> больше или равно числу <i>i2</i>

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

! – логическое отрицание;

-a – логическое И;

-o – логическое ИЛИ.

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

Для *shell* характерно то, что запись логического выражения еще не означает его автоматического вычисления. Для этого требуется, чтобы элементы логического выражения были записаны в качестве параметров команды *test*. Результатом выполнения этой команды является код завершения: 0 – логическое выражение истинно, 1 – ложно. Команду *test*

можно задать одним из двух способов: 1) обычным способом; 2) заключив логическое выражение в квадратные скобки.

Примеры.

```
$ var1=10; var2=20; var3=30
1)$test $var1 -gt $var2 && echo "var1 > var2"
2)$[ $var1 -gt $var2 ] || echo "var1 <= var2"
   var1 <= var2
3)$test $var1 = $var2 && echo "var1 = var2"
4)$[$var1 = $var2 -o $var2 != $var3 ]&&echo "=="
   ==
5)$test \( $var1-eq$var2\) && echo "var1 = var2"
6)$[! \( $var1 -eq 0 \)]&& echo "var1 не равно 0"
   var1 не равно 0
7)test \( $ var1 != $var2 \) -a \( $var1 -eq
$var2 \) || echo "?????"
   ?????
8) [ abc ] && echo true
   true
```

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

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

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

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

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

1.5.5. Управляющие операторы

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

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

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

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

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

```
if      команда-условие
  then
```

```
    последовательность команд 1
else
    последовательность команд 2
fi
```

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

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

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

Наиболее общая структура условного оператора:

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

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

вершения 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 много альтернативный 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):

```
$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.

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:

```
$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.

1.5.6. Командные файлы

Командный файл – файл, содержащий список команд интерпретатора команд ОС. Применение командных файлов позволяет избежать повторения набора часто используемых команд, и фактически каждый

такой файл представляет собой виртуальную программу, записанную на входном языке ИК.

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

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

```
$cat >file
# Копирование всех файлов пользователя с суффик-
com 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 п.3.)

Отличием первой команды от второй является использование в ней простого имени файла-скрипта. В связи с этим напомним, что *shell* не производит поиск исполняемого файла в текущем каталоге по умолчанию, и для этого требуется явное задание текущего каталога в переменной *PATH* с помощью команды: *PATH=\${PATH}:/*. 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 породить для этого новый экземпляр *shell*. Одним из следствий этого является то, что при выполнении скрипта могут использоваться любые переменные текущего *shell*, а не только переменные окружения. Что касается отличий между командами 4 и 5, то они аналогичны различиям между командами 1 и 2.

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

- 1) \$ file
- 2) \$. file

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

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

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

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

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

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

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

```
$ file .txt k2
```

Для того, чтобы при выполнении командного файла *shell* мог использовать значения позиционных параметров, полученные от пользователя, каждый из этих параметров имеет свое имя, в качестве которого используется порядковый номер той позиции, которую занимает параметр в командной строке. При этом в качестве параметра 0 рассматривается имя скрипта. Например, в рассматриваемом примере позиционные параметры имеют следующие значения: параметр 0 – *file*; параметр 1 – *.txt*; параметр 2 – *k2*.

Как и для переменной, значение позиционного параметра обозначается его именем, которому предшествует символ “\$”. При выполнении скрипта каждое значение его позиционного параметра заменяется его значением, полученным из командной строки. Само это значение в командном файле 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 предвзается символом “\$”.

С учетом сделанных замечаний выполним запись рассматриваемого скрипта:

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

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

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

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

```
$ cat >k1
# Меняет значения своих параметров на a и b
# параметры 1 и 2 - любые слова
echo $1; echo $2
set a b
echo $1; echo $2
<Ctrl>&<D>
```

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

```
$ ./k1 7 8
7
8
```

a
b

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

- ? – код завершения последней выполненной команды;
- \$ – системный номер процесса, выполняющего *shell*;
- ! – системный номер фонового процесса, запущенного последним;
- # – число позиционных параметров, переданных в *shell*. Имя скрипта (параметр 0) в это число не входит;
- * – перечень позиционных параметров, переданных в *shell*. Этот перечень представляет собой строку, словами которой являются позиционные параметры.

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

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

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

В отличие от предыдущего скрипта, число параметров при вызове данного скрипта может быть задано любое. Если это число меньше двух, то выполнение скрипта не приводит к изменению содержимого

какого-либо файла. Заметим также, что оператор *for i in \$** во второй строке скрипта может быть заменен на оператор *for i*. При этом используется следующее свойство оператора *for*: при отсутствии части этого оператора, начинающейся со слова *in*, в качестве перечня значений заданной переменной (*i*) используется перечень позиционных параметров, заданный при вызове скрипта.

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

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

2. СИСТЕМНАЯ ПОДДЕРЖКА МУЛЬТИПРОГРАММИРОВАНИЯ

2.1. Мультипрограммирование

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

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

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

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

В последующих разделах данной главы рассматриваются постановки задач, решение которых обеспечивает наличие мультипрограммирования в системе, а также рассматриваются методы решения этих задач.

2.2. Процессы

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

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

После того, как ИК (*shell*) будет создан и инициирован процессом *init*, он перейдет к ожиданию команды пользователя, набираемой на клавиатуре, или вводимой с помощью мыши в качестве выбранного варианта из меню, предлагаемого пользователю. В любом случае ИК по-

лучает имя программы, подлежащей выполнению путем создания и инициирования соответствующего программного процесса. Принципиальной особенностью мультипрограммной системы является то, что запуск новой «дочерней» программы может быть выполнен до завершения предыдущей «дочерней» программы. Поэтому в отличие от однопрограммной системы количество одновременно существующих дочерних процессов может быть более одного. Например, одновременно запускаются программы, образующие конвейер (см. п.1.5.3). Параллельно с программой в оперативном режиме могут выполняться программы в фоновом режиме.

Например, пусть пользователь ввел следующую команду:

```
$ find ~/ -name 'f*' | tee file1 | fgrep fl
>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 рис.11. Обратим внимание, что выполнение команды *echo* не приводит к появлению нового процесса, так как эту команду выполняет не отдельная утилита, а подпрограмма самого *shell*. Кроме того заметим, что не могут существовать одновременно два процесса *cat*, так как каждый из них должен выполняться в оперативном режиме (требуется экран), и поэтому один из процессов изображен пунктиром.

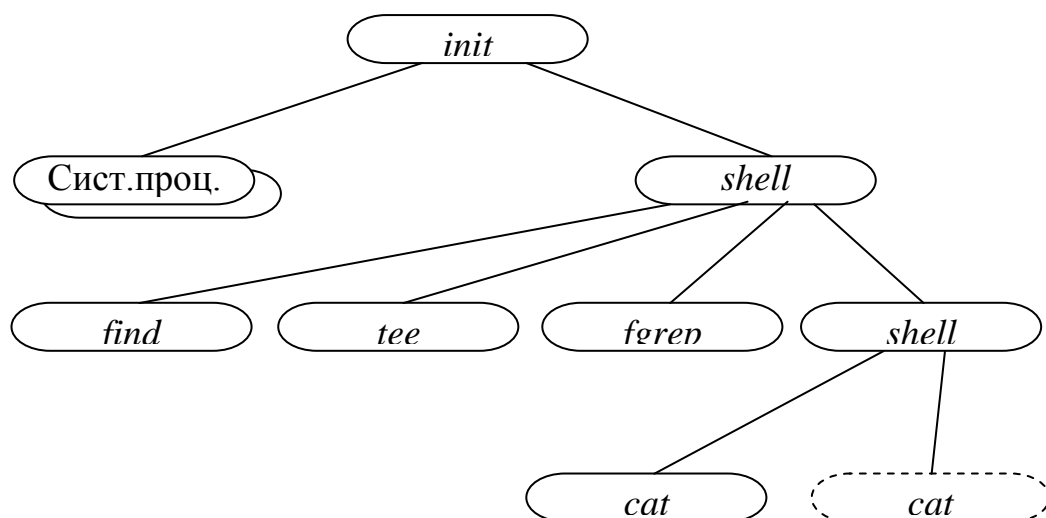


Рис.11. Пример дерева процессов

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

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

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

В выдаче команды:

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

Некоторые флаги этой команды:

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

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

белой, 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. Неизменяемую область памяти программы будем называть *сегментом кода*, а изменяемую – *сегментом данных*. При этом сегмент кода может содержать не только команды (код) программы, но и ее неизменяемые данные (константы). При совместном использовании несколькими процессами одной и той же программы каждый из них использует единственный общий экземпляр сегмента кода, но 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. 1.4). Так как подобное связывание программы производится до начала ее выполнения, то оно называется *статическим связыванием*. При этом предполагается, что если несколько программ вызывают одну и ту же подпрограмму (процедуру), то каждая из них будет обладать своим отдельным экземпляром этой процедуры. Следствием такого «тиражирования» являются повышенные затраты памяти. Использование реентерабельных подпрограмм позволяет эти затраты существенно сократить.

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

После того, как процесс создан, он может вступать с другими процессами в управляющие и информационные взаимодействия. Примера-

ми управляющих взаимодействий являются операции создания и уничтожения процессов. Примером информационного взаимодействия является обмен информацией между процессами, образующими конвейер.

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

2.3. Ресурсы

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

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

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

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

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.3.3).



Рис.12. 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.3.3).

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

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

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

- 1) распределение повторно используемых ресурсов между процессами, учитывающее: а) свойства распределяемого ресурса; б) потребности тех процессов, которым ресурс распределяется;
- 2) синхронизация параллельных процессов, совместно использующих информационные ресурсы, потребляемые или повторно используемые. Методы решения этой задачи рассматриваются в п.2.4.3, 2.4.4 и 2.5;
- 3) оказание помощи процессам при совместном использовании ими повторно используемых ресурсов, по устранению такого неприятного явления как тупики;
- 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

2.4. Синхронизация параллельных процессов

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

- 1) обработка программой процесса некоторых аппаратных прерываний;
- 2) терминальное управление процессами;
- 3) синхронизация параллельных процессов, выполняющих действия с общей областью ОП;
- 4) синхронизация параллельных процессов, выполняющих информационный обмен, используя общую область ОП.

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

2.4.1. Синхронизация с помощью сигналов

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

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

Вспомним, что в однопрограммной системе (например, в *MS-DOS*) возникающие в системе прерывания обрабатываются или подпрограммами ядра или подпрограммами прикладной программы. При этом прикладная программа может перехватывать и обрабатывать любые прерывания. Подобный подход совершенно неприемлем для мультипрограммной системы, в которой обработка прерываний должна находиться под управлением ОС. С другой стороны, полное отстранение процессов от обработки прерываний делает их выполнение негибким, лишая их многих возможностей. Следствием этого является предоставление процессам возможности участвовать в обработке некоторых типов прерываний. При этом процесс может обрабатывать не сам аппаратный сигнал прерывания, а его преобразованную форму – сигнал, выдаваемый обработчиком прерываний.

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

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;

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

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;

SIGTRAP – сигнал о возникновении исключения «трассировка». Такой сигнал используется отладчиками программ;

SIGPWR – сигнал угрозы потери питания;

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:

SIGKILL – сигнал уничтожения процесса. Единственно возможная реакция процесса на этот сигнал – немедленное завершение;

SIGSTOP – сигнал останова процесса. Единственно возможная реакция процесса на этот сигнал – переход в состояние «Останов»;

SIGCONT – продолжение работы остановленного процесса. Если процесс не был ранее остановлен, то он игнорирует данный сигнал;

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**;

SIGCHLD – сигнал, посылаемый процессу-отцу при останове или при завершении дочернего процесса. Источник сигнала – дочерний процесс;

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

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

При записи системных вызовов нами будет использоваться следующее правило. Во-первых, с целью пояснения смысла вызова его имя будет записываться прописными русскими буквами курсивом. Во-вторых, параметры вызова перечисляются через запятую в круглых

скобках после имени вызова. В-третьих, входные параметры вызова отделяются от его выходных параметров символами “||”. Причем входные параметры располагаются слева, а выходные – справа от этих символов. В-четвертых, справа от системного вызова в круглых скобках помещается аналогичный системный вызов UNIX, записанный на языке СИ.

Системный вызов для отправки сигнала:

ПОСЛАТЬ_СИГНАЛ ($i_p, S ||$) (на СИ – *kill*)

где i_p – номер процесса, которому посылается сигнал;

S – имя сигнала.

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

2.4.2. Терминальное управление процессами

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

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

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

Отношение процессов сеанса к управляющему терминалу неодинаково. Среди них есть процессы-изгои, выполняемые в фоновом режиме. (Напомним, что в фоновом режиме программа процесса не выпол-

няет операций ввода-вывода с терминалом. Для запуска из командной строки процесса или конвейера процессов в фоновом режиме, необходимо в конце командной строки набрать символ «&».)

При этом процессы, запущенные из одной командной строки, обычно информационно связаны друг с другом. Так как пользователю удобно выполнять с терминала совместное управление этими процессами, то процесс-*shell* объявляет свои совместно запускаемые дочерние программные процессы единой **группой процессов**. Каждая группа процессов имеет уникальный для всей системы **номер группы процессов**, в качестве которого *shell* назначает номер одного из процессов-членов группы. Такой процесс называется **лидером группы процессов**. Для создания новой группы, или для включения процесса в уже существующую группу, *shell* обращается к ядру, используя системный вызов:

УСТАНОВИТЬ_ГРУППУ_ПРОЦЕССОВ($i_g, i_p //$), (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.

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 не досягаем 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 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:

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

5) *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. 3.1, а пока лишь заметим, что они используются для доступа к системе удаленных пользователей.

Лидер нового сеанса вообще не открывает управляющий терминал в том случае, если он хочет оградить себя и своих потомков от воздействия сигналов с управляющего терминала. Именно с этой целью в дан-

ном 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. 2.2). Кроме того, для задания номера процесса иногда оказывается полезной внутренняя переменная *shell* с именем “!”. Эта переменная содержит номер того процесса, который был запущен последним в фоновом режиме. Как и другие внутренние переменные *shell*, значение переменной “!” задается самим *shell* и поэтому ни в командных строках, ни в скриптах имя данной переменной никогда не встречается без предварительной записи символа “\$”, наличие которого означает «значение переменной».

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

\$kill -SIGKILL \$!

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

2.4.3. Синхронизация конкурирующих процессов

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

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

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

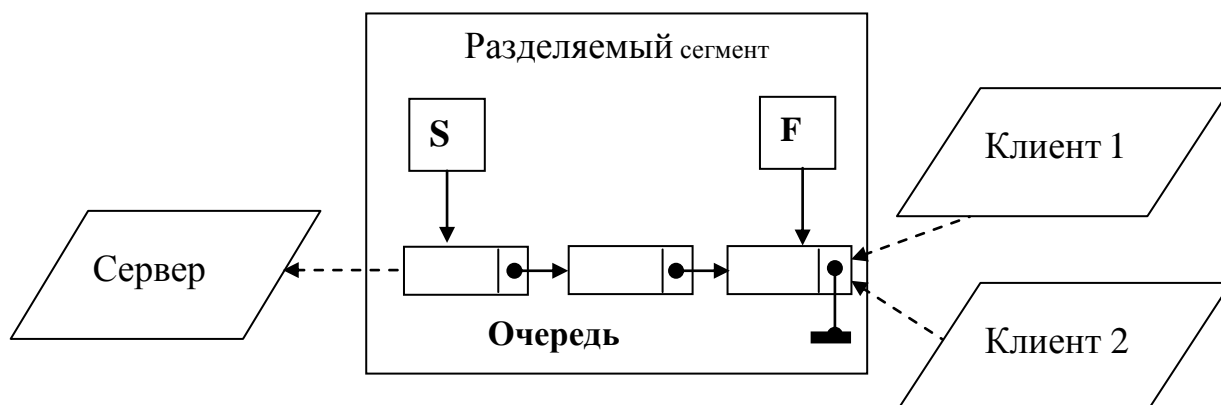


Рис. 13. Три процесса разделяют сегмент памяти

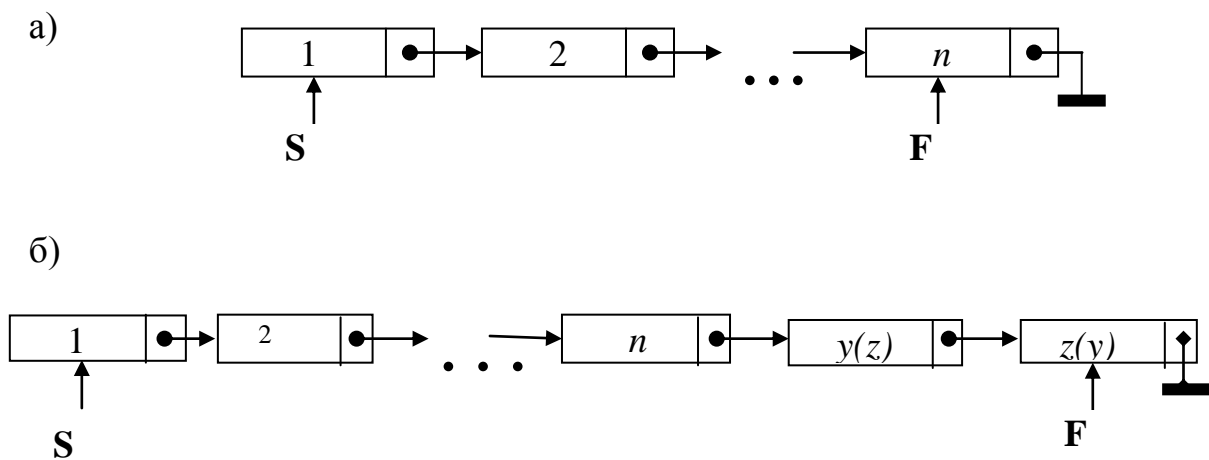


Рис.14. Состояния очереди: а) – исходное;
 б) – правильное включение элементов y и z

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. Покажем, что конкурирующие процессы обязательно должны быть синхронизированы.

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

ветствующие изменения очереди приведены на рис.15. В результате единая очередь оказалась разорванной на две несвязанные части, что не допустимо. Рассмотрим, как это можно избежать.

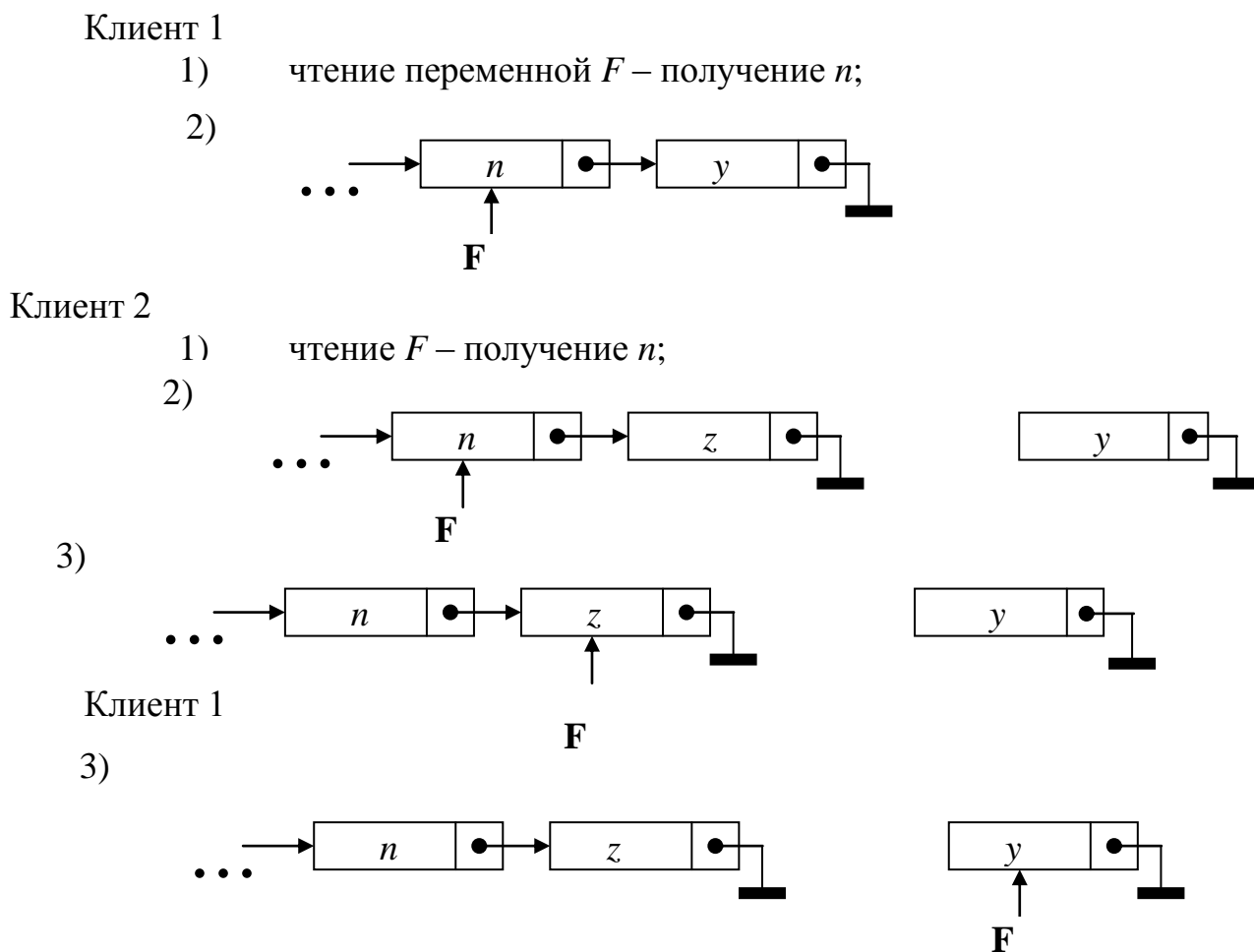


Рис.15. Возможное преобразование очереди при отсутствии синхронизации

Пусть два процесса p_1 и p_2 конкурируют из-за переменной x . При выполнении действия конкурирующего процесса с переменной, как правило, выполняется не один, а целая группа операторов соответствующей программы. Каждая такая группа называется критической секцией. В принципе никаких ошибочных взаимодействий между процессами не будет, если им запретить 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 дожидается, и время ЦП будет потрачено зря. Другой недостаток: проверку установки флага и переход в случае успеха должна выполнять одна неделимая машинная команда. Если это не так, то возможна ситуация, когда после выполнения команды проверки флага процесс будет прерван своим конкурентом, что может привести к непредсказуемому результату.

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

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

- 1) $v(S)$ – переменная S увеличивается на 1;
- 2) $p(S)$ – переменная S уменьшается на 1, если это возможно. Если $S=0$, то ее уменьшить нельзя и процесс, содержащий $p(S)$, будет ждать (в состоянии «Сон») до тех пор, пока S не станет >0 , и операция $p(S)$ не станет возможной.

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

```

p(S):      ЕСЛИ      S>0
           ТО        S ← S -1;
           продолжение выполнения процесса
           ИНАЧЕ     блокирование процесса по S
                   вызов диспетчера ЦП

```

```

v(S):      ЕСЛИ      (очередь процессов, ожидающих S, непустая)
           ТО        деблокирование процесса, заблокированного по S
           ИНАЧЕ     S ← 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) в программе каждого процесса выделить критические секции, каждая из которых выполняет некоторое действие над x ;

- 2) задать начальное значение двоичного семафора S , равное 1;
- 3) перед каждой критической секцией записать оператор $p(S)$, а после нее – $v(S)$.

2.4.4. Синхронизация кооперирующихся процессов

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

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

Допустим, что несколько процессов-«писателей» выполняют запись элементов данных в буфер, а несколько других процессов-«читателей» выполняют чтение элементов данных из этого же буфера. Пусть емкость буфера составляет n элементов. Тогда для синхронизации их деятельности достаточно ввести два семафора – $S1$ 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
.....

2.5. Информационные взаимодействия между процессами

2.5.1. Разделяемая память

Напомним, что использование несколькими процессами разделяемой памяти обеспечивает наиболее быстрое информационное взаимодействие между этими процессами. Реализация такого эффективного средства обеспечивается, в основном, используемой аппаратурой управления памятью. Сейчас нашей задачей является рассмотрение той совокупности системных вызовов, которая позволит программе процесса использовать разделяемую память.

Любой процесс, желающий работать с разделяемым сегментом памяти, должен выполнить системный вызов:

СОЗДАТЬ_РАЗДЕЛЯЕМЫЙ_СЕГМЕНТ ($K, L, F // i$) (на СИ – *shmget*)

где:

K – ключ разделяемого сегмента;

L – длина разделяемого сегмента в байтах;

F – флаги, задающие права доступа к разделяемому сегменту (аналогично файлам);

i – номер разделяемого сегмента. Он используется как программное, так и системное имя разделяемого сегмента и обладает свойством системной уникальности.

Ключ – численное имя объекта (в данном случае – разделяемого сегмента), используемое программистом в качестве первоначального программного имени сегмента. Ключ обладает свойством системной

уникальности – среди всех разделяемых сегментов в данной системе не должно существовать двух сегментов с одинаковым ключом. Кроме того ключ должен быть известен всем процессам, работающим с данным разделяемым сегментом. Получение ключа, удовлетворяющего таким требованиям, представляет некоторую проблему. Аналогичная проблема возникает при использовании семафоров, а также очередей сообщений.

Проблема выбора ключа решается путем вызова процессом, создающим соответствующий объект (разделяемый сегмент, семафор или очередь) библиотечной подпрограммы, выполняющей генерацию ключа. Эта подпрограмма получает в качестве одного из своих параметров имя какого-то существующего файла, а в качестве другого параметра – какой-то символ. Действие данной подпрограммы основано на том, что каждый файл имеет уникальное системное имя (номер). На основе данного номера подпрограмма генерирует уникальный номер, корректируемый с помощью заданного символа. Эту же самую подпрограмму вызывает любой процесс прежде, чем выполнять операции с данным разделяемым сегментом. Естественно, что файл, имя которого используется для определения ключа разделяемого сегмента (семафора или очереди), должен существовать на протяжении всего времени существования этого сегмента (семафора или очереди). Так как иначе процесс, начинающий работу с разделяемым сегментом, после уничтожения (или переименования) файла, этот сегмент не найдет.

При выполнении этого и других системных вызовов по работе с разделяемой памятью ядро использует *таблицу разделяемой памяти*, которая содержит дескрипторы всех разделяемых сегментов системы. Каждый дескриптор содержит поле ключа, поле прав доступа, а также счетчик числа процессов, работающих в данный момент с разделяемым сегментом. Индекс дескриптора в таблице и выдается процессам в качестве номера разделяемого сегмента.

Выполняя системный вызов *СОЗДАТЬ_РАЗДЕЛЯЕМЫЙ_СЕГМЕНТ*, ядро начинает работу с того, что ищет в своей таблице разделяемой памяти строку с заданным полем ключа. Если ядро обнаружит, что дескриптор сегмента с заданным ключом в таблице разделяемой памяти отсутствует, то оно ищет в ОП сегмент заданной длины и создает для него дескриптор в таблице разделяемой памяти. Для заполнения полей этого дескриптора используются входные параметры системного вызова. Индекс разделяемого сегмента возвращается в прикладную часть процесса в качестве программного имени сегмента.

Если при выполнении данного системного вызова ядро обнаружит, что в таблице разделяемой памяти уже существует дескриптор с заданным полем ключа, то выполняется проверка запрашиваемых прав доступа. В случае успешного ее завершения в процесс возвращается номер разделяемого сегмента.

Получив номер разделяемого сегмента процесс должен выполнить его отображение на свою линейную виртуальную память (ЛВП). Для такого отображения используется системный вызов:

ПРИСОЕДИНИТЬ_РАЗДЕЛЯЕМЫЙ_СЕГМЕНТ (*i* // *A*) (на СИ – *shmat*)

где:

i – номер разделяемого сегмента;

A – начальный адрес размещения разделяемого сегмента в ЛВП процесса.

После успешного завершения данного системного вызова процесс может работать с разделяемым сегментом так же, как с обычной областью памяти, не используя никаких дополнительных системных вызовов. Интересно отметить, что процесс может отображать разделяемый сегмент на ЛВП не один, а любое число раз. В результате разные фрагменты программы могут работать с одной и той же областью реальной памяти, используя различающиеся между собой структуры данных.

Закончив работу с разделяемой памятью, процесс отсоединяет ее с помощью системного вызова:

ОТСОЕДИНИТЬ_РАЗДЕЛЯЕМЫЙ_СЕГМЕНТ (*A* //) (на СИ – *shmdt*)

где:

A – начальный адрес размещения разделяемого сегмента в ЛВП процесса.

При выполнении данного вызова ядро уменьшает на единицу счетчик процессов в дескрипторе разделяемого сегмента. Если значение счетчика становится нулевым, то дескриптор разделяемого сегмента уничтожается, а область ОП освобождается.

Соблюдение прав доступа к разделяемому сегменту явно недостаточно для обеспечения совместной работы нескольких процессов с одним и тем же сегментом памяти. Как было показано в п.2.4.3, для этого требуется выполнить синхронизацию процессов, совместно использующих разделяемый сегмент, с помощью семафоров Дейкстры. В отличие от ранее рассмотренных системных вызовов *p(S)* и *v(S)*, используемые для работы с семафорами в системе UNIX системные вызовы весьма громоздки и нами рассматриваться не будут.

Кроме разделяемой памяти, требующей дополнительной синхронизации, ядро UNIX предоставляет другие средства межпроцессного информационного обмена, не требующие такой синхронизации. Такие

средства предполагают создание между процессами информационных каналов.

2.5.2. Понятие информационного канала

Разделяемая память является наиболее быстрым средством меж-процессного информационного взаимодействия, так как при ее применении, во-первых, не требуется применение ВП, а во-вторых, не требуются какие-либо переносы данных внутри ОП. Но возникающие при ее использовании проблемы синхронизации делают необходимым применение семафоров Дейкстры, что существенно ограничивает возможности применения данного средства информационного взаимодействия по следующим причинам: во-первых, программы всех процессов, разделяющих какие-то области ОП, должны содержать правильно записанные системные вызовы $p(S)$ и $v(S)$. При этом пропуск хотя бы одного такого вызова или нарушение порядка их записи приведет к ошибкам синхронизации. Во-вторых, каждый взаимодействующий процесс должен сам знать, где находится разделяемая переменная, так как используемый метод синхронизации об этом ничего не говорит. Подобным требованиям могут отвечать только процессы, находящиеся в «дружественных отношениях». Такие отношения обычно имеются у процессов, совместно выполняющих единую параллельную программу. Другие параллельные процессы не находятся в таких отношениях ни между собой, ни по отношению к разделяемой переменной. В этом случае все операции по синхронизации доступа к разделяемой переменной должны быть выведены из прикладной части процесса.

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

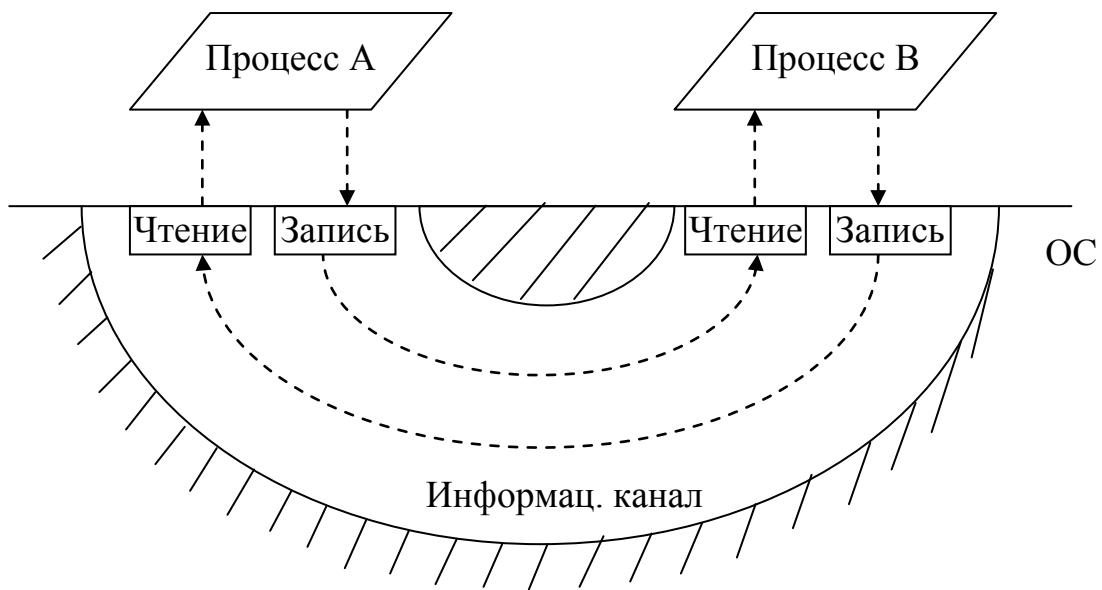


Рис.16. Взаимодействие процессов через информационный канал

Порция информации, передаваемая или принимаемая процессом за одно обращение к информационному каналу, называется *сообщением*. Любой информационный канал позволяет подключенным к нему процессам использовать для приема и передачи сообщений два системных вызова:

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

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

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

2) *структурированность передаваемых сообщений*. Будем различать информационные каналы, передающие неструктурированные сообщения и передающие пакеты. **Неструктурированное сообщение** – сообщение, в состав которого не входят какие-то особые байты, используемые информационным каналом. **Пакет** – последовательность байтов, предваряемая специальными служебными байтами, используемыми информационным каналом. Заметим, что данная характеристика от-

носится лишь к интерфейсу между процессом и информационным каналом, так как внутри канала поступающие от процесса неструктурированные сообщения обычно преобразуются в пакеты. Но для процесса подобное преобразование «не заметно»;

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

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

2.5.3. Обыкновенные программные каналы

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

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

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

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

Обыкновенный канал создается в результате системного вызова:

СОЗДАТЬ_КАНАЛ ($\| i_1, i_2$), (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 неструктурированных сообщений.

Операция чтения из канала может завершиться одним из следующих вариантов:

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

2) при чтении большего числа байтов, чем находится в канале, считывается имеющееся число байтов. Процесс сам должен 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 не родственных процессов номера открытых файлов не могут использоваться в качестве идентификаторов совместно используемых каналов. Другим существенным недостатком обыкновенных каналов является то, что они существуют недолго, лишь во время существования использующих их процессов.

Несмотря на перечисленные недостатки, обыкновенные каналы относятся к самым используемым средствам информационного взаимодействия между процессами. В частности такими каналами соединяются все процессы, команды запуска которых образуют конвейер, набираемый пользователем в командной строке *shell* (см. п. 1.5.3.). В данном случае каналы создает *shell*, дочерними процессами которого и являются процессы, образующие конвейер.

2.5.4. Именованные программные каналы

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

СОЗДАТЬ_ИМЕННОЙ_КАНАЛ (*I, m*||), (на СИ – *mkfifo*)

где *I* – символьное имя файла-канала,

m – права доступа к каналу (рассматриваются в п.3.2).

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

ОТКРЫТЬ_ФАЙЛ(*I, r*|| *i*), (на СИ – *open*)

где *r* – режим работы процесса с файлом-каналом (чтение и (или) запись);

i – программное имя (номер) файла, уникальное для данного процесса.

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

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

2.5.5. Очереди сообщений

В п. 2.5.2 было дано определение структурированного сообщения – пакета, согласно которому он представляет собой последовательность байтов, предваряемую специальными служебными байтами. Таким образом, в отличие от неструктурированного потока данных пакет представляет собой некоторую структуру данных, предназначенную для обмена между процессами. В нашем случае сообщение-пакет имеет три поля: 1) *тип сообщения* – целое положительное число; 2) длина сообщения в байтах (может быть нулевой); 3) собственно данные – любая последовательность байтов (необязательно символы). Первые поля сообщения, содержащие служебную информацию, образуют *заголовок сообщения*.

Сообщение формируется процессом и записывается им с помощью соответствующего системного вызова в *очередь сообщений*. Прежде, чем выполнять операции обмена с очередью сообщений, эта очередь должна быть создана с помощью системного вызова:

СОЗДАТЬ_ОЧЕРЕДЬ_СООБЩЕНИЙ (*K, F // i*) (на СИ – *msgget*)

где:

K – ключ очереди сообщений. Данный ключ используется в качестве первоначального программного имени очереди;

F – флаги, задающие права доступа к разделяемому сегменту (аналогично файлам);

i – номер очереди сообщений. Он используется как программное, так и системное имя очереди сообщений и обладает свойством системной уникальности.

Если очередь сообщений, которой соответствует заданный ключ, уже существует, то в результате выполнения данного системного вызова в программу возвратится номер этой очереди. Так как этот номер ис-

пользуется далее всеми процессами, работающими с очередью, в качестве ее программного имени, то, следовательно, каждый из этих процессов обязан выполнить системный вызов *СОЗДАТЬ_ОЧЕРЕДЬ_СООБЩЕНИЙ*.

При создании очереди сообщений для нее в пространстве памяти ядра создается блок управления, имеющий следующие поля: 1) права доступа; 2) длина очереди в байтах; 3) число сообщений в очереди; 4) указатель на первый элемент; 5) указатель на последний элемент. Теперь с очередью могут выполняться операции чтения и записи подобно тому, как эти операции выполняются с файлами. Подобно файлу, с каждой очередью связаны права доступа, регламентирующие доступ к очереди со стороны пользователя-владельца, со стороны пользователя-группы, а также со стороны других пользователей. Благодаря этим правам доступа процесс, создавший очередь, может, например, запретить другим процессам выполнять запись сообщений в очередь, а разрешить лишь чтение этих сообщений.

Как и блок управления очередью сообщений, сама эта очередь также размещается в пространстве памяти ядра. Это обеспечивает защиту очереди от несанкционированных операций со стороны процессов в состоянии «Задача». Логическому размещению очереди в пространстве ядра соответствует ее физическое размещение в ОП и в области свопинга ядра на магнитном диске. Какая часть очереди находится в ОП, а какая на диске, зависит от длины очереди, а также от текущего наличия аппаратных ресурсов в системе. В любом случае логическая организация очереди представляет собой связанный список, доступ к которому осуществляется через блок управления.

Процесс, выполняющий запись сообщения в очередь, обращается к ядру с помощью системного вызова:

ЗАПИСЬ_СООБЩЕНИЯ_ОЧЕРЕДЬ (*i,A,L//*) (на СИ – *msgsnd*)

где:

i – номер очереди сообщений;

A – начальный адрес области памяти, в которой находится подготовленное сообщение;

L – длина сообщения в байтах.

Процесс, выполняющий чтение сообщения из очереди, обращается к ядру с помощью системного вызова:

ЧТЕНИЕ_СООБЩЕНИЯ_ОЧЕРЕДЬ (*i,A,L,t//*) (на СИ – *msgrcv*)

где:

i – номер очереди сообщений;

A – начальный адрес области памяти, в которую следует считать сообщение из очереди;

L – длина сообщения в байтах;

t – тип сообщения.

Обратим внимание, что процесс, выполняющий считывание записи из очереди сообщений, указывает тип сообщения. При выполнении данного системного вызова ядро ищет сообщение заданного типа в очереди и выдает его процессу. Поэтому фактически, очередь сообщений является не обыкновенной, а приоритетной очередью. Операция чтения из такой очереди выдает то сообщение, которое поступило в очередь первым среди сообщений заданного типа. Наличие данного свойства приводит к тому, что информационный межпроцессный канал на базе очереди сообщений является мультиплексным (см. п.2.5.2). Для мультиплексного канала характерно то, что в нем одновременно могут находиться данные нескольких типов. Причем передача данных каждого типа производится по информационному каналу логически параллельно с передачей данных других типов. В результате мультиплексный канал может рассматриваться как объединение нескольких логически параллельных моноканалов.

Пример. В п.2.4.3 нами был рассмотрен пример, в котором процесс-сервер выполнял запросы процессов-клиентов по распечатке текстовых файлов на принтере. При этом в качестве средства информационного взаимодействия между сервером и клиентами использовался разделяемый сегмент ОП. Рассмотрим реализацию этой же задачи с использованием очереди сообщений. Причем внесем в условие задачи следующее добавление: выполнив задание клиента, сервер посылает ему «отчет о проделанной работе».

Присвоим всем сообщениям клиентов тип 1, а каждому ответному сообщению сервера – тип, совпадающий с номером того процесса-клиента, которому это сообщение предназначено. Для того, чтобы сервер мог определить номер процесса-клиента, сообщение клиента должно содержать этот номер. Возможное содержимое очереди сообщений показано на рис.17.

В заключение сравним очередь сообщений с почтовым ящиком, в который процессы-писатели кладут письма-сообщения (при наличии соответствующих прав доступа), а процессы-читатели забирают эти сообщения. Причем множества писателей и читателей могут пересекаться или даже совпадать.

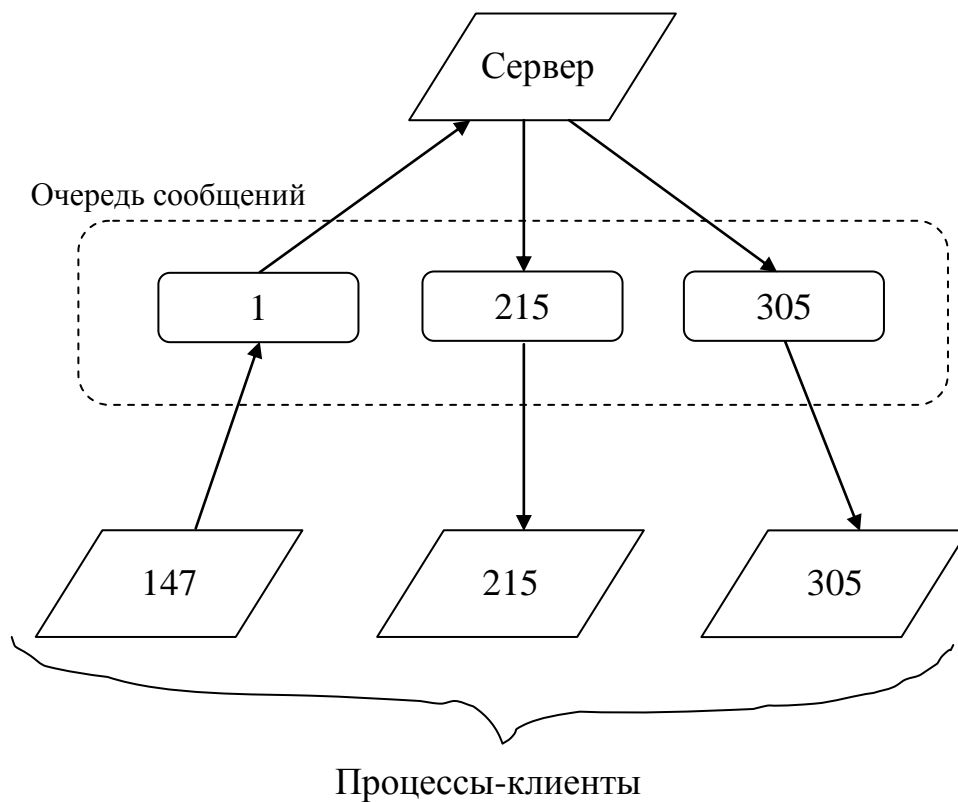


Рис. 17. Взаимодействие клиентов и сервера через очередь сообщений

3. ПОДДЕРЖКА МНОГОПОЛЬЗОВАТЕЛЬСКОЙ РАБОТЫ И СТРУКТУРА СИСТЕМЫ

3.1. Управление доступом пользователя в систему

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

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

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

Все пользователи системы делятся ее администратором на пересекающиеся группы пользователей (не путать с группами процессов из п.2.4.2). За счет включения пользователя в группу он наделяется дополнительными правами, которыми обладают все члены группы. Каждая группа имеет два имени, уникальных по всей системе – символьное имя группы и номер группы. Например, административная группа может иметь имена: root и 0. Имеющееся здесь совпадение имен группы и пользователя вполне допустимо, так как каждое из имен должно быть уникально только в одной группе объектов (пользователи или группы).

Каждый конкретный пользователь может быть одновременно членом одной, двух или большего числа групп. Группа пользователей, первая для данного пользователя, является для него первичной группой.

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

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

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

- символьное имя пользователя;

- 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 “!”;

- номер пользователя;

- номер первичной группы пользователя;

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

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

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

Таким образом, программа одного и того же процесса заменяется дважды на другую программу: `getty` → `login` → `shell`. Это не трудно обеспечить, так как любая программа может перекрыть себя другой программой, используя соответствующий системный вызов (рассматривается в п.4.2). Кроме того, отметим, что `login` сравнивает не сами пароли, а их закодированные варианты. В ходе своей работы в системе пользователь может заменить пароль, используя утилиту `passwd`.

Рассмотрим теперь, как осуществляет доступ в систему удаленный пользователь, соединенный с UNIX-системой не с помощью локального терминала, а с помощью линии связи или, даже, целой сети передачи данных. Для обеспечения такого доступа изложенная выше схема должна быть скорректирована (рис.18).

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, порождаемый 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 заместителя терминала – псевдотерминал и порождает дочерний процесс login.

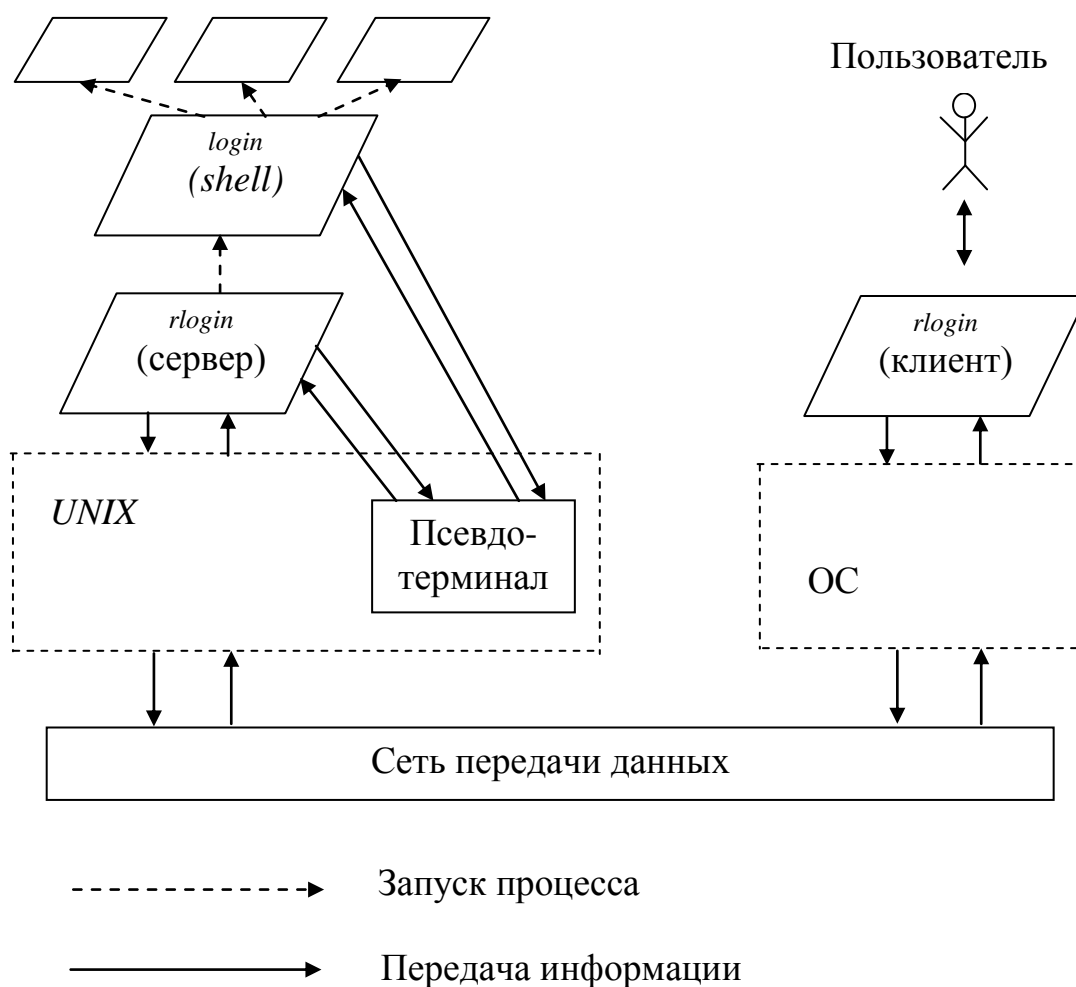


рис.18. Доступ удаленного пользователя в UNIX-систему

Псевдотерминал – программный модуль ОС, предназначенный для передачи информации между двумя процессами, а также для некоторого преобразования этой информации. Это преобразование нужно

для того, чтобы один из процессов, соединяемых псевдотерминалом, выполнялся бы так, как будто он взаимодействует с настоящим терминалом. В нашем случае таким “обманываемым” процессом является login. Перед его созданием процесс-отец rlogin (сервер) производит перенаправление стандартных ввода и вывода на псевдотерминал. Такое перенаправление выполняется очень просто: достаточно открыть псевдоустройство на ввод под программным именем 0, а на вывод – под именем 1. Так как login является дочерним процессом rlogin (сервер), то он наследует эти имена псевдоустройства и будет пользоваться ими так, как будто речь идет о настоящем терминале.

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

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

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

3.2. Защита файлов

Каждый пользователь, зарегистрированный в системе, имеет в ней «информационный след» – совокупность информации, связанной с данным пользователем. Этот информационный след схематично показан на рис.19. На этом рисунке пунктиром показана краткосрочная информа-

ция, связанная с выполнением процессов. Долгосрочная информация хранится на устройствах ВП в виде файлов.

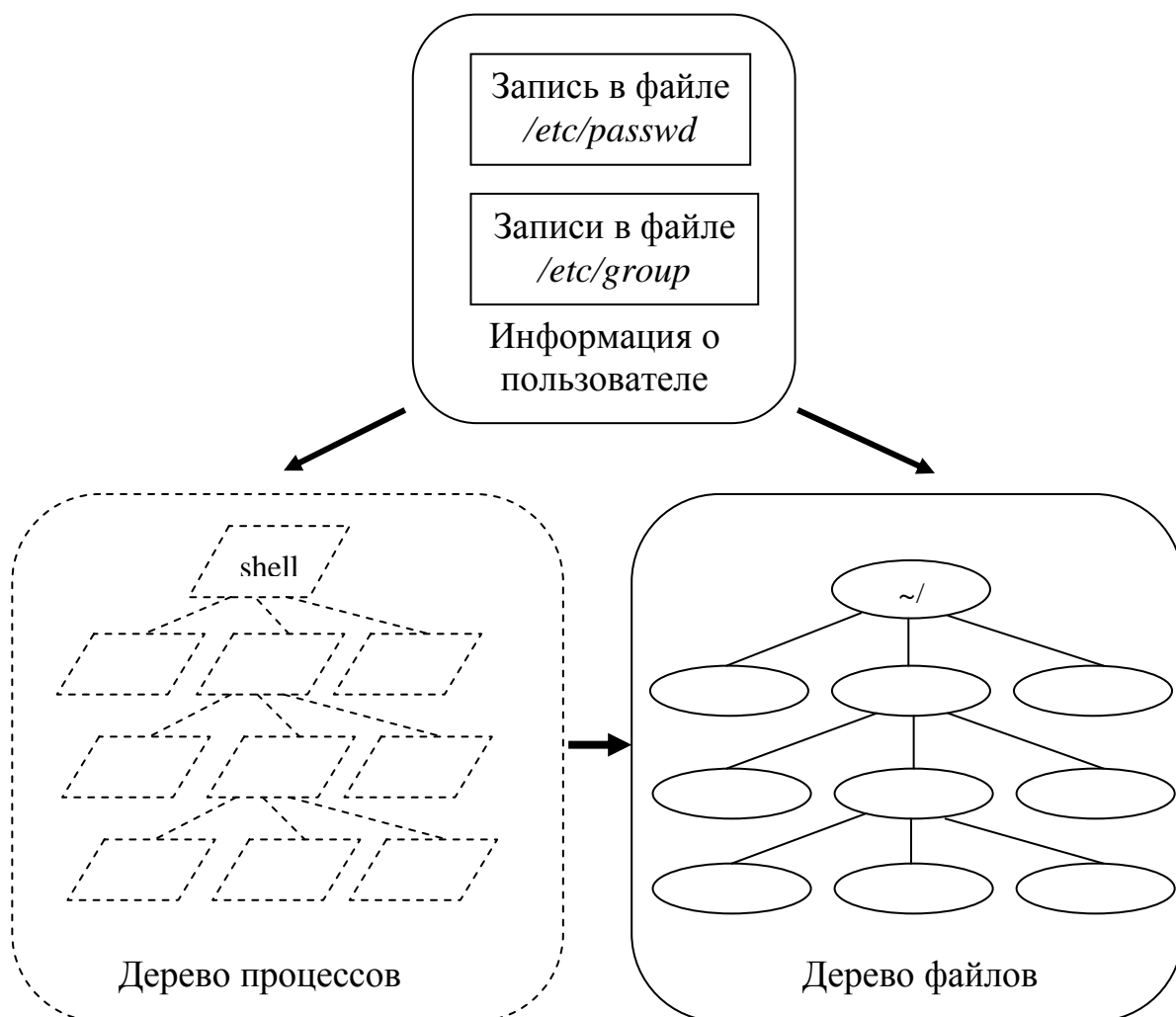


Рис.19. Информационный след пользователя в системе

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

файла */etc/passwd*, соответствующей пользователю, переписывается имя его первичной группы.

Пользователь-владелец и пользователь-группа могут быть изменены с помощью команд *shell* – *chown* и *chgrp*. В качестве первого параметра этих команд 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 или исполняемым файлом программы и может быть выполнен после загрузки в ОП как машинная программа, или файл является командным (скриптом).

Например, суперпользователь имеет по отношению к любому файлу неограниченные права доступа – *rwX*.

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

Для получения дополнительной информации о дочерних файлах каталога, необходимо иметь к нему форму доступа *x* (выполнение). Эта же форма необходима для того, чтобы сделать каталог текущим (с помощью команды *cd*). Более того, чтобы сделать каталог текущим, необ-

ходимо иметь форму доступа x ко всем каталогам в имени-пути данного каталога. Иначе, в данный каталог мы не попадем.

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

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

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

Информация о файле, выданная *UNIX*, состоит из 8 колонок. В первой позиции первой колонки записан тип файла (d – каталог; «-» – прочий файл). В остальных девяти позициях первой колонки записаны права доступа к файлу: первые три символа – права доступа пользователя-владельца; вторые три символа – права доступа пользователя-группы; последние три символа – права доступа прочих пользователей.

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

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

- 1) число жестких связей – количество каталогов, в которых «зарегистрирован» данный файл;
- 2) имя пользователя-владельца файла;
- 3) имя пользователя-группы;
- 4) длина файла в байтах;

- 5) дата последнего изменения файла;
- 6) время последнего изменения;
- 7) имя файла.

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

```

      | 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 пользовательской группе. В случае принадлежности к группе пользователь процесса наделяется правами группы, иначе – правами доступа к файлу для остальных пользователей.

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

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

может заменить только свой пароль, не влияя на пароли других пользователей;

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

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

4) 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*.

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

3.3. Укрупненная структура операционной системы

На рис.20 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 си-

стеме (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.20. Укрупненная структура ОС

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

После того, как последовательность внутренних процедур ядра завершена, «заглушка», возможно, возвратит управление в прикладную программу. Такой возврат выполняется в том случае, если выполняемый системный вызов является синхронным. При 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. 3.1). Для выполнения распределенных программ требуется помощь со стороны ОС, которая для этого должна быть сетевой. Рассмотренная выше общая структура ОС может быть использована и для реализации сетевой операционной системы.

3.4. Структура сетевой операционной системы

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

что ОС не отличает системные обрабатывающие программы от обычных прикладных программ).

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

Прикладные программы, выполняемые в сети, можно разделить на локальные и распределенные. Распределенные программы разрабатываются специально для выполнения в сети. Примером такой программы является рассмотренная в п.3.1 программа `login`. 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.2.5).

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

программе, существуют два основных подхода к разработке таких программ.

В первом из подходов статика и динамика прикладного протокола распределенной программы реализуются ее разработчиками без использования какой-то особой помощи со стороны системных программ. Единственная дополнительная помощь, которая требуется при выполнении распределенной программы от сетевых ОС – предоставление ей информационного канала, выполняющего передачу удаленных сообщений. Допустим, что такой канал предоставляет модуль ОС – транспорт сообщений (рис.21). Реализация данного модуля не относится к теме данного пособия, и для нас важен лишь тот факт, что модуль транспорта обеспечивает передачу сообщения требуемому удаленному программному процессу.

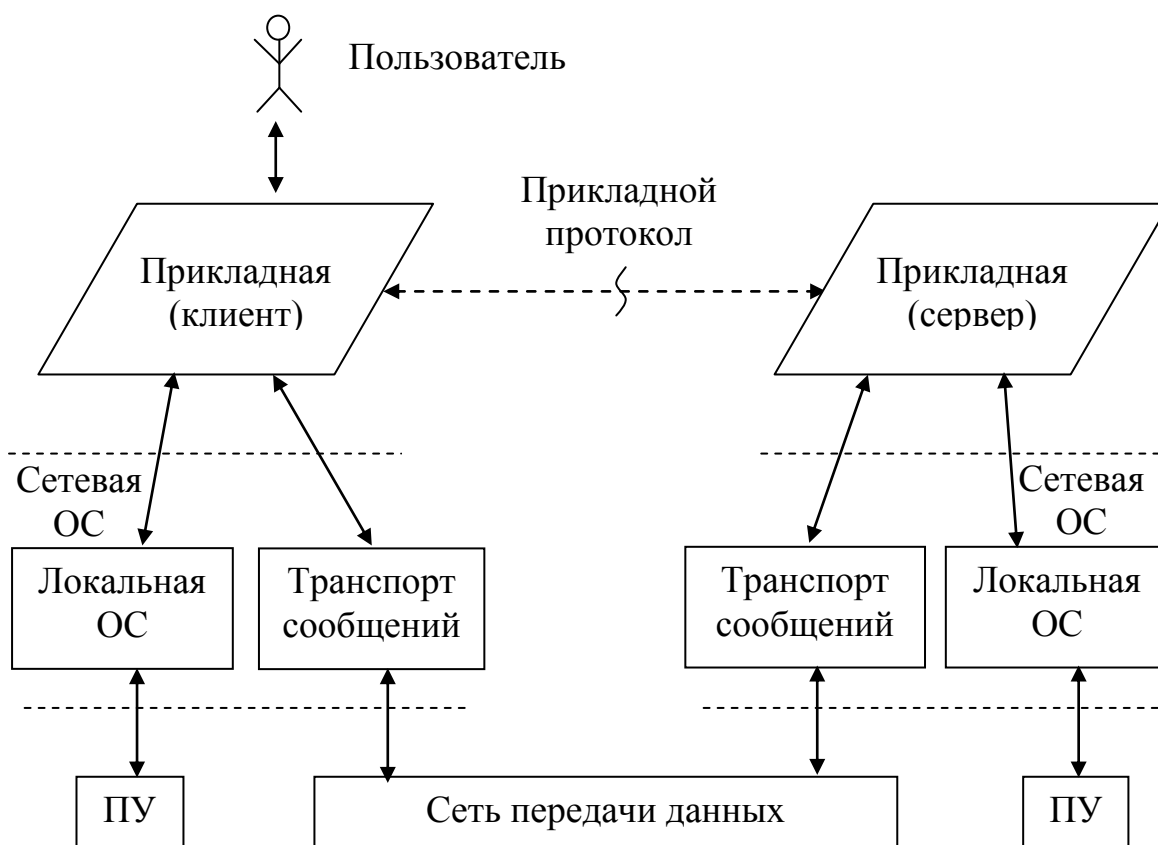


Рис.21. Выполнение распределенной программы в сети

Второй подход к реализации распределенной программы называется удаленным вызовом процедур – RPC (remote procedure call). В этом подходе прикладная программа освобождена от выполнения прикладно-

го 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.22).

При проектировании распределенной программы производится распределение ее подпрограмм между клиентом и сервером. Все процедуры сервера, которые могут потребоваться клиенту, нумеруются, и для каждой из них создаются две процедуры-заглушки, одна из которых подсоединяется к программе-клиенту, а другая – к серверу. Такое связывание выполняется или статически, или динамически. В первом случае заглушка является модулем обычной библиотеки, а во втором – модулем DLL.

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 подсо-

единяются к частям прикладной программы (клиенту и серверу) как модули DLL

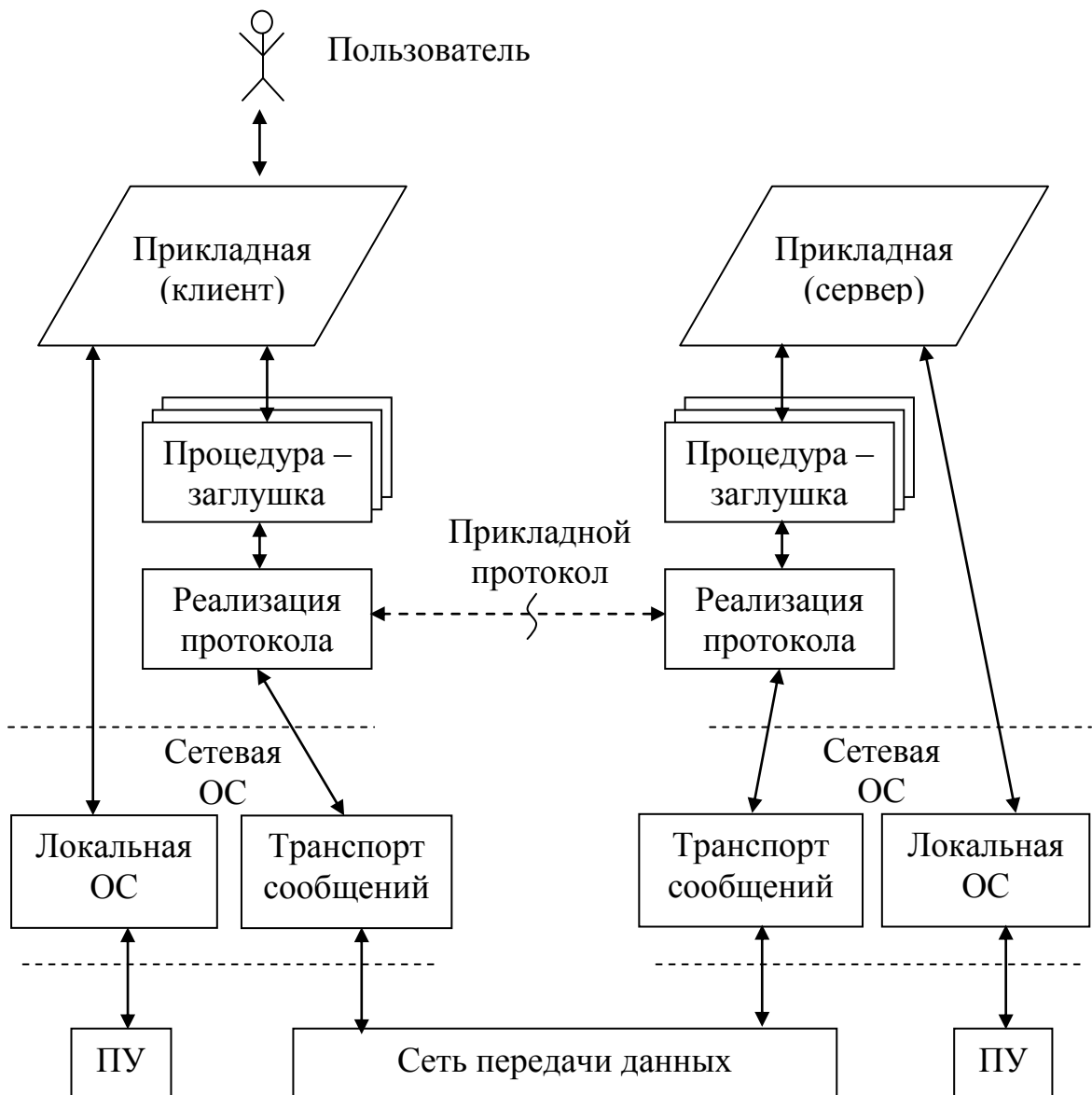


Рис.22. Распределенная программа с удаленным вызовом процедур

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:

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;

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;

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 – то же 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, обеспечивающих сетевое взаимодействие, входят в состав ядра ОС. Так как в данном методе учитываются особенности реализации файловой системы, то он будет рассмотрен позже в п.5.2.

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.23).

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

Реализация редиректора зависит от типа локальной ОС. Например, в среде MS-DOS редиректор может быть реализован в виде резидентной программы, перехватывающей системные вызовы (программные прерывания), предназначенные для инициирования файловой подсистемы.

Если локальной ОС является UNIX, то редиректор может быть реализован как подпрограмма ядра, запускаемая из интерфейса системных вызовов при поступлении запроса на работу с файлами. При этом, как и другие подпрограммы ядра, редиректор подсоединяется к программе процесса, выдавшего системный вызов.

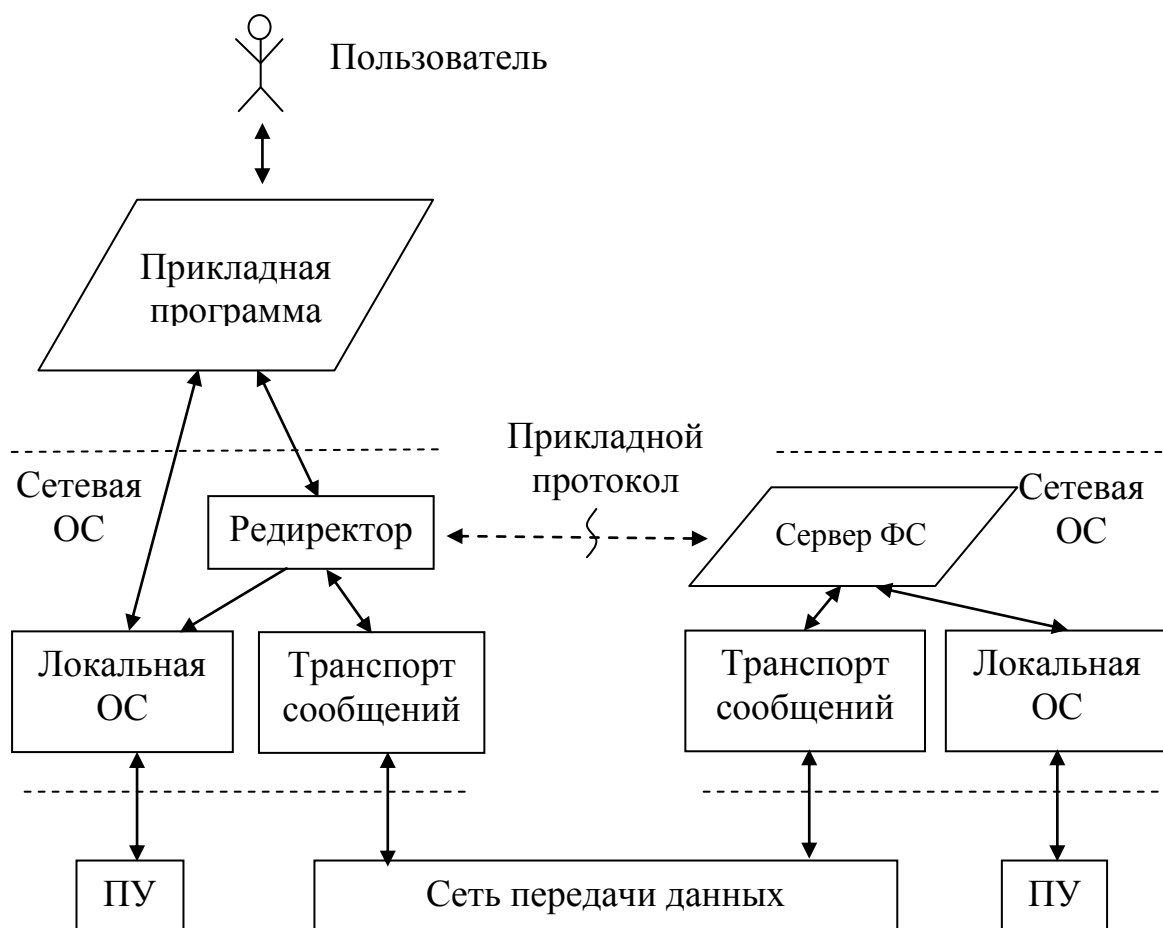


Рис.23. Сетевое обслуживание локальной прикладной программы

Сервер файловой системы – процесс (демон), предназначенный для выполнения удаленных системных вызовов. Этот процесс инициируется при поступлении сообщения от какого то удаленного редиrektора. Так как правомочность операций с файлами зависит от имени пользователя, то это имя является предметом обсуждения между редиrektором 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 редиrektору.

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

В рассмотренной выше схеме предполагалось, что конкретная ЭВМ является или узлом-сервером, или узлом-клиентом, участвующим в реализации одной из трех схем взаимодействия удаленных программных процессов. В действительности, один и тот же узел может выполнять функции и клиента, и сервера, одновременно выполняя как обслуживание «чужих» прикладных программ, так и обращаясь в сеть с целью обслуживания «своих» программ. При этом различные клиенты и серверы в данном узле могут использовать разные схемы сетевого взаимодействия.

4. ПОДСИСТЕМА УПРАВЛЕНИЯ ПРОЦЕССАМИ

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

4.1. Состояния процесса

Являясь важнейшим объектом управления со стороны ОС, процесс может находиться в состояниях, изображенных на рис.24. 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(|| i_p), (на СИ – *fork*)

где i_p – номер нового процесса.

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

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

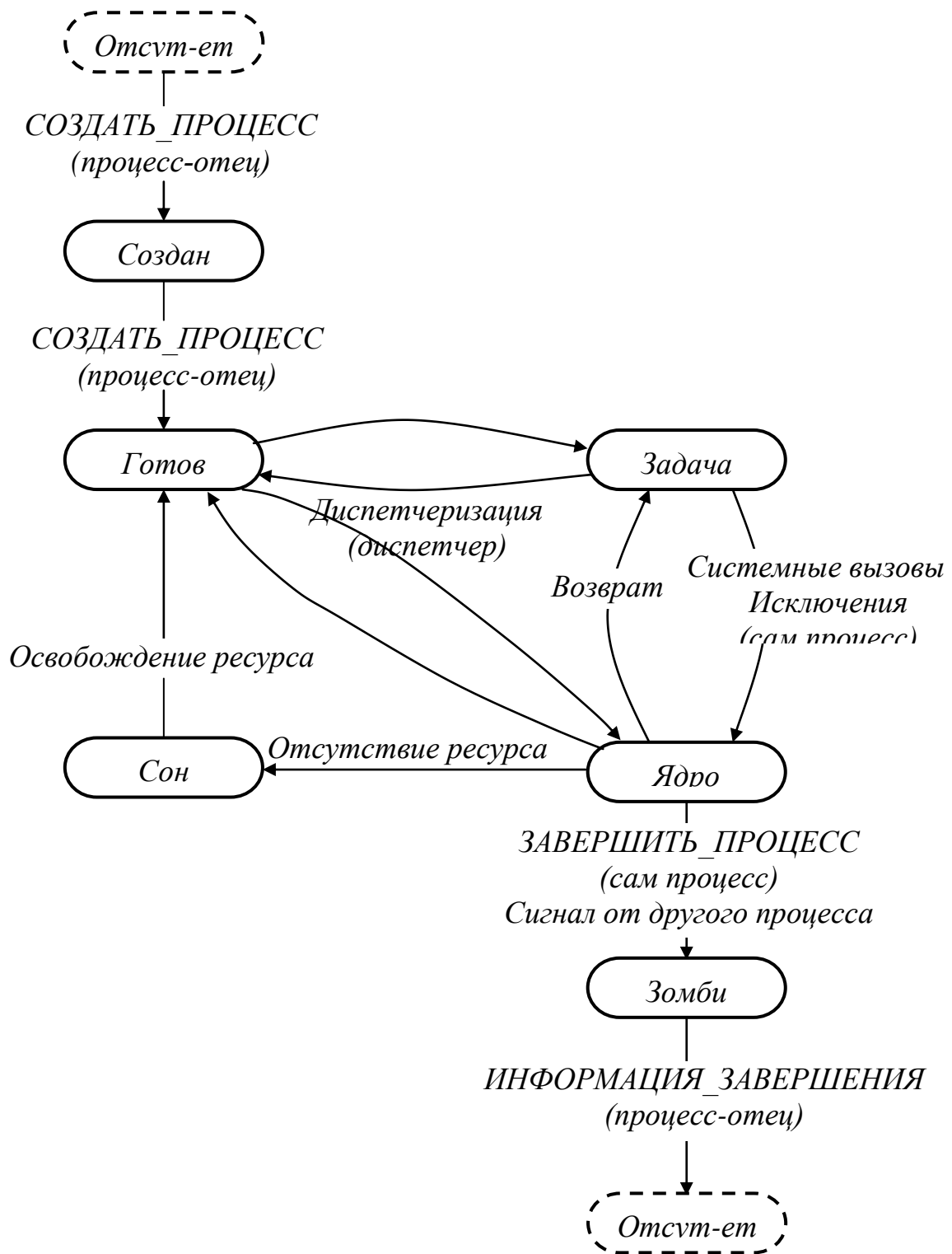


Рис.24. Состояния процесса

Если очередная выполняемая команда программы есть или команда, приведшая к исключению (исключение – внутреннее аппаратное прерывание), или команда системного вызова, то на ЦП начинается исполняемая 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 в последствии 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 не освобождаемый ресурс – одна строка в системной таблице процессов *proc*, содержащая код завершения процесса и информацию о затратах времени ЦП на его выполнение.

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

ИНФОРМАЦИЯ_ЗАВЕРШЕНИЯ (// i_p, k) (на СИ – *wait*)

где i_p – номер завершеного дочернего процесса;

k – код завершения процесса.

Для выполнения этого системного вызова будет инициирована подпрограмма ядра *wait*, которая выдаст номер завершеного 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 $\langle \text{Ctrl} \rangle \& \langle \text{Z} \rangle$, или сигналов *SIGTTIN* и *SIGTTOU*, выдаваемых драйвером клавиатуры. Обратный

переход из состояния «Останов» в состояние «Готов» производится в результате поступления сигнала *SIGCONT*.

4.2. Создание процесса

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

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

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

- 1) указатель на область памяти, содержащую **заголовок исполняемого файла**. Это служебная информация, которая предваряет код и данные прикладной программы, и которая может

использоваться как при создании процесса, так и впоследствии им самим (аналог *PSP* в *MS-DOS*);

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

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

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

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

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

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

Особенностью создания нового процесса в *UNIX* является то, что новый процесс, полученный в результате выполнения ядром системного вызова *СОЗДАТЬ_ПРОЦЕСС*, является почти полной копией своего процесса-отца. При этом он имеет такую же программу, переменные окружения, структуры *proc* и *user*. Основное различие между процессами в том, что новый процесс имеет другое имя (номер). Таким образом, результатом выполнения данного системного вызова является одновременное существование в данный момент двух процессов, выполняющих одну и ту же программу, причем в одной и той же ее точке: указатель команды *EIP* содержит адрес команды, которая следует в программе за командой, реализующей системный вызов *СОЗДАТЬ_ПРОЦЕСС*.

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

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

ЗАГРУЗИТЬ_ПРОГРАММУ (I_F ||), (на СИ – *exec*)

где I_F - имя файла, содержащего запускаяемую 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. п.1.5.6). Если файл не является ни бинарным, ни скриптом, явно задающим *shell*, то будет загружен тот *shell*, который загружается по умолчанию в данной системе.

Сама загрузка программы в отличие, например, от *MS-DOS*, не сводится к записи ее сегментов в ОП. Эти сегменты копируются из исполняемого файла в область ВП, называемую *областью свопинга*. Непосредственно в ОП записывается лишь самая начальная часть сегмента кода программы, с которой начнется выполнение программы в результате передачи управления или из ядра (при завершении системного вызова *ЗАГРУЗИТЬ_ПРОГРАММУ*), или из динамического редактора связей (при использовании в программе *DLL*). Далее загрузка требуемых фрагментов программы из области свопинга будет производиться динамически, то есть во время выполнения программы. В том случае, если программа процесса является реентерабельной, то ее сегмент кода во-

обще 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.

4.3. Обработка сигналов

Вне зависимости от того, кто является источником сигнала (обработчик прерываний или процесс), его выдача сводится к установке одного бита в поле «сигналы» структуры *proc*. (Вспомним, что наряду с *user* это одна из двух основных управляющих структур процесса.) Длина этого поля в битах совпадает с числом типов сигналов, существующих в системе. Так как одному типу сигнала соответствует всего один бит, то до начала обработки сигнала и, следовательно, до сброса бита в поле «сигналы», все последующие сигналы данного типа будут процессом игнорироваться.

Наличие сигнала означает, что он должен быть обработан процессом. В общем случае существуют три варианта обработки сигнала:

- 1) обработка по умолчанию. Она выполняется одной из подпрограмм ядра и для большинства типов сигналов сводится к прекращению существования процесса;
- 2) игнорирование сигнала;
- 3) обработка сигнала собственной подпрограммой процесса.

В любой момент времени для каждого типа сигналов, поступающих в процесс, задан один (и только один) вариант обработки. Для этого используется строка **диспозиция сигналов** в управляющей структуре *user*. Каждое поле данной строки соответствует одному типу сигналов и содержит вариант его обработки. При этом, если для обработки сигнала используется собственная (пользовательская) подпрограмма, то указанное поле содержит ее адрес в ОП.

Сразу же после создания процесса с помощью соответствующего системного вызова, диспозиция сигналов нового процесса точно такая же, как и у процесса-отца (наследование диспозиции). Но если далее процесс выдаст системный вызов **ЗАГРУЗИТЬ_ПРОГРАММУ**, то для всех сигналов устанавливается диспозиция «по умолчанию». Это объясняется тем, что все прежние обработчики сигналов уничтожены (как и все другие прикладные подпрограммы) в результате загрузки. Если та-

кая начальная диспозиция не устраивает, то для ее изменения процесс выполняет системный вызов:

УСТАНОВИТЬ_ОБРАБОТКУ_СИГНАЛА ($I_s, i, A_a//$) (на СИ – *sigaction*)

где I_s – имя сигнала (номер или символьное имя);
 i – вариант обработки сигнала (по умолчанию, игнорирование, свой обработчик сигнала);

A_a – стартовый адрес своего обработчика сигнала.

Воспользовавшись этим системным вызовом, программа процесса может сама задать вариант обработки любого своего входного сигнала, за исключением сигналов *SIGKILL* и *SIGSTOP*, для каждого из которых допустима только обработка по умолчанию (уничтожение процесса).

Установка в единицу бита в поле «сигналы» структуры *proc* не означает, что соответствующий сигнал начнет немедленно обрабатываться. Такая обработка может быть начата ядром в один из следующих моментов:

- 1) непосредственно перед переходом процесса из состояния «Готов» в состояние «Задача»;
- 2) непосредственно перед переходом процесса из состояния «Ядро» в состояние «Задача»;
- 3) непосредственно перед переходом процесса в состояние «Сон»;
- 4) непосредственно перед переходом процесса из состояния «Сон» в состояние «Готов».

В любой из этих моментов времени подпрограмма ядра *issig* проверяет наличие поступивших сигналов, читая соответствующее поле в структуре *proc*. Если сигнал обнаружен, и если он не должен быть проигнорирован, то инициируется или системный обработчик сигнала, или собственная подпрограмма процесса. В последнем случае запуску подпрограммы процесса предшествует его перевод в состояние «Задача».

Если в момент появления сигнала процесс находился в состоянии «Ядро», то никакого воздействия на процесс сигнал не окажет до выхода из этого состояния. С другой стороны, сигнал не может появиться в то время, когда процесс находится в состоянии «Задача», так как источником сигнала в это время может быть только один из обработчиков прерываний. А любой из таких обработчиков прерываний может выполняться только в состоянии «Ядро».

Если запущенный обработчик сигнала не выполняет завершение процесса, то после выполнения этого обработчика выполнение процесса продолжится с той точки, в которой оно было прервано для обработки сигнала.

4.4. Диспетчеризация процессов

Наряду с ОП время ЦП является важнейшим ресурсом системы, подлежащим распределению между процессами, существующими в данный момент времени в системе и находящимися в состоянии «Готов». Система *UNIX* является системой *с разделением времени*, так как она предоставляет время ЦП процессам по-очереди небольшими порциями (квантами). Что касается частоты предоставления квантов процессу и их фактической величины, то это зависит от типа процесса, предыдущей истории его обслуживания на ЦП, а также от типа других процессов, существующих в системе.

В зависимости от требуемой дисциплины диспетчеризации все существующие в *UNIX* процессы можно разбить на три группы:

1) **интерактивные процессы**. Они выполняются в 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.

Текущая величина **PRI** приоритета процесса хранится в одном из полей управляющей структуры процесса *proc*. Значение величины **PRI** зависит от состояния процесса, а также от того, как процесс попал в это состояние. Специальная константа разделяет величины приоритетов **PRI** на два непересекающихся подмножества системных и прикладных приоритетов. Допустим, что величина этой константы 76 (рис.25). Тогда системный приоритет процесса находится в диапазоне: $0 \leq PRI \leq 76$, а прикладной приоритет: $76 < PRI \leq 140$. В свою очередь системные приоритеты делятся на высокие ($0 \leq PRI \leq 25$) и низкие ($25 < PRI \leq 76$).

Процесс обладает системным приоритетом в следующих случаях: а) в состоянии «Ядро»; б) в состоянии «Сон»; в) в состоянии «Готов», если процесс перешел в это состояние из состояния «Сон». Покажем, что величина и использование системного приоритета в перечисленных случаях может быть различной.

Диапазон **PRI**

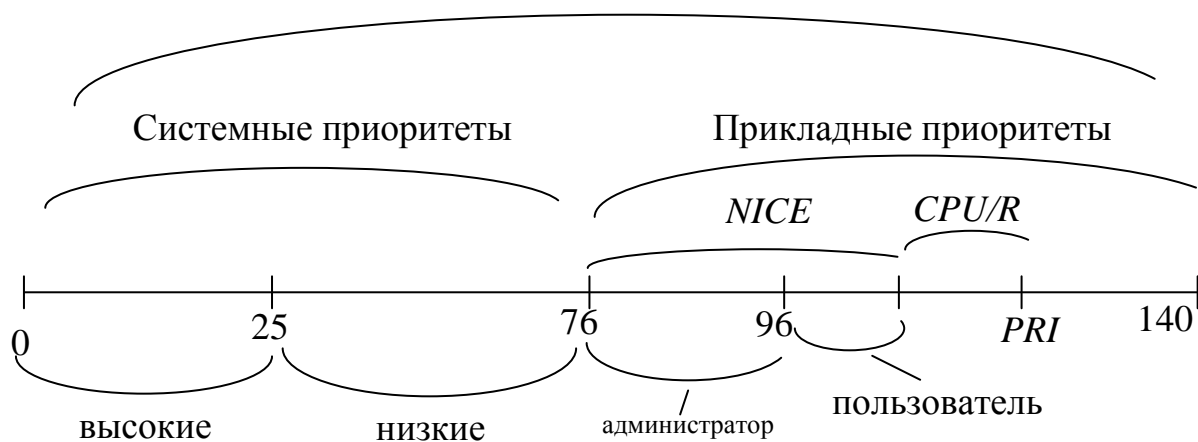


Рис. 25. Приоритеты процесса

Переход процесса в состояние «Ядро» является или результатом системного вызова, или результатом исключения, или результатом внешнего аппаратного прерывания. В любом из этих случаев процесс не снимается с ЦП, а переходит в другое состояние выполнения. Если процесс перешел в состояние «Ядро» в результате системного вызова или в результате исключения, то ему назначается достаточно низкий системный приоритет. Это не имеет никакого значения по отношению к другим процессам, так как пока процесс находится в состоянии «Ядро», никакой другой процесс не может заменить его на ЦП. Данный системный приоритет используется лишь для обеспечения целостности данных ядра во время обработки внешних аппаратных прерываний.

Дело в том, что в отличие от исключений, происходящих вследствие выполнения программы самого процесса, внешние аппаратные прерывания никак не связаны с выполняемым на ЦП процессом. Для того чтобы обработчик внешнего прерывания не нарушил данные ядра, он снабжается своим приоритетом диспетчеризации, который сравнивается с приоритетом выполняемого на ЦП процесса. Уровень приоритета обработчика внешнего прерывания достаточен, чтобы прервать выполнение любого процесса в состоянии «Задача». Что касается состояния «Ядро», то приоритет процесса может быть как лучше, так и хуже приоритета обработчика. Это определяется тем, занимается ли в данный момент процесс обработкой данных ядра или нет. Если да, то процесс выполняет свою критическую секцию (см. п.2.4.3) и его прерывать нежелательно. На время выполнения критической секции приоритет процесса улучшается настолько, чтобы он был лучше приоритетов опасных обработчиков прерываний. Затем восстанавливается прежнее значение приоритета.

Если при выполнении системного вызова процесс переводится в состояние «Сон», то при вызове внутренней процедуры ядра *sleep*, выполняющей такой перевод, в качестве ее входного параметра задается требуемый системный приоритет процесса. Величина этого приоритета зависит от типа операции ядра, завершения которой ожидает процесс. При этом каждой такой операции присущ фиксированный уровень системного приоритета процесса. Этот уровень приоритета процесса влияет на обработку сигналов, поступивших процессу в состоянии «Сон». При этом высокий системный приоритет временно запрещает, а низкий, наоборот, разрешает обработку поступивших сигналов.

После того как операция, ожидаемая процессом в состоянии «Сон», завершится, процесс окажется в состоянии «Готов» с тем системным приоритетом, который он получил при переходе в состояние «Сон». С

этим приоритетом он конкурирует за время ЦП с другими процессами, находящимися в СГ, а также с процессом, выполняемом на ЦП в состоянии “Задача”.

Если процесс находится на этапе выполнения прикладной программы, то его приоритет PRI рассчитывается по формуле:

$$PRI = 76 + NICE + CPU/R,$$

где $NICE$ - пользовательский приоритет;

CPU – степень использования ЦП;

R – коэффициент учета CPU . Подобно PRI , текущие значения $NICE$ и CPU содержатся в структуре *proc* процесса.

Пользовательский приоритет $NICE$ по умолчанию имеет значение 20. Именно на столько отличается число 96 (первоначальный прикладной приоритет процесса) от числа 76 (наилучший прикладной приоритет). Пользователь может влиять на $NICE$ (а следовательно и на PRI) у требуемого процесса с помощью команд *shell* – *nice* и *renice*. При этом следует отметить, что обычный пользователь может только ухудшать $NICE$ у своих процессов. А администратор системы может не только ухудшать, но и улучшать $NICE$ для любого процесса.

Команда *nice* используется для запуска нового процесса с приоритетом $NICE$, отличным от принятого по умолчанию. В следующем примере программа *prog5* (а точнее, соответствующий процесс) запускается с $NICE$, ухудшенным на 10:

```
$nice -10 prog5
```

Пример команды, улучшающей $NICE$ (команда доступна только администратору):

```
$nice - -10 prog5
```

Команда *renice* позволяет изменить значение $NICE$ для уже существующего процесса. В следующем примере у процесса с номером 168 значение $NICE$ увеличивается на 20:

```
$renice 20 168
```

CPU – степень использования ЦП, которая увеличивается при выполнении процесса на ЦП. При этом через каждый тик таймера (10 или 20 мсек) диспетчер увеличивает CPU на единицу. Величина CPU для процессов, находящихся в СГ, периодически также изменяется диспетчером, но, во-первых, в сторону уменьшения, а во-вторых, это происходит гораздо реже. Что касается величины этого уменьшения, то в разных системах она определяется по разному. Например, в некоторых системах CPU процесса в состоянии «Готов» корректируется каждую секунду в сторону уменьшения по формуле:

$$CPU_{i,t} := CPU_{i,t-1} * (2 * n_{t-1}) / (2 * n_{t-1} + 1),$$

где $CPU_{i,t}$ – величина CPU для i -процесса на t -й секунде времени;

n_{t-1} – среднее число процессов, находившихся в СГ в течение последней секунды. Нетрудно заметить, что чем n больше, тем меньше улучшение приоритета каждого процесса в СГ.

Коэффициент учета CPU R используется для пересчета величины CPU в величину приоритета. Значение этого параметра зависит от типа системы UNIX. В рассматриваемом примере $R = 8$.

В п.2.2 мы применяли команду *shell* – *ps* для получения краткой информации о процессах. Применение этой команды с флагом **-l** позволяет вывести на экран полную информацию о процессах, в том числе, и их приоритеты. Пример выдачи приведен на рис.26.

```
$ ps -l
```

F	S	UID	PID	PPID	CPU	PRI	NICE	TTY	TIME	CMD
1	S	vlad	701	683	0	8	0	P1	0:00.04	bash
1	R	vlad	712	701	137	133	20	P1	0:54.04	sh scl
1	R	vlad	713	701	288	132	0	P1	2:37.91	sh scl
1	O	vlad	720	701	0	96	0	P1	0:00:00	ps -l

Рис. 26. Пример выдачи на экран команды “ps -l ”

В выдаче команды появились новые столбцы:

1) **F** – комбинация битов (записанная в восьмеричной системе), определяющая тип процесса (1 – программа процесса находится в ОП; 2 – системный процесс; 4 – программа процесса занимает фиксированное место в ОП (используется в процессах-драйверах); 10 – программа процесса выгружена из ОП; 20 – процесс трассируется другим процессом). Например, если $F=3$, то это означает, что процесс системный и его программа находится в ОП;

2) **S** – состояние процесса (O – «Задача»; S – «Сон»; R – «Готов»; I – создается; Z – «Зомби»);

3) **UID** – имя пользователя-владельца процесса;

4) **PPID** – номер процесса-родителя;

5) **CPU** – степень использования ЦП;

6) **PRI** – текущий приоритет процесса;

7) **NICE** – относительный пользовательский приоритет процесса. Значение 0 в данном столбце означает, что пользовательский приоритет имеет значение по умолчанию – 20;

8) **ADDR** – адрес программы процесса (в ОП – для резидентных процессов; на диске (в области свопинга) – для остальных процессов);

- 9) *SZ* – размер программы процесса в блоках (по 512 байт);
- 10) *WCHAN* – номер события, которого ожидает процесс в состоянии сна.

Последние три столбца из перечисленных, на рис.26 не показаны. Что касается остальных столбцов, то мы рассмотрим их более внимательно. Во-первых, заметим, что выдача команды *ps* отражает состояние процессов на момент времени, непосредственно предшествующий попаданию на ЦП той машинной команды программы *ps*, которая реализует системный вызов для вывода на экран. Во-вторых, на этот момент в системе существовали четыре процесса, принадлежащих данному пользователю *vlad*:

1) интерактивный процесс *shell* (программа *bash*), выполняющий обслуживание пользователя в оперативном режиме. На рассматриваемый момент времени данный процесс находился в состоянии “Сон”, так как ожидал завершения своего дочернего процесса, выполняющего в оперативном режиме программу *ps*;

2) два фоновых процесса, порожденных оперативным *shell* в ответ на команды пользователя. Каждый из этих двух процессов выполняет свой *shell*, производящий интерпретацию одного и того же скрипта *sc1*, содержащего бесконечный цикл. Оба фоновых процесса запущены пользователем с разными пользовательскими приоритетами, но в один и тот же момент времени, с помощью следующего конвейера:

```
$nice -20      sh sc1 | sh sc1 &;
```

3) На момент получения выдачи программой *ps*, оба процесса находились в состоянии “Готов”;

4) интерактивный процесс выполнения программы *ps* в оперативном режиме. На рассматриваемый момент времени данный процесс находился в состоянии «Задача».

Что касается приоритетов процессов, то только один из них обладает системным приоритетом (8). Это интерактивный *shell* с номером 701. Остальные процессы обладают лишь прикладными приоритетами – 133, 132, 96. Последнее из этих значений присуще интерактивному процессу, выполняющему программу *ps*. Что касается фоновых процессов 712 и 713, то имея близкие значения PRI (это следствие описываемой ниже работы диспетчера), повышенное значение пользовательского приоритета NICE в процессе 712 компенсируется более высоким значением CPU у процесса 713. Разница значений CPU у процессов является следствием разницы полученного ими времени ЦП (колонка TIME в выдаче команды *ps*).

Теперь мысленно предположим, что интерактивный процесс не перейдет в состояние «Ядро» (в результате системного вызова для вывода на экран), а останется на ЦП в состоянии «Задача». По истечению каждых 10 мс его CPU будет увеличено диспетчером на единицу. Так как CPU влияет на величину приоритета процесса с коэффициентом 1/8, то приоритет PRI изменится на единицу через время: $8 \cdot 10 \text{ мс} = 80 \text{ мс}$.

Величина времени, в течение которого приоритет PRI процесса, исполняемого на ЦП, увеличивается на единицу, называется квантом. В рассматриваемом примере величина кванта равна 80 мс. По истечению кванта диспетчер сравнивает приоритет процесса, находящегося в состоянии «Задача» с приоритетом процесса, находящегося в начале СГ. При этом, если приоритет исполняемого на ЦП процесса хуже (то есть PRI больше), то этот процесс переходит с ЦП на то место в СГ, которое соответствует его приоритету PRI. А на ЦП ставится процесс из начала СГ. В нашем примере таким процессом будет процесс с номером 713. Напомним, что сама установка процесса на ЦП производится путем загрузки в регистры ЦП аппаратного контекста процесса.

Изложенная схема диспетчеризации учитывает интересы как интерактивных, так и фоновых (вычислительных) процессов. Интерактивные процессы большую часть своего времени находятся в состоянии «Сон» в ожидании завершения пользовательского ввода. Выходя из этого состояния, они имеют достаточно хороший системный приоритет, и поэтому достаточно быстро начинают выполняться на ЦП. Что касается фоновых процессов, то находясь подолгу в СГ, их приоритет улучшается, и они гарантированно получают доступ к ЦП.

Если система рассчитана на выполнение не только интерактивных и вычислительных процессов, но и процессов реального времени, то пространство текущих приоритетов разделяют не на две, а на три части. Лучшая часть приоритетов отводится для процессов РВ, хуже – для системных приоритетов, еще хуже – для прикладных приоритетов. Приоритет процесса РВ фиксирован. Он не меняется ОС и одинаков для состояний процесса «Ядро» и «Задача». Поэтому на продолжительность выполнения процесса РВ никак не влияют не только интерактивные и вычислительные процессы, но и менее приоритетные процессы РВ.

Диспетчер 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 истечению кванта времени рассматривать возможность замены процесса на ЦП, если эта замена не произошла ранее по другой причине.

В основе обработки отложенных вызовов лежат прерывания от таймера. Это устройство играет большую роль в механизме функционирования всей системы.

4.5. Использование таймера для управления процессами

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

Обработчик прерываний таймера выполняет следующие действия:

- 1) обновление статистики использования процессора для текущего процесса. Отдельно подсчитываются затраты времени ЦП на выполнение процесса в режиме «Ядро» и в режиме «Задача»;
- 2) 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. **Отложенный вызов** – выполнение требуемой подпрограммы ядра через требуемое время. Заявки на отложенные вызовы поступают от подпрограмм ядра;
- 5) обработка алармов. **Аларм** – отправка сигнала процессу по истечению заданного интервала времени. Заявки на алармы могут поступать от любых процессов;
- 6) пробуждение тех процессов ядра, которые должны выполняться периодически, через заданный интервал времени.

Некоторые из перечисленных действий выполняются не на каждом тике. Если требуемое действие требует некоторых затрат времени ЦП,

то оно реализуется не самим обработчиком прерываний таймера, а какой-то другой подпрограммой ядра, вызываемой с помощью отложенных вызовов. Любая подпрограмма ядра может сделать заявку на отложенный вызов себя или другой подпрограммы ядра, передав на вход подпрограммы обработки отложенных вызовов два параметра: A – стартовый адрес требуемой подпрограммы; t – величина 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 истечению которого процесс получит сигнал.

При выполнении данного системного вызова ядро создаст новую переменную – *таймер интервала* (не путать с аппаратным таймером), содержащую величину заданного интервала, измеренную в тиках. Данная переменная будет уменьшаться обработчиком прерываний таймера на единицу при обработке каждого тика. Как только таймер интервала станет равным нулю, соответствующему процессу будет послан сигнал *SIGALRM*. Следует отметить, что данный системный вызов не блокирует выполнение процесса до получения сигнала через время t , а сразу возвращает ему управление. Чтобы обеспечить такое блокирование, системный вызов `ЗАДАТЬ_ВРЕМЯ` следует использовать в программе процесса совместно с системным вызовом:

`ПАУЗА(|)` (на СИ – *pause*)

Данный вызов приостанавливает выполнение вызывающего процесса до получения любого сигнала. В качестве такого сигнала особенно удобно использовать сигнал *SIGALRM*, выдаваемый системным вызовом `ЗАДАТЬ_ВРЕМЯ`. Например, пара системных вызовов `ЗАДАТЬ_ВРЕМЯ` и `ПАУЗА` используются при выполнении команды

shell – sleep, выполняющей задержку программы на заданное число секунд (см. п. 1.5.5).

5. УПРАВЛЕНИЕ ФАЙЛАМИ

5.1. Виртуальная файловая система

5.1.1. 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.3.3) и имеет укрупненную структуру, приведенную на рис. 27. Как видно из данного рисунка, файловая подсистема включает виртуальную файловую систему, программные части реальных файловых систем, а также КЭШ.

В п.1.2 мы рассмотрели файлы и файловую структуру системы с точки зрения пользователя. Теперь расширим эти представления до уровня, используемого в программах процессов. Так как программисты являются частным случаем пользователей, то такое расширение является вполне обоснованным.

Во-первых, заметим, что программа процесса имеет дело не с реальным, а с логическим файлом. В отличие от реального файла, представляющего собой уникально поименованную битовую строку на конкретном носителе, *логический файл* не связан с конкретным носителем информации, а его программное имя не является уникальным в пределах всей системы.

Несмотря на то, что и физический, и соответствующий ему логический файл содержат одну и ту же битовую строку, разбиение этой строки на записи в обоих файлах различно. В то время как записи физического файла выбираются исходя из удобства их размещения на носителе, разбиение логического файла на записи производится по смысловому признаку. Например, если логический файл содержит сведения о сотрудниках какой-то организации, то одна запись такого файла содержит информацию об одном сотруднике.

Многие прежние ОС поддерживали разбиение логических файлов на логические записи. В результате программа могла, например, выполнить чтение или запись логической записи, используя для ее идентификации или номер записи в файле или ее ключ (здесь ключ – символьное имя записи). Современные ОС, в том числе *UNIX*, не поддерживают разбиение логических файлов на записи. Они позволяют программе работать с логическим файлом так, как с длинной последовательностью

байтов. Естественно, что разбиение логического файла на записи есть. Но это разбиение известно только самой обрабатывающей программе. Для ОС оно 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.

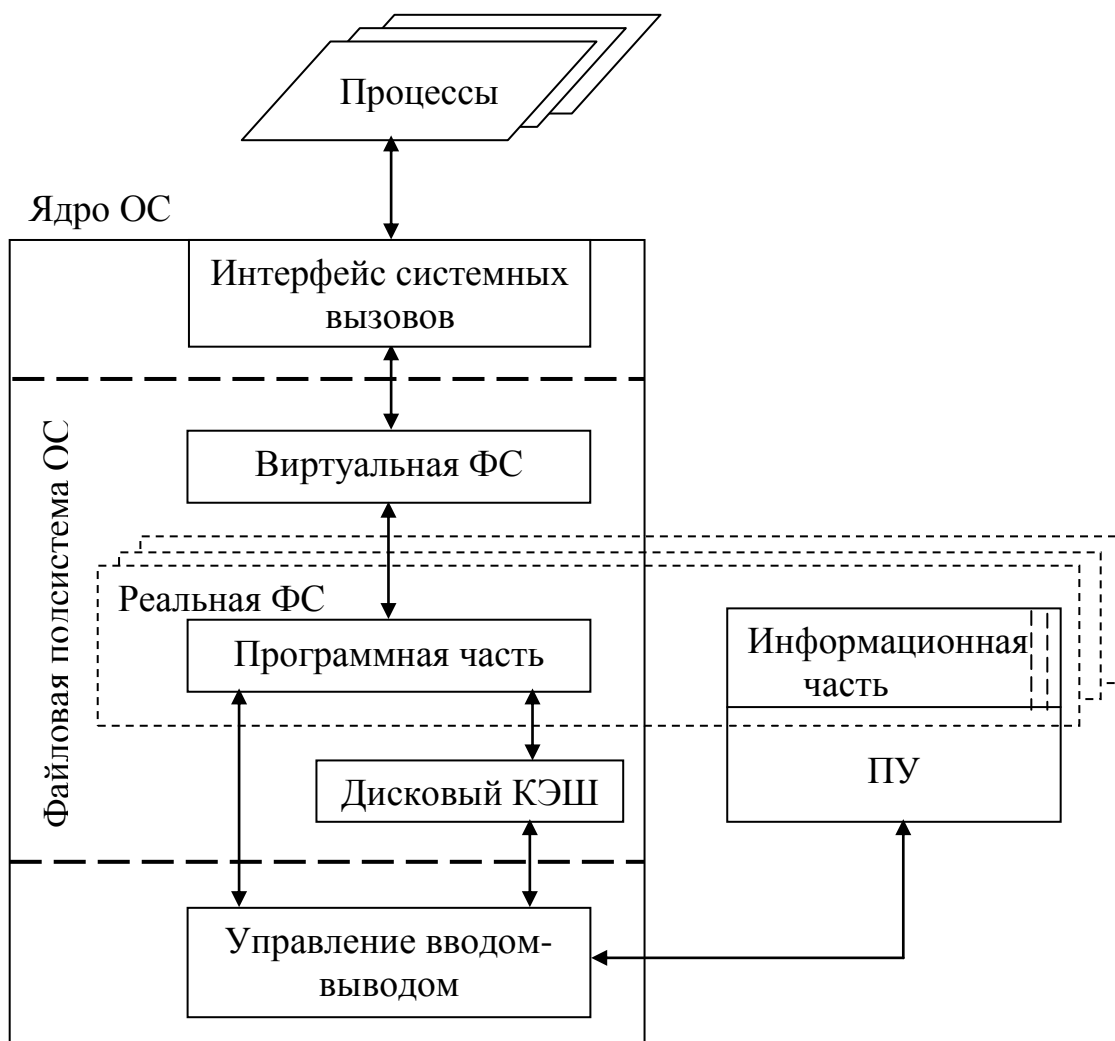


Рис.27. Структура файловой подсистемы ОС

В п. 1.2 была рассмотрена файловая структура системы, объединяющая в единое целое все файлы системы. (Напомним, что для *UNIX* файловая структура представляет собой единое дерево.) С учетом того, что нигде в системе данная структура целиком не хранится, будем называть ее **логической файловой структурой**. Если считать, что элементами логической файловой структуры являются логические файлы,

то получим *логическую файловую систему* – представление совокупности файлов с точки зрения пользователя-программиста.

Заметим, что термин «файловая система» является общепринятым, но не однозначным. Точно так же принято называть совокупность подпрограмм, выполняющих обработку соответствующих файлов. Во избежание путаницы, будем называть ту совокупность подпрограмм, которая выполняет обработку логических файлов, *виртуальной файловой системой*.

После того, как интерфейс системных вызовов выполнит «техническую» подготовку поступившего в ядро системного вызова, виртуальная ФС приступает к его действительному выполнению. Наличие этой подсистемы позволяет использовать в программах процессов стандартную совокупность системных вызовов для обработки файлов, независимую ни от физической реализации файлов, ни от физической реализации соответствующей файловой системы.

5.1.2. 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.1.2. Это имя используется для идентификации физического файла, задавая его расположение в логической файловой структуре;

r – требуемый режим работы с файлом (чтение, запись, чтение и запись, добавление в конец файла);

i – программное имя (номер) файла, уникальное для данного процесса. Это имя логического файла, которое может использоваться далее программой процесса для выполнения операций с этим файлом.

Получив данный системный вызов, виртуальная файловая система ищет требуемый файл, обращаясь за помощью к реальной файловой системе. Подробно реальные файловые системы рассматриваются в п.5.2, 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.5.3.

В зависимости от заданного символического 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.4.2). В любом случае поиск искомого файла начинается с анализа того *vnode*, который соответствует исходному каталогу.

Vnode (от *virtual inode* – виртуальный индексный дескриптор) – часть дескриптора файла, содержащая ту информацию о реальном файле, состав которой универсален для любого типа файлов. Перечислим наиболее интересные поля *vnode*:

- 1) число ссылок на данный *vnode*. Это суммарное число открытий файла во всех процессах на данный момент времени;
- 2) указатель на тот элемент списка монтирования (рассматривается в п.5.3), 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.5.2, 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, каж-

дый из которых представляет собой начальный адрес той процедуры реальной ФС, которая выполняет соответствующую операцию над физическим файлом;

- 7) тип файла, которому соответствует данный *vnode*: обычный файл, каталог, специальный файл устройства, символическая связь, сокет, удаленный файл.

Обратим внимание на поле 2, содержащее указатель на реальную ФС в списке монтирования, и поле 4, содержащее номер *inode* в реальной ФС. Пара этих чисел является уникальным системным именем файла на данный момент времени. Другим уникальным системным именем файла является номер самого *vnode* в *таблице vnode*. (рис.28). 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*). Отметим, что это одна из тех процедур реальной ФС, доступ к которой выполняется с помощью указателя на перечень операций, расположенном в поле *vnode*. Сама трансляция имени сводится к пошаговому “движению” по цепочке каталогов до тех пор, пока или не будет достигнут требуемый файл, или при прохождении очередного каталога будет выяснено, что права доступа к нему данного пользователя не отвечают минимально необходимым требованиям (см. п.3.2). 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.

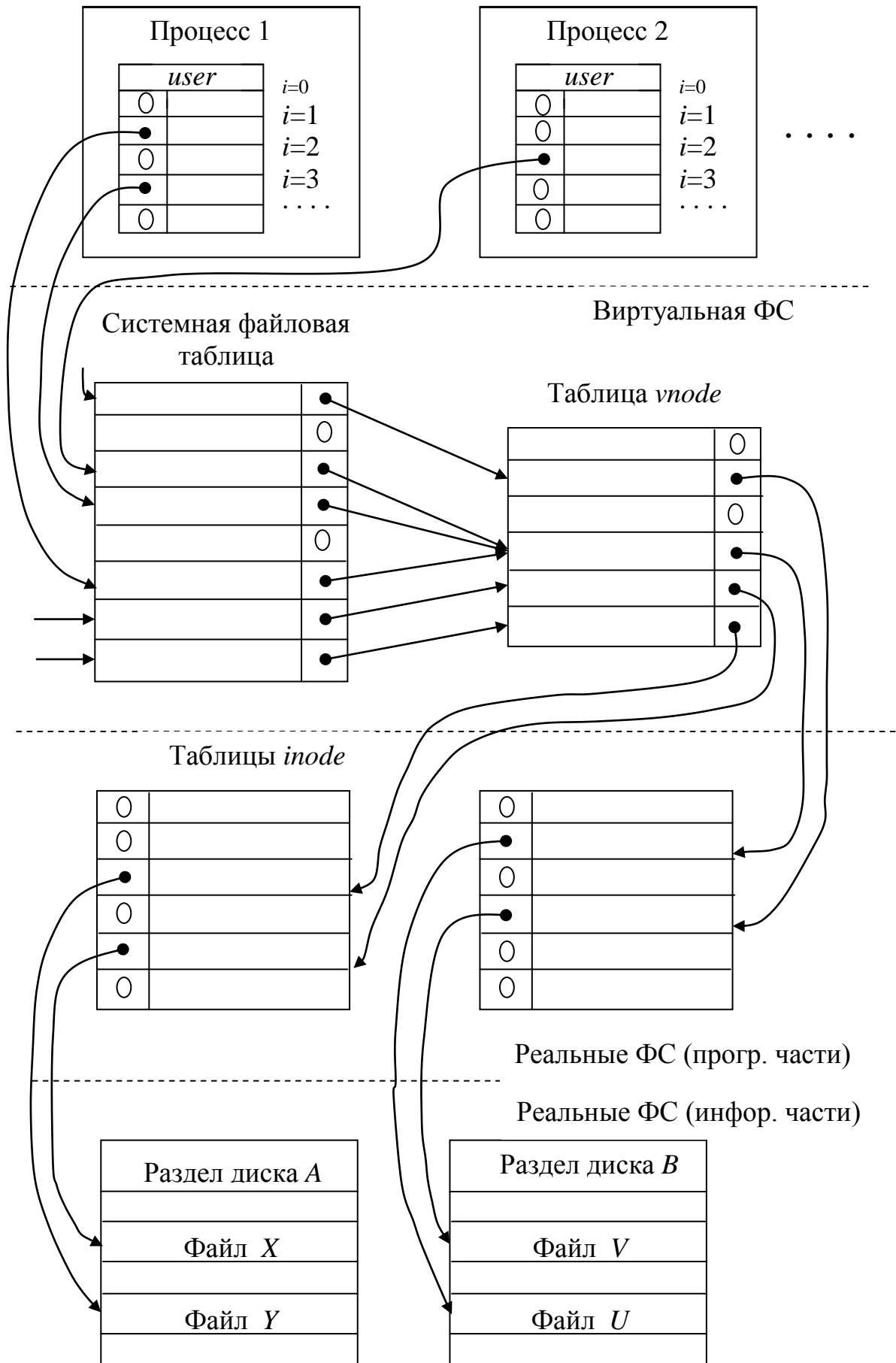


Рис. 28. Системные структуры данных для доступа процессов к файлам

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.

Рис. 28. Системные структуры данных для доступа процессов к файлам

Если требуемый *vnode* обнаружен, то число ссылок в его соответствующем поле увеличивается на единицу. Если виртуальная ФС не обнаружила *vnode* открываемого файла в своей таблице, то она создает новый *vnode*. Для этого ищется первый свободный элемент в таблице *vnode*, и его порядковый номер становится номером нового *vnode*. При заполнении полей этого *vnode* виртуальная ФС использует информацию, возвращенную ей процедурой реальной ФС.

После того, как *vnode* открываемого файла или найден, или создан заново, виртуальная ФС добавляет новую запись в свою **системную файловую таблицу**. Каждый процесс имеет в этой таблице свои записи, каждая из которых соответствует одному открытому процессом файлу. Если один и тот же файл открыт процессом несколько раз, то каждому открытию соответствует своя запись в системной файловой таблице. Перечислим поля этой записи:

- 1) режим доступа к файлу (чтение, запись, чтение и запись, добавление в конец файла);
- 2) **текущее значение файлового указателя** – номер байта файла, начиная с которого будет выполняться следующая операция информационного обмена с файлом (чтение или запись). Сразу после открытия файла эта переменная указывает в зависимости от режима открытия на самый первый или самый последний байт файла;
- 3) указатель на *vnode* файла.

Далее виртуальная ФС находит первую свободную строку в **таблице открытых файлов процесса**, являющейся частью структуры *user* процесса, а затем заполняет эту строку. Данная строка имеет всего два поля: указатель на соответствующую строку в системной файловой таблице и **флаг наследования открытого файла**. Если этот флаг установлен, то при создании дочернего процесса (с помощью соответствующего системного вызова) данный файл закрывается и его номер не наследуется дочерним процессом.

После того, как строка в таблице открытых файлов процесса заполнена, ее номер *i* возвращается системным вызовом в прикладную программу процесса в качестве программного имени открытого файла. Это имя действует в пределах данного процесса, а также наследуется

(если это не запрещено флагом наследования) его дочерними процессами.

Использование для открытия файла строки таблицы открытых файлов процесса с наименьшим номером нашло применение, в частности, для перенаправления ввода-вывода. Например, если пользователь запустил с командной строки какую-то программу, указав при этом замену стандартного вывода (то есть экрана) на какой-то файл, то процесс-*shell*, выполняющий обработку этой команды, во-первых, закроет файл с программным именем 0. Во-вторых, он откроет заданный файл. При открытии файла будет выбран свободный элемент таблицы открытых файлов процесса с наименьшим номером, то есть с номером 0. Так как структура *user*, включающая таблицу открытых файлов процесса, наследуется дочерним процессом, то весь его вывод на экран будет записан в требуемый файл. Естественно, что процесс-*shell* после запуска нового процесса должен «восстановить справедливость», закрыв файл и открыв экран. Заметим, что многие файлы, которые *shell* открывает для себя, не предназначены для использования в дочерних процессах. Для таких файлов выполняется установка флага наследования.

На рис. 28 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.

5.1.3. 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.3.2;

i – программное имя (номер) файла, уникальное для данного процесса.

Данный системный вызов не только создает новый файл, но и выполняет его открытие для записи, возвращая в качестве выходного параметра внутрипроцессный номер файла. В качестве владельцев файла записываются владелец-пользователь и владелец-группа того процесса, который выдал данный системный вызов. Если файл с именем I_f уже существует, то его длина обнуляется, а его права доступа и владельцы остаются прежними.

Виртуальная файловая система начинает выполнение данного системного вызова с того, что определяет наличие файла с заданным име-

нем If. 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 так же 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 If 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 иницировании той процедуры реальной ФС, которая производит чтение из заданного физического файла с указанного места заданного числа байтов.

5. Запись в файл. Операция записи выполняется с того байта файла, номер которого содержит файловый указатель. После завершения операции значение файлового указателя увеличивается на число записанных байтов.

Для выполнения записи в файл используется системный вызов:

ЗАПИСЬ_В_ФАЙЛ($i, n, A //$) (на СИ – *write*)

где i – программный номер файла;

n – количество записываемых байтов;

A – начальный адрес прикладной области в ОП, из которой производится запись в файл.

Данный системный вызов выполняется виртуальной ФС совместно с реальной ФС аналогично тому, как выполняется чтение из файла.

6. **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 – *map*)

где *i* – программный номер файла. Для получения данного номера отображаемый файл должен быть предварительно открыт в программе процесса;

l – величина смещения в файле на диске;

n – количество отображаемых байтов файла;

A – начальный адрес прикладной области в ОП, на которую производится отображение файла.

Выполняя данный системный вызов, ФС выполняет округление величины *n* в большую сторону до ближайшей границы логической страницы. Далее вызывается подсистема управления памятью с целью выделения такой области в прикладной части линейного виртуального адресного пространства процесса, которая достаточна для размещения отображаемой части файла (с учетом округления длины). Напомним, что в данном случае речь идет не о выделении реальной ОП, а лишь о создании таблицы страниц и записи в нее дескрипторов страниц.

Само же назначение реальной ОП (физической страницы) производится в результате страничного свопинга, реализация которого не зависит от того, что находится в странице ОП – фрагмент отображаемого файла, или обычные данные программы. Единственное отличие: при свопинге отображения файла информационный обмен производится не с областью свопинга на системном устройстве ВП, а с тем дисковым файлом, отображение которого выполнено.

Следует добавить, что применение отображения файла не позволяет изменить длину этого файла.

5.2. Реальные файловые системы

5.2.1. Критерии оценки файловых систем

Рассмотренная выше виртуальная ФС обеспечивает стандартный программный интерфейс, но с реальными (физическими) файлами эта подсистема ФС не работает. Эту функцию выполняет реальная ФС (программная часть). Наличие пояснения в круглых скобках обусловле-

но 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.5.3, 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:

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;

производительность – средняя скорость информационного обмена с физическими файлами, принадлежащими реальной ФС;

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;

используемость ВП – отношение общей длины физических файлов, содержащихся в информационной части реальной ФС к объему соответствующего раздела носителя;

предельная длина имени файла – максимальная длина простого имени файла;

предельное число файлов – максимальное количество физических файлов, которые могут находиться в одной информационной части реальной ФС;

сложность алгоритмов – сложность алгоритмов подпрограмм, принадлежащих программной части реальной ФС.

При выборе реальной ФС, как и при выборе любой другой системы, набор критериев, используемых для оценки вариантов системы, является противоречивым. Например, производительность многих типов реальных ФС зависит от используемости ВП: при высокой степени использования ВП производительность реальной ФС существенно падает. Аналогично повышение надежности реальной ФС обычно связано с дублированием информации, хранящейся на носителе, что приводит к снижению реальной производительности системы и к снижению используемости ВП.

Интересно выполнить сравнение распределенной и локальной реальных ФС. По критерию функциональности явно выигрывает распределенная ФС, так как она позволяет своим информационным частям размещаться на удаленных ЭВМ. Но по критериям надежности, сложности алгоритмов и, особенно, по критерию производительности, локальная реальная ФС явно предпочтительней. Так как логика использования распределенной реальной ФС имеет некоторые отличия от логики работы распределенной (сетевой) ОС, рассмотренной в п. 3.4, то в п.5.3 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 рассмотре-

ния всех этих реальных ФС. Вместо этого далее рассматриваются некоторые принципы проектирования таких систем, позволяющие существенно улучшить их оценки по основным из перечисленных выше критериям.

При рассмотрении принципов проектирования реальных ФС далее используются три типа таких систем: s5fs, ffs и fat. Первые две из этих систем являются для UNIX «родными». Несмотря на то, что эти ФС, особенно s5fs, разработаны давно, они до сих пор поддерживаются многими UNIX-системами. Присущая этим ФС простота делает их удобными для нашего рассмотрения.

Третья из перечисленных реальных ФС (fat) является для UNIX «чужой», так как она разрабатывалась для операционных систем MS-DOS и WINDOWS. Но учитывая ее очень широкое распространение, она входит в состав реальных ФС, поддерживаемых UNIX. Обратим внимание, что только информационные части fat-систем, принадлежащих UNIX аналогичны информационным частям таких систем, принадлежащих MS-DOS и WINDOWS. Что касается программных частей, то эти части реальной ФС сильно различаются в зависимости от типа ОС. Кроме того, заметим, что название fat является собирательным, объединяющим три реальных ФС: fat12, fat16 и fat32.

5.2.2. Физическое размещение информации на носителе

Оценки реальной ФС по критериям производительности, надежности, а так же используемости ВП, в значительной степени зависят от физического размещения информации, используемого в данной системе. Так как наиболее распространенными носителями ВП являются магнитные диски, то далее будет рассмотрено размещение информации на этих носителях. В данном подразделе рассмотрим общие свойства таких носителей, а в последующих подразделах рассмотрим влияние этих свойств на реальную ФС.

Упрощенное представление магнитного диска приведено на рис.29. 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. 29).

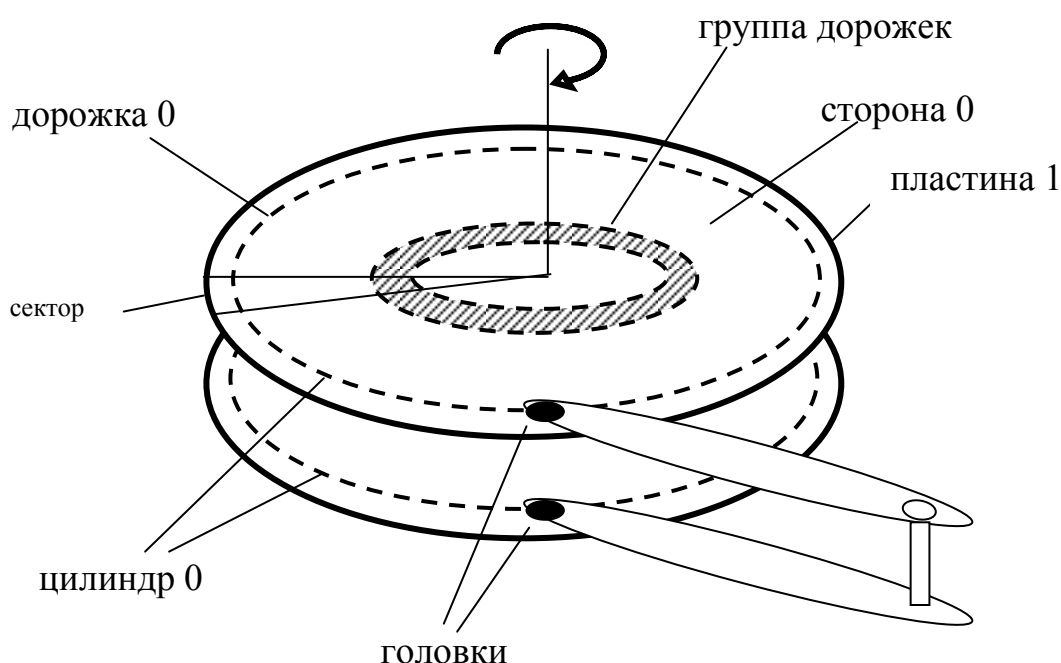


Рис. 29. Упрощенное представление магнитного диска

Для чтения (записи) информации с пластины (на пластину) используется один или несколько элементов, называемых головками. Количество головок зависит от используемого варианта конструкции дисководов. В первом из этих вариантов каждую поверхность пластины обслуживает единственная головка, которая может перемещаться по радиусу диска дискретными шагами. При этом каждый шаг соответствует одной дорожке пластины. После того, как головка «зависнет» над требуемой пластиной, она может выполнять информационный обмен с этой дорожкой. Так как все головки диска связаны жестко в единую «гребенку», то при установке одной из головок на требуемую дорожку все остальные головки «зависнут» над дорожками этого же цилиндра. Следствием этого является то, что переход между дорожками одного цилиндра не требует каких-либо механических перемещений головок. Для такого перехода достаточно лишь выполнить электронное переключение головок. Другой вариант конструкции дисководов предполагает, что каждую дорожку поверхности пластины обслуживает своя отдельная головка. В этом случае никакого перемещения головок не про-

изводится, 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 (вследствии 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. 29). Возьмем размер углового сектора таким, чтобы он «вырезал» из каждой дорожки на поверхности пластины дугу, содержащую 512 бит. Эта дуга дорожки называется сектором. Как видно из рис.29, один угловой сектор «вырезает» секторы одинаковой длины (измерение в битах) не только на одной поверхности одной пластины, но и на всех поверхностях всех пластин. Разбиение дорожки диска на секторы производится во время низкоуровневого форматирования диска. Во время этого форматирования начало каждого сектора помечается специальной последовательностью битов. Кроме того для каждого сектора записывается его номер.

С точки зрения ОС сектор является минимальной единицей информационного обмена между диском и ОП. Для выполнения такого обмена на каждый сектор задается своим адресом:

(номер цилиндра; номер поверхности; номер сектора)

Среднее время чтения (записи) одного сектора T_c можно найти по формуле:

$$T_c = T_d + T_v + T_r ,$$

где T_d – среднее время установки головки на требуемую дорожку. Типичное значение этого времени для современных дисков составляет от 5 до 10 мс;

T_v – среднее время вращения диска до достижения требуемого сектора. Нетрудно заметить, что это время выполнения диском половины своего оборота, которое в свою очередь определяется скоростью вращения диска: $T_v = 1/2n$, где n – скорость вращения диска (об/сек). Гибкие

магнитные диски имеют скорость вращения 5 – 10 об/сек, а жесткие – 90 – 170 об/сек. Например, при скорости вращения 170 об/сек среднее время вращения диска до достижения требуемого сектора составляет примерно 3 мс;

T_r – время непосредственного считывания (записи) одного сектора. Так как это время поворота диска на заданный угловой сектор, то его можно определить по формуле: $T_r = 1/(n*L)$, где L – число секторов на дорожке. Например, если диск имеет на каждой дорожке 320 секторов, а его скорость вращения $n = 170$ об/сек, то для него $T_r = 1/(170*320) = 0,018$ мс.

Приведенные формулы могут быть использованы для приблизительной оценки производительности различных реальных ФС. Но при этом надо учесть, что в качестве единицы информационного обмена с диском любая реальная ФС использует не сектор, а более крупную величину – блок. **Блок** – часть дорожки диска, состоящая из одного или нескольких секторов. Длина блока для каждой информационной части реальной ФС является величиной фиксированной и равняется размеру сектора диска, умноженному на степень двойки: 512 байт $\times N$, где $N = 1, 2, 4, \dots$. Заметим, что блок в *MS-DOS* и *WINDOWS* называется **кластером**. В отличие от сектора имя блока не связано с его физическим размещением на диске, а представляет собой порядковый номер в пределах данной информационной части реальной ФС. Блок является той единицей пространства ВП, которая используется ОС при распределении этого пространства между файлами.

После того, как проведено разбиение диска на разделы (если это требуется), для каждого раздела производится свое отдельное **высокоуровневое форматирование**. Во время этого форматирования в разделе размещаются управляющие структуры той реальной ФС, одна из информационных частей которой будет размещаться в данном разделе. При этом среди прочей управляющей информации записывается размер блока, выбранный для данного раздела. Заметим, что размеры блоков разных разделов диска могут быть разными.

Допустим, что размер блока составляет $4K$ (8 секторов). С учетом того, что секторы одного блока расположены последовательно на одной и той же дорожке, для приведенных выше численных параметров среднее время чтения (записи) одного блока составит T_b :

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

Очень часто требуется считать (записать) не один блок файла, а целую последовательность таких блоков. В этом случае затраты на обработку каждого блока приведенного выше среднего времени приведут к очень большому суммарному времени на обработку данного файла.

Например, пусть требуется считать 1000 блоков файла. Если эти блоки разбросаны (в результате распределения ВП файлу) по всему разделу, то общее время чтения будет очень велико:

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

Для увеличения производительности реальной ФС необходимо локализовать размещение блоков файла на диске так, чтобы они находились если не в пределах одного цилиндра, то в пределах близких цилиндров. Например, если все 1000 блоков нашего файла расположены на 4-х дорожках одного и того же цилиндра, то среднее время их чтения или записи составит:

$$T_2 = T_d + 4 * T_v + 1000 * 8 * T_r = 10 \text{ мс} + 4 * 3 \text{ мс} + 1000 * 8 * 0,018 \text{ мс} = 0,166 \text{ с}$$

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. 29). В начале этой группы находится управляющая информация, в том числе, дисковые блоки управления файлами – *inode*. При создании файла его *inode* помещается в одну из групп цилиндров. При выборе этой группы используются соображения:

- 1) если создается не каталог, а файл данных, то желательно поместить *inode* файла в ту группу цилиндров, куда ранее был помещен родительский каталог. Это позволяет повысить производительность реальной ФС при обслуживании многих программ, выполняющих обработку файлов из одного каталога;
- 2) если создается каталог, то его *inode* желательно поместить в группу цилиндров, отличную от группы цилиндров, в которой находится родительский каталог. Это позволяет улучшить равномерность распределения данных по диску.

Если при выполнении очередной операции записи в файл ему требуется выделить новый блок памяти, то этот блок выбирается, по возможности, из той же группы цилиндров, где находится *inode* файла. Это позволяет сократить не только время перехода между блоками данных, но и время перехода от *inode* файла к его блокам данных.

Если раздел диска, занимаемый системой *ffs*, включает всего одну группу цилиндров, то эта система не обладает лучшей производительностью по сравнению с *s5fs*. При наличии нескольких групп цилиндров система *ffs* имеет достаточно хорошую производительность лишь при ограниченной используемости ВП (не более 90%). При более высокой степени загрузки диска распределение свободных блоков фактически производится случайным образом, то есть без соблюдения перечисленных выше рекомендаций.

В заключение данного подраздела рассмотрим вопрос о выборе размера блока (кластера) с точки зрения производительности реальной ФС и присущей ей используемости ВП. С учетом того, что содержимое блока всегда находится на одной дорожке (или в одном цилиндре) увеличение размера блока приводит к увеличению производительности системы, так как при одних и тех же накладных расходах времени ($T_d + T_v$) удастся выполнить больший информационный обмен между ОП и

диск. Но с другой стороны, увеличение размера блока в системах *s5fs* и *fat* приводит к существенному снижению используемости ВП, так как в этих системах блок является минимальной единицей распределяемой ВП. И поэтому, даже очень небольшому файлу выделяется целый блок диска. В результате при наличии большого числа таких файлов в системе значительный объем дисковой памяти фактически не используется. Во-вторых, каждому файлу, занимающему более одного блока, в среднем соответствует половина неиспользуемого дискового блока, в котором находятся последние байты файла.

Для того, чтобы обеспечить хорошую используемость ВП при использовании блоков большого размера, в реальной ФС *ffs* применяется фрагментация блоков – логическое разбиение блока на два, четыре, восемь или более фрагментов одинаковой длины. При этом минимально допустимый размер фрагмента определяется размером сектора (512 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.

5.2.3. Каталоги

Напомним (см. п. 1.2), что каталог представляет собой служебный файл, содержащий сведения о других файлах, в том числе, возможно и о других каталогах. Благодаря наличию каталогов все физические файлы, принадлежащие одной и той же информационной части реальной файловой системы, оказываются логически связанными в единую информационную структуру в виде дерева. Одна логическая запись (элемент) каталога содержит сведения об одном дочернем файле или каталоге.

Система *s5fs*. В реальной файловой системе *s5fs* одна логическая запись каталога содержит минимум информации о дочернем файле (каталоге):

- 1) номер *inode* файла (2 байта). Этот номер представляет собой уникальное имя файла в пределах конкретной информационной части реальной ФС. Нулевой номер *inode* обозначает удаленный файл. Поэтому при удалении файла реальная ФС (программная часть) заменяет номер его *inode* на 0;
- 2) простое имя файла (14 байтов). Первый элемент каталога содержит имя «.», обозначающее сам этот каталог. Второй элемент каталога содержит имя «..», обозначающее родительский каталог по отношению к рассматриваемому каталогу.

Длины этих полей логической записи каталога определяют две характеристики всей реальной ФС:

- 1) предельное число файлов: $2^{16} - 1 = 65536 - 1 = 65535$;
- 2) предельная длина имени файла – 14 символов.

Система *ffs*. Для преодоления второго ограничения реальная файловая система *ffs* использует следующую структуру каталога:

- 1) номер *inode* файла (2 байта);
- 2) длина (в байтах) поля, содержащего простое имя файла (2 байта);
- 3) длина (в байтах) простого имени файла (2 байта);
- 4) простое имя файла (поле имеет переменную длину до 255 байтов). Данное поле дополняется нулями до 4-х 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 за-

писи 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.3.2).

5.2.4. Управляющие структуры данных

Информационная часть реальной ФС включает физические файлы, размещенные на носителе ВП или в его разделе, а также управляющие структуры данных, находящиеся на этом же носителе (разделе носителя). Реальные ФС, разработанные для *UNIX*, содержат управляющие структуры данных:

1) *суперблок* – дескриптор информационной части реальной ФС. Этот блок управления содержит информацию, необходимую для управления всей информационной частью реальной ФС;

2) массив *inode*. Каждый *inode* представляет собой дескриптор какого-то одного физического файла, принадлежащего данной информационной части реальной ФС. При открытии файла *inode* считывается с носителя в ОП и становится элементом таблицы активных *inode*.

Система *s5fs*. Каждая информационная часть этой реальной ФС имеет всего один суперблок, который находится в начале раздела и содержит поля:

- 1) тип реальной ФС (*s5fs*);
- 2) размер информационной части реальной ФС в блоках;
- 3) размер массива *inode*;
- 4) число свободных блоков;
- 5) число свободных *inode*;
- 6) размер блока (512, 1024, 2048,...);
- 7) список номеров свободных *inode*;
- 8) список номеров свободных блоков.

Размер списка номеров свободных *inode* во много раз меньше предельного числа файлов, которое для *s5fs* равно 65535. Поэтому при ис-

черпаниии этого списка программная часть реальной ФС просматривает массив *inode* с целью поиска его свободных элементов (каждый свободный *inode* имеет специальную отметку в своем первом поле). Найденные номера помещаются в список номеров свободных *inode*.

Что касается списка номеров свободных блоков, то его приходится хранить целиком. Первой причиной этого является то, что блоки не имеют специальных отметок об их занятости. Другой причиной являются большие затраты времени на считывание блоков для их просмотра из-за их значительных размеров. Для того, чтобы не ограничивать размеры списка номеров свободных блоков, поступают следующим образом. Во-первых, сам суперблок содержит лишь начальную часть этого списка. Первый элемент этой части списка, в отличие от других элементов, содержит не номер свободного блока, а номер блока, содержащего продолжение списка. Если требуется, то первый элемент этого блока с продолжением также содержит номер блока, используемого для продолжения списка и т.д.

Дескриптор физического файла *inode* в реальной ФС *s5fs* содержит поля:

- 1) тип файла, права доступа;
- 2) число ссылок на файл, совпадающее с числом его имен в логической файловой структуре системы;
- 3) имена пользователя-владельца и пользователя-группы;
- 4) размер файла в байтах;
- 5) время последнего доступа к файлу;
- 6) время последней модификации;
- 7) время последней модификации *inode* (кроме модификации полей 5 и 6);
- 8) массив номеров блоков, занимаемых данными файла.

Последнее поле содержит массив из 13 элементов, первые десять из которых содержат номера первых десяти блоков файла. Последние три элемента используются, соответственно, для косвенной, двойной косвенной и тройной косвенной адресации блоков файла. Косвенная адресация заключается в том, что 11-й элемент массива содержит номер того блока раздела, который содержит номера блоков, занимаемых файлом. Двойная косвенная адресация реализуется за счет того, что 12-й элемент массива содержит номер блока, используемого для хранения номеров блоков, косвенно адресующих блоки файла. Тройная косвенная адресация заключается в том, что 13-й элемент массива содержит номер блока, в котором находятся номера блоков, используемых для двойной косвенной адресации блоков файла.

Система *ffs*. В этой реальной ФС каждая информационная часть имеет с целью повышения надежности не один экземпляр суперблока, а несколько – по одному суперблоку на каждую группу цилиндров. Для того, чтобы не вносить в каждую копию суперблока текущие изменения (во время этой операции может произойти сбой в системе), суперблок содержит лишь постоянную информацию о разделе. Поэтому суперблок *ffs* не имеет таких полей, присущих суперблоку *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 Кбайт дисковой памяти, принадлежащей данной группе цилиндров. **Битовая карта блоков** – последовательность битов, длина которой (в битах) определяется общим числом фрагментов блоков, имеющих 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 *FFFFFFFh*).

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*.

5.3. 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. 30). Для выполнения такого монтирования, во-первых, требуется выбрать в корневой ФС точку монтирования – каталог, к которому будет непосредственно подсоединена монтируемая ФС. Обязательным требованием к такому каталогу является его неиспользуемость в качестве точки монтирования для другой ФС. Кроме того, желательно, чтобы этот каталог был пуст. В примере в качестве точки монтирования используется каталог */home/vlad/prog*.

Для того, чтобы виртуальная ФС выполнила монтирование, она должна быть инициализирована с помощью системного вызова:

МОНТИРОВАТЬ (*t, I_u, I_k ||*) (на СИ – *mount*)

где *t* – тип монтируемой реальной ФС;

I_u – имя-путь специального файла, соответствующего устройству или разделу устройства, на котором расположена монтируемая информационная часть реальной ФС;
 I_k – имя-путь того каталога, который является точкой монтирования.

Первичным источником данного системного вызова является системная обрабатывающая программа *mount*, запускаемая суперпользователем из командной строки *UNIX*, и имеющая те же входные параметры, что и системный вызов.

Что касается выполнения системного вызова *МОНТИРОВАТЬ*, то виртуальная ФС начинает с того, что находит *vnode*, 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. **Коммутатор файловых систем** – массив, элементами которого являются описатели точек входа в программные части реальных ФС. Каждый элемент этого коммутатора соответствует одному типу реальных ФС и содержит поля:

- 1) тип реальной ФС;
- 2) адрес процедуры инициализации реальной ФС;
- 3) указатель на вектор операций реальной ФС.

Используя поле 2 найденного элемента коммутатора, виртуальная ФС вызывает процедуру инициализации реальной ФС. Эта процедура, во-первых, выполняет считывание в ОП суперблока монтируемой информационной части реальной ФС. Во-вторых, данная процедура добавляет новый элемент в список монтирования – односвязанный список, элементами которого являются дескрипторы смонтированных информационных частей реальных ФС. Элемент этого списка содержит поля:

- 1) указатель на следующий элемент в списке монтирования;
- 2) указатель на вектор операций реальной ФС. Это копия соответствующего поля элемента коммутатора файловых систем;
- 3) номер *vnode* того каталога, который является точкой монтирования;
- 4) размер блока монтируемой системы;
- 5) указатель на суперблок смонтированной реальной ФС.

Таким образом, в отличие от коммутатора файловых систем, элементы которого используются для доступа к программным частям реальных ФС, элементы списка монтирования используются для доступа к информационным частям этих ФС. Напомним, что в системе может быть только одна программная часть реальной ФС, но может существовать любое число ее информационных частей. Первым элементом списка монтирования всегда является корневая реальная ФС.

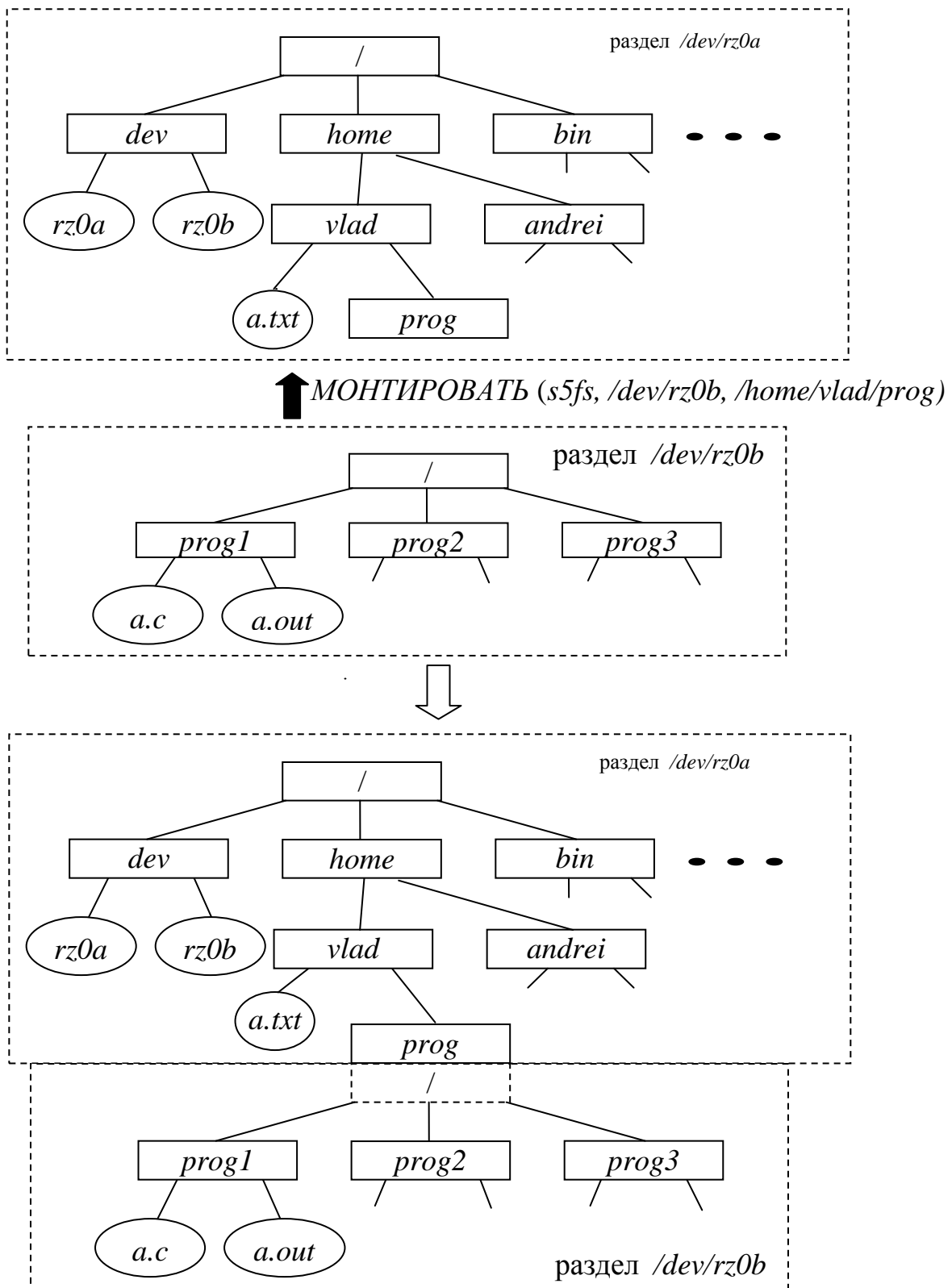


Рис. 30. Пример монтирования реальной файловой системы

После того, как в список монтирования добавлен новый элемент, указатель на этот элемент помещается в одно из полей того vnode, который соответствует каталогу, являющемуся точкой монтирования. Кроме того, создается новый vnode для корневого элемента монтируемой реальной ФС.

Интересно отметить, что в результате монтирования происходит логическое слияние двух каталогов, один из которых является точкой монтирования, а второй есть корень монтируемой ФС. Подобное слияние происходит лишь с точки зрения пользователя (на логической файловой структуре системы), так как каждому каталогу по-прежнему соответствует своя физическая реализация в своем разделе диска, а также свой отдельный vnode. Данное свойство влияет, в частности, на выполнение трансляции имен файлов. Если при трансляции очередного имени файла реальная ФС (программная часть) обнаружит, что достигнута точка монтирования, то она возвратит в виртуальную ФС не системное имя искомого файла, а указатель на подключенную ФС в списке монтирования. Далее виртуальная ФС должна 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 поддерева должны быть смонтированы с файловой структурой узла-клиента.

Получив системный вызов МОНТИРОВАТЬ, виртуальная ФС в узле-клиенте начинает с того, что по первому параметру системного вызова определяет тип монтируемой системы – NFS. Далее в список монтирования добавляется новый элемент (информационная часть NFS), а указатель на этот элемент помещается в тот vnode, который соответствует точке монтирования. После этого виртуальная ФС вызывает процедуру инициализации реальной ФС NFS. Данная процедура (как и другие процедуры NFS) выполняет свои функции, обмениваясь сообщениями с удаленной серверной частью NFS. При этом адрес требуемого узла-сервера определяется из параметра системного вызова

МОНТИРОВАТЬ, который соответствует разделу монтируемой системы. При монтировании удаленной ФС этот параметр содержит не имя раздела, а полное сетевое имя того каталога, который является корнем монтируемого поддерева файловой структуры.

Запрос с просьбой разрешить монтирование посылается в требуемый узел-сервер с помощью модуля транспорта, имеющегося в узле-клиенте (рис. 31). Приход этого сообщения инициирует процесс ядра «NFS-сервер», который большую часть времени находится в заблокированном состоянии. Реализация NFS-сервера в виде процесса обусловлена тем, что программа процесса в отличие от процедур не должна «подсоединяться» к программе какого-то другого процесса, выдавшего системный вызов. Реализация процесса «NFS-сервер» вне ядра, то есть в виде демона, привела бы к существенному снижению производительности сетевой ФС, так как данный процесс был бы вынужден конкурировать за время ЦП с обычными прикладными процессами.

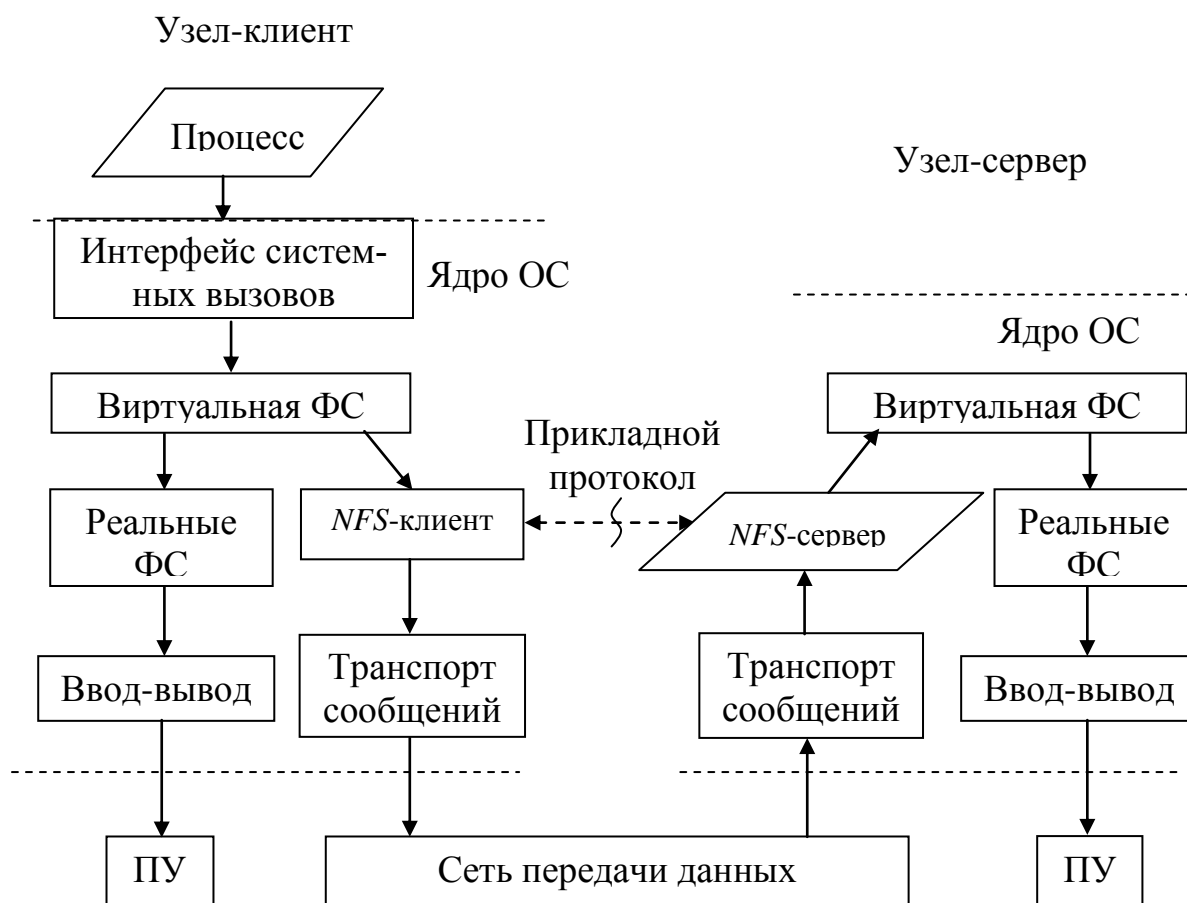


Рис. 31. Распределенная реальная файловая система NFS

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.

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 и как 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.

5.4. Кэширование блоков данных

Как видно из рис.27, 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.

6. ЛАБОРАТОРНЫЙ КУРС

Целью выполнения данного лабораторного курса является обучение студентов навыкам использования 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 него. При отсутствии такого класса допускается выполнение лабораторных работ в среде локальных *UNIX*-систем.

6.1 Лабораторная работа №1

Первоначальное знакомство с *UNIX*

6.1.1 Цель работы

Целью выполнения настоящей лабораторной работы является получение начальных навыков работы в среде *UNIX*: 1) вход в систему; 2) знакомство с текстовым редактором *ed*; 3) применение команд *shell* для работы с файлами; 4) работа в среде *Midnight Commander*.

6.1.2 Подготовка к выполнению работы

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

- 1) управление доступом пользователя в систему (п. 4.1) – понятия имени пользователя и пароля; псевдотерминал;
- 2) файлы с точки зрения пользователя (п. 2.2) – понятие файла; простое имя файла; каталоги; файловая структура системы; текущий каталог; относительное и абсолютное имена файла;
- 3) простые команды *shell* (утилиты) для работы с файловой структурой (п. 2.3.2) – *pwd*; *ls*; *cd*; *mkdir*; *rmdir*; *rm* с флагом *-r*; *cp* с флагом *-r*;
- 4) простые команды *shell* (утилиты) для работы с файлами (п. 2.3.3, п.2.3.4) – *cat*; *cp*; *mv*; *rm*;
- 5) упрощение пользовательского интерфейса (п. 2.3.6);
- 6) справочная команда *man* (п. 2.3.5).

В процессе изучения перечисленных вопросов обязательно следует выполнить примеры, приведенные в пособии. Некоторые из этих при-

меров требуют предварительного создания вспомогательных текстовых файлов и (или) каталогов. Для этого можно использовать команды *cat* и *mkdir*.

6.1.3 Вход в систему и выход из нее

При работе с многопользовательской системой, каковой является *UNIX*, каждый пользователь имеет свой уникальный идентификатор – имя пользователя. Кроме того, для избежания использования имени пользователя другим лицом, каждый пользователь может иметь пароль пользователя.

Имя пользователя – символьная строка. Например, оно может включать (одновременно) вашу фамилию, имя и отчество. Другой вариант – имя содержит номер студенческой группы и инициалы пользователя. В любом случае имя пользователя должно быть согласовано с администратором системы, который регистрирует его в «журнале» системы.

Работа с ВС, на которой установлен *UNIX*, начинается с включения терминала. В качестве терминала может использоваться совокупность экрана и клавиатуры или, даже, целая периферийная ЭВМ. В любом случае на экране появляется сообщение *UNIX – login*, в ответ на которое следует ввести имя пользователя. Пример ввода имени, зарегистрированного в системе:

```
UNIX → login:
польз. → vlad <Enter>
UNIX → $
```

Символ *\$* есть приглашение *UNIX* к вводу команды. Его появление на экране говорит об успешном входе в систему. Пример ввода имени, незарегистрированного в системе:

```
UNIX → login:
польз. → vlid <Enter>
UNIX → passwd:
```

Слово *passwd* означает просьбу ввести пароль. Может показаться странным, что в ответ на неправильное имя пользователя *UNIX* выводит такую просьбу. Дело в том, что таким образом система скрывает от лица, использующего неверное имя, сам факт такого использования. Поэтому вместо того, чтобы угадывать правильное имя, данное лицо будет

заниматься угадыванием пароля, что заведомо безнадежно, так, как ввод любого пароля приведет к повторению вывода на экран сообщения “*login*”.

Применение пароля – дело добровольное. Но если вы хотите, чтобы никто (или почти никто) в будущем не разрушил результат вашей работы, или не воспользовался бы им без вашего разрешения, без пароля не обойтись. Пароль пользователя регистрируется в системе не администратором, а самим пользователем с помощью команды *passwd* .
Пример:

```
UNIX → $
польз. → passwd <Enter>
UNIX → Changing password for vlad
      New password:
польз. → dracon <Enter>
UNIX → Retype new password:
польз. → dracon <Enter>
UNIX → $
```

Набор пароля производится пользователем «вслепую» – без отображения набираемых на клавиатуре символов на экране. Это позволяет избежать «подсматривание» пароля.

Требования к паролю: 1) если для написания пароля используются обычные алфавитно-цифровые символы, его длина должна быть не менее 6 символов (в примере пароль пользователя *dracon* отвечает этому требованию); 2) при использовании специальных и управляющих символов допускается меньшая длина пароля.

Если пароль отвечает хотя бы одному из приведенных двух требований, *UNIX* выводит просьбу повторить ввод нового пароля (с целью избежания ошибки при наборе пароля). Иначе *UNIX* выведет сообщение: “*Please use a longer password*” («Пожалуйста, введите длинный пароль»).

После того, как пароль зарегистрирован, он будет запрашиваться всякий раз, когда вы будете входить в систему. Пример такого входа:

```
UNIX → login:
польз. → vlad <Enter>
UNIX → passwd:
польз. → dracon <Enter>
UNIX → $
```

Если вы забудете пароль, то не сможете войти в систему без помощи администратора. Он удалит прежний пароль, а затем вы зарегистрируете в системе новый пароль. Замену пароля может выполнить и сам пользователь при условии, что прежний пароль им не забыт. Для этого вновь используется команда *passwd*. Пример:

```
UNIX → $
польз. → passwd <Enter>
UNIX → Changing password for vlad
        Old password:
польз. → dracon <Enter>
UNIX → New password:
        польз. → dracon7 <Enter>
UNIX → Retype new password:
        польз. → dracon7 <Enter>
UNIX → $
```

Зарегистрируйте свое имя пользователя у администратора *UNIX*-системы. Выполните вход в систему и установку пароля.

Примечание. Допустим, что после включения терминала вы оказываетесь не в среде *UNIX*, а в среде другой операционной системы, позволяющей связываться с *UNIX*. Тогда прежде, чем ввести команды по входу в *UNIX*, необходимо войти в исходную операционную систему, а затем ввести ее команду по запуску «терминала» в среде *UNIX*. При этом под «терминалом» понимается не аппаратное устройство, а псевдотерминал – программный процесс, связанный с *UNIX*.

Например, если ваш терминал находится в локальной сети, управляемой операционной системой *WindowsXP* (либо *старше*), то вы сначала входите в эту систему под своим логическим именем, которое зарегистрировано в этой локальной сети. После этого следует запустить программу *putty* (которую можно найти в свободном доступе в среде Internet, объем ее 450kb), которая осуществляет запуск программного процесса, имитирующего терминал *UNIX*. Необходимо заполнить следующие поля: Host name(or IP address): kcup.tusur.ru, Port: 22, Connection type: SSH и нажать кнопку Open. В дальнейшем действовать в терминале как изложено выше, и помнить при вводе пароля введенные символы не отображаются на экране!

Вопрос о взаимодействии различных операционных систем весьма интересен, но сейчас нас интересуют лишь его технические аспекты.

Выход из системы. При завершении сеанса работы необходимо сообщить об этом *UNIX* командой `<Ctrl>&<D>` (одновременное нажатие клавиш `<Ctrl>` и `<D>`). Пример:

```
UNIX   →   $
польз. →   <Ctrl>&<D>
UNIX   →   login:
```

Другой способ выхода из *UNIX* – использование команды *exit*.
В ы й д и т е из системы, а затем опять войдите в нее.

6.1.4 Текстовый редактор *ed*

Редактор *ed* ориентирован на построчное редактирование текстовой информации. При этом под строкой понимается текст, введенный до нажатия клавиши `<Enter>`. Запуск редактора выполняется командой *ed* с параметром, в качестве которого выступает имя редактируемого текстового файла. Так как *ed* весьма старый редактор, то простое имя файла не должно иметь длину более 14 символов. При этом, как и в любой команде *UNIX*, команда *ed* отделяется от параметра одним или несколькими пробелами. Пример:

```
UNIX   →   $
польз. →   ed letter<Enter>
ed     →   ?letter
```

В данном примере файл *letter* новый и в нем еще ничего нет. Поэтому в ответном сообщении редактора присутствует символ “?”. Если бы мы задали имя уже существующего файла, то *ed* сообщил бы длину редактируемого файла в байтах. При этом последняя строка могла бы выглядеть, например, так:

```
ed → 1234
```

Допустим пока, что мы создали новый (пустой) файл. Для того, чтобы поместить в него текст, необходимо перейти в режим добавления. Это делается командой *a* редактора. Пример:

```
Польз   → a<Enter>
ed       →   ответа нет; переход на новую строку
польз   → This is a test to see if I am<Enter>
entering text in the file "letter".<Enter>
```

```
Once I have completed it I shall find<Enter>
that I have created 4 new lines of  data<Enter>
<Enter>
```

Обратите внимание на точку в последней строке. Она представляет собой команду выхода из режима добавления. Другого способа выхода из этого режима нет. Так как только одиночная точка в строке будет воспринята редактором как команда, а не как часть вводимого текста.

После того, как текст файла набран, его можно выводить на экран и редактировать. Для вывода файла используется команда *p* редактора. Вывод файла можно выполнять построчно или в виде группы строк. Для построчного вывода редактору передается номер первой требуемой строки, в ответ, на что *ed* выводит текст этой строки на экран. Далее вывод каждой следующей строки файла предваряется нажатием клавиши *<Enter>*. Пример:

```
Польз    →1p<Enter>
ed        →This is a test to see if I am
польз    →<Enter>
ed        →entering text in the file "letter".
Польз    →<Enter>
ed        →Once I have completed it I shall find
Польз    →<Enter>
ed        →that I have created 4 new lines
Польз    →<Enter>
ed        →?
```

Символ “?” означает, что текст файла закончился.

Для вывода на экран группы строк, в редактор передаются одновременно номер первой и номер последней выводимой строки файла, разделенные запятой. При этом для обозначения последней строки файла можно использовать символ *\$*. Пример:

```
польз.   → 1,$p<Enter>
ed        → This is a test to see if I am
entering text in the file "letter".
Once I have completed it I shall find
that I have created 4 new lines of  data
```


Для выполнения редактирования файла, во-первых, устанавливается требуемая *текущая строка*, а во-вторых, выполняется требуемая операция редактирования. Для установки требуемой текущей строки можно использовать построчный вывод текста. Получив на экране требуемую строку, следует ввести одну из команд редактирования: *a* – добавление текста после текущей строки; *i* – добавление текста перед текущей строкой; *c* – замена текущей строки текстом новой строки (строк); *d* – удаление текущей строки. Пример:

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

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

```
польз.   → 3p<Enter>
          ed → Once I have completed it I shall find
польз    → i<Enter>
I am now inserting two lines of<Enter>
text to demonstrate how it works.<Enter>
.<Enter>
```

Пример замещения текста текущей строки текстом другой строки:

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

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

```

польз.   →   3p<Enter>
ed   →   I am now inserting two lines of
польз.   →   d<Enter>
польз.   →   d<Enter>

```

Одной командой можно удалить целую группу строк. В следующем примере производится удаление всего содержимого файла:

```

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

```

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

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

```

польз.   →q<Enter>
UNIX     →$

```

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.

6.1.5 Работа в среде *Midnight Commander*

1. Представление на экране файловой структуры. Утилита *Midnight Commander* предназначена для того, чтобы предоставить пользователю *UNIX* удобный интерфейс для общения с этой системой. Это обеспечивается, во-первых, наглядным выводом на экран информации о файловой структуре системы. Во-вторых, *Midnight Commander* существенно упрощает для пользователя ввод команд *UNIX*.

Для того, чтобы запустить *Midnight Commander*, достаточно набрать команду *UNIX* – *mc*. Часто эту команду включают в командный файл, выполняемый после загрузки системы, и поэтому она выполняется автоматически.

В любом случае в верхней части экрана появляются две серые панели, каждая из которых содержит перечень файлов, «зарегистрированных» в одном из каталогов файловой структуры системы. При этом в заголовке каждой панели указано имя каталога. Ниже располагается командная строка *UNIX* с обычным ее приглашением и мерцающим курсором. В последней строке экрана находится список клавиш $\langle F1 \rangle$ – $\langle F10 \rangle$ с кратким обозначением их функций.

Каталоги, отображаемые на левой и правой панелях, могут совпадать или не совпадать, но в любом случае одна из панелей, называемая активной, отображает текущий каталог. Заголовок активной панели выделяется белым цветом. Кроме того, одно из имен файлов на активной панели выделено псевдокурсором. В отличие от обычного курсора (он находится в командной строке *UNIX*) *псевдокурсор* генерируется не аппаратурой, а программой (в графическом режиме экрана). Переключение активной панели производится нажатием клавиши $\langle Tab \rangle$.

Перемещение псевдокурсора внутри активной панели производится с помощью клавиш управления курсором – вниз, вверх, влево, вправо. Нажатие клавиши $\langle End \rangle$ приводит к установке псевдокурсора на последнюю, а $\langle Home \rangle$ – на первую строку панели. Щелчком левой клавиши мыши можно установить псевдокурсор в любую позицию не только активной, но и соседней панели (происходит смена активной панели).

В общем случае панель содержит строки трех типов:

- 1) строку “..”, обозначающую выход в “родительский каталог” данного каталога;
- 2) строки с именами подкаталогов данного каталога;
- 3) строки с именами файлов данного каталога.

В последней строке панели, как правило, записано имя выделенного файла, его длина, дата и время создания или последней модификации.

Для того, чтобы перейти на подкаталог текущего каталога, достаточно установить на него псевдокурсор и нажать $\langle Enter \rangle$. Для возврата в родительский каталог требуется установить псевдокурсор на “..” и нажать $\langle Enter \rangle$. Перемещаясь подобным образом вверх-вниз по файловой структуре логического диска, можно сделать текущим любой каталог на нем.

Прекращение работы *Midnight Commander* происходит в результате нажатия клавиши $\langle F10 \rangle$. В ответ на появившийся затем вопрос о намерении прекратить работу следует нажать $\langle Enter \rangle$.

2. Команды для работы с файлами. *Midnight Commander* предоставляет пользователю команды для работы с файлами, намного более удобные, чем соответствующие команды *UNIX*. Рассмотрим их.

Создание каталога. Для этого достаточно установить в заголовке активной панели “родительский” каталог по отношению к вновь создаваемому каталогу, а затем нажать клавишу **<F7>**. На экране появится диалоговое окно с приглашением набрать имя нового каталога. В ответ следует набрать имя каталога прописными или строчными буквами и нажать клавишу **<Enter>**. В результате на активной панели появится имя нового каталога.

Копирование файла. Для этого требуется установить в заголовке одной из панелей “родительский” каталог по отношению к копируемому файлу, а в заголовке другой панели – “родительский” каталог по отношению к файлу-копии. Далее следует установить псевдокурсор на копируемый файл и нажать клавишу **<F5>**. На экране появится диалоговое окно с сообщением о готовности выполнить копирование. В ответ достаточно нажать клавишу **<Enter>**. (Для отмены этой или другой команды следует нажать **<Esc>**.) В результате на второй панели появится имя скопированного файла.

Для того, чтобы создать копию файла в том же каталоге, что и исходный файл, необходимо обеспечить, чтобы старый и новый файл имели разные имена. Для этого следует в диалоговом окне, появившемся после нажатия **<F5>**, набрать имя файла-копии, а уж затем нажать **<Enter>**.

Копирование каталога выполняется аналогично копированию обычного файла данных. При этом копируется все поддерево файловой структуры, начинающееся с данного каталога.

Копирование нескольких “дочерних” файлов (каталогов) текущего каталога можно ускорить, используя выделение группы файлов. Для выделения файла необходимо установить на его имя псевдокурсор и нажать клавишу **<Ins>**. В результате имя файла высвечивается белым цветом, и оно включается в группу. (Для исключения файла из группы необходимо повторить эти же два действия – установку псевдокурсора и нажатие **<Ins>**.) После того, как требуемая группа файлов выделена (в нижней строке панели информация об общем числе выделенных файлов и их общем объеме), ее можно скопировать последовательным нажатием **<F5>** и **<Enter>**.

Уничтожение файла. Для этого требуется установить псевдокурсор на имя уничтожаемого файла и нажать клавишу **<F8>**. На экране появится диалоговое окно с просьбой подтвердить намерение удалить файл. В ответ достаточно нажать **<Enter>** (для отмены – **<Esc>**).

Выделив группу файлов (аналогично выделению при копировании) ее можно уничтожить нажатием <F8> и последующими нажатиями <Enter> в ответ на вопросы *Midnight Commander*.

Переименование файла. Для этого следует установить псевдокурсор на требуемый файл и нажать клавишу <F6>. В появившемся диалоговом окне следует набрать новое имя файла, а затем нажать <Enter>.

Создание текстового файла. Для создания нового файла необходимо запустить с командной строки один из текстовых редакторов, например, *ed* или *vi* (этот редактор встроен в *Midnight Commander*).

Редактирование текстового файла. Для работы с ранее созданным текстовым файлом с помощью редактора *vi* необходимо установить псевдокурсор в каталоге файлов на нужный файл и нажать клавишу <F4>.

Примечание. Нажатие функциональной клавиши <Fi> можно заменить последовательным нажатием двух клавиш – <Esc> и <i>, где *i* – номер функциональной клавиши.

3. Ввод команд UNIX. В отличие от рассмотренных выше операций с файлами, для выполнения которых *Midnight Commander* предоставляет пользователю свои команды, формат остальных команд UNIX остается без изменений. При этом помощь *Midnight Commander* заключается в ускорении набора этих команд.

Крайний случай – пользователь набирает команду в командной строке UNIX полностью вручную, не пользуясь помощью *Commander*. Такой метод используется для ввода коротких или редко используемых команд.

Другой метод позволяет обойтись вообще без записи в командную строку. Установив псевдокурсор на имени исполняемого файла (такое имя начинается с символа «*»), следует нажать <Enter>. Данный метод обычно применяется тогда, когда команда не имеет параметров.

Третий метод заключается в том, что команда UNIX переписывается в командную строку из протокола команд. Протокол – список последних команд (не более 16), сохраненный в *Commander*. Для вывода на экран протокола команд достаточно нажать последовательность клавиш:

<F9> → *Command* → *Command History* → *Ok*

Установив псевдокурсор на требуемую команду в протоколе, следует нажать <Enter>.

Если программа, запускаемая любым способом из *Commander*, выводит какие-то данные на экран, то чтение их будет невозможно из-за присутствия на экране панелей. Для того, чтобы убрать с экрана обе па-

нели, требуется одновременно нажать **<Ctrl>&<O>**. Для восстановления панелей достаточно опять нажать **<Ctrl>&<O>**.

Удаление (восстановление) только левой панели производится нажатием последовательности клавиш:

<F9> → Left → Listing mode → Ok

Удаление (восстановление) только правой панели:

<F9> → Right → Listing mode → Ok

4. Настройка Midnight Commander. *Commander* предоставляет своим пользователям возможность выполнять настройку формата информации, выводимой на экран.

В результате нажатия клавиши **<F9>** в верхней части экрана появляется главное (горизонтальное) меню из пяти пунктов: **Left** (левая), **Files** (файлы), **Commands** (команды), **Options** (параметры), **Right** (правая). Одна из позиций меню выделена псевдокурсором. Для выбора требуемого пункта меню достаточно установить на него псевдокурсор (с помощью клавиш управления курсором) и нажать **<Enter>**. Это же можно сделать щелчком левой клавиши мыши. В результате подобного выбора на экране появится ниспадающее меню, выбор в котором позволит выполнить требуемое действие.

Пункт горизонтального меню **Files** позволяет с помощью своих ниспадающих меню выдавать команды по работе с файлами. Многие из этих команд могут быть введены другим способом – нажатием клавиш **<F1>—<F10>**. Примеры других команд: **Chmod** – установка прав доступа к файлу; **Chown** – замена владельца файла.

В пункте **Command** находится большое количество вспомогательных команд *MC*. Примеры таких команд: **Find File** – поиск файла; **Swap panels** – переключение панелей; **Show directories size** – показ размера каталогов.

Пункты горизонтального меню **Left** и **Right** предназначены для настройки левой и правой панелей соответственно. При этом, установив в ниспадающем меню режим **Listing mode**...режим **Brief** (краткий), мы обеспечим вывод на соответствующей панели лишь одних имен файлов. Задание режима **Full** (полный) позволяет выводить на экран не только имена файлов, но и их основные характеристики (размер в байтах, дату и время создания или последней модификации). **Name, Extension, Time, Size** – группа полей в ниспадающем меню, позволяющая выполнить сортировку имен файлов, выводимых на панель, соответственно по имени, расширению имени, времени создания или модификации, размеру файла. Для интеграции в сетевое пространство предусмотрены пункты **Network link** и **FTP link**.

Пункт горизонтального меню *Options* позволяет настраивать интерфейс *MC*. Например, в ниспадающем меню можно установить режимы: *Layout* – задание информации, выводимой вне панелей; *Learn Keys* – задание функциональных клавиш, используемых для работы с *MC*; *Virtual FS* – задание параметров для установления связи. В этом же ниспадающем меню можно выбрать пункт *Configuration* (конфигурация). На экране появится диалоговое окно, в котором можно установить дополнительные параметры настройки *Commander*.

6.1.6 Задание

Для успешной сдачи работы требуется выполнить наизусть следующие операции:

- 1) войти в *UNIX* с паролем;
- 2) создать трехуровневое поддерево каталогов;
- 3) с помощью *ed* создать в одном из новых каталогов текстовый файл;
- 4) вывести файл на экран;
- 5) выполнить добавление текста в начало, в середину и в конец файла;
- 6) вывести файл на экран;
- 7) произвести переименование файла;
- 8) выполнить копирование файла (исходный файл и файл-копия должны располагаться в разных каталогах);
- 9) удалить созданные файлы и каталоги;
- 10) выполнить операции 2-9 с помощью *Midnight Commander*;
- 11) выйти из *UNIX*.

6.2 Лабораторная работа №2

Дальнейшее знакомство с командами UNIX

6.2.1 Цель работы

Целью выполнения настоящей лабораторной работы является развитие навыков работы в среде *UNIX*: 1) использование в командах *shell* метасимволов и перенаправление ввода-вывода; 2) запуск конвейеров программ; 3) применение в командах *shell* переменных; 4) построение командных файлов; 5) изменение прав доступа к файлам.

6.2.2 Подготовка к выполнению работы

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

- 1) команда вывода строки символов на экран *echo* (п. 2.3.3);
- 2) использование метасимволов и перенаправление ввода-вывода (п. 2.5.2);
- 3) конвейеры программ (п. 2.5.3);
- 4) переменные (п.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;
- 5) командные файлы (п. 2.5.6) – понятие скрипта; способы запуска скрипта; вложенные скрипты; комментарии; позиционные параметры; инициализационный скрипт *.profile*;
- 6) 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.

6.2.3 Задание

Выполнить (желательно наизусть) следующую последовательность действий:

- а) создать два каталога и поместить в один из них четыре текстовых файла, два из которых имеют в своем имени одинаковую символьную последовательность, называемую далее «словом»;
- б) поместить во второй каталог скрипт, имеющий два входных параметра: имя каталога и набор символов. Скрипт выполняет действия:
 - вывод на экран перечня файлов, «дочерних» к заданному каталогу, которые имеют в своем имени заданный набор символов;
 - уничтожение всех остальных файлов заданного каталога;
 - любые другие действия (по вашему желанию);
- в) создать свой инициализационный скрипт, выполняющий действия:
 - здороваются;

- «переделывает» приглашения *shell*;
 - запускает вложенный скрипт, созданный в (б), задав ему в качестве параметров каталог и «слово» из (а);
 - любые другие действия (по вашему желанию);
- г) выйти из *UNIX*, а затем войти вновь с целью демонстрации результатов выполнения инициализационного скрипта.

Примечание 1. При выполнении задания разрешается использовать те средства *shell*, которые рассмотрены в данной и предыдущей работах. В частности нельзя применять управляющие операторы (рассматриваются в следующей работе).

Примечание 2. Для избирательного удаления файлов в (б) рекомендуется использовать команду *rm* с флагом *-i*, предварительно установив права доступа к файлам. При этом, для обеспечения автоматического выполнения *rm*, ее стандартный ввод должен быть переключен на вспомогательный файл, содержащий любой символ кроме “у”.

6.3 Лабораторная работа №3

Управляющие операторы командного языка

6.3.1 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*.

6.3.2 Подготовка к выполнению работы

В начале выполнения данной работы обязательно следует изучить следующие вопросы из теоретической части пособия (п. 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.

6.3.3 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.)

6.4 Лабораторная работа №4

Процессы в UNIX

6.4.1 Цель работы

Целью выполнения настоящей лабораторной работы является развитие навыков работы в среде *UNIX*: 1) управление программными процессами, используя команды *shell*; 2) обмен сообщениями с другими пользователями этой же системы *UNIX*.

6.4.2 Подготовка к выполнению работы

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

- 1) общие сведения о процессах (п. 3.2) – понятие процесса; дерево процессов; имя процесса; получение сжатой информации о процессах с помощью команды *ps* без флагов;
- 2) запуск процессов в фоновом режиме и ожидание завершения дочернего фонового процесса с помощью команды *wait* (п. 2.5.2);
- 3) задержка выполнения текущего *shell* на заданное число секунд с помощью команды *sleep* (п. 2.5.5);
- 4) уничтожение процессов с помощью команды *shell* – *kill* (п. 3.4.2);

- 5) приоритеты процессов (п. 5.4) – приоритеты *CPU*, *PRI* и *NICE*; команда задания приоритета *nice*; получение подробной информации о процессах с помощью команды *ps* с флагом *-l*.

В процессе изучения перечисленных вопросов обязательно следует выполнить примеры, приведенные в пособии.

6.4.3 Сообщения другому пользователю

Рано или поздно у вас могло появиться желание побеседовать с другими пользователями *UNIX*. Для этого, во-первых, желательно ознакомиться со списком пользователей, работающих в данный момент в системе. Это можно сделать с помощью команды *who*. Она выводит список пользователей с указанием для каждого из них имени, терминала и времени входа в систему.

Посылка сообщения на экран другого пользователя может быть выполнена командой *write* с указанием имени получателя сообщения. Например, пусть мы вошли в систему под именем *vlad* и хотим послать сообщение пользователю с именем *sergei*. Для этого набираем команду:

```
$write sergei
```

Ввод этой команды приведет к последствиям, как для нашего терминала, так и для терминала другого пользователя. Во-первых, наш *shell* будет ждать ввода сообщения до тех пор, пока мы не введем конец файла, то есть *<Ctrl>&<D>*. Во-вторых, ввод нами этой команды приведет к появлению на другом экране сообщения:

```
message from vlad tty1      (сообщение от vlad и терминала tty1)
```

Получив эту строку, пользователь *sergei* может поступить двояко. Во-первых, он может перейти к ожиданию собственно сообщения (оно поступит после нажатия нами *<Ctrl>&<D>*). Во-вторых, *sergei* может набрать ответную команду: *write vlad*. Если отправка нашего сообщения (момент нажатия *<Ctrl>&<D>*) совпадет по времени с работой другого пользователя на клавиатуре, то на его экране произойдет наложение двух текстов. Аналогично, наложение двух текстов может произойти и на нашем терминале, если во время набора нами сообщения начнет поступать ответное сообщение. Для предотвращения подобных неприятных ситуаций необходимо, чтобы диалог двух пользователей соответствовал некоторому протоколу.

Протокол – алгоритм диалога двух удаленных партнеров, предотвращающий порчу информации. Например, простейший протокол применения команд *write* заключается в следующем. Допустим, что мы получили на своем экране сообщение «*message from . . .*». Тогда мы будем ждать появления в конце строки одиночной буквы «o», которая означа-

ет, что пришла наша очередь набирать сообщение. Набрав его, мы поместим в конец строки букву «о», а уж затем нажмем `<Ctrl>&<D>`. Если мы хотим прекратить диалог, то набираем в конце строки не одну, а две буквы «о».

Вне зависимости от того, применяем ли мы этот или другой подобный протокол, важно помнить, что за его выполнение *UNIX* не несет никакой ответственности. Выполнение подобного «прикладного» протокола полностью основано на добровольном согласии партнеров диалога соблюдать требования протокола.

Возможно, что в данное время вы не хотите вести какие-то диалоги со своими «соседями» по *UNIX*. Тогда вам следует набрать команду *mesg* с параметром *n*. В результате все сообщения, идущие на ваш терминал, будут отменены до тех пор, пока вы не наберете эту же команду *mesg*, но с параметром *y*.

6.4.4 Задание

Выполните наизусть следующую последовательность действий:

- 1) запуск из командной строки в фоновом режиме скрипта, выполняющего задержку на 10 секунд;
- 2) автоматическое ожидание завершения запущенного скрипта;
- 3) одновременный запуск в фоновом режиме двух одинаковых скриптов, выполняющих бесконечные циклы, с разными приоритетами;
- 4) по истечению нескольких минут получение на экране и объяснение информации о затратах времени ЦП двух запущенных ранее бесконечных процессов;
- 5) уничтожение всех созданных фоновых процессов;
- 6) обмен двумя-тремя сообщениями с другим пользователем, работающим в системе.

Примечание. При выполнении задания 3 можно обеспечить одновременность запуска скриптов, поместив их в общий конвейер (несмотря на отсутствие информационного обмена между скриптами).

6.5 Лабораторная работа № 5

Операции с файлами в программе на языке СИ

6.5.1 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.

6.5.2 Подготовка к выполнению работы

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

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

6.5.3 Задание

Требуется разработать одну из следующих программ (по указанию преподавателя):

- 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;
- б) ввод с клавиатуры содержимого текстового файла, имя которого задается в командной строке, используя стандартные функции.

6.6 Лабораторная работа № 6

Системные вызовы для управления процессами

6.6.1 Цель работы

Целью выполнения настоящей лабораторной работы является получение навыков использования в программах на языке СИ системных вызовов *UNIX*, предназначенных для управления процессами: *fork*, *exec*, *wait*, а также работа с переменными окружения.

6.6.2 Подготовка к выполнению работы

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

- 1) состояния процесса (п. 5.1) [1,2];
- 2) создание процесса (п. 5.2) [1,2];
- 3) переменные окружения (п. 2.5.4) [1,2].

6.6.3 Создание процесса

Любой процесс может порождать дочерние процессы, используя системный вызов *fork*. Его описание:

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

В случае успешного завершения вызова *fork* создается новый процесс, который является почти полной копией своего процесса-отца. При этом он имеет такую же программу, переменные окружения, а также структуры данных ядра, обслуживающие процесс. Единственное различие между процессами в том, что новый процесс имеет другой *pid* (Вспомним, что *pid* – номер процесса, уникальный для всей системы.) Таким образом, результатом выполнения *fork* является параллельное (одновременное) существование в данный момент двух процессов, выполняющих одну и ту же программу, причем в одной и той же ее точке. Эта точка - команда, которая следует в программе за командой вызова подпрограммы *fork*.

Дальнейшее выполнение одной и той же программы процессом-отцом и дочерним процессом различно по той причине, что различается код возврата подпрограммы *fork* в эти два процесса. В процесс-отец возвращается код возврата, который совпадает с *pid* нового процесса, а код возврата в новый процесс равен 0. Поэтому после оператора вызова подпрограммы *fork* программа должна содержать или оператор *if* или

оператор *switch*, выполняющий ветвление программы в зависимости от кода возврата подпрограммы *fork*.

6.6.5 Задание

Требуется разработать программу, выполняющую:

- 1) создание файла и размещение в нем небольшого текста;
- 2) создание двух дочерних процессов, каждый из которых выполняет запись в файл из п. 1 своих переменных окружения, предварительно внося в некоторые из них любые изменения. После этого дочерний процесс завершается;
- 3) дождавшись завершения обоих потомков, вывод результирующего файла на экран;
- 4) выполнение над результирующим файлом одного из действий (по вашему усмотрению):
 - 1) вывод на экран количества слов в файле;
 - 2) вывод на экран строк с заданным ключевым словом.

6.7 Лабораторная работа № 7

Обработка сигналов

6.7.1 Цель работы

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

6.7.2 Подготовка к выполнению работы

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

- 1) синхронизация процессов с помощью сигналов (п. 3.4.1) [1,2] – понятие сигнала; типы сигналов; выдача сигнала процессом;
- 2) терминальное управление процессами (п. 3.4.2) [1,2] – управляющий терминал; сеанс; группа процессов; получение информации о процессах с помощью команды *shell - ps* с флагом *-j*; применение команды *shell - kill* с целью посылки сигнала процессу;
- 3) обработка сигналов (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.

6.7.3 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.

6.7.4 Задание

Разработать одну из следующих программ (по указанию преподавателя).

1. Процесс-отец порождает четыре дочерних процесса, каждый из которых выполняет бесконечный цикл. Далее в течение 10 секунд процесс-отец выводит на экран какое-то сообщение. По истечению 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. При этом каждый из дочерних процессов особым образом реагирует на сигнал *SIGINT*:

- процесс 1 при получении сигнала *SIGINT* выводит сообщение на экран и продолжается;

- процесс 2 обрабатывает сигнал *SIGINT* в бесконечном цикле, выдавая свое сообщение на экран. Данный цикл защищен от воздействия сигнала *SIGQUIT*;

- вся программа процесса 3 защищена от воздействия сигнала *SIGINT*;

- перед входом процесса 4 в бесконечный цикл для него меняется идентификатор сеанса.

Сразу после порождения дочерних процессов процесс-отец завершается.

Далее следует проверить реакцию оставшихся процессов на сигналы *SIGINT* и *SIGQUIT*.

3. Процесс-отец открывает существующий текстовый файл, а затем порождает два дочерних процесса, которые по очереди выводят содержимое этого файла фиксированными порциями по 10 символов, предваряя каждый вывод номером своего процесса. Вывод на экран заканчивается или при достижении конца файла, или по истечению интервала времени в 10 секунд.

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

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

6.8 Лабораторная работа № 8

Управление терминалом

6.8.1 Цель работы

Целью выполнения настоящей лабораторной работы является получение навыков по программному управлению терминалом.

6.8.2 Функции терминальной линии

Напомним, что терминал - устройство, выполняющее ввод и вывод символьной информации. Чаще всего это совокупность клавиатуры (устройство ввода) и экрана (устройство вывода), но могут использоваться и другие символьные устройства. Подобно другим периферийным устройствам, терминал представлен в файловой структуре системы специальным файлом в каталоге */dev*. Примеры имен таких файлов: */dev/console*, */dev/tty03*, */dev/tty*. Последнее имя является «собирательным», позволяя в любой программе обращаться к своему управляющему терминалу стандартным образом.

Если пользователь использует не терминал системы *UNIX*, а работает на терминале другой ЭВМ, соединенной с *UNIX*-системой каналами связи, то *UNIX* предоставляет в распоряжение такого пользователя псевдотерминал, который с точки зрения пользователя и его программных процессов в *UNIX*-системе, выглядит точно также, как и реальный терминал. Поэтому сказанное далее о терминалах справедливо также и для псевдотерминалов.

Драйвер терминала – программный модуль ядра ОС, выполняющий низкоуровневое программное управление терминалом подобно обычному строковому драйверу, обеспечивающему обмен символьной

информацией между программным процессом и устройством (а точнее – с его интерфейсным устройством (контроллером)). Перечислим дополнительные функции программного управления терминалом, которые не обеспечивает драйвер терминала:

1) вывод на экран терминала «эха» символов, вводимых с клавиатуры. «Эхо» - изображение на экране той клавиши, которая была только что нажата. Наличие данной функции обусловлено тем, что устройство ввода (клавиатура) и устройство вывода (экран) никак не связаны между собой на аппаратном уровне;

2) предоставление возможности пользователю выполнять простейшее редактирование информации, вводимой с клавиатуры. Примером является удаление символов, расположенных левее или правее позиции, отмеченной курсором;

3) выдача сигналов (см. работу №7) при нажатии пользователем определенных клавиш, или их комбинаций.

Для выполнения перечисленных функций, между процессом и драйвером терминала включается дополнительный программный модуль, называемый дисциплиной линии. Данное название обусловлено тем, что данный модуль совместно с драйвером терминала образуют терминальную линию - канал связи между процессом и терминалом.

6.8.3 Задание

Требуется разработать программу, выполняющую действия:

1) задает старт-стопное управление выводом на экран. Причем для продолжения вывода может использоваться любой символ;

2) задает новые значения (по вашему выбору) символов стирания *erase*, удаления строки *kill*, прерывания *intr*, временной приостановки выполнения *susp*.

Проверить произведенные установки: а) с помощью утилиты *cat* выполнить вывод на экран достаточно большого текстового файла; б) выполнить редактирование строки символов, используя символы *erase* и *kill*; в) из командной строки запустить один оперативный и один фоновый процесс, выполняющие бесконечные циклы, а затем проверить на них действие сигналов *intr* и *susp*.

Примечание. После выполнения перечисленных действий требуется восстановить прежнее состояние терминала, так как иначе другие процессы «не поймут» новые настройки терминала.

6.9 Лабораторная работа № 9

Датаграммные локальные каналы

6.9.1 Цель работы

Целью выполнения настоящей лабораторной работы является получение навыков создания и использования локальных датаграммных информационных каналов между процессами, используя метод сокетов.

6.9.2 Подготовка к выполнению работы

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

- 1) понятие информационного канала и его характеристики (п. 3.5);
- 2) сокет (п. 3.5.4) [1,2] – понятие сокета; имена сокетов; системные вызовы для реализации датаграммного канала;
- 3) понятие протокола (п. 4.4) [1,2].

6.9.3 Задание

Требуется разработать три процесса, запускаемые из командной строки *UNIX*: процесс-сервер, запускаемый в оперативном режиме и два процесса-клиента, запускаемые в фоновом режиме. Между этими процессами должно существовать датаграммное информационное взаимодействие, обеспечивающее решение одной из следующих задач, указываемой преподавателем:

1. Сервер экрана. Каждый клиент выводит содержимое своего текстового файла на экран, находящийся под управлением процесса-сервера. Размер каждого файла соответствует нескольким экранным строкам, каждая из которых передается в виде одной датаграммы. При этом вывод каждой такой строки на экран сервер предваряет выводом номера клиента.

2. Сервер клавиатуры. Каждый клиент вводит содержимое своего текстового файла с клавиатуры, находящейся под управлением процесса-сервера. При этом каждая датаграмма сервера соответствует одной введенной строке текста. Размер каждого клиентского файла многократно превосходит размер одной строки-датаграммы.

3. Сервер файла для записи. Каждый клиент выводит содержимое своего текстового файла в файл на диске, находящийся под управлением процесса-сервера. При этом каждая датаграмма клиента соответствует одной строке файла. Размер каждого клиентского файла превосходит размер одной строки-датаграммы. Запись каждой такой датаграммы в свой файл сервер предваряет записью номера клиента.

4. Сервер файла для чтения. Каждый клиент выполняет запись в свой текстовый файл на диске информации из файла, находящегося под управлением процесса-сервера. При этом каждая датаграмма сервера соответствует одной строке файла. Размер каждого файла многократно превосходит размер одной строки-датаграммы. Заметим, что в данном задании производится не копирование серверного файла, а распределение его копии между клиентами.

Примечание 1. Первый байт каждой датаграммы клиента должен содержать номер этого клиента. (Датаграммы в заданиях 2 и 4 не содержат другой полезной информации.)

Примечание 2. Ответная датаграмма сервера должна содержать или полезную информацию (задания 2 и 4), или является лишь подтверждением со стороны сервера правильности выполненной им операции (задания 1 и 3).

6.10 Лабораторная работа № 10

Сетевые датаграммные каналы

6.10.1 Цель работы

Целью выполнения настоящей лабораторной работы является получение навыков создания на основе сокетов датаграммных *UDP*-каналов, используемых для связи между удаленными процессами.

В начале выполнения данной работы следует ознакомиться с п. из теоретической части пособия.

6.10.2 Состав и порядок выдачи системных вызовов

При создании и использовании сетевого *UDP*-канала порядок выдачи системных вызовов удаленными частями сетевого приложения (клиентской и серверной) не отличается от порядка выдачи системных вызовов при создании и использовании локального датаграммного канала. Например, если рассмотренную в предыдущей лабораторной работе задачу передачи “эха” реализовать в виде распределенного приложения, то сохранится порядок выдачи системных вызовов, приведенный на рис.100. Единственное отличие, не влияющее на логику работы приложения, заключается в том, что клиентская и серверная части приложения выдают системные вызовы разным ОС, так как выполняются на разных хостах.

Несмотря на то, что состав и порядок выдачи системных вызовов для *UDP*-канала и для локального канала совпадают, некоторые параметры этих вызовов в обоих случаях различны.

6.10.3 Задание

Требуется разработать клиентскую и серверную части, принадлежащие разным приложениям, использующим сетевой *UDP*-канал. При этом серверная часть предназначена для обслуживания запросов, поступающих от клиентских частей, принадлежащих другим студентам, а клиентская часть выполняет выдачу запросов к какому-то чужому серверу.

Функции серверной части:

- 1) запись датаграммы, поступившей от какого-то клиента, а также текущего времени в качестве строки текстового файла, расположенного на своем хосте;
- 2) отправка клиенту ответной датаграммы, имеющей длину не более 100 байтов, которая содержит имя и фамилию владельца сервера, а также, возможно, другую информацию по желанию владельца.

Функции клиентской части:

- 1) отправка одному из «известных» серверов (список серверов на доске) датаграммы-запроса, имеющей длину не более 40 байтов, содержащей имя и фамилию владельца клиентской части;
- 2) прием ответной датаграммы от сервера и вывод ее содержимого на экран.

Порядок выполнения работы:

- 1) выяснение у преподавателя *IP*-адреса своего хоста;
- 2) написание программ клиентской и серверной частей;
- 3) совместная отладка своих клиентской и серверной частей в локальном режиме. В процессе данной отладки клиентская и серверная части располагаются на одном и том же хосте, хотя и используют сетевой *UDP*-канал. При этом *UDP*-модуль (входит в состав ОС), обнаружив в качестве адреса назначения свой собственный *IP*-адрес, никакой отправки датаграммы по сети не выполняет, а направляет ее в указанный локальный сокет;
- 4) отлаженный в локальном режиме сервер запускается в бесконечном цикле до конца занятия в фоновом режиме. При этом владелец сервера записывает на доске адрес сокета сервера в виде пары чисел: (*IP*-адрес; номер порта);
- 5) клиентская часть запускается несколько раз в оперативном режиме с целью получения обслуживания от соответствующего числа «известных» серверов.

В процессе защиты лабораторной работы студент должен, в частности, продемонстрировать обслуживание своей клиентской части со стороны чужого сервера, а также вывести на экран (например, с помощью команды *shell – cat*) результирующий файл своей серверной части.

6.11 Лабораторная работа № 11 Локальные виртуальные соединения

6.11.1 Цель работы

Целью выполнения настоящей лабораторной работы является получение навыков создания и использования локальных виртуальных соединений между процессами, используя метод сокетов.

В начале выполнения данной работы следует ознакомиться с той частью п. из теоретической части пособия, в которой рассматривается вопрос построения локальных виртуальных соединений.

6.11.2 Порядок выдачи системных вызовов

В качестве простейшего примера создания и использования локального виртуального соединения опять рассмотрим получение двумя процессами «эха» символьной строки. В этом примере процесс-клиент посылает другому процессу-серверу любую символьную строку. Далее сервер посылает эту строку обратно клиенту. Получив символьную строку, процесс-клиент выводит ее на экран, сопроводив этот вывод пояснительной надписью. На рис. 63 приведен возможный порядок выдачи системных вызовов во времени.

При выполнении сервером системных вызовов *socket* (создать сокет) и *bind* (связать сокет с адресом), а также клиентом вызова *socket* не существует отличий от применения этих вызовов при работе с локальным датаграммным каналом (см. п.), за исключением – в вызове *socket* в качестве параметра *type* (определяет устойчивость создаваемого канала) задается значение *SOCK_STREAM* – виртуальное соединение.

6.11.3 Задание

Требуется разработать процессы, запускаемые из командной строки *UNIX*: процесс-сервер, запускаемый в оперативном режиме и два процесса-клиента, запускаемые в фоновом режиме. Между этими процессами должны существовать информационные взаимодействия в виде виртуальных соединений, предназначенные для решения той же самой задачи, что и в работе № 9:

1. Сервер экрана. Каждый клиент выводит содержимое своего текстового файла на экран, находящийся под управлением процесса-сервера. Размер каждого файла соответствует нескольким экранным строкам, каждая из которых передается клиентом в виде одного сообщения. Причем вывод каждой такой строки на экран сервер предваряет выводом номера клиента.

2. Сервер клавиатуры. Каждый клиент вводит содержимое своего текстового файла с клавиатуры, находящейся под управлением процесса-сервера. При этом каждое сообщение сервера соответствует одной введенной строке текста. Размер каждого клиентского файла многократно превосходит размер одной строки-сообщения.

3. Сервер файла для записи. Каждый клиент выводит содержимое своего текстового файла в файл на диске, находящийся под управлением процесса-сервера. При этом каждое сообщение клиента соответствует одной строке файла. Размер каждого клиентского файла превосходит размер одной строки-сообщения. Запись каждого такого сообщения в свой файл сервер предваряет записью номера клиента.

4. Сервер файла для чтения. Каждый клиент выполняет запись в свой текстовый файл на диске информации из файла, находящегося под управлением процесса-сервера. При этом каждое сообщение сервера соответствует одной строке файла. Размер каждого файла многократно превосходит размер одной строки-сообщения. Заметим, что в данном задании производится не копирование серверного файла, а распределение его копии между клиентами.

Примечание 1. Первый байт каждого сообщения клиента должен содержать номер этого клиента. (Сообщения клиентов в заданиях 2 и 4 не содержат другой полезной информации.)

Примечание 2. Ответное сообщение сервера должно содержать или полезную информацию (задания 2 и 4), или является лишь подтверждением со стороны сервера правильности выполненной им операции (задания 1 и 3).

6.12 Лабораторная работа № 12

Сетевые виртуальные соединения

6.12.1 Цель работы

Целью выполнения настоящей лабораторной работы является получение навыков создания на основе сокетов виртуальных *TCP*-соединений, используемых для связи между удаленными процессами.

В начале выполнения данной работы следует ознакомиться с п. из теоретической части пособия.

6.12.2 Состав и порядок выдачи системных вызовов

При создании и использовании сетевого *TCP*-канала порядок выдачи системных вызовов удаленными частями сетевого приложения (клиентской и серверной) не отличается от порядка выдачи системных вы-

зовов при создании и использовании локального виртуального канала. Например, если рассмотренную в предыдущей лабораторной работе задачу передачи “эха” реализовать в виде распределенного приложения, то сохранится порядок выдачи системных вызовов, несмотря на то, что клиентская и серверная части приложения выполняются на разных хостах.

Несмотря на то, что состав и порядок выдачи системных вызовов для *TCP*-канала и для локального канала совпадают, некоторые параметры этих вызовов в обоих случаях различны. Далее рассмотрим эти отличия.

Во-первых, вызов *socket*, выполняющий в серверной части приложения создание главного сокета сервера, а в клиентской части – создание сокета клиента, имеет единственное отличие - в качестве первого параметра вызова (*domain*) задается константа *AF_INET*. Ее наличие сообщает ядру ОС о том, что соединяемые информационным каналом процессы выполняются на удаленных ЭВМ, подключенных к сети *Internet*.

6.12.3. Задание

Требуется разработать клиентскую и серверную части, принадлежащие разным приложениям, использующим сетевое виртуальное соединение на основе транспортного протокола *TCP*. При этом серверная часть предназначена для обслуживания запросов, поступающих от клиентских частей, принадлежащих другим студентам, а клиентская часть выполняет выдачу запросов к какому-то чужому серверу.

Функции серверной части:

1) выполнение запросов, поступающих от клиентов, на создание виртуальных соединений;

2) выполнение запросов на обслуживание, поступающих от клиентов по виртуальным соединениям. Содержимое каждого сообщения-запроса клиента добавляется к содержимому учетного текстового файла сервера. В ответ сервер отправляет клиенту по виртуальному соединению содержимое другого своего текстового файла, имеющего длину не менее 100 байтов и содержащего среди прочих данных обязательно имя и фамилию владельца сервера. Длина сообщений сервера – 20 байтов.

Функции клиентской части:

1) отправка одному из «известных» серверов (список серверов на доске) запроса на создание виртуального соединения;

2) отправка по созданному виртуальному соединению серверу сообщения-запроса на обслуживание. Это сообщение имеет длину не бо-

лее 40 байтов и содержит имя и фамилию владельца клиентской части, а также любую прочую информацию;

3) прием от сервера текстового файла и вывод его содержимого на экран.

Порядок выполнения работы:

6) выяснение у преподавателя *IP*-адреса своего хоста;

7) написание программ клиентской и серверной частей;

8) совместная отладка своих клиентской и серверной частей в локальном режиме. В процессе данной отладки клиентская и серверная части располагаются на одном и том же хосте, хотя и используют сетевой *TCP*-канал. При этом *TCP*-модуль (входит в состав ОС), обнаружив в качестве адреса назначения свой собственный *IP*-адрес, никакой отправки сообщения по сети не выполняет, а направляет его в указанный локальный сокет;

9) отлаженный в локальном режиме сервер запускается в бесконечном цикле до конца занятия в фоновом режиме. При этом владелец сервера записывает на доске адрес главного сокета сервера в виде пары чисел: (*IP*-адрес; номер порта);

10) клиентская часть запускается несколько раз в оперативном режиме с целью получения обслуживания от соответствующего числа «известных» серверов.

В процессе защиты лабораторной работы студент должен, в частности, продемонстрировать обслуживание своей клиентской части со стороны чужого сервера, а также вывести на экран (например, с помощью команды *chell – cat*) учетный файл своей серверной части.

7. ИНДИВИДУАЛЬНОЕ ТВОРЧЕСКОЕ ЗАДАНИЕ

7.1. Методические указания по выполнению индивидуального задания

В течение учебного семестра студентам предлагается подготовить реферат по одной из предложенных ниже тем для последующего представления информации студентам своей группы. Желательно, чтобы реферат содержал функциональную схему исследуемой операционной или файловой системы, а также описания ее пользовательского и программного интерфейсов.

Допускается самостоятельная конкретизация студентом предложенной темы курсовой работы, когда вместо конкретной ОС или ФС рассматривается одна из ее подсистем. Например, вместо темы курсо-

вой работы «» допускается выбор темы «подсистема управления процессами» или «подсистема ввода-вывода».

Оформление работы должно соответствовать следующим требованиям:

- 1) результаты исследования оформляются в виде пояснительной записки;
- 2) объем пояснительной записки должен составлять 15-20 страниц;
- 3) на титульном листе указываются фамилия и инициалы студента, преподавателя, тема курсовой работы;
- 4) пояснительная записка представляется в печатном и электронном виде (формат *RTF*, *HTML* или *WORD*).

Защита творческой работы проводится в виде выступления (5-10 минут) с презентацией, оформленной в *Power Point*.

7.2. Задания на творческую работу

- 1) Структура и архитектура Windows server 2008 R2;
- 2) Управление процессами и потоками в Windows server 2008 R2;
- 3) Понятие, инсталляция и администрирование службы каталога Active Directory (на примере Samba, GFARM);
- 4) Структура и архитектура Windows Embedded (версий Windows CE);
- 5) Структура и архитектура Java Desktop System;
- 6) Структура и архитектура FreeBSD, либо TrueBSD, либо OpenBSD;
- 7) Структура и архитектура Windows Mobile (на базе Windows CE);
- 8) Конфигурация DNS (структура DNS и использование DNS для разрешения имен), интеграция DNS с Active Directory, планирование и администрирование DNS;
- 9) Управление маршрутизацией (основные принципы маршрутизации, таблицы маршрутизации, протоколы маршрутизации RIP и OSPF, администрирование маршрутизацией);
- 10) Структура и архитектура IOS от CISCO;
- 11) Структура и архитектура IOS XR от CISCO на основе QNX;
- 12) Структура и архитектура операционных систем QNX в сравнении с VX WORKS;
- 13) Общие сведения о файловых системах FAT(12, 16 и 32), вопросы совместимости и ограничения;
- 14) Общие сведения о файловых системах FATX и VFAT, вопросы совместимости и ограничения;

- 15) Общие сведения о файловых системах EFS и Ext (ext2, ext3 и ext3cow), вопросы совместимости и ограничения;
 - 16) Общие сведения о файловых системах NTFS (Qnx4fs, ReiserFs, Reiser4), вопросы совместимости и ограничения;
 - 17) Общие сведения о файловых системах UFS (UFS2, XFS и ZFS), вопросы совместимости и ограничения;
 - 18) Общие сведения о файловых системах Network File System и Server Message Block, вопросы совместимости и ограничения;
 - 19) Общие сведения о файловых системах GoogleFile System, Hadoop Distributed File System, GmailFS, вопросы совместимости и ограничения;
 - 20) Общие сведения о файловых системах Secure Sell File System и EFS, вопросы совместимости и ограничения;
 - 21) Способы организации и хранения данных в системах MS SQL, MySQL, Postgre SQL, Light SQL;
 - 22) Способы организации и передачи данных в сетях Infiny Band;
 - 23) Способы организации и передачи данных в сетях Ethernet;
 - 24) Сервис DNS: назначение, структура, принципы организации.
- Сервис E-mail: назначение, структура, принципы организации..

ЛИТЕРАТУРА

1. Одиноков В.В., Коцубинский В.П. Операционные системы. – Томск: ТУСУР, 2006. – 342 с.
2. Одиноков В.В., Коцубинский В.П. Операционные системы и сети. – Томск: ТУСУР, 2008. – 395 с.
3. Робачевский А. Операционная система *UNIX*. – СПб.: БХВ – Санкт-Петербург, 1999. – 528 с.
4. Дунаев С. *UNIX SYSTEM V. Releas 4.2: Общее руководство*. – М.: ДИАЛОГ-МИФИ, 1995. – 287 с.
5. Рейчард К., Фостер-Джонсон Э. *Unix: Справочник*. – СПб.: Питер, 2000. – 384 с.
6. Столлингс В. Операционные системы. Внутреннее устройство и принципы проектирования. – М.: Вильямс, 2002. – 848 с.
7. Олифер В.Г., Олифер Н.А. Сетевые операционные системы. – СПб.: Питер, 2001. – 544 с.
8. Соловьев А.А. Программирование на *shell*. – М.: «Финансы и статистика», 1999. – 318 с.

Учебное издание

КОЦУБИНСКИЙ Владислав Петрович

Операционные системы и сети

Учебное пособие

Научный редактор

В авторской редакции

Верстка В.П. Коцубинского


**Отпечатано в Издательстве ТПУ в полном соответствии
с качеством предоставленного оригинал-макета**

Подписано к печати Формат 60×84/16.
Бумага «Снегурочка». Печать Хегох.
Усл. печ. л. 5,46. Уч.-изд. л. 4,95.
Заказ . Тираж экз.



Национальный исследовательский
Томский политехнический университет
Система менеджмента качества
Издательства Томского политехнического университета сертифицирована
NATIONAL QUALITY ASSURANCE по стандарту BS EN ISO 9001:2008



ИЗДАТЕЛЬСТВО  **ТПУ**. 634050, г. Томск, пр. Ленина, 30.
Тел./факс: 8(3822)56-35-35, www.tpu.ru