

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ
Государственное образовательное учреждение высшего профессионального образования
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ТОМСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

А.В. Замятин

**ОПЕРАЦИОННЫЕ СИСТЕМЫ.
Теория и практика**

Издательство
Томского политехнического университета
2011

УДК 681.3.066(075.8)

ББК 32.973-018.2я73

3-26

Замятин А.В.

- 3-26 Операционные системы. Теория и практика: учебное пособие / А.В. Замятин. – Томск: Изд-во Томского политехнического университета, 2011. – 281 с.

В учебном пособии изложены особенности функционирования, назначения и архитектуры современных операционных систем (ОС). В работе отражены: понятие и эволюция операционных систем, архитектурные особенности и классификация ОС по различным критериям, особенности управления процессами и памятью, основы организации файловых систем и некоторые их конкретные примеры, рассмотрены консолидированные серверные системы хранения данных большого объема RAID. Для получения практических навыков работе в операционных системах (на примере систем семейства Unix-Linux) учебное пособие освещает вопросы разработки программных проектов с использованием специализированных утилит, а также по управлению процессами и потоками и средствами их синхронизации.

Предназначено для студентов, обучающихся по направлению 230100 Информатика и вычислительная техника.

УДК 681.3.066(075.8)

ББК 32.973-018.2я73

Рецензенты

Доктор технических наук, декан факультета информатики Томского государственного университета, заведующий кафедрой прикладной информатики
С.П. Сущенко

Кандидат технических наук, заместитель начальника Управления информационных технологий ОАО «Востокгазпром»,
П.М. Острасть

© Замятин А.В., 2011

© Томский политехнический университет, 2011

© Оформление. Издательство ТПУ, 2011

СОДЕРЖАНИЕ

ВВЕДЕНИЕ	7
1. ПОНЯТИЕ И ЭВОЛЮЦИЯ ОПЕРАЦИОННЫХ СИСТЕМ...9	
1.1 Основные понятия, назначения и функции ОС	9
1.2 Эволюция вычислительных и операционных систем.....	12
1.2.1 История развития ОС	12
1.2.2 Основные функции ОС	20
1.2.3 Особенности современного этапа развития ОС	21
1.3 Вопросы для самопроверки	22
2. АРХИТЕКТУРНЫЕ ОСОБЕННОСТИ ОС.	
КЛАССИФИКАЦИЯ ОС	23
2.1 Архитектурные особенности ОС	23
2.1.1 Монолитное ядро	23
2.1.2 Микроядерная архитектура	24
2.1.3 Многоуровневые системы	26
2.1.4 Виртуальные машины	27
2.1.5 Смешанные системы	28
2.2 Классификация ОС	28
2.3 Вопросы для самопроверки	31
3. УПРАВЛЕНИЕ ПРОЦЕССАМИ.....33	
3.1 Мультипрограммирование.....	33
3.1.1 Мультипрограммирование в системах пакетной обработки	34
3.1.2 Мультипрограммирование в системах разделения времени	35
3.1.3 Мультипрограммирование в системах реального времени	36
3.1.4 Мультипрограммирование при мультипроцессорной обработке.....	37
3.1.5 Роль прерываний при мультипрограммировании	40
3.2 Планирование процессов и потоков	46
3.2.1 Понятие процесса и потока.....	46
3.2.2 Создание процессов и потоков.....	49
3.2.3 Управляющие структуры процессов и потоков	50
3.2.4 Состояния процесса.....	53
3.2.5 Критерии планирования.....	56
3.2.6 Цели и свойства алгоритмов планирования.....	57
3.2.7 Виды планирования.....	59
3.2.8 Алгоритмы планирования.....	60
3.3 Взаимодействие и синхронизация процессов и потоков.....	64
3.3.1 Независимые и взаимодействующие вычислительные процессы.....	64

3.3.2	Цели и средства синхронизации	65
3.3.3	Пример необходимости синхронизации	66
3.3.4	Механизмы синхронизации	68
3.3.5	Проблемы синхронизации	75
3.3.6	Механизмы межпроцессного взаимодействия	78
3.4	Вопросы для самопроверки	82
4.	УПРАВЛЕНИЕ ПАМЯТЬЮ	85
4.1	Основные положения	85
4.1.1	Задачи по управлению памятью.....	86
4.1.2	Типы адресации	87
4.2	Распределение памяти	91
4.2.1	Общие принципы управления памятью в однопрограммных ОС	91
4.2.2	Особенности организации управления памятью в мультипрограммных ОС	93
4.2.3	Распределение фиксированными разделами	94
4.2.4	Распределение динамическими разделами	96
4.2.5	Распределение перемещаемыми разделами.....	98
4.2.6	Сегментное распределение	99
4.2.7	Страничное распределение.....	106
4.2.8	Особенности эффективного использования таблиц страниц	111
4.2.9	Сегментно-страничное распределение.....	116
4.3	Вопросы для самопроверки	118
5.	ФАЙЛОВЫЕ СИСТЕМЫ.....	120
5.1	Физическая организация жесткого диска.....	120
5.2	Принципы построения файловой системы	124
5.2.1	Интерфейс файловой системы	124
5.2.2	Функциональная схема организации файловой системы...	125
5.2.3	Типовая структура файловой системы на диске	127
5.2.4	Способы выделения дискового пространства	128
5.2.5	Управление дисковым пространством	132
5.2.6	Размер логического блока.....	133
5.3	Особенности загрузки ОС.....	134
5.4	Файлы и файловая система.....	136
5.4.1	Цели и задачи файловой системы	136
5.4.2	Типы файлов.....	138
5.4.3	Атрибуты файла.....	141
5.4.4	Доступ к файлам	141
5.4.5	Операции над файлами	144

5.4.6	Иерархическая структура каталогов.....	145
5.4.7	Операции над директориями	146
5.5	Особенности организации некоторых файловых систем	147
5.5.1	FAT	147
5.5.2	VFAT	152
5.5.3	NTFS.....	154
5.5.4	HPFS	157
5.5.5	UFS	160
5.6	Дисковые массивы RAID	164
5.7	Вопросы для самопроверки	166
6.	ПРАКТИЧЕСКОЕ ЗАНЯТИЕ № 1. ЗНАКОМСТВО С ОПЕРАЦИОННОЙ СИСТЕМОЙ UNIX.....	168
6.1	Цель работы.....	168
6.2	Задание.....	168
6.3	Основы работы в операционной системе UNIX.....	168
6.3.1	Интерфейс командной строки в системах <i>Unix</i>	168
6.3.2	Основы интерактивной работы в оболочке <i>bash</i>	178
6.3.3	Файловая система	185
6.4	Последовательность выполнения работы	196
7.	ПРАКТИЧЕСКОЕ ЗАНЯТИЕ № 2. ЗНАКОМСТВО СО СТАНДАРТНОЙ УТИЛИТОЙ <i>GNU MAKE</i> ДЛЯ ПОСТРОЕНИЯ ПРОЕКТОВ В ОС <i>UNIX</i>	198
7.1	Цель работы.....	198
7.2	Задание.....	198
7.3	Основы использования утилиты построения проектов <i>Make</i>	198
7.3.1	Основные правила работы с утилитой <i>make</i>	198
7.3.2	Пример практического использования утилиты <i>make</i>	204
7.4	Последовательность выполнения работы	215
8.	ПРАКТИЧЕСКОЕ ЗАНЯТИЕ № 3. ЗНАКОМСТВО С ПОТОКАМИ И ИХ СИНХРОНИЗАЦИЕЙ В ОС <i>UNIX</i>.....	217
8.1	Цель работы.....	217
8.1	Задание.....	217
8.2	Управление потоками.....	217
8.2.1	Программирование потоков	217
8.2.2	Основные функции для работы с потоками.....	217
8.2.3	Запуск и завершение потока	218
8.2.4	Досрочное завершение потока	219
8.2.5	Синхронизация потоков.....	221
8.3	Мьютексы	222

8.3.1	Функции синхронизации потоков с использованием мьютексов	222
8.3.2	Инициализация и удаление объекта атрибутов мьютекса	222
8.3.3	Область видимости мьютекса	223
8.3.4	Инициализация мьютекса	224
8.3.5	Запирание мьютекса	225
8.3.6	Удаление мьютекса.....	226
8.3.7	Компиляция многопоточной программы	230
8.3.8	Особенности отладки многопоточной программы	231
8.3.9	Примеры практической реализации	231
8.4	Последовательность выполнения работы	240
9.	ПРАКТИЧЕСКОЕ ЗАНЯТИЕ № 4. ЗНАКОМСТВО С ПРОЦЕССАМИ, ПЕРЕДАЧЕЙ ДАННЫХ МЕЖДУ ПРОЦЕССАМИ И ИХ СИНХРОНИЗАЦИЕЙ	241
9.1	Цель работы.....	241
9.2	Задание.....	241
9.3	Основы оперирования процессами в оболочке bash.....	241
9.3.1	Задания и процессы	241
9.3.2	Передний план и фоновый режим	242
9.3.3	Перевод заданий в фоновый режим и уничтожение заданий	243
9.3.4	Приостановка и продолжение работы заданий	245
9.4	Механизмы межпроцессного взаимодействия в ОС Unix.....	247
9.4.1	Очереди сообщений.....	255
9.4.2	Работа с разделяемой памятью.....	257
9.4.3	Примеры практической реализации	259
9.5	Последовательность выполнения работы	278
	ЛИТЕРАТУРА.....	280

Введение

С каждым годом использование вычислительных систем становится все более широким, существенно повышая эффективность функционирования самых различных бизнес-процессов больших и малых предприятий, облегчая и делая более разнообразной и интересной работу с персональными компьютерами рядовых пользователей по всему миру. Во многом, эта заслуга операционной системы (ОС) – неотъемлемой части любой современной вычислительной системы от простого и недорогого пользовательского компьютера до мощного суперкомпьютера с десятками процессоров.

В связи с вышеизложенным, очевидна ведущая роль современных ОС и особое внимание, которое следует уделять их изучению. Предлагаемое учебное пособие направлено в некоторой степени на решение этой задачи. В нем сделана попытка изложить самые основные элементы особенностей функционирования, назначения и архитектуры современных ОС без существенной «привязки» к какой-либо конкретной ОС и лишних деталей функционирования, усложняющих процесс понимания.

В пособии изложены общетеоретические вопросы построения и функционирования ОС: базовые понятия ОС, основные этапы эволюции ОС, составляющие функционирования мультипроцессной системы – мультипрограммирование, планирование процессов и потоков и механизмы их взаимодействия и синхронизации, основные задачи по управлению памятью, особенности реализации распределения оперативной памяти и физической организации жесткого диска, базовые принципы построения типовой файловой системы, особенности загрузки ОС и организация консолидированных серверных систем хранения данных большого объема RAID различного уровня.

Значительная часть пособия посвящена приобретению практических навыков работы в ОС. На сегодняшний день существует множество различных ОС, построенных на закрытой (ОС семейства *Windows*) и открытой архитектурах (ОС семейства *Unix*). При этом, благодаря своему «открытому» характеру, именно последние более подходят для изучения базовых аспектов функционирования ОС.

Unix-подобные ОС являются достаточно популярными уже более трех десятилетий, что для ОС – очень серьезный срок. Несмотря на большое число разновидностей систем *Unix*, все их объединяет ряд основных черт, таких как язык высокого уровня *C*, положенный в основу кода всей системы, наличие стандартов в архитектуре и интерфейсных решений (*POSIX*, *System V*), использование единой, легко обслуживаемой иерархической файловой системы, различных дополнительных средств, включая средства, предназначенные для упрощения сборки программных проектов.

Современные ОС, включая ОС семейства *Unix*, поддерживают многозадачность и многопоточность. При многопоточности в системе одновременно может работать несколько задач, и каждая из задач может быть разбита на подзадачи, выполняемые параллельно. Для этого в современных ОС предусмотрен ряд стандартных механизмов, позволяющих достаточно эффективно обеспечить синхронизацию процессов и передачу данных между ними, что дает возможность решать задачи и подзадачи как независимо, так и с кооперацией между ними.

В рамках предлагаемого учебного пособия предоставляется возможность получить практические навыки работы в одной из разновидностей ОС *Unix* – *Linux* на примере решения практических задач: разработка программного проекта с помощью утилиты для автоматизированной сборки программных проектов *GNU make*; разработка многопоточного приложения и разработка многопроцессного приложения с использованием стандартных межпроцессных средств синхронизации и передачи данных между клиентом и сервером.

Пособие предназначено для студентов направления 230100 «Информатика и вычислительная техника», однако оно может быть полезно и студентам других специальностей, в образовательных программах которых присутствует дисциплина «Операционные системы», а также широкому кругу подготовленных пользователей, желающих углубить свои познания в области ОС. Более глубокие знания основных принципов организации и функционирования ОС позволят обоснованно принимать решения по приобретению того или иного вычислительного оборудования и программного обеспечения, организовывать его эффективную и надежную эксплуатацию, принимать решения о необходимости его частичного обновления или замены.

1. ПОНЯТИЕ И ЭВОЛЮЦИЯ ОПЕРАЦИОННЫХ СИСТЕМ

1.1 Основные понятия, назначения и функции ОС

Для того, чтобы ответить на вопрос, что представляет собой операционная система, необходимо сначала рассмотреть вопрос, из чего состоит вычислительная система (ВС) в целом. Обобщенно структура вычислительной системы представлена на рис. 1.



Рисунок 1 – Пользователь и обобщенная структура вычислительной системы

Во-первых, ВС состоит из того, что называют *аппаратным* или *техническим обеспечением* (англ. *hardware*): процессоры, память, мониторы, дисковые устройства, накопители на магнитных лентах, сетевая коммуникационная аппаратура, принтеры и т.д., объединенные магистральным соединением (шиной).

Во-вторых, ВС состоит из *программного обеспечения* (ПО), в котором выделяют две части – *системное* и *прикладное*. Системное ПО – это набор программ, которые управляют компонентами ВС, такими как процессор, коммуникационные и периферийные устройства, и предназначены для обеспечения функционирования и работоспособности си-

системы в целом. Большинство из них отвечают непосредственно за контроль и объединение в единое целое различных компонентов аппаратного оборудования ВС, обеспечение работы компьютера самого по себе и выполнение различных прикладных программ. Системное ПО противопоставляется прикладному ПО, которое напрямую решает проблемы пользователя и предназначено для выполнения определенных пользовательских задач и рассчитано на непосредственное взаимодействие с пользователем. К прикладному ПО, как правило, относят разнообразные вспомогательные программы (игры, текстовые процессоры и т.п.).

Следует отметить, что деление на прикладное и системное ПО является отчасти условным и зависит от того, кто осуществляет такое деление. Так, обычный пользователь, неискушенный в программировании, может считать текстовый процессор *Microsoft Word* системной программой, а с точки зрения программиста, это – приложение. Компилятор языка *C* для обычного программиста – системная программа, а для системного – прикладная.

Принимая во внимание вышеизложенное, следует отметить, что операционная система является фундаментальным компонентом системного программного обеспечения. Именно эта часть ПО будет основным предметом детального рассмотрения далее.

Очевидно, что операционная система является основным компонентом любой вычислительной системы и во многом определяет эффективность ее функционирования в целом. При этом, дать однозначное определение операционной системе затруднительно. Главным образом это связано с тем, что операционная система выполняет целый ряд разнородных функций, начиная от обеспечения пользователю-программисту удобств посредством предоставления удобного интерфейса к аппаратной части вычислительной системы и заканчивая обеспечением рационального управления ресурсами вычислительной системы. В связи с этим целесообразно дать несколько различных определений и сделать акцент на цели создания операционных систем, их функции и предназначение.

Главными целями разработчиков операционных систем являются следующие:

1. Эффективное использование всех компьютерных ресурсов.
2. Повышение производительности труда программистов.
3. Простота, гибкость, эффективность и надежность организации вычислительного процесса.
4. Обеспечение независимости прикладного ПО от аппаратного ПО.

Операционная система (ОС) – это программа, которая обеспечивает возможность рационального использования оборудования компьютера удобным для пользователя образом.

ОС – базовый комплекс компьютерных программ, обеспечивающий управление аппаратными средствами компьютера, работу с файлами, ввод и вывод данных, а также выполнение прикладных программ и утилит.

Кроме различных определений ОС, два из которых приведены выше, пользователи выделяют ряд различных «точек зрения» на ОС:

- ОС как виртуальная машина;
- ОС как система управления ресурсами;
- ОС как защитник пользователей и программ;
- ОС как постоянно функционирующее ядро.

Для более полного представления об ОС рассмотрим основные «точки зрения» пользователей более подробно.

ОС как виртуальная машина. Использование архитектуры персонального компьютера на уровне машинных команд является крайне неудобным для использования прикладными программами. Так, работа с диском предполагает знание внутреннего устройства его электронного компонента – контроллера для ввода команд вращения диска, поиска и форматирования дорожек, чтения и записи секторов и т.д. Работа по организации прерываний, работы таймера, управления памятью и т. д. также может требовать при программировании знания и учета большого количества деталей.

В связи с этим необходимо обеспечить интерфейс между пользователем и компьютером, скрывая лишние подробности за счет использования относительно простых и высокоуровневых абстракций. Например, представлять информационное пространство диска как набор файлов, которые можно открывать для чтения или записи, использовать для получения или сброса информации, а затем закрывать, создавать иллюзию неограниченного размера операционной памяти, числа процессоров и прочее. Обеспечением такого высокоуровневого абстрагирования занимается ОС, что позволяет представлять ее пользователю в виде *виртуальной машины*, с которой проще иметь дело, чем непосредственно с оборудованием компьютера.

ОС как система управления ресурсами. В случае, если несколько программ, работающих на одном компьютере, будут пытаться одновременно осуществлять вывод на принтер, то можно получить «мешанину» строчек и страниц. ОС должна предотвращать такого рода хаос за счет буферизации подобной информации и организации очереди на печать.

Не менее актуальная проблема – проблема управления ресурсами для многопользовательских компьютеров.

Таким образом, ОС как *менеджер ресурсов* осуществляет упорядоченное и контролируемое распределение процессоров, памяти и других ресурсов между различными программами.

ОС как защитник пользователей и программ. Если в вычислительной системе требуется обеспечение совместной работы нескольких пользователей, то возникает проблема организации их безопасной деятельности. Так, необходимо обеспечить:

- сохранность информации на диске, защиту от повреждения или удаления файлов;
- разрешение программам одних пользователей произвольно вмешиваться в работу программ других пользователей;
- пресечение попыток несанкционированного использования вычислительной системы.

Эти задачи, как правило, возложены на ОС как организатора безопасной работы пользователей и их программ.

ОС как постоянно функционирующее ядро. Можно говорить об ОС, как о программе (программах), постоянно работающей на компьютере и взаимодействующей с множеством прикладных программ. Очевидно, что такое определение верно лишь отчасти, т.к. во многих современных ОС постоянно работает на компьютере лишь часть ОС, которую принято называть ее ядром.

Учитывая рассмотренное многообразие точек зрения на ОС, целесообразно выполнить обзор предназначений и функций ОС, для чего, в свою очередь, стоит рассмотреть эволюцию развития вычислительных систем в целом и операционных систем, в частности.

1.2 **Эволюция вычислительных и операционных систем**

1.2.1 История развития ОС

Поколения ОС, так же как и аппаратные средства связаны с достижениями в области создания электронных компонентов: ламп (1-е поколение), транзисторов (2-е поколение), интегральных микросхем (ИС, 3-е поколение), больших и сверхбольших интегральных схем (БИС и СБИС, 4-е и 5-е поколения). Рассмотрим эволюцию ОС более подробно.

Первое поколение (1940-е – 50-е гг.). В эти годы ОС отсутствуют. Первые шаги в области разработки электронных вычислительных машин были предприняты в конце Второй мировой войны. В середине 40-х гг. были созданы первые ламповые вычислительные устройства и появился принцип программы, хранящейся в памяти машины (*John Von*

Neumann, июнь 1945 г.). В то время одна и та же группа людей участвовала и в проектировании, эксплуатации и в программировании вычислительной машины. Это была скорее научно-исследовательская работа в области вычислительной техники, а не регулярное использование компьютеров в качестве инструмента решения каких-либо практических задач из других прикладных областей. Программирование осуществлялось исключительно на машинном языке. Об ОС не было и речи, все задачи организации вычислительного процесса решались вручную каждым программистом с пульта управления. За пультом мог находиться только один пользователь. Программа загружалась в память машины в лучшем случае с колоды перфокарт, а обычно с помощью панели переключателей.

Вычислительная система выполняла одновременно только одну операцию (ввод-вывод или собственно вычисления). Отладка программ велась с пульта управления с помощью изучения состояния памяти и регистров машины. В конце этого периода появляется первое системное программное обеспечение: в 1951–1952 гг. возникают прообразы первых компиляторов с символических языков (*Fortran* и др.), а в 1954 г. *Nat Rochester* разрабатывает Ассемблер для *IBM-701*.

Существенная часть времени уходила на подготовку запуска программы, а сами программы выполнялись строго последовательно. Такой режим работы называется последовательной обработкой данных. В целом первый период характеризуется крайне высокой стоимостью вычислительных систем, их малым количеством и низкой эффективностью использования.

Второе поколение (1950-е – 60-е гг.). С середины 50-х гг. начался следующий период в эволюции вычислительной техники, связанный с появлением новой технической базы – полупроводниковых элементов. Применение транзисторов вместо часто перегоравших электронных ламп привело к повышению надежности компьютеров. Теперь машины непрерывно могут работать достаточно долго, чтобы на них можно было возложить выполнение практически важных задач. Снижается потребление вычислительными машинами электроэнергии, совершенствуются системы охлаждения. Размеры компьютеров уменьшились. Снизилась стоимость эксплуатации и обслуживания вычислительной техники. Началось использование ЭВМ коммерческими фирмами. Одновременно наблюдается бурное развитие алгоритмических языков (*LISP*, *COBOL*, *ALGOL-60*, *PL-1* и т.д.). Появляются первые настоящие компиляторы, редакторы связей, библиотеки математических и служебных подпрограмм. Упрощается процесс программирования. Пропадает необходимость вваливать на одних и тех же людей весь процесс разработки и использования компьютеров. Именно в этот период происходит разде-

ление персонала на программистов и операторов, специалистов по эксплуатации и разработчиков вычислительных машин.

Изменяется процесс отладки программ. Теперь пользователь приносит программу с входными данными в виде колоды перфокарт и указывает необходимые ресурсы. Такая колода получает название *задания*. Оператор загружает задание в память машины и запускает его на исполнение. Полученные выходные данные печатаются на принтере, и пользователь получает их обратно через некоторое (довольно продолжительное) время.

Смена запрошенных ресурсов вызывает приостановку выполнения программ, в результате процессор часто простаивает. Для повышения эффективности использования компьютера задания с похожими ресурсами начинают собирать вместе, создавая пакет заданий.

Появляются первые системы *пакетной обработки*, которые просто автоматизируют запуск одной программы из пакета за другой и тем самым увеличивают коэффициент загрузки процессора. При реализации систем пакетной обработки был разработан формализованный язык управления заданиями, с помощью которого программист сообщал системе и оператору, какую работу он хочет выполнить на вычислительной машине. Системы пакетной обработки стали прообразом современных ОС, они были первыми системными программами, предназначенными для управления вычислительным процессом.

Следует отметить основные недостатки, присущие вычислительным системам второго поколения:

1. Использование части машинного времени (времени процессора) на выполнение системной управляющей программы.

2. Программа, получившая доступ к процессору, обслуживается до ее завершения. При этом, если возникает потребность в передаче данных между внешними устройствами и памятью, то процессор простаивает, ожидая завершения операции обмена. С другой стороны, при работе процессора простаивают внешние устройства. Для персонального компьютера проявление фактора простоя процессора не столь существенно, так как стоимость его не велика, чего не скажешь о больших и дорогих ЭВМ.

Третий период развития вычислительных машин относится к началу 1960 – 70 гг. В это время в технической базе произошел переход от отдельных полупроводниковых элементов типа транзисторов к интегральным микросхемам. Вычислительная техника становится более надежной и дешевой. Растет сложность и количество задач, решаемых компьютерами. Повышается производительность процессоров.

Повышению эффективности использования процессорного времени мешает низкая скорость работы механических устройств ввода-вывода.

Вместо непосредственного чтения пакета заданий с перфокарт в память начинают использовать его предварительную запись, сначала на магнитную ленту, а затем и на диск. Когда в процессе выполнения задания требуется ввод данных, они читаются с диска. Точно так же выходная информация сначала копируется в системный буфер и записывается на ленту или диск, а печатается только после завершения задания. Вначале действительные операции ввода-вывода осуществлялись в режиме *off-line*, т.е. с использованием других, более простых, отдельно стоящих компьютеров. В дальнейшем они начинают выполняться на том же компьютере, который производит вычисления (уже в режиме *on-line*). Такой прием получает название *spooling* (сокращение от *Simultaneous Peripheral Operation On Line*) или подкачки-откачки данных. Введение техники подкачки-откачки в пакетные системы позволило совместить реальные операции ввода-вывода (в основном – печати) одного задания с выполнением другого задания, но, в то же время потребовало разработки аппарата прерываний для извещения процессора об окончании этих операций.

Магнитные ленты обеспечивали *последовательный доступ* (информация считывалась с них в том порядке, в каком была записана). Появление магнитного диска, для которого не важен порядок чтения информации (реализующего механизм *прямого доступа* к дорожке и *последовательного* – внутри дорожки), привело к дальнейшему развитию вычислительных систем. При обработке пакета заданий на магнитной ленте очередность запуска заданий определялась порядком их ввода. При обработке пакета заданий на магнитном диске появилась возможность выбора очередного выполняемого задания. Пакетные системы начинают заниматься планированием заданий: в зависимости от наличия запрошенных ресурсов, срочности вычислений и т.д. на выполнение выбирается то или иное задание.

Дальнейшее повышение эффективности использования процессора было достигнуто с помощью *мультипрограммирования*. Идея мультипрограммирования заключается в следующем: пока одна программа выполняет операцию ввода-вывода, процессор не простаивает, как это происходило при однопрограммном режиме, а выполняет другую программу. Когда операция ввода-вывода заканчивается, процессор возвращается к выполнению первой программы. При этом каждая программа загружается в свой участок оперативной памяти, называемый разделом, и не должна влиять на выполнение другой программы.

Появление мультипрограммирования требует настоящей революции в строении вычислительной системы. Особую роль здесь играет аппаратная поддержка (многие аппаратные новшества появились еще на

предыдущем этапе эволюции), наиболее существенные особенности которой перечислены ниже.

- Реализация защитных механизмов. Программы не должны иметь самостоятельного доступа к распределению ресурсов, что приводит к появлению привилегированных и непривилегированных команд. Привилегированные команды, например команды ввода-вывода, могут исполняться только операционной системой. Говорят, что она работает в привилегированном режиме. Переход управления от прикладной программы к ОС сопровождается контролируемой сменой режима. Кроме того, это защита памяти, позволяющая изолировать конкурирующие пользовательские программы друг от друга, а ОС – от программ пользователей.

- Наличие прерываний. Внешние прерывания оповещают ОС о том, что произошло асинхронное событие, например завершилась операция ввода-вывода. Внутренние прерывания (сейчас их принято называть исключительными ситуациями) возникают, когда выполнение программы привело к ситуации, требующей вмешательства ОС, например деление на ноль или попытка нарушения защиты.

- Развитие параллелизма в архитектуре. Прямой доступ к памяти и организация каналов ввода-вывода позволили освободить центральный процессор от рутинных операций.

Не менее важна в организации мультипрограммирования роль собственно ОС. Она отвечает за следующие операции:

- Организация интерфейса между прикладной программой и ОС при помощи системных вызовов.

- Организация очереди из заданий в памяти и выделение процессору одному из заданий потребовало планирования использования процессора.

- Переключение с одного задания на другое требует сохранения содержимого регистров и структур данных, необходимых для выполнения задания, иначе говоря, контекста для обеспечения правильного продолжения вычислений.

- Поскольку память является ограниченным ресурсом, нужны стратегии управления памятью, то есть требуется упорядочить процессы размещения, замещения и выборки информации из памяти.

- Организация хранения информации на внешних носителях в виде файлов и обеспечение доступа к конкретному файлу только определенным категориям пользователей.

- Поскольку программам может потребоваться произвести санкционированный обмен данными, необходимо их обеспечить средствами коммуникации.

- Для корректного обмена данными необходимо разрешать конфликтные ситуации, возникающие при работе с различными ресурсами и предусмотреть координацию программами своих действий, т.е. снабдить систему средствами синхронизации.

Мультипрограммные системы обеспечили возможность более эффективного использования системных ресурсов (например, процессора, памяти, периферийных устройств), но они еще долго оставались пакетными. Пользователь не мог непосредственно взаимодействовать с заданием и должен был предусмотреть с помощью управляющих карт все возможные ситуации. Отладка программ по-прежнему занимала много времени и требовала изучения многостраничных распечаток содержимого памяти и регистров или использования отладочной печати.

Параллельно внутренней эволюции вычислительных систем происходила и внешняя их эволюция. До начала этого периода вычислительные комплексы были, как правило, несовместимы. Каждый имел собственную ОС, свою систему команд и т.д. В результате программу, успешно работающую на одном типе машин, необходимо было полностью переписывать и заново отлаживать для выполнения на компьютерах другого типа. В начале третьего периода появилась идея создания семейств программно совместимых машин, работающих под управлением одной и той же ОС. Первым семейством программно совместимых компьютеров, построенных на интегральных микросхемах, стала серия больших машин IBM/360. Разработанное в начале 60-х годов, это семейство значительно превосходило машины второго поколения по критерию цена/производительность. За ним последовала линия мини компьютеров PDP (несовместимых с линией IBM), и лучшей моделью в ней стала PDP-11.

Сила «одной семьи» была одновременно и ее слабостью. Широкие возможности этой концепции (наличие всех моделей: от мини-компьютеров до гигантских машин; обилие разнообразной периферии; различное окружение; различные пользователи) порождали сложную и громоздкую ОС. Миллионы строчек Ассемблера, написанные тысячами программистов, содержали множество ошибок, что вызывало непрерывный поток публикаций о них и попыток исправления. Только в операционной системе OS/360 содержалось более 1000 известных ошибок. Тем не менее, идея стандартизации ОС была широко внедрена в сознание пользователей и в дальнейшем получила активное развитие.

К этому же периоду относится появление первых *операционных систем реального времени* (ОСРВ), в которых ЭВМ применяется для управления техническими объектами, такими, например, как станок, спутник, научная экспериментальная установка, или технологическими процессами, такими, как гальваническая линия, доменный процесс и т.п.

Во всех этих случаях существует предельно допустимое время, в течение которого должна быть выполнена та или иная программа, управляющая объектом. В противном случае может произойти авария: спутник сойдет с орбиты, экспериментальные данные могут быть потеряны, толщина гальванического покрытия не будет соответствовать норме и т.п. Характерным для ОСРВ является обеспечение заранее заданных интервалов времени реакции на предусмотренные события для получения управляющего воздействия – *реактивность*. Поскольку в технологических процессах промедление может привести к нежелательным и даже опасным последствиям, это реализовано за счет проектирования ОСРВ для работы в наихудших условиях (например, при возникновении аварийной ситуации, запускающей все оговоренные заранее программы).

Системное программное обеспечение этого периода решало множество проблем, связанных с защитой результатов работы различных программ, защитой данных в оперативной памяти и других данных. Кроме того, ОС должна управлять новыми устройствами, входящими в состав аппаратного обеспечения. Для решения этих задач системное программное обеспечение сформировалось в сложную систему, требующую для реализации своих возможностей значительных вычислительных ресурсов.

ОС четвертого поколения (1970-80-е гг.) были многорежимными системами, обеспечивающими пакетную обработку, разделение времени, режим реального времени и мультипроцессорный режим. Они были громоздкими и дорогостоящими (например, ОС OS/360 фирмы IBM). Такие ОС, будучи прослойкой, между пользователем и аппаратурой ЭВМ, привели к значительному усложнению вычислительной обстановки. Для выполнения простейшей программы необходимо было изучать сложные языки управления заданием (*Job Control Language – JCL*). К этому периоду относится появление вытесняющей многозадачности (*Preemptive scheduling*) и использование концепции баз данных для хранения больших объемов информации для организации распределенной обработки. Программисты перестали использовать перфокарты и магнитные ленты для хранения своих данных. Вводится приоритетное планирование (*Prioritized scheduling*) и выделение квот на использование ограниченных ресурсов компьютеров (процессорного времени, дисковой памяти, физической (оперативной) памяти).

Появление электронно-лучевых дисплеев и переосмысление возможностей применения клавиатур поставили на очередь решение этой проблемы. Логическим расширением систем мультипрограммирования стали системы *разделения времени* (*time-sharing* системы). В них процессор переключается между задачами не только на время операций ввода-вывода, но и через определенные интервалы времени. Эти пере-

ключения происходят так часто, что пользователи могут взаимодействовать со своими программами во время их выполнения, то есть интерактивно. В результате появляется возможность одновременной работы нескольких пользователей на одной компьютерной системе. У каждого пользователя для этого должна быть хотя бы одна программа в памяти. Чтобы уменьшить ограничения на количество работающих пользователей была внедрена идея неполного нахождения исполняемой программы в оперативной памяти. Основная часть программы находится на диске, и фрагмент, который необходимо в данный момент выполнять, может быть загружен в оперативную память, а ненужный – выкачан обратно на диск. Это реализуется с помощью механизма виртуальной памяти. Основным достоинством такого механизма является создание иллюзии неограниченной оперативной памяти ЭВМ.

В системах разделения времени пользователь получил возможность эффективно производить отладку программы в интерактивном режиме и записывать информацию на диск, не используя перфокарты, а непосредственно с клавиатуры. Появление *on-line*-файлов привело к необходимости разработки развитых файловых систем.

Пятое поколение (с середины 1980-х гг. по н.в.). Период характеризуется уменьшением стоимости компьютеров и увеличением стоимости труда программиста. Появление персональных компьютеров позволило установить компьютер практически каждому пользователю на рабочем столе. Благодаря широкому распространению вычислительных сетей и средств оперативной обработки (работающих в режиме *on-line*), пользователи получают доступ к территориально распределенным компьютерам. Появились микропроцессоры, на основе которых создаются все новые и новые персональные компьютеры, которые могут быть использованы как автономно, так и в качестве терминалов более мощных вычислительных систем. При передаче информации по линиям связи усложняются проблемы защиты информации, шифрования данных. Возникло понятие сетевого компьютера (*Network computer*), способного получать все ресурсы через компьютерную сеть. Понятие файловой системы распространяется на данные, доступные по различным сетевым протоколам.

Число людей, пользующихся компьютером, значительно возросло, что выдвигает требование наличия дружественного интерфейса пользователя, ориентации на неподготовленного пользователя. Появились системы с управлением с помощью меню и элементов графического интерфейса. Начала широко распространяться концепция виртуальных машин. Пользователь более не заботится о физических деталях построения ЭВМ или сетей. Он имеет дело с функциональным эквивалентом компьютера, создаваемым для него операционной системой, представ-

ляющим виртуальную машину. Таким образом, возникла концепция *виртуализации ресурсов* ЭВМ – создание функциональных программно моделируемых эквивалентов реального монопольного ресурса, допускающих их совместное использование многими процессами. Например, мультипрограммирование – виртуализация центрального процессора, буферный ввод-вывод – виртуализация устройств ввода и вывода.

Широкое внедрение получила концепция распределенной обработки данных. Развитием распределенной обработки данных стала технология «*клиент – сервер*», в которой серверный процесс предоставляет возможность использовать свои ресурсы клиентскому процессу по соответствующему протоколу взаимодействия. Название сервера отображает вид ресурса, который предоставляется клиентским системам (сервер печати, сервер вычислений, сервер баз данных, сервер новостей, сервер FTP, сервер WWW и т.д.).

1.2.2 Основные функции ОС

Обзор этапов развития вычислительных и операционных систем позволяет все функции ОС условно разделить на две различные группы – *интерфейсные* и *внутренние*. К интерфейсным функциям ОС относятся:

- управление аппаратными средствами;
- управление устройствами ввода- вывода;
- поддержку файловой системы;
- поддержку многозадачности (разделение использования памяти, времени выполнения);
- ограничение доступа, многопользовательский режим работы, планирование доступа пользователей к общим ресурсам;
- интерфейс пользователя (команды в MS DOS, Unix; графический интерфейс в ОС Windows);
- поддержка работы с общими данными в режиме коллективного пользования;
- поддержка работы в локальных и глобальных сетях.

К внутренним функциям ОС, которые выделились в процессе эволюции вычислительных и операционных систем, следует отнести:

- реализацию обработки прерываний;
- управление виртуальной памятью;
- планирование использования процессора;
- обслуживание драйверов устройств.

1.2.3 Особенности современного этапа развития ОС

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

Во второй половине 90-х годов все производители ОС резко усилили поддержку средств работы с *Internet* (кроме производителей *Unix*-систем, в которых эта поддержка всегда была существенной). Кроме самого стека *TCP/IP* в комплект поставки начали включать утилиты, реализующие такие популярные сервисы *Internet* как *telnet*, *ftp*, *DNS* и *Web*. Влияние *Internet* проявилось и в том, что компьютер превратился из чисто вычислительного устройства в средство коммуникаций с развитыми вычислительными возможностями.

Особое внимание в течение всего последнего десятилетия уделялось корпоративным сетевым ОС. Очевидно, что и в обозримом будущем они продолжают свое развитие. Корпоративная ОС характеризуется единой информационной структурой, включающей пользователей системы с указанием всех доступных им ресурсов, предоставляемых, в частности, другими серверами корпоративной сети. Такие ОС способны устойчиво работать в крупных сетях, характерных для больших предприятий, имеющих отделения в десятках городов и, возможно, в разных странах. Таким сетям органически присуща высокая степень гетерогенности программных и аппаратных средств, поэтому корпоративная ОС должна успешно взаимодействовать с ОС разных типов и работать на различных аппаратных платформах.

На современном этапе развития операционных систем на передний план вышли средства обеспечения безопасности. Это связано с возросшей ценностью информации, обрабатываемой компьютерами, а также с повышенным уровнем угроз, существующих при передаче данных по сетям, особенно по публичным, таким как Интернет. Многие операционные системы обладают сегодня развитыми средствами защиты ин-

формации, основанными на шифрации данных, аутентификации и авторизации.

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

1.3 **Вопросы для самопроверки**

1. Какие основные компоненты входят в обобщенную структуру вычислительной системы?
2. Что такое техническое и программное обеспечение ЭВМ?
3. В чем отличие системного и прикладного программного обеспечения?
4. Какие основные цели преследуют разработчики ОС?
5. Какие определения операционной системы вам известны?
6. Что понимают под ОС как виртуальной машиной?
7. Что понимают под ОС как системой управления ресурсами?
8. Что понимают под ОС как «защитника» пользователей и программ?
9. Что понимают под ОС как постоянно функционирующее ядро?
10. Какие этапы эволюции вам известны? В чем их суть?
12. В чем преимущества пакетной обработки заданий? Что такое spooling?
13. В чем заключается основная идея мультипрограммирования?
14. Какие изменения потребовало мультипрограммирование в строении вычислительной системы? В чем их суть?
15. Какие основные операции при организации мультипрограммирования реализуются ОС?
16. Что такое операционная система реального времени? Какое ее предназначение и основные характеристики?
17. Что такое системы разделения времени?
18. Какие функции ОС относят к интерфейсным, а какие к внутренним?

2. АРХИТЕКТУРНЫЕ ОСОБЕННОСТИ ОС. КЛАССИФИКАЦИЯ ОС

2.1 *Архитектурные особенности ОС*

Рассмотрев эволюцию развития вычислительных и операционных систем, функции ОС «извне», рассмотрим, что представляют собой ОС «изнутри» и какие подходы существуют к их построению.

2.1.1 Монолитное ядро

По сути, ОС – это программа, которую можно реализовать с использованием процедур и функций. Если при этом ОС компонуется как одна программа, работающая в *привилегированном режиме*¹ и использующая быстрые переходы с одной процедуры на другую, не требующие переключения из привилегированного режима в *пользовательский режим*, и наоборот, то такая архитектура построения ОС называется *монолитным ядром* (англ. *monolithic kernel*).

Архитектура «монолитное ядро» характеризуется тем, что:

- каждая процедура может вызвать каждую;
- все процедуры работают в привилегированном режиме;
- все части монолитного ядра работают в одном адресном пространстве;
- ядро «совпадает» со всей ОС;
- сборка (компиляция) ядра осуществляется отдельно для каждого компьютера, при установке, добавлении или исключении отдельных компонент требуется перекомпиляция;
- старейший способ организации ОС.

Архитектура «монолитное ядро» имеют долгую историю развития и усовершенствования и, на данный момент, являются наиболее архитектурно зрелыми и пригодными к эксплуатации. Вместе с тем, монолитность ядер усложняет их отладку, понимание кода ядра, добавление новых функций и возможностей, удаление кода, унаследованного от предыдущих версий. «Разбухание» кода монолитных ядер также повышает требования к объёму оперативной памяти, требуемому для функционирования ядра ОС. Это делает монолитные ядерные архитектуры мало пригодными к эксплуатации в системах, сильно ограниченных по

¹ Аппаратура компьютера должна поддерживать как минимум два режима работы — *пользовательский режим* (*user mode*) и *привилегированный режим*, который также называют режимом ядра (*kernel mode*) или режимом *супервизора* (*supervisor mode*). Подразумевается, что ОС или некоторые ее части работают в привилегированном режиме (с доступом к оборудованию и ресурсам), а приложения — в пользовательском режиме.

объёму ОЗУ, например, встраиваемых системах, производственных микроконтроллерах и т.д.

Старые монолитные ядра требовали перекомпиляции при любом изменении состава оборудования. Следует отметить, что большинство современных ядер позволяют во время работы динамически подгружать модули, выполняющие части функций ядра. Такие ядра называются модульными ядрами. Возможность динамической подгрузки модулей не нарушает монолитности архитектуры ядра, так как динамически подгружаемые модули загружаются в адресное пространство ядра и в дальнейшем работают как интегральная часть ядра. Не следует путать модульность ядра с *гибридной* или *микроядерной* архитектурой (см. ниже).

Примером систем с монолитным ядром служит большинство *Unix*-подобных систем, таких как *BSD*, *Linux* или *NetWare*.

2.1.2 Микроядерная архитектура

При разработке ОС используют подход, при котором значительную часть системного кода переносят на уровень пользователя с одновременной минимизацией ядра. Системы, разработанные с использованием такого подхода, называют реализованными в микроядерной архитектуре (англ. *microkernel architecture*). В этом случае построение ядра ОС осуществляется так, что большинство составляющих ОС являются самостоятельными программами, а взаимодействие между ними обеспечивает специальный модуль ядра – *микроядро*, работающее в привилегированном режиме и обеспечивающее взаимодействие между программами, планирование использования процессора, первичную обработку прерываний, операции ввода-вывода и базовое управление памятью (рис. 2).



Рисунок 2 – Микроядерная архитектура операционной системы

В микроядерных ОС выделяют центральный компактный модуль, относящийся к супервизорной части системы. Этот модуль имеет очень небольшие размеры и выполняет относительно небольшое количество управляющих функций, но позволяет передать управление на другие

управляющие модули, которые и выполняют затребованную функцию. Микроядро – это минимальная главная (стержневая) часть ОС, служащая основой модульных и переносимых расширений. Микроядро само является модулем системного ПО, работающим в наиболее приоритетном состоянии компьютера и поддерживающим связи с остальной частью операционной системы, которая рассматривается как набор серверных приложений (служб).

Основная идея, заложенная в технологию микроядра, заключается в том, чтобы создать необходимую среду верхнего уровня иерархии, из которой можно легко получить доступ ко всем функциональным возможностям уровня аппаратного обеспечения. При этом микроядро является стартовой точкой для создания всех остальных модулей системы. Остальные модули, реализующие необходимые системе функции, вызываются из микроядра и выполняют сервисную роль, получая при этом статус обычного процесса.

Важнейшая задача при разработке микроядра заключается в выборе базовых примитивов, которые должны находиться в микроядре для обеспечения необходимого и достаточного сервиса. В микроядре содержится и исполняется минимальное количество кода, необходимое для реализации основных системных вызовов. В число этих вызовов входят передача сообщений и организация другого общения между внешними по отношению к микроядру процессами, поддержка управления прерываниями, а также ряд других весьма немногочисленных функций. Остальные системные функции, характерные для «обычных» (не микроядерных) операционных систем, обеспечиваются как модульные дополнения-процессы, взаимодействующие главным образом между собой посредством передачи сообщений.

К преимуществам построения ОС в данной архитектуре относят:

- упрощенное добавление и отладка компонентов ОС без необходимости перезапуска системы за счет высокой степени модульности ядра;
- возможность без прерывания работы системы, загружать и выгружать новые драйверы, файловые системы и т.д.
- возможность отладки компонентов ядра с помощью обычных программных средств;
- повышенная надежность системы (ошибка на уровне непривилегированной программы менее опасна, чем отказ на уровне режима ядра).

К недостаткам построения ОС в данной архитектуре относят:

- дополнительные накладные расходы, связанные с передачей сообщений и возникающие за счет частого переключения из защищенного

режима ядра в незащищенный, в котором функционируют остальные модули;

- усложнение процесса проектирования при попытке снижения возможных накладных расходов (требуется «аккуратное» проектирование, разбиение системы на компоненты, минимизация взаимодействия между ними).

2.1.3 Многоуровневые системы

Обеспечивая строгую структуризацию, можно представить всю вычислительную систему в виде ряда уровней с хорошо определенными связями между ними. При этом объекты уровня N могут вызывать только объекты уровня $N-1$. Чем ниже уровень, тем более привилегированные команды и действия может выполнять модуль, находящийся на этом уровне. Впервые такой подход был применен при создании системы *THE* (*Technische Hogeschool Eindhoven*) в 1968 г. Дейкстрой (*Dijkstra*) и его студентами (рис. 3).

5	Интерфейс пользователя
4	Управление вводом-выводом
3	Драйвер устройства связи оператора и консоли
2	Планирование задач и процессов
1	Управление памятью
0	Аппаратное обеспечение

Рисунок 3 – Структура системы *THE*

Вычислительные системы, реализованные в подобной архитектуре, называют *многоуровневыми системами* (англ. *layered systems*).

В качестве достоинства многоуровневых систем отмечают:

- простоту реализации (за счет того, что при использовании операций нижнего слоя не нужно знать, как они реализованы, нужно лишь понимать, что они делают);
- простоту тестирования (отладка осуществляется послойно и при возникновении ошибки всегда легко локализовать ошибку);
- простоту модификации (при необходимости можно заменить лишь один слой, не трогая остальные).

К недостаткам относят:

- сложность разработки (непросто верно определить порядок и состав каждого из слоев);

- меньшая по сравнению с монолитными системами эффективность за счет необходимости прохождения целого ряда слоев (например, для выполнения операций ввода-вывода программе пользователя придется последовательно проходить все слои от верхнего до нижнего).

2.1.4 Виртуальные машины

Виртуальной машиной (англ. *virtual machine*) называют программную или аппаратную среду, исполняющую некоторый код (например, байт-код² или машинный код реального процессора). Зачастую виртуальная машина эмулирует работу реального компьютера. На виртуальную машину, так же как и на реальный компьютер можно устанавливать ОС, у виртуальной машины может быть BIOS, оперативная память, жёсткий диск (выделенное место на жёстком диске реального компьютера), могут эмулироваться периферийные устройства. На одном компьютере может функционировать несколько виртуальных машин. На рис. 4 представлена обобщенная структура некоторой виртуальной машины с тремя различными ОС.

Виртуальная машина реализует для пользователя имитацию *hardware* в вычислительной системе (процессор, привилегированные и непривилегированные команды, устройства ввода-вывода, прерывания и т.д.). При обращении к «виртуальному hardware» на уровне привилегированных команд в действительности происходит системный вызов³ реальной ОС, которая и производит все необходимые действия.

Программа пользователя	Программа пользователя	Программа пользователя
<i>MS-DOS</i>	<i>Linux</i>	<i>Windows NT</i>
Виртуальное hardware	Виртуальное hardware	Виртуальное hardware
Реальная операционная система		
Реальное аппаратное обеспечение		

Рисунок 4 – Обобщенная структура некоторой виртуальной машины

Недостатками реализации ОС в подобных архитектурах является снижение эффективности виртуальных машин по сравнению с реальным

² **Байт-код** (англ. byte-code) – машинно-независимый код низкого уровня, генерируемый транслятором и исполняемый интерпретатором. Большинство инструкций байт-кода эквивалентны одной или нескольким командам ассемблера. Трансляция в байт-код занимает промежуточное положение между компиляцией в машинный код и интерпретацией.

³ **Системный вызов** (англ. system call) – обращение прикладной программы к ядру ОС для выполнения какой-либо операции с использованием привилегированных команд.

компьютером, и, как правило, их громоздкость. Преимуществом является использование в рамках одной вычислительной системы программ, созданных для разных ОС. Примерами ОС, реализованных в подобной архитектуре, являются *CP/CMS (VM/370)* для семейства машин *IBM/370*, *VMWare Workstation* компании *VMWare*.

2.1.5 Смешанные системы

В большинстве случаев современные ОС используют различные комбинации подходов, рассмотренных в пп. 2.1.1-2.1.4, реализуя смешанные (гибридные) ОС. Например, ядро ОС *Linux* представляет собой монолитную систему с элементами микроядерной архитектуры. Системы *4.4BSD* и *MkLinux* – монолитные ОС, работающие на микроядре *Mach* (микроядро обеспечивает управление виртуальной памятью и работу низкоуровневых драйверов; остальные функции, в том числе взаимодействие с прикладными программами, осуществляется монолитным ядром). Совместно элементы микроядерной архитектуры и элементы монолитного ядра используются в ядре *Windows NT*:

- компоненты ядра *Windows NT* располагаются в вытесняемой памяти и взаимодействуют друг с другом путем передачи сообщений, как и положено в микроядерных ОС;
- все компоненты ядра работают в одном адресном пространстве и активно используют общие структуры данных.

2.2 Классификация ОС

В зависимости от выбранного признака, по которому один объект отличают от другого, вариантов классификации может быть множество. Что касается ОС, здесь уже давно сформировалось относительно небольшое количество классификаций: по назначению, по режиму обработки задач, по способу взаимодействия с системой и, наконец, по способам построения (архитектурным особенностям системы).

Прежде всего, традиционно различают ОС *общего* и *специального* назначения. Системы специального назначения, в свою очередь, подразделяются на ОС для носимых микрокомпьютеров и различных встроенных систем, организации и ведения баз данных, решения задач реального времени и т.п. Еще недавно ОС для персональных компьютеров относили к ОС специального назначения. Сегодня современные мультизадачные ОС для персональных компьютеров уже многими относятся к ОС общего назначения, поскольку их можно использовать для самых разнообразных целей.

По режиму обработки задач различают ОС, обеспечивающие *однопрограммный* и *мультипрограммный*⁴ (мультизадачный, многозадачный) режимы. Любая задержка в решении программы (например, для осуществления операций ввода-вывода данных) используется для выполнения других программ. Однозадачные ОС (например, *MS-DOS*, *MSX*) выполняют функцию предоставления пользователю виртуальной машины, делая более простым и удобным процесс взаимодействия пользователя с компьютером, а также включают средства управления периферийными устройствами, средства управления файлами, средства общения с пользователем.

Следует различать понятия «*мультипрограммный режим*» и «*мультизадачный режим*». Принципиальное отличие этих понятий заключается в том, что мультипрограммный режим обеспечивает параллельное выполнение нескольких приложений, и при этом программисты, создающие эти программы, не должны заботиться о механизмах организации их параллельной работы (эти функции берет на себя сама ОС). Мультизадачный режим, наоборот, предполагает, что забота о параллельном выполнении и взаимодействии приложений ложится как раз на прикладных программистов. Современные ОС для персональных компьютеров реализуют мультипрограммный и мультизадачный режимы работы.

Среди множества существующих вариантов реализации многозадачности можно выделить две группы:

1) *Невытесняющая многозадачность* (*NetWare*, *Windows 3.x*). В этом случае активный процесс выполняется до тех пор, пока он сам, по собственной инициативе, не отдаст управление ОС для того, чтобы та выбрала из очереди другой готовый к выполнению процесс.

2) *Вытесняющая многозадачность* (*Windows NT*, *OS/2*, *Unix*). При вытесняющей многозадачности решение о переключении процессора с одного процесса на другой принимается ОС, а не самим активным процессом.

Также многозадачные ОС подразделяют на различные типы в соответствии с использованными при их разработке критериями эффективности:

- *системы пакетной обработки* (например, *ES*, критерий – коэффициент загрузки процессора);
- *системы разделения времени* (*Unix*, *VMS*, критерий – удобство и эффективность работы пользователей при одновременном выполнении нескольких пользовательских приложений);

⁴ Способ организации выполнения нескольких программ одновременно на одном компьютере.

- *системы реального времени (QNX, RT/11, критерий – реактивность).*

Информация о системах пакетной обработки и разделения времени приведена выше в п. 1.2. Как отмечено выше, основной особенностью ОСРВ является обеспечение обработки поступающих заданий в течение заданных интервалов времени, которые нельзя превышать. Поток заданий в общем случае не является плановым и не может регулироваться оператором (характер следования событий можно предсказать лишь в редких случаях), то есть задания поступают в непредсказуемые моменты времени и без всякой очередности. Лучшие характеристики по производительности для систем реального времени обеспечиваются однопользовательскими ОСРВ. Средства организации мультитерминального режима всегда замедляют работу системы в целом, но расширяют функциональные возможности системы. Одной из наиболее известных ОСРВ для персональных компьютеров является ОС QNX [16].

Если принимать во внимание способ взаимодействия с компьютером, то можно говорить о *диалоговых* системах и системах *пакетной обработки*. Доля последних хоть и не убывает в абсолютном исчислении, но в процентном отношении она существенно сократилась по сравнению с диалоговыми системами.

При организации работы с вычислительной системой в диалоговом режиме можно говорить об *однопользовательских* (однопользовательских) и *многопользовательских* (*мультитерминальных*) ОС. В мультитерминальных ОС с одной вычислительной системой одновременно могут работать несколько пользователей, каждый со своего терминала. При этом у пользователей возникает иллюзия, что у каждого из них имеется собственная вычислительная система. Очевидно, что для организации мультитерминального доступа к вычислительной системе необходимо обеспечить мультипрограммный режим работы. В качестве одного из примеров мультитерминальных ОС для персональных компьютеров можно назвать *Linux*. Некая имитация мультитерминальных возможностей имеется и в системе *Windows XP*. В этой ОС каждый пользователь после регистрации (входа в систему) получает свою виртуальную машину. Если необходимо временно предоставить компьютер другому пользователю, вычислительные процессы первого можно не завершать, а просто для этого другого пользователя система создает новую виртуальную машину. В результате компьютер будет выполнять задачи и первого, и второго пользователя. Количество параллельно работающих виртуальных машин определяется имеющимися ресурсами. Главным отличием многопользовательских систем от однопользовательских является наличие средств защиты информации каждого пользователя от несанкционированного доступа других пользователей.

Кроме того, если в ОС отсутствуют или присутствуют средства поддержки многопроцессорной обработки, они могут быть разделены на *многопроцессорные* и *однопроцессорные*. Как правило, функции мультипроцессорирования имеются в операционных системах *Solaris 2.x* фирмы *Sun*, *Open Server 3.x* компании *Santa Crus Operations*, *OS/2* фирмы *IBM*, *Windows NT* фирмы *Microsoft*, *NetWare 4.1* фирмы *Novell*, однако, очевидно, их наличие усложняет алгоритмы управления ресурсами. В свою очередь, многопроцессорные ОС могут классифицироваться по способу организации вычислительного процесса в системе с многопроцессорной архитектурой: *асимметричные* ОС и *симметричные* ОС. Асимметричная ОС целиком выполняется только на одном из процессоров системы, распределяя прикладные задачи по остальным процессорам. Симметричная ОС полностью децентрализована и использует весь пул процессоров, разделяя их между системными и прикладными задачами.

Следует отметить еще один признак, по которому разделяют ОС – организация работы с вычислительной сетью. По этому признаку выделяют *сетевые ОС* и *распределенные ОС* (следует отметить, что иногда в литературе такое разделение отсутствует). Сетевая ОС характеризуется тем, что наделена развитыми функциями работы с сетью, а также контроля доступа к файлам (систему прав доступа). К сетевым ОС относят как системы для рабочих мест (*Novell for DOS*, *MS Windows*, *GNU/Linux*), так серверные ОС (*GNU/Linux*, семейство *BSD*-систем, серверные версии *MS Windows*, *NetWare*), а также специализированные ОС сетевого оборудования (*Cisco IOS*⁵) [17].

При использовании распределенной ОС пользователь не знает, на локальной или удаленной машине хранятся его файлы и выполняются его программы, он может не знать, подключен ли его компьютер к сети. Внешне распределенная ОС выглядит как обычная автономная система, а ее внутреннее строение имеет существенные отличия от автономных систем.

Также ОС классифицируют по архитектуре, в которой они реализованы. Виды архитектур, в которых реализуются ОС, достаточно подробно изложены выше в п. 2.1.

2.3 Вопросы для самопроверки

1. В каких архитектурах реализуют операционные системы?
2. Чем характеризуется ОС, реализованная в архитектуре «монолитное ядро»?

⁵ Первоначально – Internetwork Operating System

3. В чем отличие работы аппаратуры компьютера в пользовательском режиме и в режиме ядра?
4. Какие недостатки вызывает использование ОС в архитектуре монолитное ядро? Что означает модульное ядро?
5. Чем характеризуется ОС, реализованная в микроядерной архитектуре?
6. В чем заключаются достоинства и недостатки ОС, реализованных в микроядерной архитектуре?
7. Чем характеризуется многоуровневая ОС? В чем ее достоинства и недостатки?
8. Какова обобщенная структура виртуальной машины? Назовите примеры современных виртуальных машин?
9. Что такое смешанные (гибридные) ОС? Какие ОС реализованы в этой архитектуре?
10. По каким признакам классифицируют ОС?
11. В чем отличие мультипрограммного режима работы ОС от мультизадачного?
12. Что означают понятия вытесняющая и невытесняющая многозадачность?
13. Что означает симметричная и ассиметричная мультипроцессорная обработка?
14. В чем отличие сетевых и распределенных ОС?

3. УПРАВЛЕНИЕ ПРОЦЕССАМИ

3.1 Мультипрограммирование

Мультипрограммирование – это режим обработки данных, при котором ресурсы вычислительной системы предоставляются каждому процессу из группы процессов обработки данных, находящихся в вычислительной системе, на интервалы времени, длительность и очередность предоставления которых определяется управляющей программой этой системы с целью обеспечения одновременной работы в интерактивном режиме.

Суть мультипрограммного режима работы ОС заключается в том, что пока одна программа (один вычислительный процесс) ожидает завершения очередной операции ввода-вывода (подпись «Вв» на оси ординат), другая программа (процесс) может быть поставлена на решение (рис. 5). Это позволяет более полно использовать имеющиеся ресурсы (например, как видно из рисунка, центральный процессор начинает меньше простаивать) и уменьшить общее (суммарное) время, необходимое для решения некоторого множества задач.

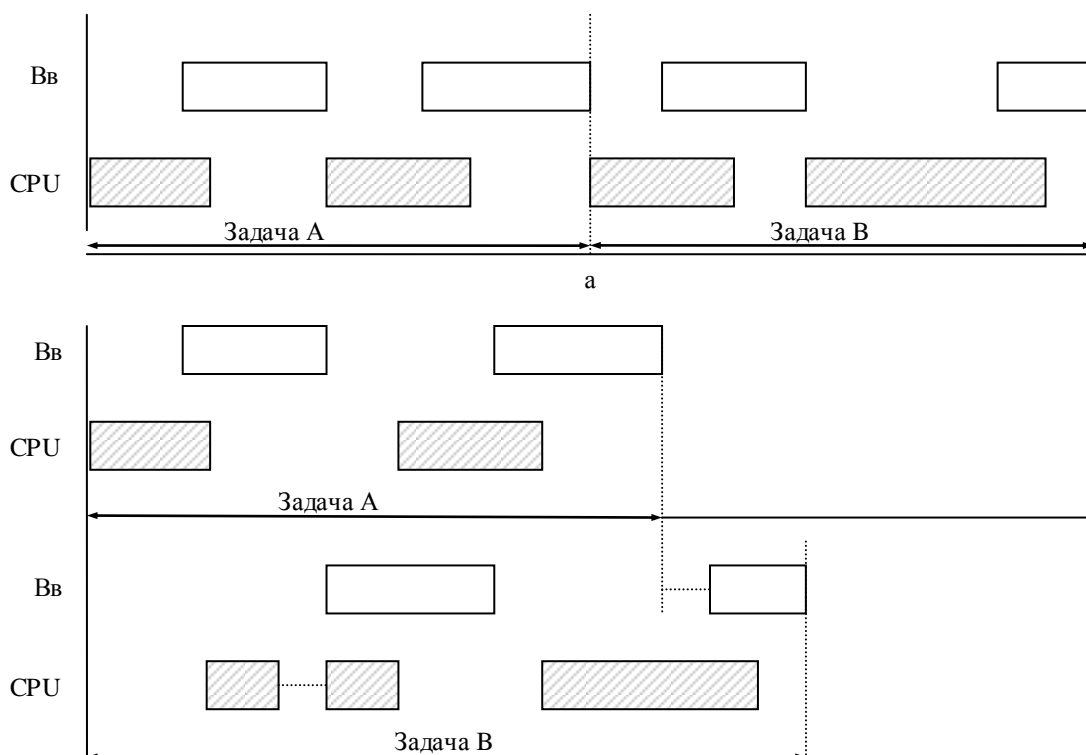


Рисунок 5 – Пример выполнения двух программ в мультипрограммном режиме

На рис. 5 в качестве примера изображена гипотетическая ситуация, при которой благодаря совмещению во времени двух вычислительных

процессов общее время их выполнения получается меньше, чем если бы их выполняли по очереди (запуск одного начинался бы только после полного завершения другого). Видно, что время выполнения каждого процесса в общем случае больше, чем если бы каждый из этих процессов выполнялся как единственный.

При мультипрограммировании повышается пропускная способность системы, но отдельный процесс никогда не может быть выполнен быстрее, чем если бы он выполнялся в однопрограммном режиме (всякое разделение ресурсов замедляет работу одного из участников за счет дополнительных затрат времени на ожидание освобождения ресурса). При мультипрограммировании программы совместно используют не только процессор, но и другие ресурсы компьютера (оперативную и внешнюю память, устройства ввода-вывода, данные). Мультипрограммирование призвано повысить эффективность использования вычислительной системы, однако эффективность может пониматься по-разному. Наиболее характерными критериями эффективности вычислительных систем являются:

- пропускная способность – количество задач, выполняемых вычислительной системой в единицу времени;
- удобство работы пользователей, заключающееся, в частности, в том, что они имеют возможность интерактивно работать одновременно с несколькими приложениями на одной машине;
- реактивность системы – способность системы выдерживать заранее заданные (возможно, очень короткие) интервалы времени между запуском программы и получением результата.

Как было отмечено выше в п. 2.2, в зависимости от выбранного критерия эффективности, ОС делят на системы пакетной обработки, системы разделения времени и системы реального времени, каждая из которых имеет свои особенности реализации мультипрограммирования, причем некоторые ОС могут поддерживать несколько режимов. Учитывая все это, рассмотрим более подробно особенности реализации мультипрограммирования для каждого из рассмотренных типов ОС.

3.1.1 Мультипрограммирование в системах пакетной обработки

Для повышения пропускной способности компьютера стремятся к обеспечению минимизации простоев (например, из-за отсутствия соответствующих данных) всех его устройств, и прежде всего центрального процессора. Естественным решением, ведущим к повышению эффективности использования процессора, является переключение процессора на выполнение другой задачи, у которой есть данные для обработки. Такой подход к реализации мультипрограммирования положен в основу пакетных систем.

Основное предназначение систем пакетной обработки – решать задачи вычислительного характера, не требующие быстрого получения результатов. Главной целью и критерием эффективности систем пакетной обработки является пропускная способность, то есть решение числа задач в единицу времени.

Для достижения этой цели в системах пакетной обработки используется следующая схема функционирования. В начале работы формируется пакет заданий, каждое из которых содержит требование к системным ресурсам. Из пакета заданий формируется мультипрограммная «смесь», то есть множество одновременно выполняемых задач. Для одновременного выполнения выбираются задачи, предъявляющие разные требования к ресурсам, так, чтобы обеспечивалась сбалансированная загрузка всех устройств вычислительной машины. Например, в мультипрограммной смеси желательно одновременное присутствие вычислительных задач и задач с интенсивным вводом-выводом.

Таким образом, выбор нового задания из пакета заданий зависит от внутренней ситуации, складывающейся в системе, то есть выбирается «выгодное» в некотором смысле задание. Следовательно, в вычислительных системах, работающих под управлением пакетных ОС, невозможно гарантировать выполнение того или иного задания в течение определенного периода времени. Кроме того, переключение процессора с выполнения одной задачи на выполнение другой происходит по инициативе самой активной задачи, например, когда она отказывается от процессора из-за необходимости выполнить операцию ввода-вывода. Поэтому существует высокая вероятность того, что одна задача может надолго занять процессор и выполнение интерактивных задач станет невозможным.

Взаимодействие пользователя с вычислительной машиной, на которой установлена система пакетной обработки, сводится к тому, что он приносит задание, отдает его диспетчеру-оператору, а в конце дня после выполнения всего пакета заданий получает результат. Очевидно, что такой порядок повышает эффективность функционирования аппаратуры, но снижает эффективность работы пользователя.

3.1.2 Мультипрограммирование в системах разделения времени

С целью повышения удобства и эффективности работы пользователя применяют другой способ мультипрограммирования – разделения времени. В системах разделения времени пользователю (пользователям) предоставляется возможность интерактивной работы с несколькими приложениями одновременно. В системах разделения времени эта проблема решается за счет того, что ОС принудительно периодически приостанавливает приложения, не дожидаясь, когда они добровольно осво-

бодят процессор. Всем приложениям попеременно выделяется квант процессорного времени, поэтому пользователи, запустившие программы на выполнение, получают возможность поддерживать с ними диалог.

Системы разделения времени призваны исправить основной недостаток систем пакетной обработки – изоляцию пользователя-программиста от процесса выполнения его задач. Каждому пользователю в этом случае предоставляется терминал, с которого он может вести диалог со своей программой. В связи с тем, что в системах разделения времени каждой задаче выделяется только квант процессорного времени, ни одна задача не занимает процессор надолго и время ответа оказывается приемлемым. Если квант выбран небольшой, то у всех пользователей, одновременно работающих на одной и той же машине, складывается впечатление, что каждый из них использует ее единолично.

Очевидно, что системы разделения времени обладают меньшей пропускной способностью, чем системы пакетной обработки, так как на выполнение принимается каждая запущенная пользователем задача, а не та, которая «выгодна» системе. Кроме того, производительность системы снижается из-за возросших накладных расходов вычислительной мощности на более частое переключение процессора с задачи на задачу. Это вполне соответствует тому, что критерием эффективности систем разделения времени является не максимальная пропускная способность, а удобство и эффективность работы пользователя. Вместе с тем мультипрограммное выполнение интерактивных приложений повышает пропускную способность компьютера (пусть и не в такой степени, как пакетные системы). Аппаратура загружается более эффективно, поскольку в то время, пока одно приложение ждет сообщения пользователя, другие приложения могут обрабатываться процессором.

3.1.3 Мультипрограммирование в системах реального времени

Еще одна разновидность мультипрограммирования используется ОСРВ, предназначенных, как отмечено выше в п. 2.2, для управления с помощью компьютера различными техническими объектами или технологическими процессами. Критерием эффективности этих систем является способность выдерживать заранее заданные интервалы времени между запуском программы и получением результата (управляющего воздействия). Это время называется временем реакции системы, а соответствующее свойство системы – реактивностью. Требования ко времени реакции зависят от специфики управляемого процесса.

В ОСРВ «мультипрограммная смесь» представляет собой фиксированный набор заранее разработанных программ, а выбор программы на выполнение осуществляется по прерываниям (исходя из текущего состояния объекта) или в соответствии с расписанием плановых работ.

Способность аппаратуры компьютера и ОСРВ к быстрому ответу зависит в основном от скорости переключения одной задачи на другую и, в частности, от скорости обработки сигналов прерывания. Если для обработки прерывания процессор должен опросить сотни потенциальных источников прерывания, то реакция системы будет слишком медленной. Время обработки прерывания в ОСРВ часто определяет требования к классу процессора даже при небольшой его загрузке.

В ОСРВ не стремятся максимально загружать все устройства, наоборот, при проектировании программного управляющего комплекса обычно закладывается некоторый «запас» вычислительной мощности на случай пиковой нагрузки. Статистические аргументы о низкой вероятности возникновения пиковой нагрузки, основанные на том, что вероятность одновременного возникновения большого количества независимых событий очень мала, не применимы ко многим ситуациям в реальных системах управления. Например, в системе управления атомной электростанцией в случае возникновения крупной аварии атомного реактора многие аварийные датчики сработают одновременно и создадут коррелированную нагрузку. Если ОСРВ не спроектирована для поддержки подобной пиковой нагрузки, то может случиться так, что система не справится с работой именно тогда, когда она нужна в наибольшей степени.

3.1.4 Мультипрограммирование при мультипроцессорной обработке

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

Мультипроцессорные вычислительные системы начали появляться с середины 80-х гг. и к настоящему моменту обычным делом является наличие нескольких процессоров даже в архитектуре персонального компьютера. Более того, многопроцессорность теперь является одним из необходимых требований, которые предъявляются к компьютерам, используемым в качестве центрального сервера крупной вычислительной сети.

В мультипроцессорных системах несколько задач могут реально выполняться одновременно, так как имеется несколько обрабатывающих устройств – процессоров. Очевидно, что мультипроцессирование не исключает мультипрограммирования – на каждом из процессоров может попеременно выполняться некоторый закрепленный за данным процессором набор задач.

Мультипроцессорная организация системы приводит к усложнению всех алгоритмов управления ресурсами. Так, например, требуется планирование работы процессов не для одного, а для нескольких процессоров, что гораздо сложнее. Сложности заключаются и в возрастании числа конфликтов по обращению к устройствам ввода-вывода, данным, общей памяти и совместно используемым программам. Необходимо предусмотреть эффективные средства блокировки при доступе к разделяемым информационным структурам ядра. Все эти проблемы должна решать ОС путем синхронизации процессов, ведения очередей и планирования ресурсов. Более того, сама ОС должна быть спроектирована так, чтобы уменьшить существующие взаимозависимости между ответственными компонентами.

Мультипроцессорные системы разделяют на *симметричные* и *несимметричные (асимметричные)*. При этом следует четко разделять, к какому аспекту мультипроцессорной системы относится эта характеристика – к типу архитектуры или к способу организации вычислительного процесса. Сначала рассмотрим особенности такого разделения с точки зрения архитектуры.

Симметричная архитектура мультипроцессорной системы предполагает однородность всех процессоров и единообразие включения процессоров в общую схему системы. Традиционные симметричные мультипроцессорные конфигурации разделяют одну большую память между всеми процессорами и используют одну схему отображения памяти. Они могут очень быстро обмениваться данными так, что обеспечивается достаточно высокая производительность для тех приложений (например, при работе с базами данных), в которых несколько задач должны активно взаимодействовать между собой.

В настоящее время функции поддержки симметричной мультипроцессорной обработки данных имеются во всех популярных ОС, таких как *MS Windows NT Workstation 4.0*, *Microsoft Windows Server 2003*, *Windows XP Professional*, *QNX Neutrino*.

Возможность наращивания числа процессоров (*масштабируемость*) в симметричных системах ограничена вследствие того, что все они пользуются одной и той же оперативной памятью и, следовательно, должны располагаться в одном корпусе. Такая конструкция, называемая *масштабируемой по вертикали*, практически ограничивает число процессоров до четырех или восьми.

В *асимметричной архитектуре* разные процессоры могут отличаться как своими характеристиками (производительностью, надежностью, системой команд и т.д., вплоть до модели микропроцессора), так и функциональной ролью, которая поручается им в системе. Например, одни процессоры могут предназначаться для работы в качестве основ-

ных вычислителей, другие – для управления подсистемой ввода-вывода, третьи – для каких-то иных целей.

Функциональная неоднородность в асимметричных архитектурах влечет за собой структурные отличия во фрагментах системы, содержащих разные процессоры системы. Например, они могут отличаться схемами подключения процессоров к системной шине, набором периферийных устройств и способами взаимодействия процессоров с устройствами.

Масштабирование в асимметричной архитектуре реализуется иначе, чем в симметричной. В связи с тем, что требование «единого корпуса» отсутствует, система может состоять из нескольких устройств, каждое из которых содержит один или несколько процессоров – *масштабирование по горизонтали*. Каждое такое устройство называется *кластером*, а вся мультипроцессорная система – *кластерной*.

Рассмотрев особенности организации мультипроцессорных систем с точки зрения архитектуры, отметим особенности их организации с точки зрения вычислительного процесса.

Асимметричное мультипроцессирование является наиболее простым способом организации вычислительного процесса в системах с несколькими процессорами. Этот способ иногда условно называют «ведущий-ведомый». Функционирование системы по принципу «ведущий-ведомый» предполагает выделение одного из процессоров в качестве «ведущего», на котором работает ОС и который управляет всеми остальными «ведомыми» процессорами. В этом случае «ведущий» процессор берет на себя функции распределения задач и ресурсов, а «ведомые» процессоры работают только как обрабатывающие устройства и никаких действий по организации работы вычислительной системы не выполняют. Учитывая то, что ОС работает исключительно на одном процессоре и функции управления централизованы, то такая система по сложности схожа с ОС однопроцессорной системы.

Следует отметить, что асимметричная организация вычислительного процесса может быть реализована как для симметричной мультипроцессорной архитектуры, в которой все процессоры аппаратно неразличимы, так и для несимметричной, для которой характерна неоднородность процессоров, их специализация на аппаратном уровне. В *архитектурно-асимметричных системах* на роль ведущего процессора может быть назначен наиболее надежный и производительный процессор. Если в наборе процессоров имеется специализированный процессор, ориентированный, например, на матричные вычисления, то при планировании процессов ОС, реализующая асимметричное мультипроцессирование, должна учитывать специфику этого процессора. Такая специа-

лизация снижает надежность системы в целом, так как процессоры не являются взаимозаменяемыми.

Симметричное мультипроцессирование как способ организации вычислительного процесса (процессоры работают с общими устройствами и разделяемой основной памятью) может быть реализовано исключительно в системах с симметричной мультипроцессорной архитектурой. Симметричное мультипроцессирование реализуется ОС, общей для всех процессоров.

При симметричной организации все процессоры равноправно участвуют и в управлении вычислительным процессом, и в выполнении прикладных задач. Например, сигнал прерывания от принтера, который распечатывает данные прикладного процесса, выполняемого на некотором процессоре, может быть обработан совсем другим процессором. Разные процессоры могут в какой-то момент одновременно обслуживать как разные, так и одинаковые модули общей ОС, однако при этом ОС должна обладать свойством реентерабельности⁶. Следует отметить, что большим преимуществом симметричных систем перед асимметричными системами является то, что в случае отказа одного из процессоров, симметричные системы, как правило, значительно легче реконфигурируются.

3.1.5 Роль прерываний при мультипрограммировании

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

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

Механизм прерываний реализуется аппаратно-программными средствами вычислительной системы. Структуры систем прерывания (в зависимости от аппаратной архитектуры) могут быть самыми разными, но все они имеют одну общую особенность – прерывание непременно вле-

⁶ *Реентерабельность* – свойство программы или отдельной процедуры, которая разработана так, что одна и та же копия инструкций программы в памяти может быть совместно использована несколькими пользователями или процессами.

чет за собой изменение порядка выполнения команд процессором.

В зависимости от источника все прерывания делят на два класса:

- аппаратные (внешние и внутренние);
- программные.

Аппаратные (англ. *Interrupt Request – IRQ*) – события от периферийных устройств или события в микропроцессоре, возникающие вследствие подачи некоторой аппаратурой электрического сигнала, который передается на специальный вход прерывания процессора. При этом *внешними* будут прерывания, инициированные периферийными устройствами (например, нажатия клавиш клавиатуры, движение мыши, сигнал от таймера, сетевой карты или дискового накопителя), а *внутренними* – те, что происходят в микропроцессоре.

Внешние прерывания являются асинхронными по отношению к потоку инструкций прерываемой программы. Аппаратура процессора работает так, что асинхронные прерывания возникают между выполнением двух соседних инструкций, при этом система после обработки прерывания продолжает выполнение процесса, уже начиная со следующей инструкции.

Внутренние прерывания, называемые также *исключениями* (англ. *exception*), происходят в микропроцессоре и инициируются синхронно выполнению программы при появлении аварийной ситуации в ходе исполнения некоторой инструкции программы – возникают непосредственно в ходе выполнения тактов команды («внутри» выполнения). Примерами исключений являются деление на нуль, ошибки защиты памяти, обращения по несуществующему адресу, попытка выполнить привилегированную инструкцию в пользовательском режиме и т.п.

Программные прерывания возникают (синхронно) при исполнении особой команды процессора, которая имитирует прерывание, т.е. переход на новую последовательность инструкций. Этот класс прерываний отличается от предыдущих тем, что они по своей сути не являются «истинными» прерываниями. Этот механизм был специально введен для того, чтобы переключение на системные программные модули происходило не просто как переход на подпрограмму, а точно таким же образом, как и обычное прерывание. Этим, прежде всего, обеспечивается автоматическое переключение процессора в привилегированный режим с возможностью исполнения любых команд ОС.

Механизм обработки прерываний независимо от архитектуры вычислительной системы подразумевает выполнение некоторой последовательности шагов.

Шаг 1. Установление факта прерывания (прием сигнала запроса на прерывание) и идентификация прерывания (в ОС идентификация прерывания иногда осуществляется повторно, на шаге 4).

Шаг 2. Запоминание состояния прерванного процесса вычислений. Состояние процесса выполнения программы определяется, прежде всего, значением счетчика команд (адресом следующей команды), содержимым регистров процессора, и может включать также спецификацию режима (например, режим пользовательский или привилегированный) и другую информацию.

Шаг 3. Управление аппаратно передается на подпрограмму обработки прерывания. В простейшем случае в счетчик команд заносится начальный адрес подпрограммы обработки прерываний, а в соответствующие регистры – информация из слова состояния.

Шаг 4. Сохранение информации о прерванной программе, которую не удалось спасти на шаге 2 с помощью аппаратуры. В некоторых процессорах предусматривается запоминание довольно большого объема информации о состоянии прерванных вычислений.

Шаг 5. Собственно выполнение программы, связанной с обработкой прерывания. Эта работа может быть выполнена той же подпрограммой, на которую было передано управление на шаге 3, но в ОС достаточно часто она реализуется путем последующего вызова соответствующей подпрограммы.

Шаг 6. Восстановление информации, относящейся к прерванному процессу (этап, обратный шагу 4).

Шаг 7. Возврат на прерванную программу.

Следует отметить, что шаги 1-3 реализуются аппаратно, а шаги 4-7 – программно.

Главные функции механизма прерываний – это:

- распознавание или классификация прерываний;
- передача управления соответствующему обработчику прерываний;
- корректное возвращение к прерванной программе.

На рис. 6 показано, что при возникновении запроса на прерывание естественный ход вычислений нарушается и управление передается на программу обработки возникшего прерывания (*обработчик прерываний, процедура обслуживания прерываний*, англ. *Interrupt Service Routine*). При этом средствами аппаратуры сохраняется (как правило, с помощью механизмов стековой памяти) адрес той команды, с которой следует продолжить выполнение прерванной программы.

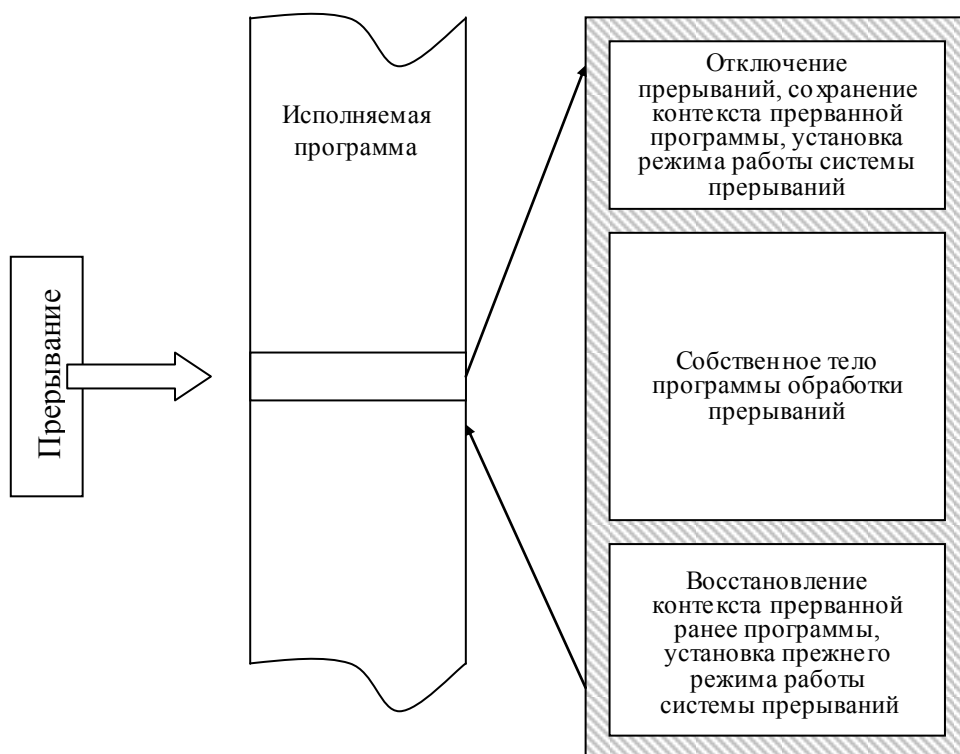


Рисунок 6 – Обобщенная схема обработки прерывания

Обработчик прерываний – специальная процедура, вызываемая по прерыванию для выполнения его обработки. Обработчики прерываний могут выполнять множество функций, в зависимости от причины, вызвавшей прерывание. Вид функции, исполняемой обработчиком в каждом конкретном случае, определяется в соответствии с *таблицей векторов прерываний*, размещаемой в определенном месте адресного пространства.

После выполнения программы обработки прерывания, управление возвращается на прерванную ранее программу посредством занесения в указатель команд сохраненного адреса команды, которую нужно было бы выполнить, если бы не возникло прерывание. Отметим, что такая схема обработки прерываний используется только в самых простых программных средах.

Из рис. 6 видно, что в схеме обработки прерывания имеется две служебные секции – в первой осуществляется сохранение контекста прерванных вычислений, а в заключительной – наоборот, восстановление этого контекста. Для того, чтобы система не среагировала повторно на сигнал запроса на прерывание, выполняется защита от повторного срабатывания, поэтому система обработки обычно автоматически «закрывает» (отключает) прерывания, и необходимо позже в подпрограмме обработки прерываний вновь включать систему прерываний. В конце первой секции подпрограммы обработки осуществляется установка ре-

жимов *приоритетов* обработки прерываний. Таким образом, на время выполнения центральной секции прерывания разрешены. На время работы заключительной секции подпрограммы обработки система прерываний вновь должна быть отключена, и после восстановления контекста опять включена. Поскольку эти действия необходимо выполнять практически в каждой подпрограмме обработки прерываний, во многих ОС первые секции подпрограмм обработки прерываний выделяются в специальный системный программный модуль – *супервизор прерываний*.

В общем случае, управление ходом выполнения задач со стороны ОС заключается в организации реакций на прерывания, в организации обмена информацией, в предоставлении необходимых ресурсов, в выполнении задачи в динамике и в организации сервиса. Причины прерываний определяет ОС (супервизор прерываний), она же и выполняет действия, необходимые при данном прерывании и в данной ситуации.

Организация обработки прерывания при участии супервизора представлена на рис. 7.

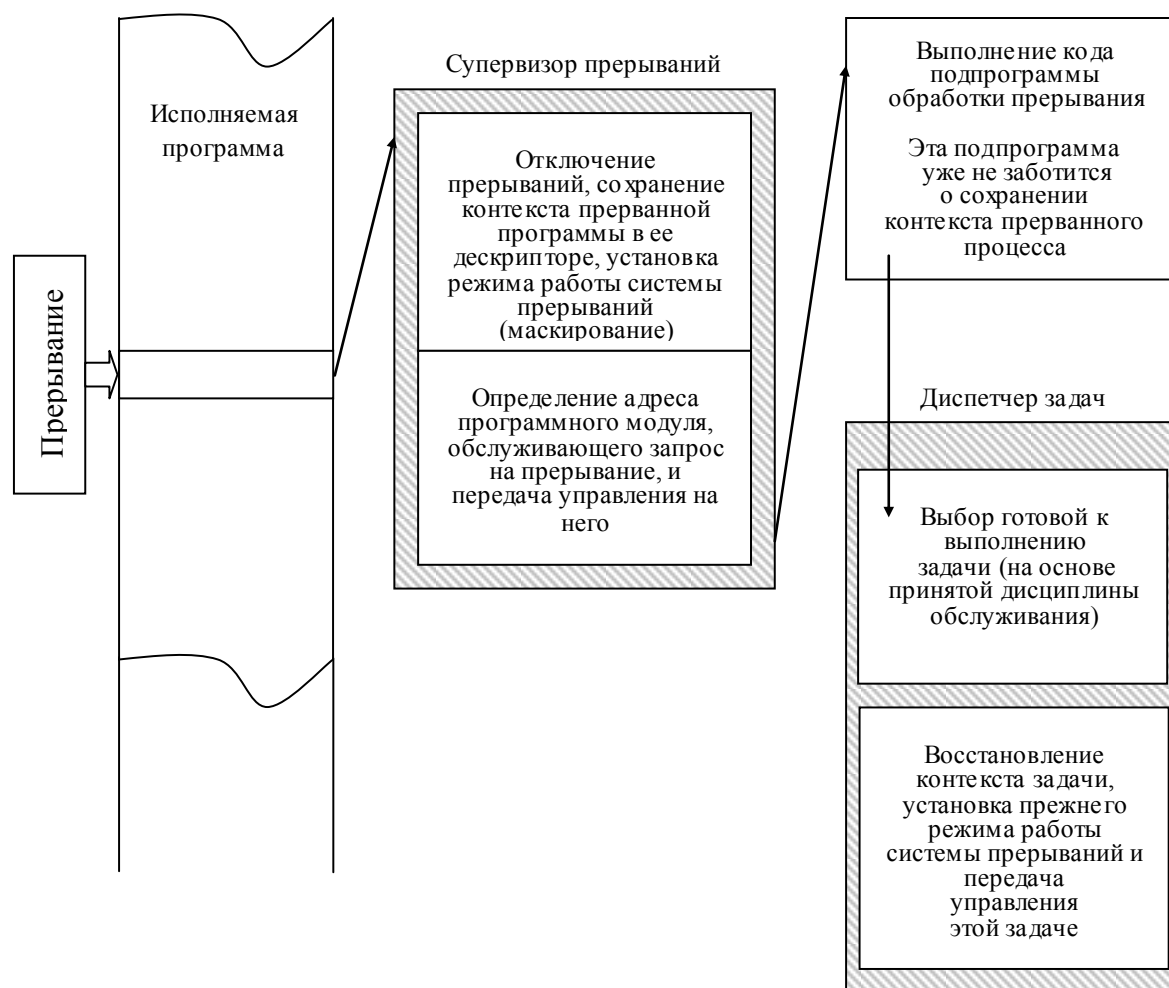


Рисунок 7 – Обработка прерывания при участии супервизора

Как видно из представленной схемы, здесь отсутствует возврат в

прерванную ранее программу непосредственно из самой подпрограммы обработки прерывания. Для прямого возврата достаточно адрес возврата сохранить в стеке, что и делает аппаратура процессора. При этом стек легко обеспечивает возможность возврата в случае вложенных прерываний, поскольку он всегда реализует дисциплину *LCFS*. Супервизор прерываний сохраняет в дескрипторе текущей задачи рабочие регистры процессора, определяющие контекст прерываемого вычислительного процесса. Далее он определяет ту подпрограмму, которая должна выполнить действия, связанные с обслуживанием настоящего (текущего) запроса на прерывание. Наконец, перед тем, как передать управление на эту подпрограмму, супервизор прерываний устанавливает необходимый режим обработки прерывания (осуществляется *маскирование*, определяющее запрет некоторых сигналов прерывания). После выполнения подпрограммы обработки прерывания управление вновь передается ядру ОС. На этот раз уже на тот модуль, который занимается диспетчеризацией задач (см. п. 3). И уже диспетчер задач, в свою очередь, в соответствии с принятой дисциплиной распределения процессорного времени (между выполняющимися вычислительными процессами) восстановит контекст той задачи, которой будет решено выделить процессор.

Сигналы, вызывающие прерывания, формируются вне процессора или в процессоре, и могут возникать одновременно. В связи с этим, еще одним неотъемлемым элементом обработки прерываний является *приоритет прерываний*, с помощью которого они ранжируются по степени важности и срочности. О прерываниях, имеющих одинаковое значение приоритета, говорят, что они относятся к одному уровню приоритета прерываний. Так, со всей очевидностью, прерывания от схем контроля процессора должны обладать наивысшим приоритетом (действительно, если аппаратура работает неправильно, то не имеет смысла продолжать обработку информации), а например, программные прерывания – самым низким.

На рис. 8 изображен обычный порядок (приоритеты) обработки прерываний в зависимости от типа прерываний.

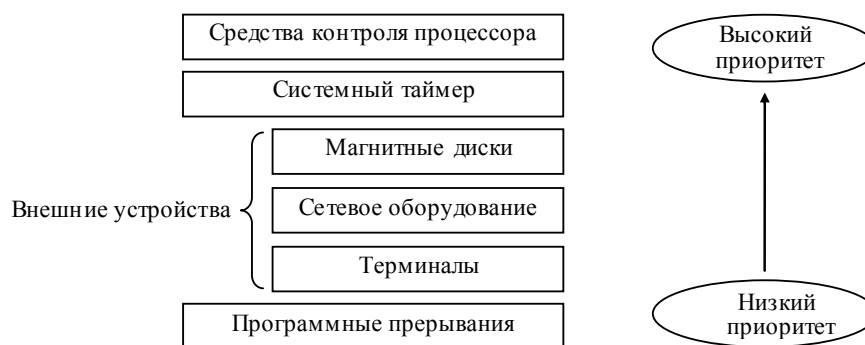


Рисунок 8 – Распределение прерываний по уровням приоритета

Например, если приоритет запрашиваемого прерывания ниже приоритета средства контроля процессора, то в этом случае прерывания не произойдет.

Учет приоритета может быть встроен в технические средства, а также определяться ОС, т.е. кроме аппаратно реализованных приоритетов прерывания большинство вычислительных машин и комплексов допускают программно-аппаратное управление порядком обработки сигналов прерывания. Второй способ, дополняя первый, позволяет применять различные дисциплины обслуживания прерываний.

3.2 Планирование процессов и потоков

3.2.1 Понятие процесса и потока

Для того, чтобы перейти к более подробному рассмотрению понятия процесс, рассмотрим примеры, позволяющие выявить различия между понятиями *программа*, *задача* или *процесс*.

Пусть два студента запускают программу извлечения квадратного корня, при этом один хочет вычислить квадратный корень из 4, а второй – из 1. С точки зрения студентов, запущена одна и та же программа; с точки зрения компьютерной системы, ей приходится заниматься двумя различными вычислительными процессами, так как разные исходные данные приводят к разному набору вычислений. Следовательно, на уровне происходящего внутри вычислительной системы нельзя использовать термин «программа» в пользовательском смысле слова. Рассмотрим другой пример. Пусть два студента сформировали идентичные задания и пытаются извлечь квадратный корень из 1, но система выполняет эти вычисления с некоторым сдвигом во времени: в то время как одно из выполняемых заданий приступило к печати результата и ждет окончания операции ввода-вывода, второе только начинает исполняться. Очевидно, что в один момент времени в системе присутствуют различные задания, так как состояние процесса их выполнения различно.

В связи с этим, термины *программа* и *задание* правомерно применять для описания некоторых статических, неактивных объектов. Для описания динамических объектов используют термин *процесс*. *Процесс* характеризует некоторую совокупность набора исполняющихся команд, ассоциированных с ним ресурсов (выделенная для исполнения память или адресное пространство, стеки, используемые файлы и устройства ввода-вывода и т.д.) и информации о текущем моменте его исполнения (значения регистров, программного счетчика, состояние стека и значения переменных), находящуюся под управлением ОС. Такая обособленность нужна для того, чтобы защитить один процесс от другого, поскольку они, совместно используя все ресурсы вычислительной систе-

мы, конкурируют друг с другом за доступ к ресурсам. В общем случае процессы никак не связаны между собой и могут принадлежать даже разным пользователям, разделяющим одну вычислительную систему.

В настоящее время в большинстве ОС для поддержки мультипрограммирования кроме понятия процесс определена еще одна, более «мелкая» единица работы – *поток*.

Очевидно, что любая работа вычислительной системы заключается в выполнении некоторой программы. Поэтому и с процессом, и с потоком связывается определенный программный код, который для этих целей оформляется в виде исполняемого модуля. Очевидно, выполнение программы невозможно без предоставления ей процессорного времени, то есть времени, в течение которого процессор выполняет коды данной программы. В ОС, где существуют и процессы, и потоки, процесс рассматривается как заявка на потребление всех видов ресурсов, кроме одного – процессорного времени. Этот последний важнейший ресурс распределяется ОС между другими единицами работы – потоками, которые и получили свое название благодаря тому, что они представляют собой последовательности (потоки выполнения) команд.

В простейшем случае процесс состоит из одного потока, а это означает, что остается только одна единица работы и потребления ресурсов – процесс. Мультипрограммирование осуществляется в таких ОС на уровне процессов. Рассмотрим основные предпосылки, способствовавшие появлению потоков.

Как отмечалось выше, для того чтобы процессы не могли вмешаться в распределение ресурсов, а также не могли повредить коды и данные друг друга, важнейшей задачей ОС является изоляция одного процесса от другого. Для этого ОС обеспечивает каждый процесс отдельным виртуальным адресным пространством, так что ни один процесс не может получить прямого доступа к командам и данным другого процесса. При необходимости взаимодействия, процессы обращаются к ОС, которая, выполняя функции посредника, предоставляет им средства межпроцессной связи – конвейеры, разделяемые секции памяти и др. Очевидно, такой способ взаимодействия параллельно действующих процессов является достаточно затратным, т.к. требуются дополнительные затраты времени на переключение контекста.

В то же время, приложение, выполняемое в рамках одного процесса, может обладать внутренним параллелизмом, который в принципе мог бы позволить ускорить его работу. Если, например, в программе предусмотрено обращение к внешнему устройству, то на время этой операции можно не блокировать выполнение всего процесса, а продолжить вычисления по другой ветви программы. Параллелизм желателен как для обслуживания различных запросов к базе данных, так и для бо-

лее быстрого выполнения отдельного запроса за счет одновременного просмотра различных записей базы.

Конечно, задача распараллеливания вычислений в рамках одного приложения может быть решена традиционными способами с использованием развитых функций программирования и стандартных средств межпроцессного взаимодействия без использования потоков.

Например, прикладной программист может взять на себя сложную задачу организации параллелизма, выделив в приложении некоторую подпрограмму-диспетчер, которая периодически передает управление той или иной ветви вычислений. При этом программа может получиться логически весьма сложной, с многочисленными передачами управления, что существенно затрудняет ее отладку и модификацию.

Другим способом является создание для одного приложения нескольких процессов для каждой из параллельных работ. Однако использование для создания процессов стандартных средств ОС не позволяет учесть тот факт, что эти процессы решают единую задачу, а значит, имеют много общего между собой – могут работать с одними и теми же данными, использовать один и тот же кодовый сегмент, наделяться одними и теми же правами доступа к ресурсам вычислительной системы. Например, создание отдельных процессов при поступлении каждого запроса к серверу баз данных, при условии, что все процессы будут выполнять один и тот же программный код и выполнять поиск в записях, общих для всех процессов файлов данных, не выглядит целесообразным. В особенности если ОС при таком подходе будет рассматривать эти процессы наравне со всеми остальными процессами и с помощью универсальных механизмов обеспечивать их изоляцию друг от друга. Очевидно, что в подобном случае достаточно громоздкие механизмы межпроцессного взаимодействия будут использоваться явно не по назначению, затрудняя обмен данными между различными частями приложения. Кроме того, на создание каждого процесса ОС тратит системные ресурсы, которые в данном случае неоправданно дублируются – каждому процессу выделяются собственное виртуальное адресное пространство, физическая память, закрепляются устройства ввода-вывода и т.п.

Из всего вышеизложенного, следует, что в ОС наряду с процессами нужен другой механизм распараллеливания вычислений, который учитывал бы тесные связи между отдельными ветвями вычислений одного и того же приложения. Для этих целей современные ОС предлагают механизм многопоточной обработки (англ. *multithreading*). При этом и необходима новая единица работы – *поток* или *нить исполнения* (*нити*, *легковесные* (англ. *thin*) *процессы*, *треды* (англ. *threads*)).

Следует отметить, что *легковесными* эти процессы называют потому, что ОС не должна для них организовывать полноценную виртуальную машину (с собственными ресурсами), а развиваются они в том же виртуальном адресном пространстве, могут пользоваться теми же файлами, виртуальными устройствами и иными ресурсами, выделенными ОС данному процессу. Единственное, что они имеют свое – это *процессорный ресурс*. Другими словами, понятию *поток* соответствует последовательный переход процессора от одной команды программы к другой, при этом ОС распределяет процессорное время между потоками. В то же время *процессу* ОС назначает адресное пространство и набор ресурсов, которые совместно используются всеми его потоками.

3.2.2 Создание процессов и потоков

Создать процесс – это прежде всего означает создать *описатель процесса*, в качестве которого выступает одна или несколько информационных структур, содержащих все сведения о процессе, необходимые ОС для управления им. Примерами описателей процесса являются блок управления задачами (*Task Control Block – TCB*) в *OS/360*, управляющий блок процесса (*Process Control Block – PCB*) в *OS/2*, дескриптор процесса в *Unix*, объект-процесс (*object-process*) в *Windows NT*.

Создание описателя процесса знаменует собой появление в системе еще одного претендента на вычислительные ресурсы. Начиная с этого момента при распределении ресурсов ОС должна принимать во внимание потребности нового процесса. Создание процесса включает загрузку кодов и данных исполняемой программы данного процесса с диска в оперативную память. Для этого ОС должна обнаружить местоположение такой программы на диске, перераспределить оперативную память и выделить память исполняемой программе нового процесса. Затем необходимо считать программу в выделенные для нее участки памяти и, возможно, изменить параметры программы в зависимости от размещения в памяти.

В системах с виртуальной памятью (подробнее особенности организации виртуальной памяти рассмотрены ниже п. 4.2) в начальный момент может загружаться только часть кодов и данных процесса, с тем чтобы «подкачивать» остальные по мере необходимости. Существуют системы, в которых на этапе создания процесса не требуется непременно загружать коды и данные в оперативную память, вместо этого исполняемый модуль копируется из того каталога файловой системы, в котором он изначально находился, в область подкачки – специальную область диска, отведенную для хранения кодов и данных процессов. При выполнении всех этих действий подсистема управления процессами

тесно взаимодействует с подсистемой управления памятью и файловой системой.

Создание процесса состоит из нескольких этапов:

- присвоения идентификатора процессу;
- включения его в список активных процессов, известных системе;
- формирования блока управления процессом;
- выделения процессу начальных ресурсов.

В многопоточной системе при создании процесса ОС создает для каждого процесса как минимум один поток. При создании потока так же, как и при создании процесса, ОС генерирует специальную информационную структуру – *описатель потока*.

В исходном состоянии поток (или процесс, если речь идет о системе, в которой понятие «поток» не поддерживается) находится в приостановленном состоянии. Момент выборки потока на выполнение осуществляется в соответствии с принятым в данной системе правилом предоставления процессорного времени и с учетом всех существующих в данный момент потоков и процессов. В случае если коды и данные процесса находятся в области подкачки, необходимым условием активизации потока процесса является также наличие места в оперативной памяти для загрузки его исполняемого модуля.

В общем случае существующий процесс может породить новый процесс, и может иметь место иерархическая структура процессов. Задача может порождать подзадачу в мультипрограммном режиме, и в этом смысле будут появляться *родительский* и *дочерний* процессы. Уничтожение процесса означает удаление его из системы. Ресурсы возвращаются системе, имя процесса удаляется из списка, блок управления процессом освобождается.

В разных ОС по-разному строятся отношения между потоками-потомками и их родителями. Например, в одних ОС выполнение родительского потока синхронизируется с его потомками, в частности после завершения родительского потока ОС может снимать с выполнения всех его потомков. В других системах потоки-потомки могут выполняться асинхронно по отношению к родительскому потоку. Потомки, как правило, наследуют многие свойства родительских потоков. Во многих системах порождение потомков является основным механизмом создания процессов и потоков.

3.2.3 Управляющие структуры процессов и потоков

Для того чтобы ОС могла управлять процессами, она должна располагать всей необходимой для этого информацией, содержащейся в описателе процесса – дескрипторе процесса (контексте процесса). В об-

щем случае дескриптор процесса, как правило, содержит следующую информацию:

- идентификатор процесса (*Process Identifier, PID*);
- тип (или класс) процесса, который определяет для супервизора некоторые правила предоставления ресурсов;
- приоритет процесса, в соответствии с которым супервизор предоставляет ресурсы (в рамках одного класса процессов в первую очередь обслуживаются более приоритетные процессы);
- переменную состояния, которая определяет, в каком состоянии находится процесс (готов к работе, выполняется, ожидает устройства ввода-вывода и т. д.);
- контекст задачи, то есть защищенную область памяти (или адрес такой области), в которой хранятся текущие значения регистров процессора, когда процесс прерывается, не закончив работы;
- информацию о ресурсах, которыми процесс владеет и/или имеет право пользоваться (указатели на открытые файлы, информация о незавершенных операциях ввода-вывода и др.);
- место (или его адрес) для организации общения с другими процессами;
- параметры времени запуска (момент времени, когда процесс должен активизироваться, и периодичность этой процедуры);
- в случае отсутствия системы управления файлами адрес задачи на диске в ее исходном состоянии и адрес на диске, куда она выгружается из оперативной памяти (ОП), если ее вытесняет другая задача (последнее справедливо для диск-резидентных задач, которые постоянно находятся во внешней памяти на системном магнитном диске и загружаются в ОП только на время выполнения).

В некоторых ОС количество описателей определяется жестко и заранее (на этапе генерации варианта ОС или в конфигурационном файле, который используется при загрузке ОС), в других по мере необходимости система может выделять участки памяти под новые описатели. Например, в уже мало кому известной системе *OS/2*, которая несколько лет тому назад многими специалистами считалась одной из лучших ОС для персональных компьютеров, максимально возможное количество описателей задач указывается в конфигурационном файле *CONFIG.SYS*. Например, строка *THREADS = 1024* в файле *CONFIG.SYS* означает, что всего в системе может параллельно существовать и выполняться до 1024 задач, включая вычислительные процессы и их потоки. В системах *Windows NT/2000/XP* количество описателей в явном виде не задается – это переменная величина и она определяется самой ОС. Текущее количество таких описателей представлено в окне *Диспетчера задач*.

Описатели задач, как правило, постоянно располагаются в ОП с целью ускорить работу супервизора, который организует их в *списки (очереди)* и отображает изменение состояния процесса перемещением соответствующего описателя из одного списка в другой. Для каждого состояния (за исключением состояния исполнения для однопроцессорной системы) ОС ведет соответствующий список задач, находящихся в этом состоянии. Однако для состояния ожидания обычно имеется не один список, а столько, сколько различных видов ресурсов могут вызывать состояние ожидания.

Очереди процессов представляют собой дескрипторы отдельных процессов, объединенные в списки – каждый дескриптор, кроме всего прочего, содержит, по крайней мере, один указатель на другой дескриптор, соседствующий с ним в очереди. Такая организация очередей позволяет легко их переупорядочивать, включать и исключать процессы, переводить процессы из одного состояния в другое.

В ОСРВ чаще всего количество процессов фиксируется, и, следовательно, целесообразно заранее определять (на этапе генерации или конфигурирования ОС) количество дескрипторов. Для использования таких ОС в качестве систем общего назначения (что в настоящее время нехарактерно) обычно количество дескрипторов бралось с некоторым запасом, и появление новой задачи связывалось с заполнением этой информационной структуры. Поскольку дескрипторы процессов постоянно располагаются в оперативной памяти (с целью ускорить работу диспетчера), то их количество не должно быть очень большим.

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

Для аппаратной поддержки работы ОС с этими информационными структурами (дескрипторами задач) в процессорах могут быть реализованы соответствующие механизмы. Так, например, в микропроцессорах *Intel 80x86* имеется специальный регистр (*Task Register – TR*), указывающий местонахождение специальной информационной структуры – сегмента состояния задачи (*Task State Segment – TSS*), в котором при переключении с задачи на задачу автоматически сохраняется содержимое регистров процессора.

3.2.4 Состояния процесса

Все, что выполняется в вычислительных системах (не только программы пользователей, но и, возможно, определенные части ОС), организовано как набор процессов. Понятно, что реально на однопроцессорной компьютерной системе в каждый момент времени может исполняться только один процесс. Для мультипрограммных вычислительных систем псевдопараллельная обработка нескольких процессов достигается с помощью переключения процессора с одного процесса на другой. Пока один процесс выполняется, остальные ждут своей очереди на получение процессора.

Каждый процесс может находиться как минимум в двух состояниях: *процесс выполняется* и *процесс не выполняется*. Диаграмма состояний процесса в такой модели изображена на рис. 9.

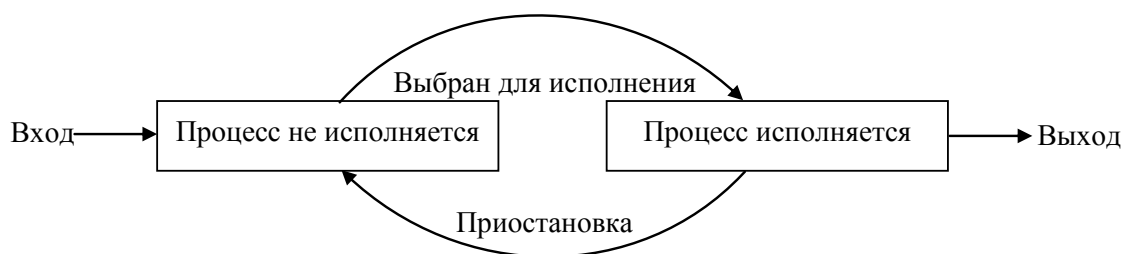


Рисунок 9 – Простейшая диаграмма состояний процесса

Процесс, находящийся в состоянии *процесс выполняется*, может через некоторое время завершиться или быть приостановлен ОС и снова переведен в состояние *процесс не выполняется*. Приостановка процесса может произойти, например, по следующим причинам: для его дальнейшей работы потребовалось возникновение какого-либо события (например, завершения операции ввода-вывода) или истек временной интервал, отведенный ОС для работы этого процесса. После этого ОС по определенному алгоритму выбирает для исполнения один из процессов, находящихся в состоянии *процесс не выполняется*, и переводит его в состояние *процесс выполняется*. Новый процесс, появляющийся в системе, первоначально помещается в состояние *процесс не выполняется*.

Приведенная модель является очень грубой. Она не учитывает, в частности то, что процесс, выбранный для исполнения, может все еще ждать события, из-за которого он был приостановлен, и реально к выполнению не готов. Для того чтобы избежать такой ситуации, разобьем состояние *процесс не выполняется* на два новых состояния: *готовность* и *ожидание* (рис. 10).

Всякий новый процесс, появляющийся в системе, попадает в состояние *готовность*. Операционная система, пользуясь каким-либо алгоритмом планирования, выбирает один из готовых процессов и переводит

его в состояние *исполнение*. В состоянии *исполнение* происходит непосредственное выполнение программного кода процесса. Покинуть это состояние процесс может по трем причинам:

- либо он завершает свою деятельность;
- либо в результате возникновения прерывания в вычислительной системе (например, прерывания от таймера по истечении дозволенного времени выполнения) его возвращают в состояние *готовность*;
- либо он не может продолжать свою работу, пока не произойдет некоторое событие, и ОС (по прерыванию) переводит его в состояние *ожидание*.

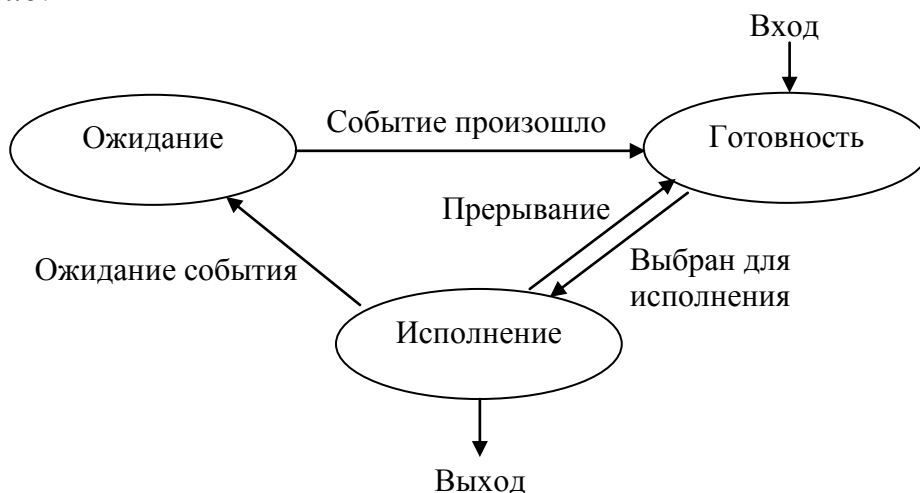


Рисунок 10 – Диаграмма 3-х состояний процесса

Эта модель хорошо описывает поведение процессов во время их жизни, но она не показывает появление процесса в системе и его исчезновение из системы. Поэтому целесообразно показать еще два состояния процессов: *рождение* и *закончил исполнение* (рис. 11).

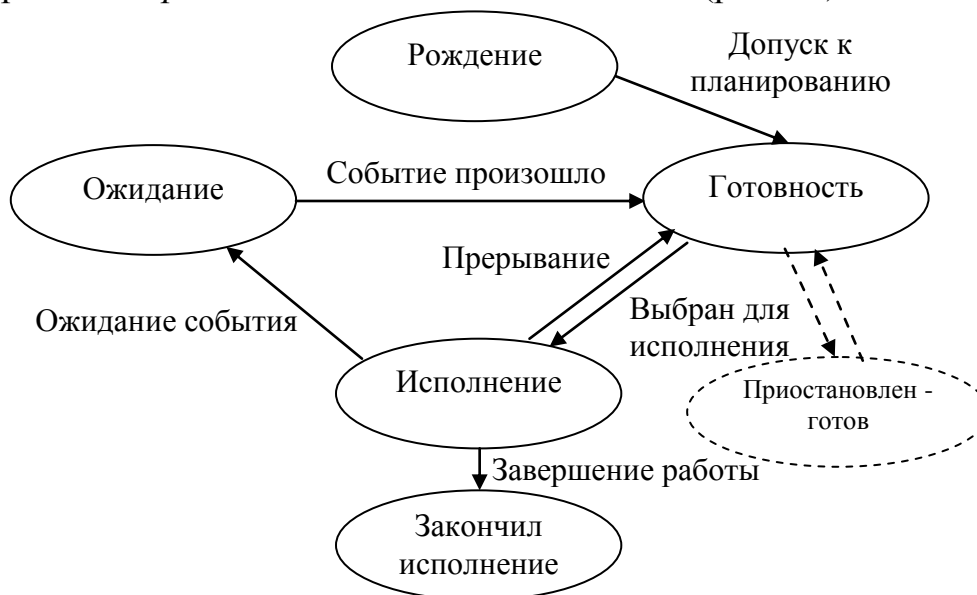


Рисунок 11 – Более детальная диаграмма состояний процесса

Теперь для появления в вычислительной системе процесс должен пройти через состояние *рождение*. При рождении процесс получает в свое распоряжение адресное пространство, в которое загружается программный код процесса, ему выделяются стек и системные ресурсы; устанавливается начальное значение программного счетчика этого процесса и т.д. Родившийся процесс переводится в состояние *готовность*. При завершении своей деятельности процесс из состояния *исполнение* попадает в состояние *закончил исполнение*.

Следует отметить, что в ОСРВ могут существовать дополнительное состояние *приостановлен-готов*, в которое процесс может перейти из состояния *готовность* через дополнительную операцию «приостановки», а вернуться – через «возобновление». Это играет важную роль в ОСРВ и используется в следующих случаях:

- при пиковой нагрузке вычислительной системы, когда она не может обеспечить требуемое быстродействие, при расходах времени на смену состояний, превышающих полезную работу;
- при ненадежной работе системы и возможном ее отказе;
- в случаях, когда промежуточные результаты работы процесса вызывают сомнение в правильности работы программы.

При приостановке (и нехватке памяти) процесс освободит ОП, его копия сбрасывается на диск в специальный свопинг файл. Также могут быть освобождены и другие ресурсы.

Следует помнить, что количество состояний процесса в различных ОС может быть различно. Например, в ОС *Windows NT* – 7 состояний, а в ОС *Unix* – 9 состояний.

Изменением состояния процессов занимается ОС, совершая операции над ними. Основные операции над процессами удобно объединить в три пары:

- *создание процесса* – *завершение процесса* (для процесса выполняются однократно);
- *приостановка процесса* (перевод из состояния *исполнение* в состояние *готовность*) – *запуск процесса* (перевод из состояния *готовность* в состояние *исполнение*);
- *блокирование процесса* (перевод из состояния *исполнение* в состояние *ожидание*) – *разблокирование процесса* (перевод из состояния *ожидание* в состояние *готовность*).

Кроме того, следует выделить еще одну – непарную операцию: *изменение приоритета процесса*.

Указанные операции на процессом выполняются в соответствии с алгоритмом планирования процессов, реализуемым в данной ОС.

3.2.5 Критерии планирования

Вообще, *планирование* – это работа по определению того, в какой момент времени прервать выполнение одного процесса и какому процессу предоставить возможность выполняться. Планирование использования процессора впервые возникает в мультипрограммных вычислительных системах, где в состоянии готовности могут одновременно находиться несколько процессов. Планирование процессов включает в себя решение следующих задач:

- определение момента времени для смены выполняемого процесса;
- выбор процесса на выполнение из очереди готовых процессов;
- переключение контекстов «старого» и «нового» процессов.

В свою очередь, *диспетчеризация* – это реализация решения, найденного в результате планирования. Задачами диспетчеризации являются:

- сохранение контекста текущего потока;
- загрузка контекста нового потока;
- запуск нового потока на выполнение.

При построении алгоритмов планирования выделяют три различных уровня, которые обуславливают особенности работы этих алгоритмов:

- долгосрочное;
- краткосрочное;
- среднесрочное.

Долгосрочное планирование характеризуется тем, что:

- отвечает за порождение новых процессов в системе, определяя ее степень мультипрограммирования;
- осуществляется достаточно редко (между появлением процессов могут проходить минуты и даже десятки минут);
- оказывает влияние на функционирование вычислительной системы на протяжении достаточно длительного времени;
- в некоторых ОС сведено к минимуму или отсутствует совсем.

Краткосрочное планирование характеризуется тем, что:

- применяется при планировании использования процессора (например, при обращении исполняющегося процесса к устройствам ввода-вывода или по завершении определенного интервала времени);
- осуществляется не реже одного раза в 100 миллисекунд;
- оказывает влияние на функционирование системы до наступления очередного аналогичного события.

Среднесрочное планирование, в свою очередь, характеризуется тем, что применяется в вычислительных системах для повышения производительности при «swapping»⁷.

3.2.6 Цели и свойства алгоритмов планирования

Для каждого уровня планирования процессов можно предложить много различных алгоритмов. Выбор конкретного алгоритма определяется классом задач, решаемых вычислительной системой, и целями, которые должны быть достигнуты планированием. К числу таких целей можно отнести следующие:

- *Справедливость*. Гарантировать каждому заданию или процессу определенную часть времени использования процессора в компьютерной системе, стараясь не допустить «захвата» процессора одним процессом.

- *Эффективность*. Постараться занять процессор на 100% рабочего времени, не позволяя простаивать ему в ожидании процессов, готовых к исполнению. В реальных вычислительных системах загрузка процессора составляет 40% - 90%.

- *Сокращение полного времени выполнения* (англ. *turnaround time*). Обеспечить минимальное время между стартом процесса или постановкой задания в очередь для исполнения и его завершением.

- *Сокращение времени ожидания* (англ. *waiting time*). Сократить время, которое проводят процессы в состоянии *готовность* в очереди для исполнения.

- *Сокращение времени отклика* (англ. *response time*). Минимизировать время, которое требуется процессу в интерактивных системах для ответа на запрос пользователя.

Многие из приведенных выше целей и свойств являются противоречивыми. Улучшая работу алгоритма с точки зрения одного критерия, можно ухудшить его с точки зрения другого, поэтому задача разработчика алгоритма планирования заключается в поиске разумного компромисса.

Кроме целей планирования, которые необходимо достигнуть, желательно также, чтобы алгоритмы обладали следующими свойствами:

- *Предсказуемость*. Одно и то же задание должно выполняться приблизительно за одно и то же время.

- *Минимальные накладные расходы*. По сути означает, что $t_{\text{исполнения процесса}} \gg t_{\text{выбора процесса}}$. Другими словами, если на каждые 100 миллисекунд, выделенные процессу для использования процессора, будет

⁷ Временное удаление какого-либо частично выполнившегося процесса из оперативной памяти на диск, а позже – его возвращение для дальнейшего выполнения.

приходиться 200 миллисекунд на определение того, какой именно процесс получит процессор в свое распоряжение, и на переключение контекста, то такой алгоритм, очевидно, применять не стоит.

- *Равномерная загрузка ресурсов* вычислительной системы, отдавая предпочтение тем процессам, которые будут занимать малоиспользуемые ресурсы.

- *Масштабируемость*. Рост количества процессов в системе в два раза не должен приводить к увеличению полного времени выполнения процессов на порядок.

Для достижения целей планирования алгоритмы планирования должны опираться на некоторые характеристики – *параметры планирования* как вычислительной системы в целом, так и самих процессов.

Среди параметров планирования *вычислительной системы* выделяют следующие:

- *Статические* (не изменяемые в ходе функционирования) – предельные значения ресурсов системы: размер оперативной памяти, максимальное количество памяти на диске для осуществления свопинга, количество подключенных устройств ввода-вывода и т.п.).

- *Динамические* (изменяемые в ходе функционирования) – значения ресурсов системы на текущий момент.

К статическим параметрам *процессов* относятся характеристики, как правило, присущие заданиям уже на этапе загрузки:

- пользователь, запустивший процесс;
- приоритетность выполнения поставленной задачи;
- соотношение процессорного времени и времени, необходимого для осуществления операций ввода-вывода;
- номенклатура (оперативная память, устройства ввода-вывода, специальные библиотеки и системные программы и т.д.) и величина необходимых заданию ресурсов вычислительной системы.

Алгоритмы долгосрочного планирования процессов используют в своей работе:

- параметры вычислительной системы (статические и динамические);
- статические параметры процессов (динамические параметры процессов на этапе загрузки заданий еще не известны);

Алгоритмы краткосрочного и среднесрочного планирования процессов дополнительно учитывают и динамические характеристики процессов. Для алгоритмов среднесрочного планирования в качестве динамических характеристик может использоваться следующая информация:

- время с момента выгрузки процесса на диск или его загрузки в ОП;

- размер занимаемой процессом ОП;
- количество процессорного времени, предоставленного процессу на данный момент.

3.2.7 Виды планирования

Существует два основных вида алгоритмов планирования процессов – *невывесняющие* (*non-preemptive*, применяются в ОС NetWare) и *вывесняющие* (*preemptive*, применяются в ОС Unix, Windows, OS/2, VMS).

Невывесняющая многозадачность (*non-preemptive multitasking*) – это способ планирования процессов, при котором активный процесс выполняется до тех пор, пока он сам, по собственной инициативе, не отдаст управление планировщику ОС для того, чтобы тот выбрал из очереди другой, готовый к выполнению процесс.

Вывесняющая многозадачность (*preemptive multitasking*) – это такой способ планирования, при котором решение о переключении процессора с выполнения одного процесса на выполнение другого процесса принимается планировщиком ОС, а не самим активным процессом.

Алгоритмы планирования могут быть:

- основаны на *квантовании*;
- основаны на *приоритетах*.

В соответствии с алгоритмами, основанными на квантовании, смена активного процесса происходит, если:

- процесс завершился и покинул систему;
- произошла ошибка;
- процесс перешел в состояние «ожидание»;
- исчерпан квант процессорного времени, отведенный данному процессу.

Другая группа алгоритмов основана на понятии *приоритет* – числе, характеризующем степень привилегированности процесса при использовании ресурсов вычислительной машины, в частности, процессорного времени (чем выше приоритет, тем выше привилегии). Приоритет может выражаться целыми или дробными, положительным или отрицательным значением. Чем выше привилегии процесса, тем меньше времени он будет проводить в очередях. Приоритет может назначаться директивно администратором системы в зависимости от важности работы или внесенной платы, либо вычисляться самой ОС по определенным правилам, он может оставаться фиксированным на протяжении всей жизни процесса либо изменяться во времени в соответствии с некоторым законом (динамические приоритеты).

Выделяют две разновидности приоритетного планирования: обслуживание с *относительными приоритетами* и обслуживание с *абсолютными приоритетами*. В обоих случаях выбор потока на выполнение из очереди готовых осуществляется одинаково – выбирается поток, имеющий наивысший приоритет. Отличие заключается в определении момента смены активного потока. В системах с относительными приоритетами активный поток выполняется до тех пор, пока он сам не покинет процессор, перейдя в состояние ожидания (или произойдет ошибка или поток завершится). В системах с абсолютными приоритетами выполнение активного потока прерывается, если в очереди готовых потоков появился поток, приоритет которого выше приоритета активного потока.

В системах, в которых планирование осуществляется на основе относительных приоритетов, минимизируются затраты на переключение процессора с одной работы на другую. С другой стороны, здесь могут возникать ситуации, когда одна задача занимает процессор долгое время. Ясно, что для систем разделения времени и реального времени такая дисциплина обслуживания не подходит: интерактивное приложение может ждать своей очереди часами, пока вычислительной задаче не потребуется ввод-вывод. А вот в системах пакетной обработки (в том числе известной ОС OS/360) относительные приоритеты использовались широко.

Во многих ОС алгоритмы планирования носят «смешанный» характер и построены как с использованием квантования, так и приоритетов. Например, в основе планирования лежит квантование, но величина кванта и/или порядок выбора потока из очереди готовых определяется приоритетами потоков. Именно так реализовано планирование в системе *Windows NT*, в которой квантование сочетается с динамическими абсолютными приоритетами.

3.2.8 Алгоритмы планирования

Рассмотрев цели, свойства и виды алгоритмов планирования, которые могут существовать в вычислительной системе, перейдем к краткому рассмотрению некоторых конкретных алгоритмов планирования (применительно задачам кратковременного планирования).

FCFS. Простейшим алгоритмом планирования является алгоритм, который принято обозначать аббревиатурой *FCFS* по первым буквам его английского названия – *First Come, First Served* (первым пришел, первым обслужен). Схема обслуживания задач согласно этой дисциплине представлена на рис. 12.

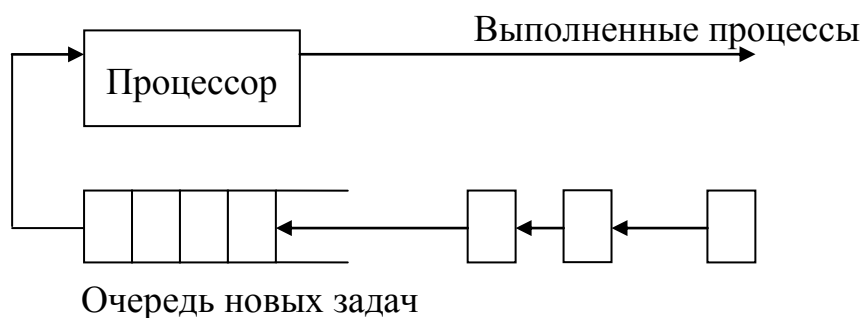


Рисунок 12 – Схема обслуживания задач согласно дисциплине FCFS

Такой алгоритм выбора процесса осуществляет невытесняющее планирование. Процесс, получивший в свое распоряжение процессор, занимает его до истечения текущего времени выполнения. После этого для выполнения выбирается новый процесс из начала очереди. Преимуществом алгоритма FCFS является легкость его реализации, недостатками – среднее время ожидания и среднее полное время выполнения для этого алгоритма существенно зависят от порядка расположения процессов в очереди. Для подтверждения этого тезиса рассмотрим следующий пример.

Пусть в состоянии готовности находятся три процесса p_0 , p_1 и p_2 , время выполнения которых в условных единицах, соответственно, составляет $t_1^{\text{вып}} = 13$, $t_2^{\text{вып}} = 4$ и $t_3^{\text{вып}} = 1$ (рис. 13). Для простоты будем полагать, что в выполнение процессов не вмешиваются операции типа ввода-вывода, а временем переключения контекста можно пренебречь.



Рисунок 13 – Временная диаграмма выполнения процессов в порядке p_0 , p_1 и p_2

Среднее время ожидания в данном случае можно рассчитать как $(0 + 13 + 17)/3 = 10$ усл. ед. времени, среднее полное время выполнения составит в этом случае $(13 + 17 + 18)/3 = 16$ усл. ед. времени.

В то же время, если очередность этих процессов изменить на обратную – p_2, p_1, p_0 , то временная диаграмма выполнения будет выглядеть так, как представлено на рис. 14.

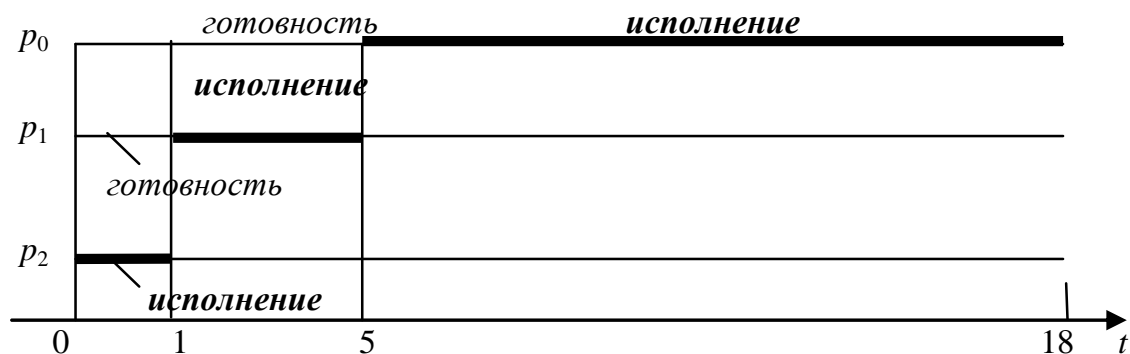


Рисунок 14 – Временная диаграмма выполнения процессов в порядке p_2 , p_1 и p_0

В этом случае среднее время ожидания будет $(5 + 1 + 0)/3 = 2$ усл. ед. времени, а среднее полное время выполнения будет $(18 + 5 + 1)/3 = 8$ усл. ед. времени.

Все это означает, что при изменении очередности выполнения одних и тех же процессов среднее время ожидания уменьшилось в 5 раз, а полное время выполнения – уменьшилось в 2 раза, что подтверждает большую чувствительность алгоритма *FCFS* к изменению порядка очередности выполнения процессов.

Round Robin. Модификацией алгоритма *FCFS* является алгоритм, получивший название *Round Robin* (детская карусель) или сокращенно *RR*. Схематично обслуживание задач согласно алгоритму *RR* представлено на рис. 15. По сути, это алгоритм *FCFS*, только реализованный в режиме вытесняющего планирования (очередной процесс передается на исполнение по таймеру по истечении определенного кванта времени).

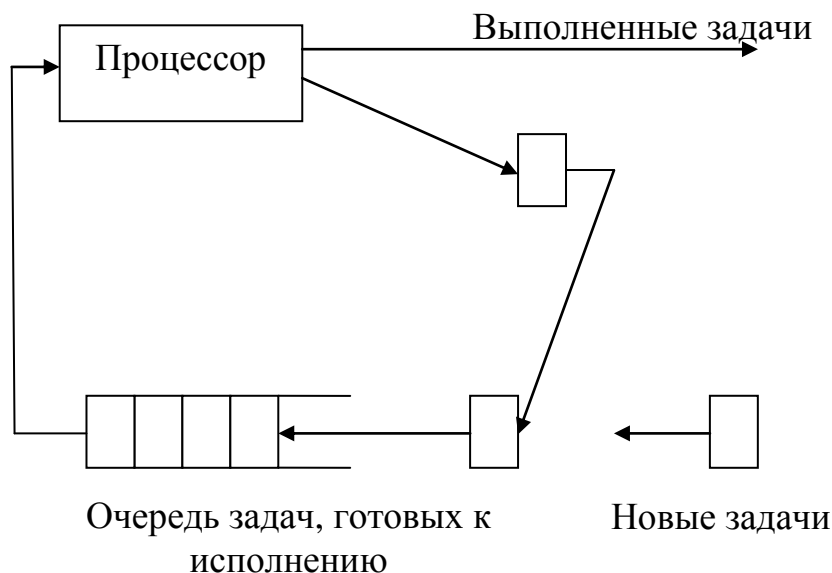


Рисунок 15 – Схема обслуживания задач согласно дисциплине *RR*

Можно представить себе все множество готовых процессов организованным циклически – процессы «сидят на карусели». Карусель враца-

ется так, что каждый процесс находится около процессора небольшой фиксированный квант времени (см. рис. 16). Пока процесс находится рядом с процессором, он получает процессор в свое распоряжение и может исполняться.

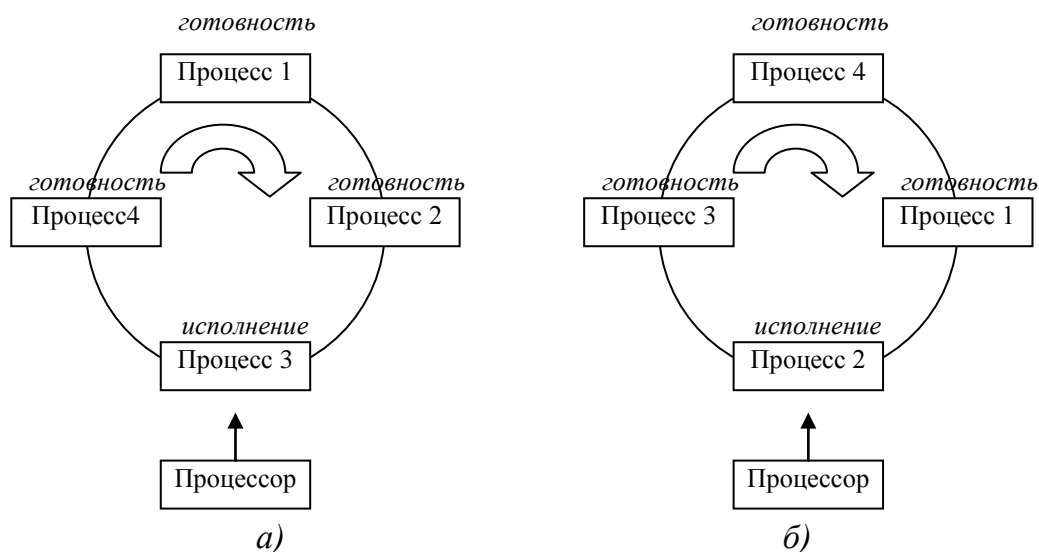


Рисунок 16 – Процессы «на карусели»:
а – момент времени t_1 , б – момент времени t_2

На производительность алгоритма *RR* существенно влияет величина кванта времени – при очень больших величинах кванта алгоритм *RR* вырождается в алгоритм *FCFS*, при очень малых – создается иллюзия того, что каждый из n процессов работает на собственном виртуальном процессоре с производительностью $\sim 1/n$ от производительности реального процессора (конечно, если не принимать во внимание время переключения контекста).

Shortest Job First. Если выбирать процесс не по порядку (как в *FCFS* и *RR*), а основываясь на его минимальном времени непрерывного использования процессора, то это позволит повысить производительность алгоритма планирования использования процессора. Описанный алгоритм получил название «кратчайшая работа первой» (англ. *Shortest Job First, SJF*).

Основную сложность при реализации алгоритма *SJF* представляет невозможность точно знать в каждом случае время исполнения очередного процесса.

3.3 **Взаимодействие и синхронизация процессов и потоков**

3.3.1 Независимые и взаимодействующие вычислительные процессы

Основной особенностью мультипрограммных ОС является то, что в их среде параллельно развивается несколько (последовательных) вычислительных процессов. С точки зрения внешнего наблюдателя эти последовательные вычислительные процессы выполняются одновременно («параллельно»). При этом под параллельными понимаются не только процессы, одновременно развивающиеся на различных процессорах, каналах и устройствах ввода-вывода, но и те последовательные процессы, которые разделяют центральный процессор и в своем выполнении хотя бы частично перекрываются во времени. Любая мультипрограммная ОС вместе с параллельно выполняющимися процессами может быть логически представлена как совокупность последовательных вычислений, которые, с одной стороны, состязаются за ресурсы, переходя из одного состояния в другое, а с другой – действуют почти независимо один от другого, но при этом образуя единую систему посредством установления разного рода связей между собой (путем пересылки сообщений и синхронизирующих сигналов).

Итак, *параллельными* называют такие последовательные вычислительные процессы, которые одновременно находятся в каком-нибудь активном состоянии. Два параллельных процесса могут быть *независимыми* (англ. *independent processes*) либо *взаимодействующими* (англ. *cooperating processes*).

Независимыми являются процессы, множества переменных которых не пересекаются. Под переменными в этом случае понимают файлы данных, а также области оперативной памяти, сопоставленные промежуточным и определенным в программе переменным. Независимые процессы не влияют на результаты работы друг друга, так как не могут изменять значения переменных друг у друга. Они могут только явиться причиной в задержках исполнения друг друга, так как вынуждены разделять ресурсы системы.

Взаимодействующие процессы используют совместно некоторые (общие) переменные, и выполнение одного процесса может повлиять на выполнение другого. Следует отметить, что при рассмотрении вопросов синхронизации вычислительных процессов (п. 3.3.2), из числа разделяемых ими ресурсов исключаются центральный процессор и программы, реализующие эти процессы, то есть с логической точки зрения каждому процессу соответствуют свои процессор и программа, хотя в реальных

системах обычно несколько процессов разделяют один процессор и одну или несколько программ. Многие ресурсы вычислительной системы могут совместно использоваться несколькими процессами, но в каждый момент времени к разделяемому ресурсу может иметь доступ только один процесс. Ресурсы, которые не допускают одновременного использования несколькими процессами, называются *критическими*.

3.3.2 Цели и средства синхронизации

Синхронизация процессов – приведение двух или более процессов к такому их протеканию, когда определенные стадии разных процессов совершаются в определенном порядке, либо одновременно. Потребность в синхронизации возникает только в мультипрограммной ОС и связана с совместным использованием аппаратных и информационных ресурсов вычислительной системы. Синхронизация⁸ необходима в любых случаях, когда параллельно протекающим процессам (потокам) необходимо взаимодействовать – для исключения гонок и тупиков при обмене данными между потоками, разделении данных, при доступе к устройствам ввода-вывода. Для ее организации используются *средства межпроцессного взаимодействия (Inter Process Communications, IPC)*, что отражает историческую первичность понятия «процесс» по отношению к понятию «поток». Среди наиболее часто используемых средств синхронизации – *сигналы, сообщения, семафоры и мьютексы*.

Выполнение процесса (потока) в мультипрограммной среде всегда имеет асинхронный характер. Очень сложно с полной определенностью сказать, на каком этапе выполнения будет находиться процесс в определенный момент времени. Даже в однопрограммном режиме не всегда можно точно оценить время выполнения задачи. Это время во многих случаях существенно зависит от значения исходных данных, которые влияют на количество циклов, направления разветвления программы, время выполнения операций ввода-вывода и т.п. В связи с тем, что исходные данные в разные моменты запуска задачи могут быть разными, то и время выполнения отдельных этапов и задачи в целом является весьма неопределенной величиной.

Еще более неопределенным является время выполнения программы в мультипрограммной системе. Моменты прерывания потоков, время нахождения их в очередях к разделяемым ресурсам, порядок выбора потоков для выполнения – все эти события являются результатом стечения многих обстоятельств и могут быть интерпретированы как случайные. В лучшем случае можно оценить вероятностные характеристики вычисли-

⁸ Все сказанное справедливо для синхронизации потоков, а если ОС не поддерживает потоки – то для синхронизации процессов.

тельного процесса, например, вероятность его завершения за данный период времени.

Учитывая вышеизложенное, можно сделать вывод, что потоки в общем случае (когда программист не предпринял специальных мер по их синхронизации) протекают независимо, асинхронно друг другу. Это справедливо как по отношению к потокам одного процесса, выполняющим общий программный код, так и по отношению к потокам разных процессов, каждый из которых выполняет собственную программу.

Для синхронизации потоков прикладных программ программист может использовать как собственные средства и приемы синхронизации, так и средства ОС. Например, два потока одного прикладного процесса могут координировать свою работу с помощью доступной для них обоим глобальной логической переменной, которая устанавливается в единицу при осуществлении некоторого события, например выработки одним потоком данных, нужных для продолжения работы другого. Однако во многих случаях более эффективными или даже единственно возможными являются средства синхронизации, предоставляемые ОС в форме системных вызовов. Так, потоки, принадлежащие разным процессам, не имеют возможности вмешиваться каким-либо образом в работу друг друга. Без посредничества ОС они не могут приостановить друг друга или оповестить о произошедшем событии.

Средства синхронизации используются ОС не только для синхронизации прикладных процессов, но и для ее внутренних нужд. Обычно разработчики ОС предоставляют в распоряжение прикладных и системных программистов широкий спектр средств синхронизации. Эти средства могут образовывать иерархию, когда на основе более простых средств строятся более сложные, а также могут быть функционально специализированными. Например, средства для синхронизации потоков одного процесса, средства для синхронизации потоков разных процессов при обмене данными и т.д.

3.3.3 Пример необходимости синхронизации

Пренебрежение вопросами синхронизации процессов, выполняющихся в режиме мультипрограммирования, может привести к их неправильной работе или даже к краху системы.

Рассмотрим пример, демонстрирующий необходимость синхронизации двух процессов, каждый из которых работает с некоторым общим ресурсом – программой печати файлов (принт-сервером, рис. 17).

Эта программа печатает по очереди содержимое файлов, имена которых последовательно в порядке поступления записывают в специальный общедоступный файл «заказов». Специальная переменная *NEXT*, доступная всем процессам-клиентам, содержит номер первой свободной

позиции для записи имени файла в файле «заказов». Процессы-клиенты считывают значение этой переменной, определяя тем самым очередную свободную позицию для записи имени файла, после чего значение переменной *NEXT* увеличивается на единицу.

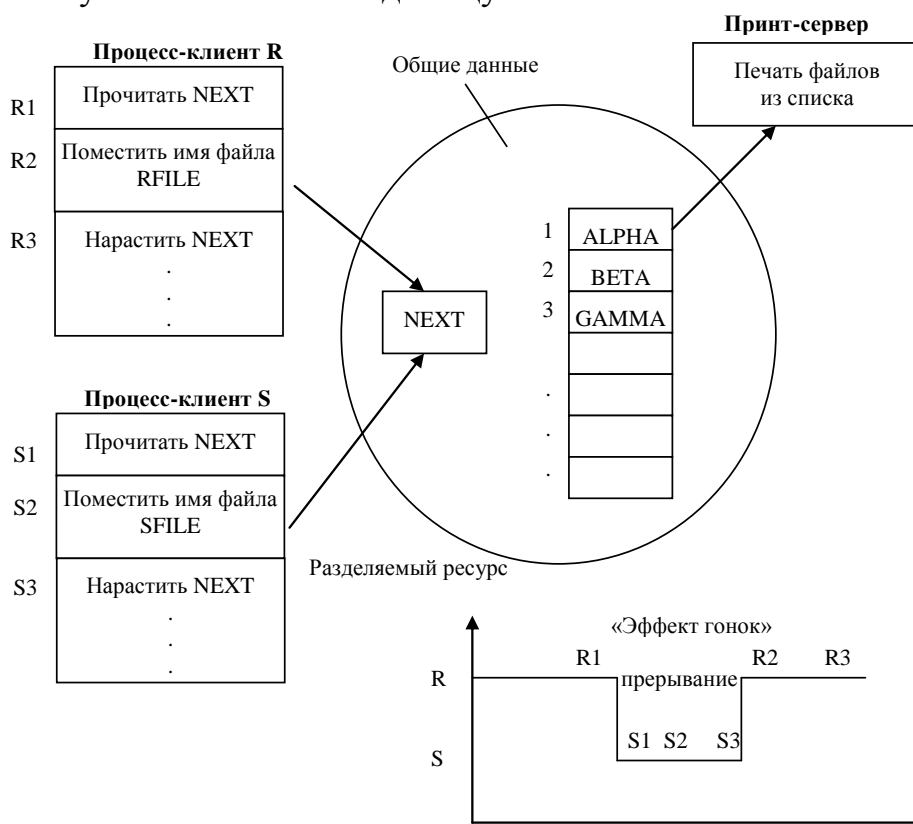


Рисунок 17 – Пример необходимости синхронизации при доступе к одному файлу заказов

Предположим, что в некоторый момент времени процесс *R* решил распечатать свой файл, для чего считал текущее значение переменной *NEXT* (пусть $NEXT = k$). Затем, например, вследствие исчерпания кванта времени, выделенного процессу, выполнение процесса было прервано.

Очередной процесс *S*, желающий также распечатать файл, считал значение k переменной *NEXT*, поместил в позицию k имя своего файла и нарастил значение переменной *NEXT* на единицу.

Когда в очередной раз управление будет передано процессу *R*, то он, продолжая свое выполнение, в полном соответствии со считанным ранее значением текущей свободной позиции, запишет имя файла для печати в файл «заказов» также в позицию k (поверх имени файла процесса *S*). Поэтому, файл процесса *S* не будет напечатан.

Очевидно, что решение проблемы синхронизации в данном случае зависит от взаимных скоростей процессов и моментов их прерывания, то есть при других названных параметрах потери файлов при печати

могло и не быть. Это подтверждает нерегулярность рассинхронизации процессов и наличие сложностей в обеспечении их синхронизации.

Ситуации подобные этой, когда два или более процессов обрабатывают разделяемые данные, и конечный результат зависит от соотношения скоростей процессов, называются *гонками*.

Важным понятием, используемым при синхронизации потоков, является понятие «критической секции» программы. *Критическая секция* – это часть программы, результат выполнения которой может непредсказуемо меняться, если переменные, относящиеся к этой части программы, изменяются другими потоками в то время, когда выполнение этой части еще не завершено. Критическая секция всегда определяется по отношению к определенным критическим данным, при несогласованном изменении которых могут возникнуть нежелательные эффекты. В примере, представленном на рис. 17, такой критической секцией является файл «заказов», являющийся разделяемым ресурсом для процессов *R* и *S*.

3.3.4 Механизмы синхронизации

Как было отмечено выше в п. 3.3.2, ОС поддерживает целый ряд различных средств синхронизации процессов, позволяющих избежать проблем, подобной той, что описана выше и представлена на рис. 17. Рассмотрим каждое из этих средств синхронизации процессов более подробно.

Блокирующие переменные. Для синхронизации потоков одного процесса прикладной программист может использовать *глобальные блокирующие переменные*. С этими переменными, к которым все потоки процесса имеют прямой доступ, программист работает, не обращаясь к системным вызовам ОС. Каждому набору критических данных ставится в соответствие двоичная переменная, которой поток присваивает значение 0, когда он входит в критическую секцию, и значение 1, когда он ее покидает.

На рис. 18 показан фрагмент алгоритма потока, использующего для реализации взаимного исключения доступа к критическим данным *D* блокирующую переменную $F(D)$. Перед входом в критическую секцию поток проверяет, не работает ли уже какой-нибудь другой поток с данными *D*. Если переменная $F(D)$ установлена в 0, то данные заняты и проверка циклически повторяется. Если же данные свободны ($F(D) = 1$), то значение переменной $F(D)$ устанавливается в 0 и поток входит в критическую секцию. После того, как поток выполнит все действия с данными *D*, значение переменной $F(D)$ снова устанавливается равным 1.



Рисунок 18 – Работа в критической секции с использованием блокирующих переменных

Блокирующие переменные могут использоваться не только при доступе к разделяемым данным, но и при доступе к разделяемым ресурсам любого вида. Если все потоки разработаны с учетом вышеописанных соглашений, то взаимное исключение гарантируется. При этом потоки могут быть прерваны ОС в любой момент и в любом месте, в том числе в критической секции. Однако следует заметить, что одно ограничение на прерывания все же имеется – нельзя прерывать поток между выполнением операций проверки и установки блокирующей переменной, т.к. это нарушит принцип взаимного исключения.

Действительно, пусть в результате проверки переменной поток определил, что ресурс свободен, но сразу после этого, не успев установить переменную в 0, был прерван. За время его приостановки другой поток занял ресурс, вошел в свою критическую секцию, но также был прерван, не завершив работы с разделяемым ресурсом. Когда управление было возвращено первому потоку, он, считая ресурс свободным, установил признак занятости и начал выполнять операции в критической секции. Таким образом, в одной критической секции производят работу два различных потока, а это потенциально может привести к нежелательным последствиям.

Во избежание подобных ситуаций в системе команд многих компьютеров предусмотрена единая, неделимая команда анализа и присвоения значения логической переменной (например, команды *BTC*, *BTR* и *BT5* процессора *Pentium*). При отсутствии такой команды в процессоре соответствующие действия должны реализовываться специальными системными примитивами – базовыми функциями ОС, которые бы запрещали прерывания на протяжении всей операции проверки и установки.

Реализация взаимного исключения с использованием глобальных блокирующих переменных имеет существенный недостаток: в течение времени, когда один поток находится в критической секции, другой поток, которому требуется тот же ресурс, получив доступ к процессору, будет непрерывно опрашивать блокирующую переменную, бесполезно растрачивая выделяемое ему процессорное время, которое могло бы быть использовано для выполнения какого-нибудь другого потока. Для устранения этого недостатка во многих ОС предусматриваются специальные системные вызовы (аппарат событий) для работы с критическими секциями.

Семафоры. В разных ОС аппарат событий реализуется по своему, но в любом случае используются системные функции аналогичного назначения, которые условно именуют $WAIT(x)$ и $POST(x)$, где x – идентификатор некоторого события. На рис. 19 показан фрагмент алгоритма процесса, использующего эти функции.

Если ресурс занят, то процесс не выполняет циклический опрос, а вызывает системную функцию $WAIT(D)$, здесь D обозначает событие, заключающееся в освобождении ресурса D . Функция $WAIT(D)$ переводит активный процесс в состояние *ожидание* и делает отметку в его дескрипторе о том, что процесс ожидает события D . Процесс, который в это время использует ресурс D , после выхода из критической секции выполняет системную функцию $POST(D)$, ОС просматривает очередь ожидающих процессов и переводит процесс, ожидающий события D , в состояние *готовность*.

Обобщающее средство синхронизации процессов с использованием изложенных выше принципов, названное *семафор*, предложил Э.В. Дейкстра (*Edsger Wybe Dijkstra*) – выдающийся нидерландский учёный, идеи которого оказали огромное влияние на развитие компьютерной индустрии. Семафор – объект, позволяющий войти в заданный участок кода не более чем n потокам. Осуществляется это путем использования специальной защищенной переменной S , значения которой можно опрашивать и менять только при помощи специальных операций $P(S)$ и $V(S)$ и операции инициализации. Семафор может принимать целое неотрицательное значение. При выполнении потоком операции P над семафором S значение семафора уменьшается на 1 при $S > 0$, или

поток блокируется, «ожидая на семафоре», при $S = 0$. При выполнении операции $V(S)$ происходит пробуждение одного из потоков, ожидающих на семафоре S , а если таковых нет, то значение семафора увеличивается на 1 (как можно заметить, по сути – операции P и V аналогичны функциям $POST$ и $WAIT$). Как следует из вышесказанного, при входе в критическую секцию поток должен выполнять операцию $P(S)$, а при выходе из критической секции – операцию $V(S)$.



Рисунок 19 – Работа в критической секции с использованием системных функций $WAIT(D)$ и $POST(D)$

В простом случае, когда семафор работает в режиме 2-х состояний ($S > 0$ и $S = 0$), его алгоритм работы полностью совпадает с алгоритмом мьютекса, а S выполняет роль *блокирующей переменной*.

Рассмотрим использование семафоров на классическом примере взаимодействия двух процессов, выполняющихся в режиме мультипрограммирования, один из процессов пишет данные в буферный пул, а другой считывает их из буферного пула. Пусть буферный пул состоит из N буферов, каждый из которых может содержать одну запись. Процесс

«писатель» должен приостанавливаться, когда все буфера оказываются занятыми, и активизироваться при освобождении хотя бы одного буфера. Напротив, процесс «читатель» приостанавливается, когда все буферы пусты, и активизируется при появлении хотя бы одной записи. Введем следующие семафоры:

- e – число пустых буферов (« e » – *empty*);
- f – число заполненных буферов (« f » – *full*);
- b – двоичный семафор, используемый для обеспечения взаимного исключения.

Операции с буфером (запись, считывание) являются критическими секциями. Процессы могут быть описаны следующим образом:

```
// Глобальные переменные
#define N 256
int e = N, f = 0, b = 1;
void Writer ()
{
    while(1)
    {
        PrepareNextRecord(); /* подготовка новой записи */
        P(e); /* Уменьшить число свободных буферов, если они есть */
        /* в противном случае ждать, пока они освободятся */
        P(b); /* Вход в критическую секцию */
        AddToBuffer(); /* Добавить новую запись в буфер */
        V(b); /* Выход из критической секции */
        V(f); /* Увеличить число занятых буферов */
    }
}
void Reader ()
{
    while(1)
    {
        P(f); /* Уменьшить число занятых буферов, если они есть */
        /* в противном случае ждать, пока они появятся */
        P(b); /* Вход в критическую секцию */
        GetFromBuffer(); /* Взять запись из буфера */
        V(b); /* Выход из критической секции */
        V(e); /* Увеличить число свободных буферов */
        ProcessRecord(); /* Обработать запись */
    }
}
```

Достоинства использования операций на семафоре:

1. Пассивное ожидание (постановка в очередь и автоматическая выдача ресурсов).
2. Возможность управления группой однородных ресурсов.

К недостаткам использования семафоров относят то, что некорректное использование операций на семафоре может привести к нарушению работоспособности параллельных систем.

Действительно, если в рассмотренном примере переставить местами операции $P(e)$ и $P(b)$ в функции «писатель», то при некотором стечении обстоятельств эти два процесса могут взаимно заблокировать друг друга. Так, пусть «писатель» первым войдет в критическую секцию и обнаружит отсутствие свободных буферов; он начнет ждать, когда «читатель» возьмет очередную запись из буфера, но «читатель» не сможет этого сделать, так как для этого необходимо войти в критическую секцию, вход в которую заблокирован процессом «писатель».

Мониторы. В связи с тем, что использование семафоров требует особой осторожности (одна незначительная ошибка может привести к останову системы), для облегчения корректного функционирования программ было предложено высокоуровневое средство синхронизации, называемое *монитором*. Мониторы представляют собой тип данных, обладающий собственными переменными, определяющими его состояние. Значения этих переменных извне могут быть изменены только с помощью вызова функций-методов монитора. Функции-методы могут использовать в работе только данные, находящиеся внутри монитора, и свои параметры. Важной особенностью мониторов является то, что в любой момент времени только один процесс может быть активен, т.е. находиться в состоянии *готовность* или *исполнение*, внутри данного монитора. Вот пример описания монитора:

```
monitor monitor_name
{
    описание переменных;
    void m1(...){... }
    void m2(...){... }
    ...
    void mn(...){... }
    { блок инициализации внутренних переменных; }
}
```

Однако одних только взаимоисключений недостаточно для того, чтобы в полном объеме реализовать решение задач, возникающих при взаимодействии процессов. Необходимы средства организации очередности процессов, подобно семафорам $f(full)$ и $e(empty)$ в примере.

Для этого в мониторах было введено понятие условных переменных, над которыми можно совершать две операции *wait* и *signal*, отчасти похожие на операции P и V над семафорами.

```
monitor ProducerConsumer
{
    condition full, empty;
    int count;
```

```

void put()
{
    if(count == N) full.wait;
    put_item;
    count += 1;
    if(count == 1) empty.signal;
}
void get()
{
    if(count == 0) empty.wait;
    get_item();
    count -= 1;
    if(count == N-1) full.signal;
}
}

Producer: while(1)
{
    produce_item;
    ProducerConsumer.put();
}

Consumer: while(1)
{
    ProducerConsumer.get();
    consume_item;
}

```

Функция монитора выполняет операцию *wait* над какой-либо условной переменной. При этом процесс, выполнивший операцию *wait*, блокируется, становится неактивным, и другой процесс получает возможность войти в монитор.

Когда ожидаемое событие происходит, другой процесс внутри функции совершает операцию *signal* над той же самой условной переменной. Это приводит к пробуждению ранее заблокированного процесса, и он становится активным.

Отличительной особенностью мониторов является то, что исключение входа нескольких процессов в монитор реализуется не программистом, а компилятором, что делает ошибки менее вероятными. Реализация мониторов требует использования специальных языков программирования и компиляторов для них (например, «*параллельный Евклид*», «*параллельный Паскаль*», *Java*).

Следует отметить, что эмуляция *семафоров* значительно проще эмуляции *мониторов* – в отличие от семафоров Дейкстры, условные переменные мониторов не запоминают предысторию, поэтому операция *signal* всегда должна выполняться после операции *wait*. Если операция *signal* выполняется над условной переменной, с которой не связано ни одного заблокированного процесса, то информация о произошедшем событии будет утеряна, и выполнение операции *wait* всегда будет приводить к блокированию процесса.

Сигналы. Вообще, *сигнал* – это некоторое значимое событие (например, прерывание), источником которого может быть ОС или иная составляющая вычислительной системы. Сигналы вызывают прерывание процесса, выполнение заранее предусмотренных действий.

Сигналы могут вырабатываться как результат работы самого процесса (т.е. вырабатываться синхронно), а могут быть направлены процессу другим процессом (т.е. вырабатываться асинхронно). Синхронные сигналы чаще всего приходят от системы прерываний процессора и свидетельствуют о действиях процесса, блокируемых аппаратурой, например деление на нуль, ошибка адресации, нарушение защиты памяти и т.д. Примером асинхронного сигнала является сигнал с терминала (например, сигнал об оперативном снятии процесса с выполнения с помощью некоторой нажатой комбинации клавиш – *Ctrl + C*, *Ctrl + Break*), в результате чего ОС вырабатывает сигнал и направляет его активному процессу.

Сигналы обеспечивают логическую связь между процессами, а также между процессами и пользователями (терминалами). Поскольку посылка сигнала предусматривает знание идентификатора процесса, то взаимодействие посредством сигналов возможно только между родственными процессами, которые могут получить данные об идентификаторах друг друга.

Следует отметить, что в распределенных системах, состоящих из нескольких процессоров, каждый из которых имеет собственную оперативную память, блокирующие переменные, семафоры, сигналы и другие аналогичные средства, основанные на разделяемой памяти, оказываются непригодными. В таких системах синхронизация может быть реализована только посредством обмена сообщениями, рассмотренными ниже в п. 3.3.6.

3.3.5 Проблемы синхронизации

В п. 3.3.4 рассмотрен ряд механизмов ОС, используемых для синхронизации параллельно действующих процессов. Однако, в некоторых случаях, при конкуренции нескольких процессов за обладание конечным числом ресурсов, может возникнуть ситуация, когда запрашиваемый процессом ресурс недоступен, и ОС переводит данный процесс в состояние ожидания. В то же время, если этот же ресурс удерживается другим ожидающим процессом, то первый процесс не сможет сменить свое состояние. Подобная ситуация называется взаимной *блокировкой*, *дедлоком* (англ. *deadlocks*), *клинчем* (англ. *clinch*), или *тупиком*. Например, тупик возникнет при перестановке местами операций $P(e)$ и $P(b)$ в примере с процессами «читатель» и «писатель», рассмотренном выше. В этом случае ни один из этих потоков не сможет завершить начатую ра-

боту и возникнет тупиковая ситуация, которая не может разрешиться без внешнего воздействия.

Тупиковые ситуации следует отличать от простых очередей хотя те и другие возникают при совместном использовании ресурсов и внешне выглядят схоже – процесс приостанавливается и ждет освобождения ресурса. Однако очередь – это нормальное явление, неотъемлемый признак высокого коэффициента использования ресурсов при случайном поступлении запросов. Очередь появляется тогда, когда ресурс недоступен в данный момент, но освободится через некоторое время, позволив потоку продолжить выполнение. Тупик же, что видно из его названия, является в некотором роде неразрешимой ситуацией. Необходимым условием возникновения тупика является потребность потока сразу в нескольких ресурсах.

Разрешение проблемы тупиков может быть осуществлено путем:

- распознавания тупиков;
- предотвращения тупиков;
- восстановления системы после тупиков;
- игнорирования.

Рассмотрим каждый из аспектов проблемы тупиков более подробно.

Распознавание тупиков. В случаях, когда тупиковая ситуация образована многими процессами, использующими много ресурсов, распознавание тупика является нетривиальной задачей. Тупиковые ситуации следует отличать от простых очередей, хотя и те и другие возникают при совместном использовании ресурсов и внешне выглядят схоже: процесс приостанавливается и ждет освобождения ресурса. Однако очередь – это нормальное явление, неотъемлемый признак высокого коэффициента использования ресурсов при случайном поступлении запросов. Очередь появляется тогда, когда ресурс недоступен в данный момент, но освободится через некоторое время, позволив потоку продолжить выполнение, а тупик – является в некотором роде неразрешимой ситуацией.

Существуют формальные, программно-реализованные методы распознавания тупиков, основанные на ведении таблиц распределения ресурсов и таблиц запросов к занятым ресурсам, анализ которых позволяет обнаружить взаимные блокировки.

Предотвращение тупиков. Тупики могут быть предотвращены на стадии проектирования и разработки программного обеспечения, чтобы тупик не мог возникнуть ни при каком соотношении взаимных скоростей процессов.

В качестве необходимых условий возникновения тупиков называют следующие четыре:

1) Условие взаимоисключения (англ. *Mutual exclusion*). Одновременно использовать ресурс может только один процесс.

2) Условие ожидания ресурсов (англ. *Hold and wait*). Процессы удерживают ресурсы, уже выделенные им, и могут запрашивать другие ресурсы.

3) Условие «неперераспределяемости» (англ. *No preemption*). Ресурс, выделенный ранее, не может быть принудительно забран у процесса до его завершения. Освобождены они могут быть только процессом, который их удерживает.

4) Условие кругового ожидания (англ. *Circular wait*). Существует кольцевая цепь процессов, в которой каждый процесс ждет доступа к ресурсу, удерживаемому другим процессом цепи.

Соответственно, решить проблему возникновения тупиков можно путем избегания описанных выше ситуаций 2-4 (ситуация взаимоисключения – объективна):

- ситуацию ожидания дополнительных ресурсов можно нарушить, если потребовать, чтобы процессы запрашивали сразу все ресурсы одновременно (очевидные недостатки – снижение уровня мультипрограммирования и нерациональное использование ресурсов);
- ситуацию «неперераспределяемости» можно нарушить, если потребовать, чтобы процесс, который не получил дополнительных ресурсов, сам освобождал удерживаемые;
- ситуацию кругового ожидания можно предотвратить, если процессы запрашивают ресурсы в заранее определенном порядке, то есть ресурсы имеют уникальные упорядоченные номера, которые распределяются в соответствии с некоторым планом (планирование распределения ресурсов).

Восстановление системы после тупиков. При возникновении тупиковой ситуации не обязательно снимать с выполнения все заблокированные процессы, а можно:

- снять только часть из них, при этом освобождая ресурсы, ожидаемые остальными процессами;
- вернуть некоторые процессы в область «свопинга»;
- совершить «откат» некоторых процессов до некоторой контрольной точки (т.е. места, где возможен тупик), в которой запоминается вся информация, необходимая для восстановления выполнения программы с данного места.

Игнорирование. Простейший подход – не замечать проблему тупиков. Для того чтобы принять такое решение, необходимо оценить ве-

роятность возникновения взаимоблокировки и сравнить ее с вероятностью ущерба от других отказов аппаратного и программного обеспечения. Подход большинства современных ОС (*Unix*, *Windows* и др.) состоит в том, чтобы игнорировать данную проблему в предположении, что маловероятный случайный тупик предпочтительнее, чем внедрение сложных и дорогостоящих средств борьбы с тупиками, и жертвовать производительностью системы или удобством пользователей (например, ограничивать пользователей в числе процессов, открытых файлов и т.п.) не стоит.

3.3.6 Механизмы межпроцессного взаимодействия

Помимо решения задачи синхронизации процессов и потоков, в ОС требуется обеспечение и обмена данными между ними. Если речь идет о необходимости обмена данными между потоками одного процесса, то решение этой задачи не представляет никакой сложности, т.к. они имеют общее адресное пространство и файлы, и получают беспрепятственный доступ к данным друг друга. Другое дело – обмен данными потоков, выполняющихся в рамках разных процессов. В этом случае обмену препятствуют развитые средства ОС по защите процессов друг от друга, находящихся к тому же в разных адресных пространствах.

Операционная система имеет доступ ко всем областям памяти, поэтому она может играть роль посредника в информационном обмене потоков: при возникновении необходимости в обмене данными поток обращается с запросом к ОС, по которому ОС, пользуясь своими привилегиями, создает различные системные средства связи, такие, например, как *каналы*, *очереди сообщений* или *разделяемую память*. Эти средства (как и рассмотренные выше средства синхронизации процессов), относятся к классу средств межпроцессного взаимодействия.

Следует помнить, что многие из средств межпроцессного обмена данными выполняют также и функции синхронизации: в том случае, когда данные для процесса-получателя отсутствуют, последний переводится в состояние ожидания средствами ОС, а при поступлении данных от процесса-отправителя процесс-получатель активизируется.

Каналы. Один из методов взаимодействия между процессами получил название *канал связи*, *конвейер* или *транспортёр* (англ. *pipe*) – однонаправленный механизм передачи данных (неструктурированного потока байтов) между процессами без необходимости создания файла на диске. Канал представляет собой буфер в оперативной памяти, поддерживающий очередь байт согласно *FIFO*. Для программиста, использующего канал, этот буфер выглядит как безымянный файл, в который можно писать и читать, осуществляя тем самым обмен данными.

Механизм каналов часто используется не только программистами, но и обычными пользователями ОС для организации конвейера команд в случае, когда выходные данные одной команды пользователя становятся входными данными для другой команды. Так, наиболее простой вариант канала создает оболочка *Unix* между программами, запускаемыми из командной строки, разделенными символом «*|*». Например, командная строка *dmesg | less* создает канал от программы *dmesg*, выводящей отладочные сообщения ядра при загрузке к программе постраничного просмотра *less*.

Обычный канал получил развитие, результатом которого стало появление *именованного канала* или *именованного конвейера* – зарегистрированного в системе канала (по сути – запись в каталоге файловой системы), который разрешено использовать различным процессам или потокам (не обязательно родственным). Реализуется это путем создания одним, а чтения – другим процессом (поток) файла типа *FIFO* с одним и тем же указанным в процессах именем.

Также можно создать канал с помощью, например, команд *mknod* или *mkfifo* и настроить *gzip* на сжатие того, что туда попадает:

```
mkfifo pipe
```

```
gzip -9 -c < pipe > out
```

Параллельно, в другом процессе можно выполнить:

```
cat file > pipe,
```

что приведёт к сжатию передаваемых данных *gzip*-ом.

Именованный канал существует в ОС и после завершения процесса, поэтому после окончания использования он должен быть «отсоединен» или удален. Следует иметь в виду, что именованные каналы используют файловую систему только для хранения имени конвейера, а данные между процессами передаются через буфер в оперативной памяти, как и в случае обычного канала.

Очереди сообщений. Механизм *очереди сообщений* (англ. *queues*) в целом схож с механизмом каналов, но позволяет процессам и потокам обмениваться структурированными сообщениями. При этом синхронизация осуществляется по сообщениям, то есть процесс, пытающийся прочесть сообщение, переводится в состояние ожидания в том случае, если в очереди нет ни одного полного сообщения.

Следует отметить, что в распределенных системах, состоящих из нескольких процессоров и неразделяемых блоков памяти, использование таких средств синхронизации как блокирующие переменные, семафоры, сигналы является непригодным. В таких системах синхронизацию следует реализовывать только посредством обмена сообщениями.

С помощью очередей можно из одной или нескольких задач независимым образом посылать сообщения некоторой задаче-приемнику. При этом только процесс-приемник может читать и удалять сообщения из очереди, а процессы-клиенты имеют право лишь помещать в очередь свои сообщения. Таким образом, очередь работает только в одном направлении, а если необходима двухсторонняя связь, то следует создать две очереди.

Очереди сообщений являются глобальными средствами коммуникаций для процессов ОС, как и именованные каналы, так как каждая очередь имеет в пределах ОС уникальное имя. В ОС *Unix* в качестве такого имени используется числовое значение – так называемый ключ. Ключ является числовым аналогом имени файла, при использовании одного и того же значения ключа процессы будут работать с одной и той же очередью сообщений.

Работа с очередями сообщений имеет ряд отличий от работы с каналами:

1) Очереди сообщений предоставляют возможность использовать несколько дисциплин обработки сообщений (*FIFO*, *LIFO*, приоритетный доступ, произвольный доступ).

2) Если при чтении сообщения оно удаляется из конвейера, то при чтении сообщения из очереди этого не происходит, и сообщение может быть прочитано несколько раз.

3) В очередях присутствуют не непосредственно сами сообщения, а их адреса в памяти и размер. Эта информация размещается системой в сегменте памяти, доступном для всех задач, общающихся с помощью данной очереди. Каждый процесс, использующий очередь, должен предварительно получить разрешение на доступ в общий сегмент памяти с помощью системных запросов *API*, ибо очередь – это системный механизм, и для работы с ним требуются системные ресурсы и, соответственно, обращение к самой ОС.

Для наглядности, при обзоре механизмов реализации средств меж-процессного взаимодействия здесь и далее будем использовать конкретные системные вызовы ОС *Unix*.

Так, для работы с очередью сообщений процесс должен воспользоваться системным вызовом *msgget*, указав в качестве параметра значение ключа. Если очередь с данным ключом в настоящий момент не используется ни одним процессом, то для нее резервируется область памяти, а затем процессу возвращается идентификатор очереди, который, как и дескриптор файла, имеет локальное для процесса значение. Если же очередь уже используется, то процессу просто возвращается ее идентификатор.

После открытия очереди процесс может помещать в него сообщения с помощью вызова *msgsnd* или читать сообщения с помощью вызова *msgrsv*. Программист может влиять на то, как ОС будет обрабатывать ситуацию, когда процесс пытается читать сообщения, которые еще не поступили в очередь, то есть на синхронизацию процесса с данными.

Разделяемая память. Разделяемая память представляет собой сегмент физической памяти, отображенной в виртуальное адресное пространство двух или более процессов. Механизм разделяемой памяти поддерживается подсистемой виртуальной памяти, которая настраивает таблицы отображения адресов для процессов, запросивших разделение памяти, так что одни и те же адреса некоторой области физической памяти соответствуют виртуальным адресам разных процессов.

Для работы с разделяемой памятью используются четыре системных вызова:

- *shmget* – создает новый сегмент разделяемой памяти или находит существующий сегмент с тем же ключом;
- *shmat* – подключает сегмент с указанным дескриптором к виртуальной памяти обращающегося процесса;
- *shmdt* – отключает от виртуальной памяти ранее подключенный к ней сегмент с указанным виртуальным адресом начала;
- *shmctl* – служит для управления разнообразными параметрами, связанными существующим сегментом.

После того как сегмент разделяемой памяти подключен к виртуальной памяти процесса, процесс может обращаться к соответствующим элементам памяти с использованием обычных машинных команд чтения и записи, не прибегая к дополнительным системным вызовам. Синтаксис системного вызова *shmget* выглядит следующим образом:

shmid = *shmget* (*key*, *size*, *flag*);

Параметр *size* определяет желаемый размер сегмента в байтах. Далее, если в таблице разделяемой памяти находится элемент, содержащий заданный ключ, и права доступа не противоречат текущим характеристикам обращающегося процесса, то значением системного вызова является дескриптор существующего сегмента (и обратившийся процесс так и не узнает реального размера сегмента, хотя впоследствии его можно узнать с помощью системного вызова *shmctl*). В противном случае создается новый сегмент, размер которого не меньше, чем установленный в системе минимальный размер сегмента разделяемой памяти, и не больше, чем установленный максимальный размер. Создание сегмента не означает немедленного выделения для него основной памяти. Это действие откладывается до первого системного вызова подключения сегмента к виртуальной памяти некоторого процесса. Аналогично, при

выполнении последнего системного вызова отключения сегмента от виртуальной памяти соответствующая основная память освобождается.

Подключение сегмента к виртуальной памяти выполняется путем обращения к системному вызову *shmat*:

virtaddr = shmat (id, addr, flags).

Здесь *id* – ранее полученный дескриптор сегмента, *addr* – требуемый процессу виртуальный адрес, который должен соответствовать началу сегмента в виртуальной памяти. Значением системного вызова является реальный виртуальный адрес начала сегмента (его значение не обязательно совпадает со значением параметра *addr*). Если значением *addr* является нуль, ядро выбирает подходящий виртуальный адрес начала сегмента.

Для отключения сегмента от виртуальной памяти используется системный вызов *shmdt*:

shmdt (addr),

где *addr* – виртуальный адрес начала сегмента в виртуальной памяти, ранее полученный с помощью системного вызова *shmat*. При этом система гарантирует (опираясь на данные таблицы сегментов процесса), что указанный виртуальный адрес действительно является адресом начала разделяемого сегмента в виртуальной памяти данного процесса.

Для управления памятью служит системный вызов *shmctl*:

shmctl (id, cmd, shsstatbuf).

Параметр *cmd* идентифицирует требуемое конкретное действие, то есть ту или иную функцию. Наиболее важной является функция уничтожения сегмента разделяемой памяти, которое производится следующим образом. Если к моменту выполнения системного вызова ни один процесс не подключил сегмент к своей виртуальной памяти, то основная память, занимаемая сегментом, освобождается, а соответствующий элемент таблицы разделяемых сегментов объявляется свободным. В противном случае в элементе таблицы сегментов выставляется флаг, запрещающий выполнение системного вызова *shmget* по отношению к этому сегменту, но процессам, успевшим получить дескриптор сегмента, по-прежнему разрешается подключать сегмент к своей виртуальной памяти. При выполнении последнего системного вызова отключения сегмента от виртуальной памяти операция уничтожения сегмента завершается.

3.4 Вопросы для самопроверки

1. Чем характеризуется мультипрограммирование в современных ОС?

2. Какие критерии используют для определения эффективности вычислительных систем при использовании мультипрограммирования?
3. Какие критерии к «мультипрограммной смеси» применяют в системах пакетной обработки?
4. Какие особенности имеет мультипрограммирование в системах пакетной обработки?
5. Какие критерии к «мультипрограммной смеси» применяют в системах разделения времени?
6. Какие особенности имеет мультипрограммирование в системах разделения времени?
7. Какие критерии к «мультипрограммной смеси» применяют в системах реального времени?
8. Какие особенности имеет мультипрограммирование в системах реального времени?
9. Чем характеризуется мультипроцессорная обработка? На какие виды ее разделяют?
10. Что означает симметричная и несимметричная мультипроцессорная обработка с точки зрения архитектуры?
11. Что означает симметричная и несимметричная мультипроцессорная обработка с точки зрения организации вычислительного процесса?
12. Что означает масштабирование в многопроцессорной архитектуре по вертикали? По горизонтали?
13. Что такое «кластерная мультипроцессорная система»?
14. Какова роль прерываний при мультипрограммировании?
15. На какие классы делят прерывания? В чем отличия этих классов между собой?
16. Какова пошаговая последовательность реализации прерываний в вычислительной системе? Каковы при этом главные функции механизма прерываний?
17. Что такое «обработчик прерывания»? В чем его функция?
19. В какие моменты при обработке прерывания система должна отключать возможности прерывания?
20. Для чего в ОС используется супервизор прерываний? Каково его место в общей схеме обработки прерываний?
21. Что такое «приоритет прерывания»? Обработка прерываний каких компонентов вычислительной системы обладает более высоким, а каких – более низким приоритетом?
22. В чем отличие между терминами программа или задание и термином процесс?
23. Что такое поток или нить исполнения? В чем заключаются основные отличия нити от процесса?

24. Каковы основные предпосылки появления потоков?
25. Какие основные этапы создания процесса?
26. Какую информацию содержит описатель потока?
27. Какие элементы выделяют на диаграмме состояний процессов?
Как при этом реализуется механизм смены состояний процессов?
28. В чем заключается планирование использования процессора?
Какие задачи предполагаются к решению при планировании?
29. В чем заключается диспетчеризация процессов? В чем ее отличие от планирования? Какие основные задачи при этом должны быть решены?
30. Какие выделяют уровни планирования? Чем характеризуется каждый из уровней?
31. Какие основные цели и свойства алгоритмов планирования?
32. Что такое параметры планирования? Какие виды параметров выделяют?
33. Какие виды планирования выделяют?
34. Какие разновидности приоритетного планирования выделяют?
35. Какие существуют алгоритмы планирования? В чем заключаются их достоинства и недостатки?
36. Что понимают под параллельно действующими процессами? В чем отличие независимых и взаимодействующих процессов?
37. В чем заключается обеспечение синхронизации процессов?
38. Какие механизмы ОС относят к средствам синхронизации процессов и потоков?
39. Какие примеры необходимости обеспечения синхронизации процессов Вам известны?
40. Каков механизм синхронизации процессов с использованием блокирующей переменной, семафоров, мониторов, сигналов? Какие характерные особенности каждого из этих методов, достоинства и недостатки?
41. В чем заключается и когда возникает взаимная блокировка процессов (тупик)? Какие подзадачи требуют разрешения для решения проблемы тупиков? Какие условия необходимы для возникновения тупиков и как их избежать? Какие существуют пути восстановления системы после тупиков?
42. Какие механизмы ОС относят к классу средств межпроцессного взаимодействия? В чем заключаются основные особенности практического использования каждого из этих средств?

4. УПРАВЛЕНИЕ ПАМЯТЬЮ

4.1 Основные положения

Запоминающие устройства компьютера разделяют, как минимум, на два вида:

- основную (главную, оперативную);
- вторичную (внешнюю) память.

Основная память (англ. *memory*) представляет собой упорядоченный массив однобайтовых ячеек, каждая из которых имеет свой уникальный адрес (номер). Процессор извлекает команду из основной памяти, декодирует и выполняет ее. Для выполнения команды могут потребоваться обращения еще к нескольким ячейкам основной памяти. Обычно основная память изготавливается с применением полупроводниковых технологий и теряет свое содержимое при отключении питания.

Вторичную память (англ. *storage*) также можно рассматривать как одномерное линейное адресное пространство, состоящее из последовательности байтов. Как правило, внешняя память реализована с использованием различного рода дисков. В отличие от ОП, она является энергонезависимой, имеет существенно большую емкость и используется в качестве расширения основной памяти.

Эту схему можно дополнить еще несколькими промежуточными уровнями. Разновидности памяти могут быть объединены в иерархию по убыванию времени доступа, возрастанию цены и увеличению емкости (рис. 20).



Рисунок 20 – Иерархическая организация памяти

Как правило, иерархическая многоуровневая схема используется следующим образом. Информация, которая находится в памяти верхнего уровня, обычно сохраняется также и на более низких уровнях. Если

процессор не обнаруживает нужную информацию на i -м уровне, он начинает искать ее на $i-1$, $i-2$,... уровнях. Когда нужная информация найдена, она переносится на уровни, находящиеся выше в иерархии и обеспечивающие более высокое быстродействие.

4.1.1 Задачи по управлению памятью

Особая роль памяти объясняется тем, что процессор может выполнять инструкции программы только в том случае, если они находятся в памяти. В ранних ОС управление памятью сводилось просто к загрузке программы и ее данных из некоторого внешнего накопителя (перфолен-ты, магнитной ленты или магнитного диска) в память.

С появлением мультипрограммирования перед ОС были поставле-ны новые задачи, связанные с распределением имеющейся памяти меж-ду несколькими одновременно выполняющимися программами. Органи-зация и управление ОП во многом определяют фактический уровень мультипрограммирования ОС, то есть возможность выполнения не-скольких параллельных процессов.

Как правило, учитывая первостепенную важность решения задач по управлению памятью компьютера, в ОС для них выделена специальная подсистема управления памятью. Ее основная цель – обеспечить макси-мальный уровень мультипрограммирования и, тем самым, максималь-ную загрузку центрального процессора. Поэтому одной из основных за-дач этой подсистемы является эффективное размещение процессов в ОП.

Сегодня функциями ОС по управлению памятью в мультипро-граммной системе являются:

- отслеживание свободной и занятой памяти;
- выделение памяти процессам и освобождение памяти по завер-шении процессов;
- вытеснение кодов и данных процессов из ОП на диск (полное или частичное), когда размеры основной памяти не достаточны для раз-мещения в ней всех процессов, и возвращение их в ОП, когда в ней освобождается место;
- настройка адресов программы на конкретную область физиче-ской памяти.

Кроме того, следует выделить такую функцию ОС как *динамиче-ское распределение* и *перераспределение* памяти после ее первоначаль-ного выделения процессам. Также необходимо решение задачи по *за-щите памяти*, которая состоит в том, чтобы не позволить выполняемо-му процессу записывать или читать данные из памяти, назначенной дру-гому процессу. Эта функция, как правило, реализуется программными модулями ОС в тесном взаимодействии с аппаратными средствами.

4.1.2 Типы адресации

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

С другой стороны, при выполнении программы идет работа с физической ОП, собственно с которой и работает процессор, извлекая из нее команды и данные и помещая в нее результаты вычислений. Физическая память представляет собой упорядоченное множество ячеек реально существующей ОП, и все они пронумерованы, то есть к каждой из них можно обратиться, указав ее порядковый номер (адрес). Количество ячеек физической памяти ограничено и фиксировано.

Системное ПО должно связать каждое указанное пользователем символьное имя с физической ячейкой памяти, то есть осуществить *отображение* пространства имен на физическую память компьютера. В общем случае это отображение осуществляется в два этапа – сначала системой программирования, а затем ОС (рис. 21).

Система программирования осуществляет трансляцию и компоновку программы, используя библиотечные программные модули. В результате работы этой системы программа получает *виртуальные адреса*, которые могут иметь двоичную или символьно-двоичную формы. Те модули, адреса для которых пока не могут быть определены, имеют по-прежнему символьную форму, и их окончательная привязка к физическим ячейкам будет осуществлена на этапе загрузки программы в память непосредственно перед ее исполнением. Физические адреса соответствуют номерам ячеек ОП, где в действительности расположены или будут расположены переменные и команды.

Второе отображение, реализуемое посредством ОС, осуществляется с помощью соответствующих аппаратных средств процессора – подсистемы управления памятью, которая использует дополнительную информацию, подготавливаемую и обрабатываемую ОС.

Между этими этапами обращения к памяти имеют форму *виртуального адреса*. Говорят, что множество всех допустимых значений виртуального адреса для некоторой программы определяет ее *виртуальное адресное пространство* или *виртуальную память*. Виртуальное адресное пространство программы зависит, прежде всего, от архитектуры

процессора и от системы программирования и практически не зависит от объема реальной физической памяти компьютера.

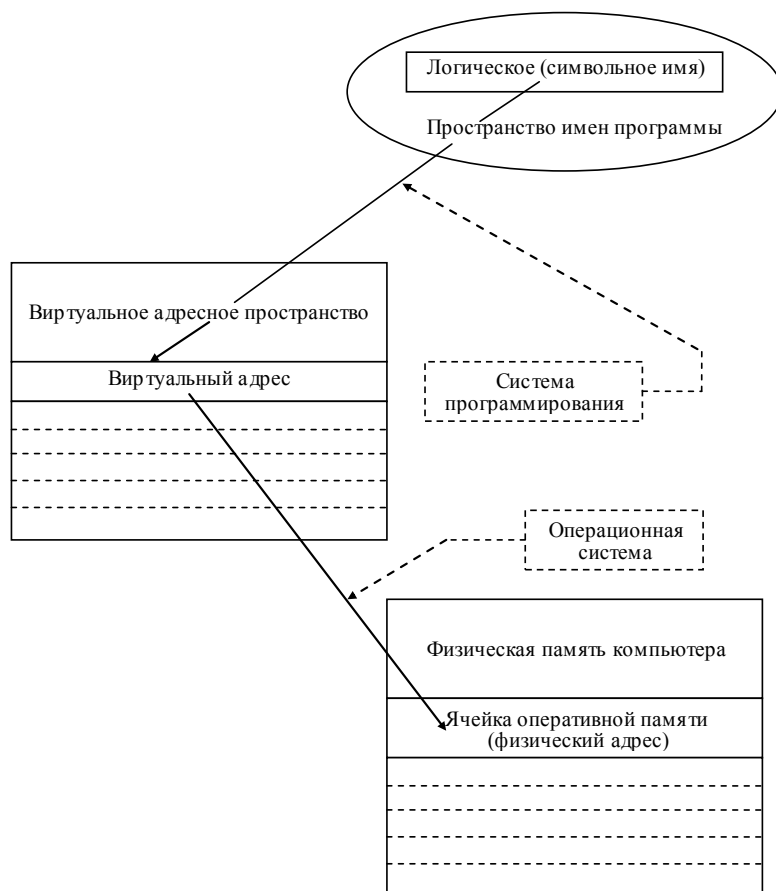


Рисунок 21 – Отображение пространства имен на физическую память компьютера

Диапазон возможных адресов виртуального пространства у всех процессов является одним и тем же. Например, при использовании 32-разрядных виртуальных адресов этот диапазон задается границами 00000000_{16} и $FFFFFFFF_{16}$. Тем не менее, каждый процесс имеет собственное виртуальное адресное пространство – транслятор независимо присваивает виртуальные адреса переменным и кодам каждой выполняемой программе. Можно еще сказать, что адреса команд и переменных в машинной программе, подготовленной к выполнению системой программирования, как раз и являются виртуальными адресами.

Возможны различные варианты перехода от символьных имен к физическим адресам. Рассмотрим ряд частных и промежуточных вариантов такого перехода.

Одним из частных случаев общей схемы трансляции адресного пространства является тождественность виртуального адресного пространства исходному логическому пространству имен. Здесь уже отображение выполняется самой ОС, которая во время исполнения исполь-

зует таблицу символьных имен. Такая схема отображения используется чрезвычайно редко, так как отображение имен на адреса необходимо выполнять для каждого вхождения имени (каждого нового имени), и особенно много времени расходуется на квалификацию имен. Данную схему можно было встретить в интерпретаторах, в которых стадии трансляции и исполнения практически неразличимы. Это характерно для простейших компьютерных систем, в которых вместо ОС использовался встроенный интерпретатор (например, *Basic*).

Другим частным случаем отображения пространства символьных имен на физическую память является полная тождественность виртуального адресного пространства физической памяти. При этом нет необходимости осуществлять второе отображение. В таком случае говорят, что система программирования генерирует абсолютную двоичную программу: в этой программе все двоичные адреса таковы, что программа может исполняться только тогда, когда ее виртуальные адреса будут точно соответствовать физическим. Некоторые программные модули любой ОС обязательно должны быть абсолютными двоичными программами. Эти программы размещаются по фиксированным адресам физической памяти, и с их помощью можно впоследствии реализовывать размещение остальных программ, подготовленных системой программирования таким образом, что они могут работать на различных физических адресах (то есть на тех адресах, на которые их разместит ОС). Примером таких программ являются программы загрузки ОС.

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

Следует отметить, что термин *виртуальная память* фактически относится к системам, которые сохраняют виртуальные адреса во время исполнения. В связи с тем, что второе отображение осуществляется в процессе исполнения задачи, то адреса физических ячеек могут изменяться. При правильном применении такие изменения улучшают использование памяти, избавляя программиста от деталей управления ею, и повышают надежность вычислений.

Если рассматривать общую схему двухэтапного отображения адресов, представленную на рис. 21, то с позиции соотношения объемов упомянутых адресных пространств можно отметить наличие следующих трех ситуаций:

- объем виртуального адресного пространства программы V_v меньше объема физической памяти V_p ($V_v < V_p$);
- объем виртуального адресного пространства программы V_v равен объему физической памяти V_p ($V_v = V_p$);
- объем виртуального адресного пространства программы V_v больше объема физической памяти V_p ($V_v > V_p$).

Ситуация ($V_v < V_p$) сейчас на практике практически не встречается, но недавно, например 16-разрядные мини-ЭВМ имели систему команд, в которых программисты могли адресовать до $2^{16} = 64$ Кбайт адресов (обычно в качестве адресуемой единицы выступала ячейка памяти размером 1 байт). При этом физически старшие модели этих мини-ЭВМ могли иметь объем ОП в несколько мегабайтов. Обращение к памяти столь большого объема осуществлялось с помощью специальных регистров, содержимое которых складывалось с адресом операнда (или команды), извлекаемым из поля операнда или указателя команды (и/или определяемым по значению поля операнда или указателя команды). Соответствующие значения в эти специальные регистры, выступающие как базовое смещение в памяти, заносила ОС. Для одной задачи в регистр заносилось одно значение, а для второй (третьей, четвертой и т.д.) задачи, размещаемой одновременно с первой, но в другой области памяти, заносилось, соответственно, другое значение. Таким образом, вся физическая память разбивалась на разделы объемом по 64 Кбайт, и на каждый такой раздел осуществлялось отображение своего виртуального адресного пространства.

Вторая ситуация ($V_v = V_p$) особенно характерна для недорогих вычислительных комплексов. Для этого случая имеется большое количество методов распределения ОП.

Третья ситуация, при которой объема виртуального адресного пространства программы превышает объем физической памяти ($V_v > V_p$), сегодня наиболее характерна. Теперь это самая обычная ситуация, и для нее имеется несколько методов распределения памяти, отличающихся как сложностью, так и эффективностью, которые более подробно будут рассмотрены ниже в п. 4.2.

4.2 *Распределение памяти*

4.2.1 Общие принципы управления памятью в однопрограммных ОС

Непрерывное распределение. Непрерывное распределение – это самая простая и распространенная схема, согласно которой вся память условно может быть разделена на три области:

- область, занимаемая ОС;
- область, в которой размещается исполняемый процесс;
- свободная область памяти.

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

Для того чтобы отвести задачам как можно больший объем памяти, ОС строится таким образом, чтобы постоянно в ОП располагалась только самая нужная ее часть – *ядро* ОС. Прежде всего, в ядро ОС входят основные модули супервизора. Для однопрограммных систем понятие супервизора вырождается в модули, получающие и выполняющие первичную обработку запросов от обрабатывающих и прикладных программ, и в модули подсистемы памяти. Ведь если программа по ходу своего выполнения запрашивает некоторое множество ячеек памяти, то подсистема управления памятью должна их выделить (если они есть), а после освобождения памяти эта подсистема должна выполнить действия, связанные с возвратом памяти в систему. Остальные модули ОС, не относящиеся к ее ядру, могут быть обычными *диск-резидентными* (или *транзитными*), то есть загружаться в ОП только по необходимости, и после своего выполнения вновь освобождать память.

Такая схема распределения влечет за собой два вида потерь вычислительных ресурсов:

1) Потерю процессорного времени, потому что процессор простаивает, пока задача ожидает завершения операций ввода-вывода.

2) Потерю самой ОП, потому что далеко не каждая программа использует всю память, а режим работы в этом случае однопрограммный. В то же время это недорогая реализация, которая позволяет отказаться от многих второстепенных функций ОС. В частности, такая сложная проблема, как защита памяти, здесь практически отсутствует. Единственное, что желательно защищать – это программные модули и области памяти самой ОС.

Оверлейное распределение. Если есть необходимость создать программу, логическое адресное пространство которой должно быть больше, чем свободная область памяти, или даже больше, чем весь возможный объем ОП, то используется распределение с перекрытием, в основе которого лежит использование так называемых *оверлейных структур* (англ. *overlay* – перекрытие, расположение поверх чего-то).

Этот метод распределения предполагает, что вся программа может быть разбита на части – сегменты. Каждая оверлейная программа имеет одну главную (*main*) часть и несколько сегментов (*segments*), причем в памяти машины одновременно могут находиться только ее главная часть и один или несколько не перекрывающихся сегментов. На рис. 22 представлен пример организации некоторой программы с перекрытием, причем в представленном случае поочередно можно загружать в память ветви А-В, А-С-D и А-С-Е программы.

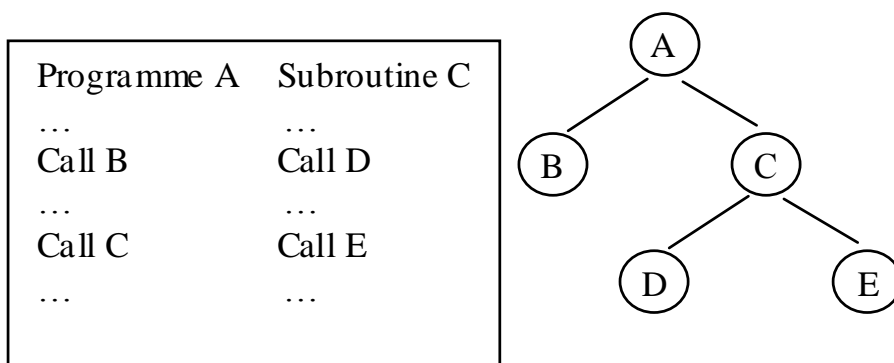


Рисунок 22 – Образное представление организации памяти с использованием структуры с перекрытием

Пока в ОП располагаются выполняющиеся сегменты, остальные находятся во внешней памяти. После того, как текущий (выполняющийся) сегмент завершит свое выполнение, возможны два варианта. Первый – сегмент сам (если данный сегмент не нужно сохранить во внешней памяти в его текущем состоянии) обращается к ОС с указанием, какой сегмент должен быть загружен в память следующим. Второй – сегмент возвращает управление главному сегменту задачи, и уже тот обращается к ОС с указанием, какой сегмент сохранить (если это нужно), а какой

сегмент загрузить в ОП, и вновь отдает управление одному из сегментов, располагающихся в памяти.

Простейшие схемы сегментирования предполагают, что в памяти в каждый конкретный момент времени может располагаться только один сегмент (вместе с главным модулем). Более сложные схемы, используемые в больших вычислительных системах, позволяют располагать в памяти несколько сегментов. В некоторых вычислительных комплексах могли существовать отдельно *сегменты кода* и *сегменты данных*. Сегменты кода, как правило, не претерпевают изменений в процессе своего исполнения, поэтому при загрузке нового сегмента кода на место отработавшего последний можно не сохранять во внешней памяти, в отличие от сегментов данных, которые сохранять необходимо.

Первоначально программисты сами должны были включать в тексты своих программ соответствующие обращения к ОС (системные вызовы) и тщательно планировать, какие сегменты могут находиться в ОП одновременно, чтобы их адресные пространства не пересекались. Однако с некоторых пор такого рода обращения к ОС системы программирования стали подставлять в код программы сами, автоматически, если в том возникает необходимость.

В известной и популярной в недалеком прошлом системе программирования *Turbo Pascal* программист просто указывал, что данный модуль является оверлейным. При обращении к нему из основной программы модуль загружался в память и получал управление. Все адреса определялись системой программирования автоматически, обращения к DOS для загрузки оверлеев тоже генерировались системой *Turbo Pascal*.

Рассмотрев основные принципы работы с памятью в однопрограммных ОС, перейдем к рассмотрению особенностей распределения памяти для более распространенных сегодня мультипрограммных ОС.

4.2.2 Особенности организации управления памятью в мультипрограммных ОС

Для организации мультипрограммного режима необходимо обеспечить одновременное расположение в ОП нескольких задач (целиком или частями). При решении этой задачи ОС должна учитывать целый ряд моментов:

- следует ли назначать каждому процессу одну непрерывную область физической памяти или фрагментами;
- должны ли сегменты программы, загруженные в память, находиться на одном месте в течение всего периода выполнения процесса или можно ее время от времени сдвигать;
- что следует предпринять, если сегменты программы не помещаются в имеющуюся память.

Разные ОС по-разному «отвечают» на подобные вопросы управления памятью. Рассмотрим ниже наиболее общие подходы к распределению памяти, которые были характерны для разных периодов развития ОС.

Все методы распределения памяти по критерию использования различных видов запоминающих устройств можно разделить на два класса (рис. 23):

- в которых используется перемещение сегментов процессов между ОП и диском;
- в которых внешняя память не привлекается.



Рисунок 23 – Методы распределения памяти

Рассмотрим эти методы более подробно, начав обзор с методов распределения без использования внешней памяти.

4.2.3 Распределение фиксированными разделами

Простейший способ управления ОП состоит в том, что память разбивается на несколько областей фиксированной величины, называемых *разделами*. Такое разбиение может быть выполнено вручную оператором во время старта системы или во время ее установки. После этого границы разделов не изменяются. Очередной новый процесс, поступивший на выполнение, помещается либо в общую очередь (рис. 24а), либо в очередь к некоторому разделу (рис. 24б). В каждом разделе в каждый

момент времени может располагаться по одной программе (задаче), и к каждому разделу в отдельности можно применить методы, используемые при распределении памяти в однопрограммных системах.

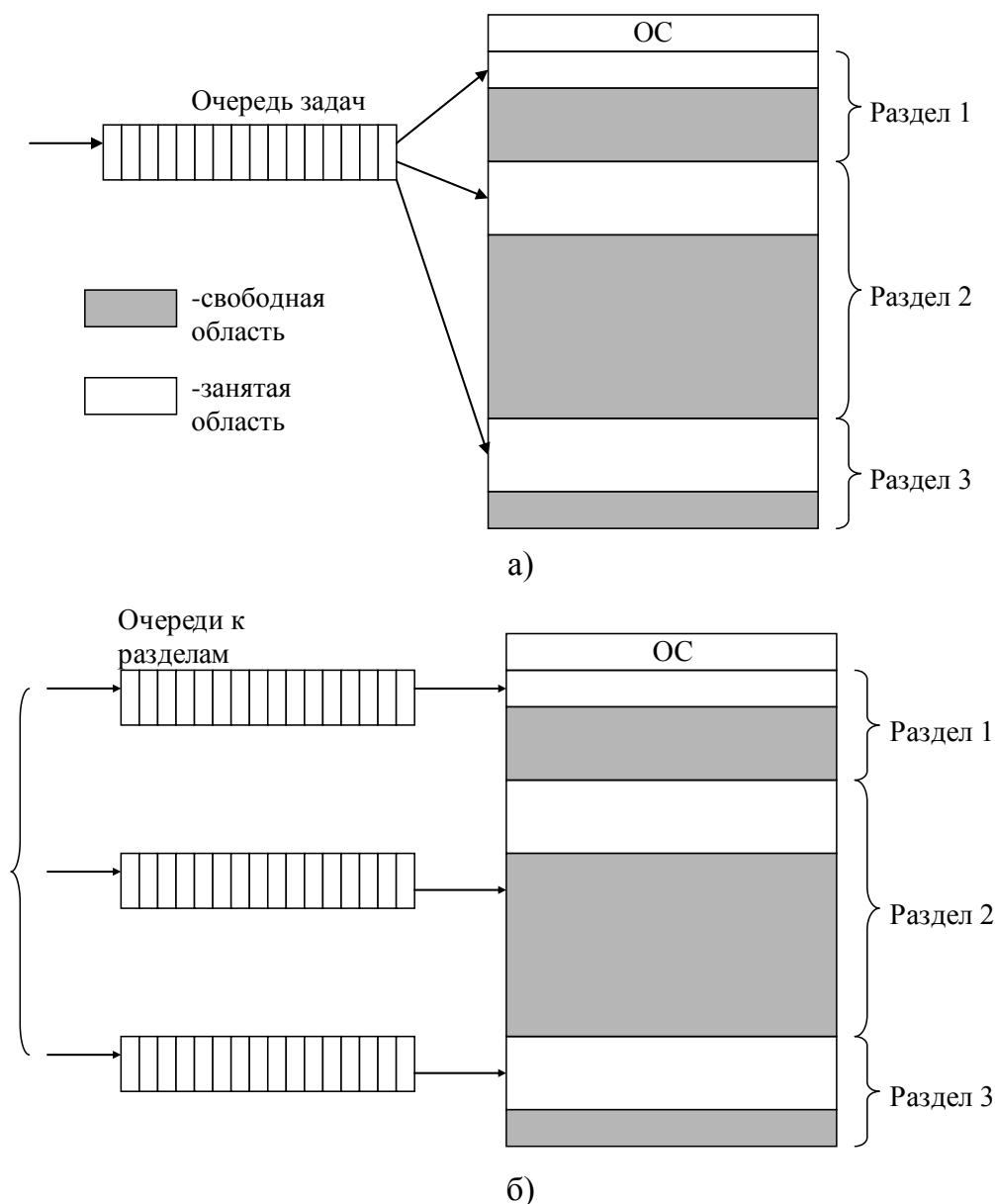


Рисунок 24 – Распределение памяти фиксированными разделами: с общей очередью (а); с отдельными очередями (б)

Подсистема управления памятью в этом случае выполняет следующие задачи:

1) Сравнивает объем памяти, требуемый для вновь поступившего процесса, с размерами свободных разделов и выбирает подходящий раздел.

2) Осуществляет загрузку программы в один из разделов и настройку адресов. Уже на этапе трансляции разработчик программы

может задать раздел, в котором ее следует выполнять. Это позволяет сразу, без использования перемещающего загрузчика, получить машинный код, настроенный на конкретную область памяти.

При очевидном преимуществе как простота реализации, данный метод имеет существенный недостаток – существенные потери памяти от внутренней фрагментации. Фрагментация возникает потому, что процесс не полностью занимает выделенный ему раздел или потому, что некоторые разделы слишком малы для выполняемых пользовательских программ. Кроме того, учитывая то, что в каждом разделе может выполняться только один процесс, уровень мультипрограммирования заранее ограничен числом разделов, т.к. процесс независимо от размера будет занимать весь раздел. Так, например, в системе с тремя разделами невозможно выполнять одновременно не более трех процессов, даже если им требуется совсем мало памяти.

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

4.2.4 Распределение динамическими разделами

Чтобы избавиться от фрагментации, присущей распределению памяти с фиксированными разделами, целесообразно размещать в ОП задачи «плотно», одну за другой, выделяя ровно столько памяти, сколько требует процесс. Именно такой принцип лежит в основе распределения памяти разделами переменной величины (динамическими разделами). В этом случае память машины не делится заранее на разделы, и сначала вся память, отводимая для приложений, свободна. Каждому вновь поступающему на выполнение приложению на этапе создания процесса выделяется вся необходимая ему память (если достаточный объем памяти отсутствует, то приложение не принимается на выполнение и процесс для него не создается). После завершения процесса память освобождается, и на это место может быть загружен другой процесс. Таким образом, в произвольный момент времени ОП представляет собой случайную последовательность занятых и свободных участков (разделов) произвольного размера. Список свободных участков памяти может быть упорядочен либо по адресам, либо по объему.

На рис. 25 показано состояние памяти в различные моменты времени при использовании динамического распределения. Так, в момент t_0 в

памяти находится только ОС, а к моменту t_1 память разделена между 5-ю процессами, причем процесс П4, завершаясь, покидает память. На освободившееся от процесса П4 место загружается процесс П6, поступивший в момент t_3 .

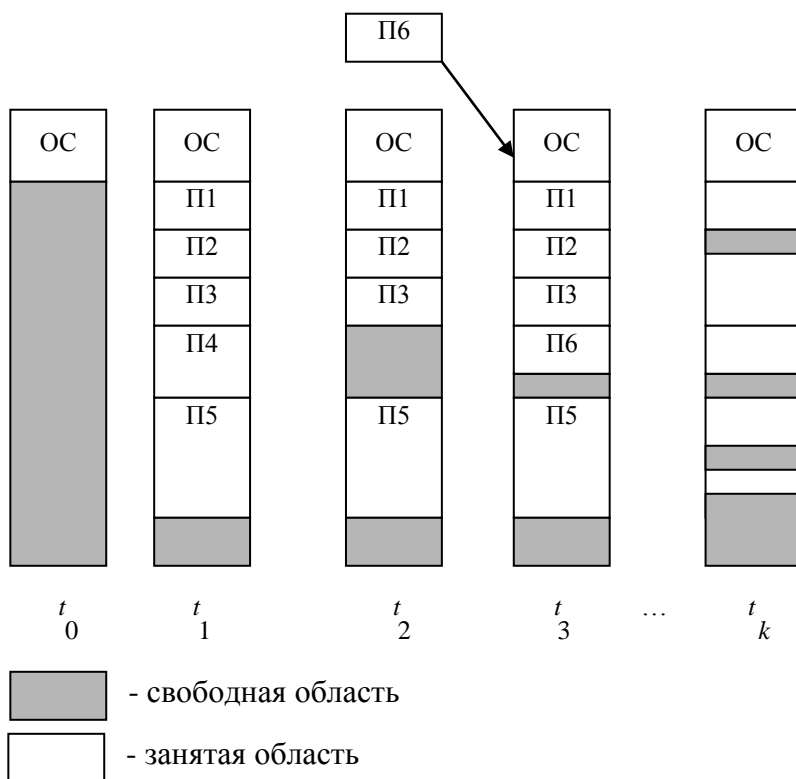


Рисунок 25 – Распределение памяти динамическими разделами

Задачами ОС при реализации данного метода распределения памяти являются:

- ведение таблиц свободных и занятых областей, в которых указываются начальные адреса и размеры участков памяти;
- при поступлении новой задачи – анализ запроса, просмотр таблицы свободных областей и выбор раздела, размер которого достаточен для размещения поступившей задачи⁹;
- загрузка задачи в выделенный ей раздел и корректировка таблиц свободных и занятых областей;
- после завершения задачи корректировка таблиц свободных и занятых областей.

По сравнению с методом распределения памяти фиксированными разделами данный метод обладает гораздо большей гибкостью, но ему также присущ недостаток, связанный с наличием фрагментации памяти – наличие столь большого числа несмежных участков свободной па-

⁹ Выбор раздела может осуществляться по разным правилам, например: «первый попавшийся раздел достаточного размера», «раздел, имеющий наименьший достаточный размер» или «раздел, имеющий наибольший достаточный размер».

мемории сравнительно маленького размера, что диспетчер памяти не может образовать новый раздел, хотя суммарный объем свободных областей больше, чем необходимо для выполнения процесса.

4.2.5 Распределение перемещаемыми разделами

Одним из методов борьбы с фрагментацией является перемещение всех занятых участков в сторону старших либо в сторону младших адресов, так, чтобы вся свободная память образовывала единую свободную область (рис. 26). В связи с этим, ОС в дополнение к функциям, которые выполняет при распределении памяти динамическими разделами, должна еще время от времени копировать содержимое разделов из одного места памяти в другое, корректируя таблицы свободных и занятых областей. Эта процедура называется *сжатием*.

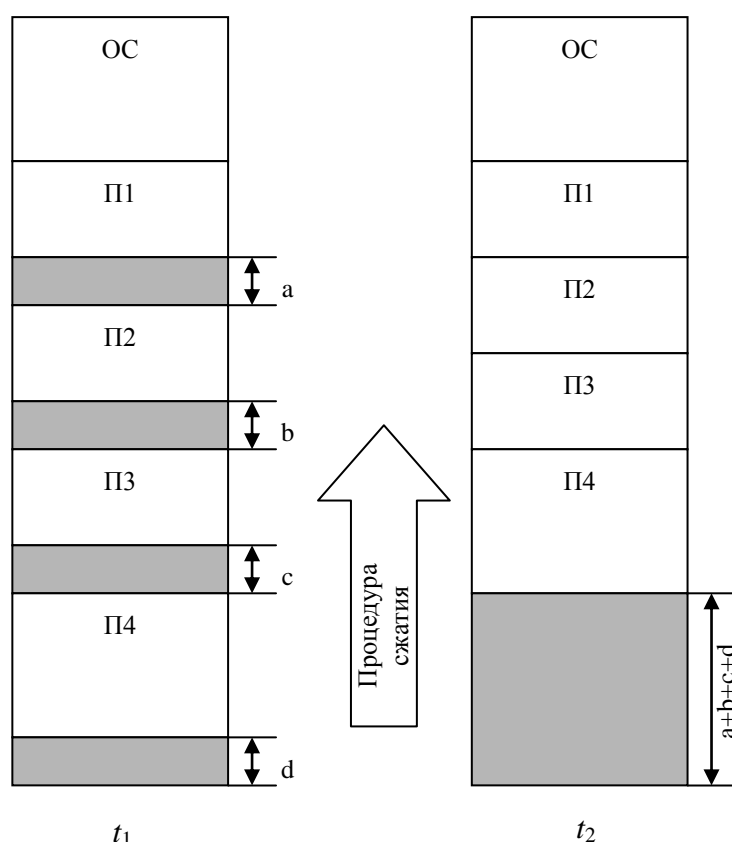


Рисунок 26 – Сжатие памяти при распределении динамическими разделами

Сжатие может выполняться:

- 1) При каждом завершении задачи (меньше однократной вычислительной работы).
- 2) В случае, когда для вновь поступившей задачи нет свободного раздела достаточного размера (процедура выполняется реже).

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

В связи с тем, что программы перемещаются по ОП в ходе своего выполнения, то невозможно выполнить настройку адресов с помощью перемещающего загрузчика. Здесь более подходящим оказывается динамическое преобразование адресов.

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

Распределение памяти динамическими разделами легло в основу подсистем управления памятью многих мультипрограммных ОС 60-70-х годов, в частности такой популярной ОС, как *OS/360*.

4.2.6 Сегментное распределение

Наряду с рассмотренными в пп. 4.2.3-4.2.5 методами непрерывного распределения памяти не использующими внешнюю память, существует целый ряд *разрывных* методов распределения памяти, при которых задаче не предоставляется сплошная (непрерывная) область памяти, и кроме того используется внешняя память.

Идея выделять память задаче не одной сплошной областью, а фрагментами позволяет уменьшить фрагментацию памяти, однако этот подход требует для своей реализации больше ресурсов, он значительно сложнее. Если задать адрес начала текущего фрагмента программы и величину смещения относительно этого начального адреса, то можно указать необходимую нам переменную или команду. Таким образом, виртуальный адрес можно представить состоящим из двух полей. Первое поле будет указывать на ту часть программы, к которой обращается процессор, для определения местоположения этой части в памяти, а второе поле виртуального адреса позволит найти нужную нам ячейку относительно найденного адреса. Программист может либо самостоятельно разбивать программу на фрагменты, либо можно автоматизировать эту задачу, возложив ее на систему программирования.

Первым среди разрывных методов распределения памяти был *сегментный*. В соответствии с этим методом программу необходимо разбивать на части и уже каждой такой части выделять физическую память. Естественным способом разбиения программы на части является разбиение ее на логические элементы – так называемые *сегменты*. В принципе, каждый программный модуль (или их совокупность) может быть воспринят как отдельный сегмент, и вся программа тогда будет представлять собой множество сегментов.

Следует отметить, что разбиение на сегменты позволяет дифференцировать способы доступа к разным частям программы (сегментам). Например, можно запретить обращаться с операциями записи и чтения в кодовый сегмент программы, а для сегмента данных разрешить только чтение. Кроме того, разбиение программы на «осмысленные» части делает принципиально возможным разделение одного сегмента несколькими процессами. Например, если два процесса используют одну и ту же математическую подпрограмму, то в ОП может быть загружена только одна копия этой подпрограммы (рис. 27).

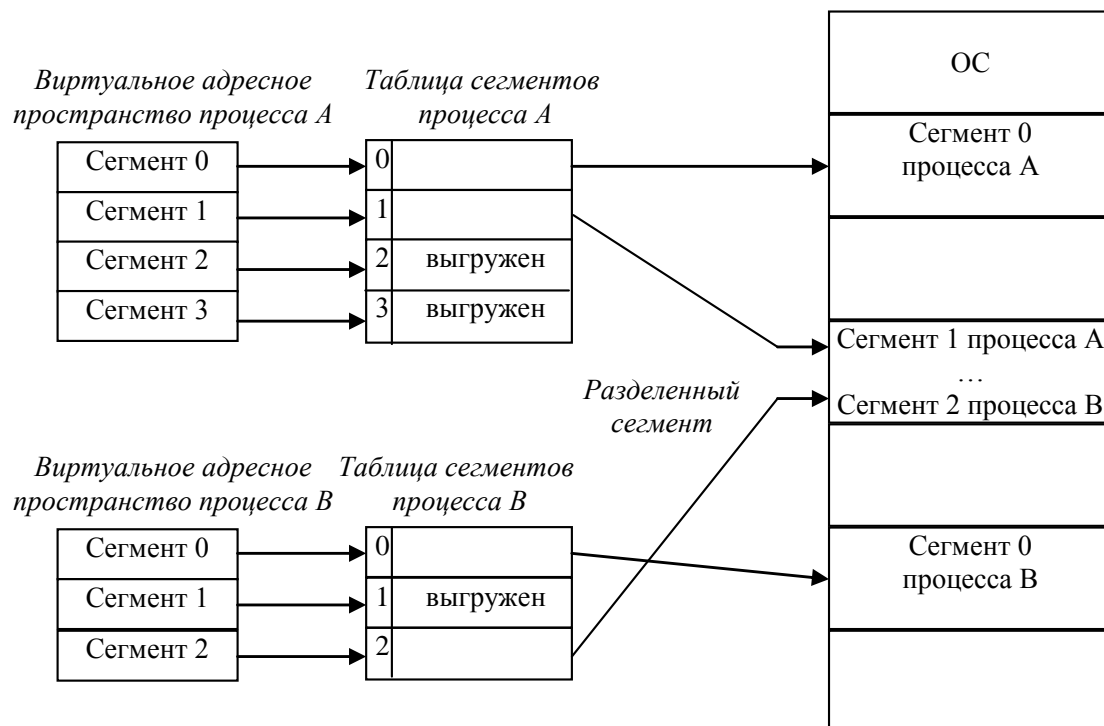


Рисунок 27 – Пример распределения памяти сегментами

Каждый сегмент размещается в памяти как до определенной степени самостоятельная единица. Логически обращение к элементам программы в этом случае будет состоять из *имени* сегмента и *смещения* относительно начала этого сегмента. Физически имя (или порядковый номер) сегмента будет соответствовать некоторому адресу, с которого этот сегмент начинается при его размещении в памяти, и смещение должно прибавляться к этому базовому адресу.

Преобразование имени сегмента в его порядковый номер осуществляет система программирования. Для каждого сегмента система программирования указывает его объем. Он должен быть известен ОС, чтобы она могла выделять ему необходимый объем памяти. Операционная система будет размещать сегменты в памяти и вести для каждого сегмента учет о местонахождении этого сегмента. Вся информация о текущем размещении сегментов задачи в памяти обычно сводится в *таб-*

лицу сегментов, которую чаще называют *таблицей дескрипторов сегментов задачи*.

Таким образом, виртуальный адрес для этого способа будет состоять из двух полей – номера сегмента и смещения относительно начала сегмента. Соответствующая иллюстрация приведена на рис. 28 для случая обращения к ячейке, виртуальный адрес которой равен сегменту с номером 11 со смещением от начала этого сегмента, равным 612. Как видно в данном случае, ОС разместила данный сегмент в памяти, начиная с ячейки с номером 19700.

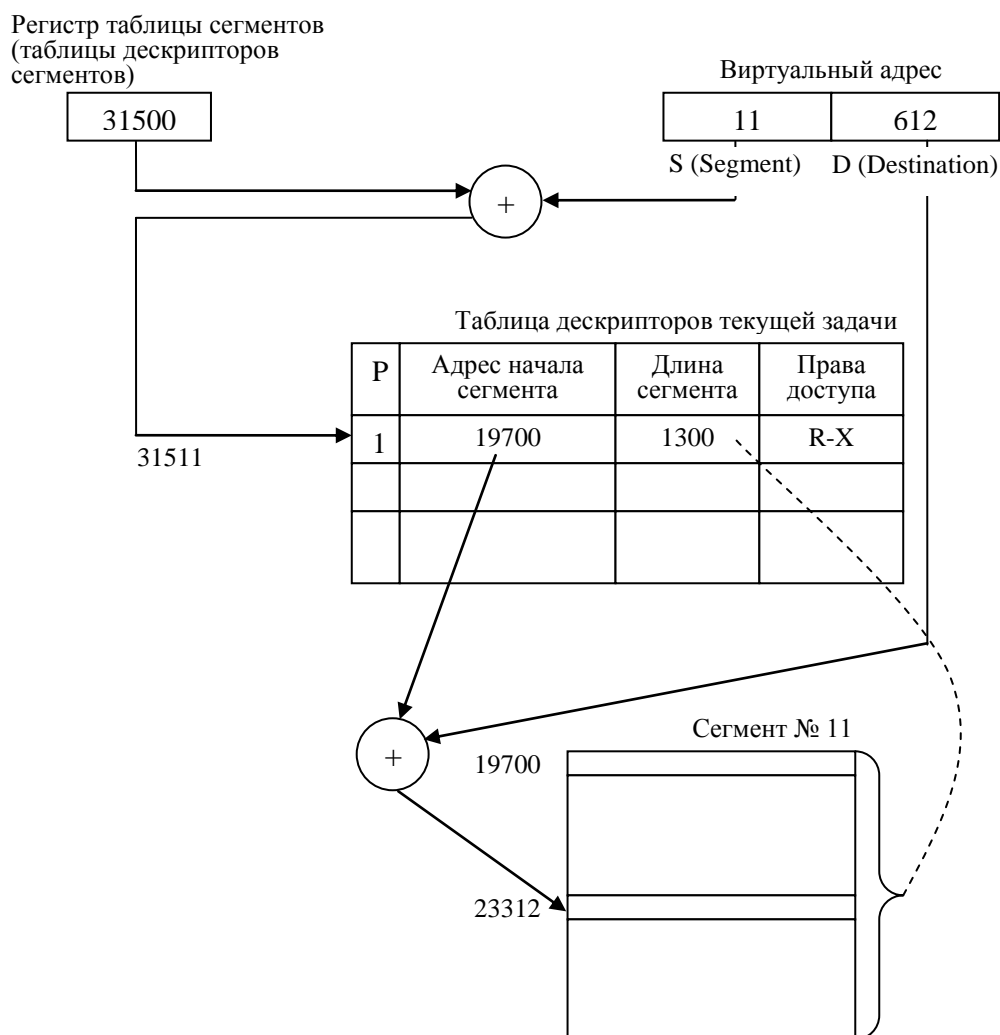


Рисунок 28 – Сегментный способ организации распределения памяти

Итак, каждый сегмент, размещаемый в памяти, имеет соответствующую информационную структуру, часто называемую дескриптором сегмента. Именно ОС строит для каждого исполняемого процесса соответствующую таблицу дескрипторов сегментов, и при размещении каждого из сегментов в оперативной или внешней памяти отмечает в дескрипторе текущее местоположение сегмента. Если сегмент задачи в данный момент находится в ОП, то об этом делается пометка в дескрип-

торе. Как правило, для этого используется бит присутствия P (от англ. *present*). В этом случае в поле адреса диспетчер памяти записывает адрес физической памяти, с которого сегмент начинается, а в поле длины сегмента (*limit*) указывается количество адресуемых ячеек памяти. Это поле используется не только для того, чтобы размещать сегменты без наложения друг на друга, но и для того, чтобы контролировать, не обращается ли код исполняющейся задачи за пределы текущего сегмента. В случае превышения длины сегмента вследствие ошибок программирования можно выявить нарушения адресации и с помощью введения специальных аппаратных средств генерировать сигналы прерывания, которые позволяют фиксировать (обнаруживать) ошибки такого рода.

Помимо информации о местоположении сегмента, в дескрипторе сегмента, как правило, содержатся данные о его типе (сегмент кода или сегмент данных), правах доступа к этому сегменту (можно или нельзя его модифицировать, предоставлять другой задаче), отметка об обращениях к данному сегменту (информация о том, как часто или как давно этот сегмент используется или не используется, на основании которой можно принять решение о том, чтобы предоставить место, занимаемое текущим сегментом, другому сегменту).

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

При таком подходе появляется возможность размещать в ОП не все сегменты задачи, а только задействованные в данный момент. Благодаря этому, с одной стороны, общий объем виртуального адресного пространства задачи может превосходить объем физической памяти компьютера, на котором эта задача будет выполняться, с другой стороны, даже если потребности в памяти не превосходят имеющуюся физическую память, можно размещать в памяти больше задач, поскольку любой задаче, как правило, все ее сегменты одновременно не нужны. Как известно, увеличение *коэффициента мультипрограммирования* μ позволяет увеличить загрузку системы и более эффективно использовать ресурсы вычислительной системы.

Очевидно, однако, что увеличивать количество задач можно только до определенного предела, т.к. если в памяти не будет хватать места для часто используемых сегментов, то производительность системы резко упадет. Ведь сегмент, находящийся вне ОП, для участия в вычислениях должен быть перемещен в ОП. При этом если в памяти есть свободное пространство, то необходимо всего лишь найти нужный сегмент во внешней памяти и загрузить его в ОП. Если свободного места нет, при-

дется принять решение – на место какого из присутствующих сегментов будет загружаться требуемый. Перемещение сегментов из ОП на жесткий диск и обратно часто называют *свопингом сегментов*.

Итак, если требуемого сегмента в ОП нет, то возникает прерывание, и управление передается через диспетчер памяти программе загрузки сегмента. Пока происходит поиск сегмента во внешней памяти и загрузка его в оперативную, диспетчер памяти определяет подходящее для сегмента место. Возможно, что свободного места нет, и тогда принимается решение о выгрузке какого-нибудь сегмента и выполняется его перемещение во внешнюю память. Если при этом еще остается время, то процессор передается другой готовой к выполнению задаче. После загрузки необходимого сегмента процессор вновь передается задаче, вызвавшей прерывание из-за отсутствия сегмента. Всякий раз при считывании сегмента в ОП в таблице дескрипторов сегментов необходимо установить адрес начала сегмента и признак присутствия сегмента.

Если свободного фрагмента памяти достаточного объема нет, но, тем не менее, сумма этих свободных фрагментов превышает требования по памяти для нового сегмента, то в принципе может быть применено сжатие памяти, упомянутое выше в п. 4.2.5.

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

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

- *FIFO (First In First Out* – первый пришедший первым и выбывает);
- *LRU (Least Recently Used* – неиспользуемый дольше других);
- *LFU (Least Frequently Used* – используемый реже других);
- *random* – случайный выбор сегмента.

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

зовании сегментов.

При использовании дисциплины *FIFO* с каждым сегментом связывается очередность его размещения в памяти. Для замещения выбирается сегмент, первым попавший в память. Каждый вновь размещаемый в памяти сегмент добавляется в хвост этой очереди. В этом случае учитывается только время нахождения сегмента в памяти, но не учитывается фактическое использование сегментов. Например, первые загруженные сегменты программы могут содержать переменные, требующиеся на протяжении всей ее работы. Это приводит к немедленному возвращению к только что замещенному сегменту.

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

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

При сегментном способе организации виртуальной памяти появляется несколько интересных возможностей по оптимизации управления памятью.

Во-первых, при загрузке программы на исполнение можно размещать ее в памяти не целиком, а «по мере необходимости». Действительно, поскольку в подавляющем большинстве случаев алгоритм, по которому работает код программы, является разветвленным, а не линейным, то в зависимости от исходных данных некоторые части программы, рас-

положенные в самостоятельных сегментах, могут быть не задействованы, а значит их можно и не загружать в ОП.

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

Однако у сегментного способа распределения памяти есть и недостатки. Согласно схеме доступа к искомой ячейке памяти, представленной на рис. 28, сначала необходимо найти и прочитать дескриптор сегмента, а уже потом, используя полученные данные о местонахождении нужного сегмента, вычислить конечный физический адрес, что требует времени. Для того чтобы уменьшить эти временные потери используется *кэширование*, то есть размещение в сверхоперативной памяти (специальных регистрах, размещаемых в процессоре) тех дескрипторов, с которыми идет работа в текущий момент.

Пример использования. Примером использования сегментного способа организации виртуальной памяти является ОС *OS/2* первого поколения, которая была создана для персональных компьютеров на базе процессора i80286. В этой ОС в полной мере использованы аппаратные средства микропроцессора, который специально проектировался для поддержки сегментного способа распределения памяти. Система *OS/2 v.1* поддерживала распределение памяти, при котором выделялись сегменты программы и сегменты данных. Система позволяла работать как с именованными, так и с неименованными сегментами. Имена разделяемых сегментов данных имели ту же форму, что и имена файлов. Процессы получали доступ к именованным разделяемым сегментам, используя их имена в специальных системных вызовах. Операционная система *OS/2 v.1* допускала разделение программных сегментов приложений и подсистем, а также глобальных сегментов данных подсистем. Вообще, вся концепция системы *OS/2* была построена на понятии разделения памяти: процессы почти всегда разделяют сегменты с другими процессами. В этом состояло существенное отличие системы *OS/2* от систем типа *Unix*, которые обычно разделяют только реентерабельные программные модули между процессами.

Сегменты, которые активно не использовались, могли выгружаться на жесткий диск. Система восстанавливала их, когда в этом возникала необходимость. Учитывая то, что все области памяти, используемые сегментом, должны были быть непрерывными, *OS/2* перемещала в основной памяти сегменты таким образом, чтобы максимизировать объем свободной физической памяти (осуществляла уплотнение памяти). Об-

ласти в младших адресах физической памяти, которые использовались для запуска *DOS*-программ и кода самой *OS/2*, в уплотнении памяти не участвовали. Кроме того, система или прикладная программа могла временно фиксировать сегмент в памяти с тем, чтобы гарантировать наличие буфера ввода-вывода в физической памяти до тех пор, пока операция ввода-вывода не завершится. Если в результате уплотнения памяти не удавалось создать необходимое свободное пространство, то супервизор выполнял операции фонового плана для перекачки достаточного количества сегментов из физической памяти, чтобы дать возможность завершиться исходному запросу.

4.2.7 Страничное распределение

Несмотря на то, что рассмотренный выше сегментный способ распределения памяти приводит к существенно меньшей фрагментации памяти, по сравнению со способами с неразрывным распределением, фрагментация все равно присутствует. Кроме того, много памяти и процессорного времени теряется на размещение и обработку дескрипторных таблиц, так как на каждую задачу необходимо иметь свою таблицу дескрипторов сегментов, а при определении физических адресов приходится выполнять достаточно затратные операции сложения.

Поэтому другим способом разрывного размещения задач в памяти стал *страничный* способ организации виртуальной памяти, при котором все фрагменты задачи считаются равными (одинакового размера), причем длина фрагмента в идеале должна быть кратна степени двойки, чтобы операции сложения можно было заменить операциями *конкатенации*.

Как уже упоминалось, при страничном способе организации виртуальной памяти все фрагменты программы, на которые она разбивается (за исключением последней ее части), получаются одинаковыми. Одинаковыми полагаются и единицы памяти, которые предоставляются для размещения фрагментов программы. Эти одинаковые части называют *страницами* и говорят, что ОП разбивается на физические страницы, а программа – на *виртуальные страницы*. Часть виртуальных страниц задачи размещается в ОП, а часть – во внешней памяти. Обычно место во внешней памяти, в качестве которой в абсолютном большинстве случаев выступают накопители на магнитных дисках (поскольку они относятся к быстродействующим устройствам с прямым доступом), называют *файлом подкачки*, или *страничным файлом* (англ. *paging file*). Иногда этот файл называют *swap-файлом*, тем самым подчеркивая, что записи этого файла – страницы – замещают друг друга в ОП. В некоторых ОС выгруженные страницы располагаются не в файле, а в специальном разделе дискового пространства.

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

Для отображения виртуального адресного пространства задачи на физическую память, как и в случае сегментного способа организации, для каждой задачи необходимо иметь *таблицу страниц* для трансляции адресных пространств. Для описания каждой страницы диспетчер памяти ОС заводит соответствующий дескриптор, который отличается от дескриптора сегмента прежде всего тем, что в нем нет поля длины – ведь все страницы имеют одинаковый размер. По номеру виртуальной страницы в таблице дескрипторов страниц текущей задачи находится соответствующий элемент (дескриптор). Если бит присутствия имеет единичное значение, значит данная страница размещена в оперативной, а не во внешней памяти, и в дескрипторе – номер физической страницы, отведенной под данную виртуальную. Если же бит присутствия равен нулю, то в дескрипторе – адрес виртуальной страницы, расположенной во внешней памяти. Именно таким образом осуществляется трансляция виртуального адресного пространства на физическую память. Этот механизм трансляции иллюстрирует рис. 29.

Защита страничной памяти. Защита страничной памяти, как и в случае сегментного механизма, основана на контроле уровня доступа к каждой странице. Как правило, возможны следующие уровни доступа:

- только чтение;
- чтение и запись;
- только выполнение.

Каждая страница снабжается соответствующим кодом уровня доступа. При трансформации логического адреса в физический сравнива-

ется значение кода разрешенного уровня доступа с фактически требуемым. При их несовпадении работа программы прерывается.

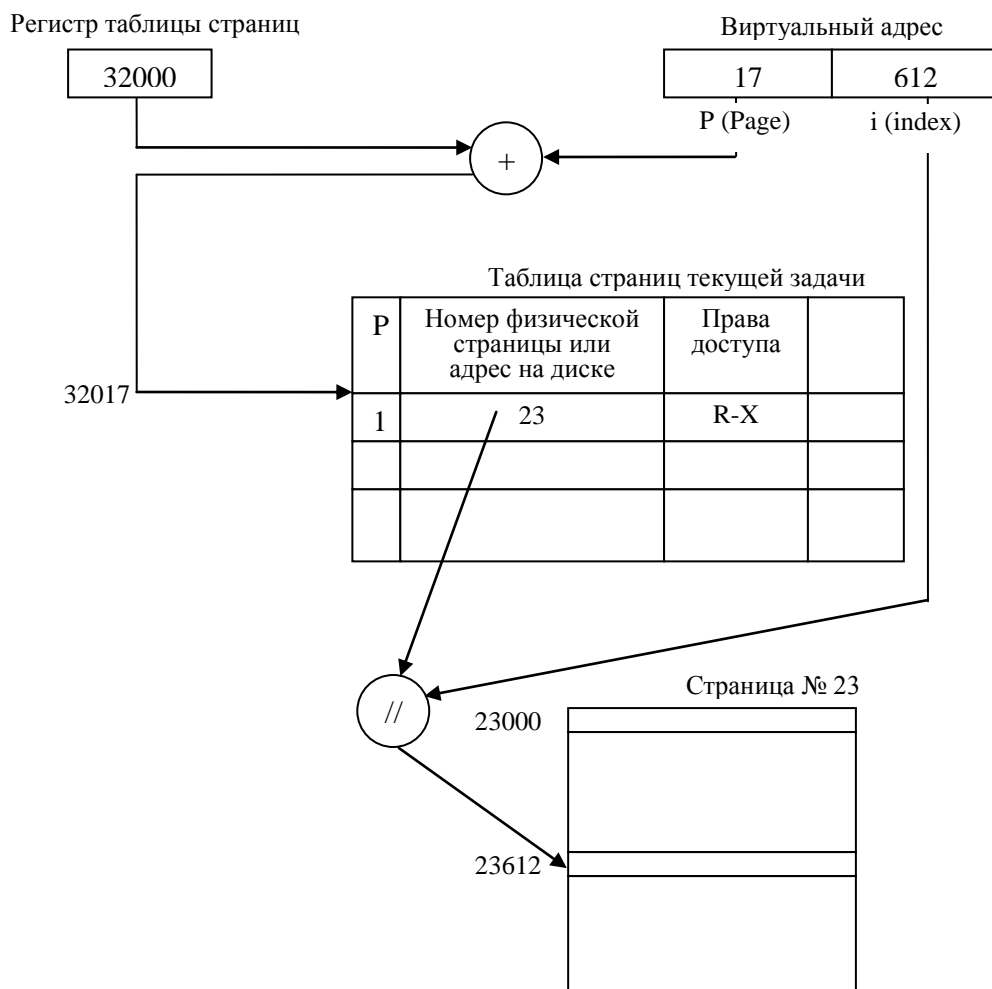


Рисунок 29 – Страничный способ организации распределения памяти

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

Напомним, что дисциплина *LFU* позволяет выбрать для замещения ту страницу, на которую не было ссылки на протяжении наиболее длительного периода времени. Дисциплина *LRU* ассоциирует с каждой страницей время ее последнего использования. Для замещения выбирается та страница, которая дольше всех не использовалась.

Для использования дисциплин *LRU* и *LFU* в процессоре должны

быть соответствующие аппаратные средства. В дескрипторе страницы размещается бит обращения, который становится единичным при обращении к дескриптору.

Если объем физической памяти небольшой и даже часто требуемые страницы не удастся разместить в ОП, возникает так называемая «пробуксовка» – ситуация, при которой загрузка нужной страницы вызывает перемещение во внешнюю память той страницы, с которой идет активная работа. Очевидно, что это очень плохое явление. Чтобы его не допускать, желательно увеличить объем ОП (учитывая сравнительно низкую стоимость – это, как правило, не сложно), уменьшить количество параллельно выполняемых задач или прибегнуть к более эффективным дисциплинам замещения.

На рис. 30 представлен пример страничного распределения памяти с участием двух процессов. Подпись «N_{вс}» в таблицах страниц означает «номер виртуальной страницы», подпись «N_{фс}» – «номер физической страницы».

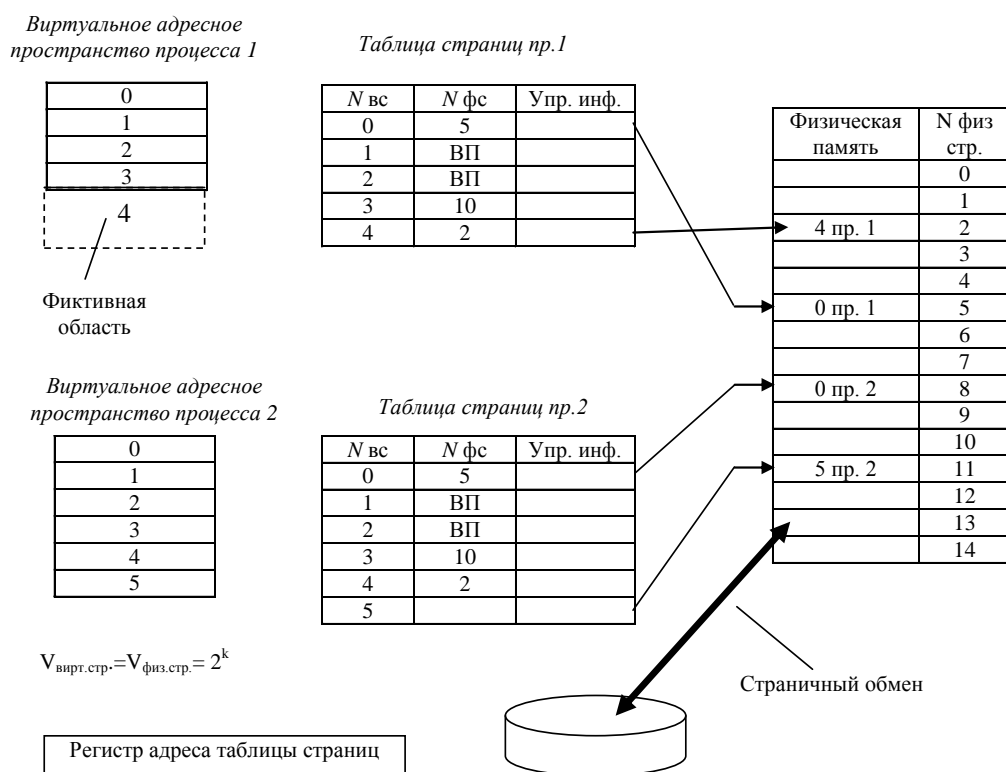


Рисунок 30 – Пример распределения памяти страницами

Примеры использования. Для абсолютного большинства современных ОС характерна дисциплина замещения страниц *LRU* как наиболее эффективная. Так, именно эта дисциплина использована в *OS/2* и *Linux*. Однако в ОС *Windows NT/2000/XP* разработчики, желая сделать их максимально независимыми от аппаратных возможностей процессора, отказались от этой дисциплины и применили правило *FIFO*.

Для того чтобы компенсировать неэффективность правила *FIFO*, была введена «буферизация» тех страниц, которые должны быть записаны в файл подкачки на диск¹⁰ или просто расформированы. Принцип буферизации следующий. Прежде чем замещаемая страница действительно окажется во внешней памяти или просто расформированной, она помечается как кандидат на выгрузку, если в следующий раз произойдет обращение к странице, находящейся в таком «буфере», то страница никуда не выгружается и уходит в конец списка *FIFO*. В противном случае страница действительно выгружается, а на ее место в «буфер» попадает следующий «кандидат». Величина такого «буфера» не может быть большой, поэтому эффективность страничной реализации памяти в *Windows NT/2000/XP* намного ниже, чем в других ОС, и явление пробуксовки проявляется даже при относительно большом объеме ОП.

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

Как и в случае с сегментным способом организации виртуальной памяти, страничный механизм приводит к тому, что без специальных аппаратных средств он существенно замедляет работу вычислительной системы, поэтому используют различные походы к повышению эффективности использования таблиц страниц. Например, одним из примеров эффективного механизма кэширования является ассоциативный кэш. Именно такой ассоциативный кэш и создан в 32-разрядных микропроцессорах i80x86. Начиная с i80386, который поддерживает страничный способ распределения памяти, в этих микропроцессорах имеется кэш на 32 страничных дескриптора. Поскольку размер страницы в этих микропроцессорах равен 4 Кбайт, возможно быстрое обращение к памяти раз-

¹⁰ В системе *Windows NT/2000/XP* файл с выгруженными виртуальными страницами носит название *PageFile.sys*.

мером 128 Кбайт. Различные походы к повышению эффективности использования таблиц страниц рассмотренные ниже в п. 4.2.8.

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

1) Страничная трансляция виртуальной памяти требует существенных накладных расходов: таблицы страниц нужно также размещать в памяти и обрабатывать при помощи диспетчера памяти.

2) Программы разбиваются на страницы случайно, без учета логических взаимосвязей, имеющих в коде. Поэтому межстраничные переходы, как правило, осуществляются чаще, нежели межсегментные, а также становится трудно организовать разделение программных модулей между выполняющимися процессами.

Для того, чтобы избежать второго недостатка, постаравшись сохранить достоинства страничного способа распределения памяти, был предложен еще один способ – *сегментно-страничный*, особенности реализации которого будут рассмотрены ниже в п. 4.2.9.

4.2.8 Особенности эффективного использования таблиц страниц

Одним из основных элементов, необходимых при страничном распределении памяти и существенно влияющих на эффективность ее использования в целом, является таблица страниц. Существуют различные варианты организации и использования таблиц страниц, направленные на повышение эффективности их функционирования, отличающиеся как структурой таблиц (*многоуровневые, инвертированные*), так и способом доступа к их записям (*ассоциативный*). Рассмотрим их более подробно.

Многоуровневые таблицы страниц. Основную проблему для эффективной реализации таблицы страниц создают большие размеры виртуальных адресных пространств современных компьютеров, которые обычно определяются разрядностью архитектуры процессора.

Например, в 32-битном адресном пространстве при размере страницы 4 Кбайт (*Intel*) получаем $2^{32}/2^{12} = 2^{20}$, т.е. приблизительно миллион страниц, а в 64-битном и того более. Таким образом, таблица должна иметь примерно миллион строк, а запись в строке состоит из нескольких байтов. Отметим, каждый процесс нуждается в своей таблице страниц (а в случае сегментно-страничной схемы, особенности реализации которой рассмотрены ниже в п. 4.2.9, желательно иметь по одной таблице страниц на каждый сегмент).

Понятно, что количество памяти, отводимое таблицам страниц, не

может быть так велико. Для того чтобы избежать размещения в памяти огромной таблицы, ее разбивают на ряд фрагментов. В ОП хранят лишь некоторые, необходимые для конкретного момента исполнения фрагменты таблицы страниц.

В силу свойства *локальности*¹¹ число таких фрагментов относительно невелико. Выполнить разбиение таблицы страниц на части можно по-разному. Наиболее распространенный способ разбиения – организация так называемой *многоуровневой таблицы страниц*.

Для примера рассмотрим двухуровневую таблицу с размером страниц 4 Кбайт, реализованную в 32-разрядной архитектуре *Intel* (рис. 31).

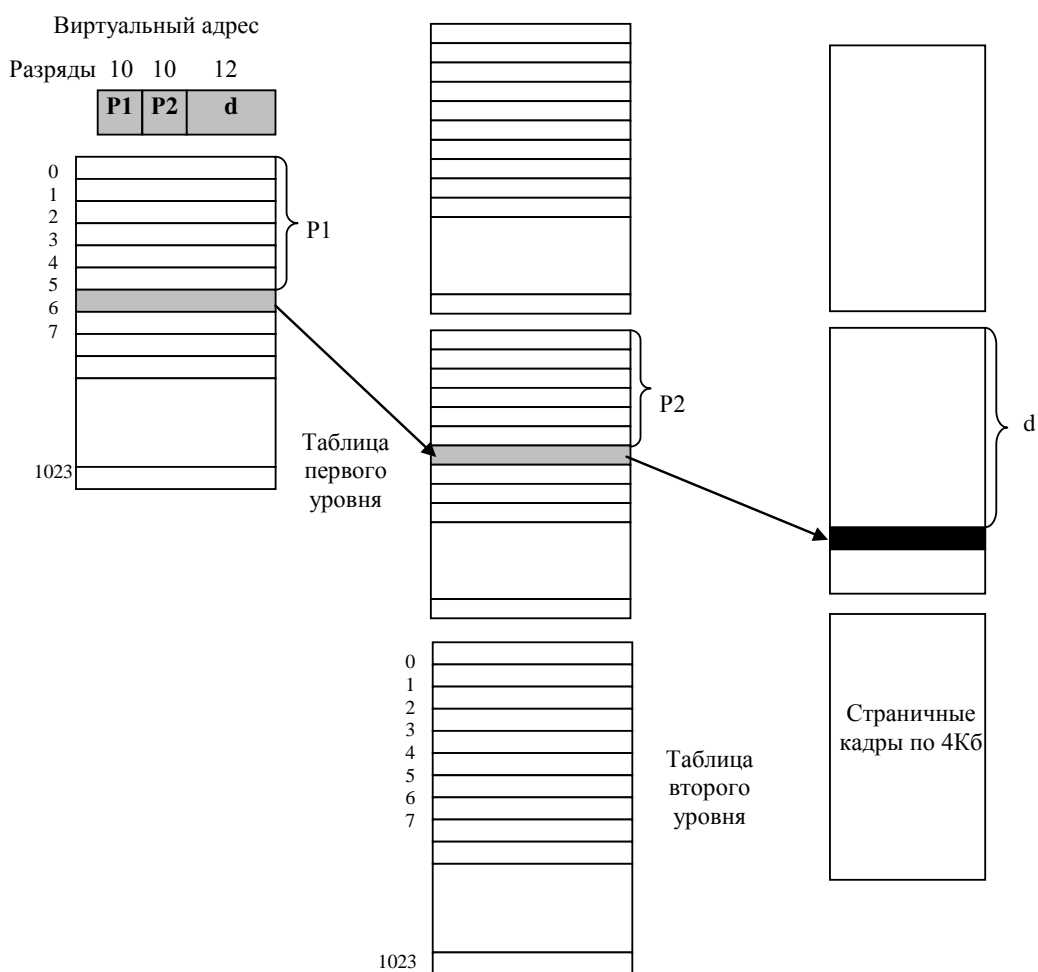


Рисунок 31 – Пример двухуровневой таблицы страниц

Таблица, состоящая из 2^{20} строк, разбивается на 2^{10} таблиц второго уровня по 2^{10} строк. Эти таблицы второго уровня объединены в общую структуру при помощи одной таблицы первого уровня, состоящей из 2^{10} строк. 32-разрядный адрес делится на 10-разрядное поле p_1 , 10-

¹¹ Локальность – способность в течение ограниченного отрезка времени работать с небольшим набором адресов памяти. Различают *временную локальность* (высокая вероятность повторного обращения по одному и тому же адресу в ближайшее время) и *пространственную локальность* (высокая вероятность повторного обращения по соседнему адресу в ближайшее время).

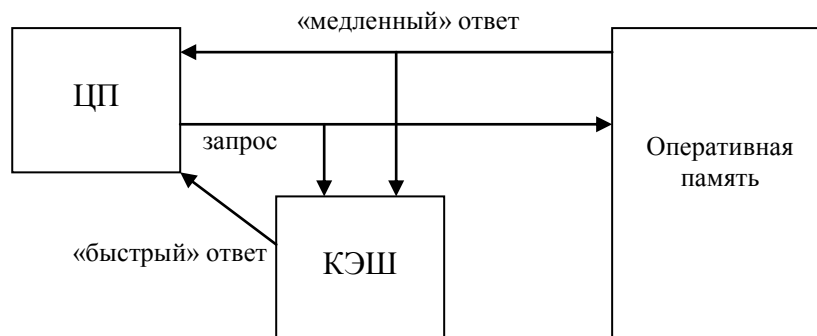
разрядное поле p_2 и 12-разрядное смещение d . Поле p_1 указывает на нужную строку в таблице первого уровня, поле p_2 – второго, а поле d локализует нужный байт внутри указанного страничного кадра.

При помощи всего лишь одной таблицы второго уровня можно охватить 4 Мбайт (4 Кбайт \times 1024) оперативной памяти. Таким образом, для размещения процесса с большим объемом занимаемой памяти достаточно иметь в памяти одну таблицу первого уровня и несколько таблиц второго уровня. Очевидно, что суммарное количество строк в этих таблицах будет много меньше 2^{20} .

По аналогии память может быть адресована и с использованием трех- и более уровневой таблицы. Количество уровней в таблице страниц зависит от конкретных особенностей архитектуры. Можно привести примеры реализации одноуровневого (*DEC PDP-11*), двухуровневого (*Intel*, *DEC VAX*), трехуровневого (*Sun SPARC*, *DEC Alpha*) пейджинга (от англ. *paging* – страничная организация памяти), а также пейджинга с заданным количеством уровней (*Motorola*). Функционирование *RISC*-процессора *MIPS R2000* осуществляется вообще без таблицы страниц. Здесь поиск нужной страницы, если эта страница отсутствует в ассоциативной памяти, должна взять на себя ОС (так называемый *zero level paging*).

Ассоциативная память. Поиск номера кадра, соответствующего нужной странице, в многоуровневой таблице страниц требует нескольких обращений к основной памяти и занимает много времени. Ускорения такого поиска добиваются на уровне архитектуры компьютера. Учитывая упомянутое выше свойство локальности, большинство обращений к памяти в течение некоторого промежутка времени осуществляется к небольшому количеству страниц. Поэтому, естественным решением проблемы ускорения – снабдить компьютер аппаратным устройством для отображения виртуальных страниц в физические без обращения к таблице страниц с использованием небольшой и быстрой кэш-памяти, хранящей необходимую на данный момент часть таблицы страниц (рис. 32). Такое устройство называют *ассоциативной памятью* или *буфером поиска трансляции* (англ. *translation lookaside buffer* – *TLB*).

Одна запись таблицы в ассоциативной памяти (один вход) содержит информацию об одной виртуальной странице: ее атрибутах и кадре, в котором она находится. Эти поля в точности соответствуют полям в таблице страниц. Рассмотрим функционирование менеджера памяти при наличии ассоциативной памяти.



Структура кэш-памяти

Адрес данных в ОП	Данные	Управл. информация	
		бит модиф.	бит обращ.
...

Рисунок 32 – Общие принципы функционирования кэш-памяти

В первый момент времени осуществляется поиск информации о необходимой странице в ассоциативной памяти. Если нужная запись найдена, то производится отображение этой страницы в физическую память, за исключением случаев нарушения привилегий, когда запрос на обращение к памяти отклоняется.

Если нужная запись в ассоциативной памяти отсутствует, отображение осуществляется через таблицу страниц: происходит замена одной из записей в ассоциативной памяти найденной записью из таблицы страниц. В этот момент необходимо решение проблемы замещения (определить какая запись подлежит изменению). Конструкция ассоциативной памяти должна организовывать записи таким образом, чтобы можно было принять решение о том, какая из старых записей должна быть удалена при внесении новых.

Основным параметром, влияющим на эффективность использования ассоциативной памяти, является *процент попаданий в кэш* (англ. *hit ratio*) – число удачных поисков номера страницы в ассоциативной памяти по отношению к общему числу поисков. Обращение к одним и тем же страницам повышает процент попаданий в кэш. Чем больше этот процент, тем меньше среднее время доступа к данным, находящимся в ОП. Предположим, например, что для доступа к памяти через таблицу страниц необходимо 100 нс, а для доступа через ассоциативную память – 20 нс. Если *hit ratio* – 90% (что соответствует значению попадания в реальных ОС), то среднее время доступа рассчитывается как

$$0,9 \times 20 + 0,1 \times 100 = 28 \text{ нс.}$$

Такая сравнительно высокая производительность современных ОС показывает эффективность использования ассоциативной памяти. Высокое значение вероятности нахождения данных в ассоциативной памяти связано с наличием у данных таких объективных свойств как пространственная и временная локальность.

Следует обратить внимание на то, что при переключении контекста процессов нужно добиться того, чтобы новый процесс «не видел» в ассоциативной памяти информацию, относящуюся к предыдущему процессу, поэтому требуется ее «очистка». Очевидно, что это ведет к дополнительным затратам на переключения контекста процесса при использовании ассоциативной памяти.

Инвертированная таблица страниц. Несмотря на многоуровневую организацию, хранение нескольких таблиц страниц большого размера по-прежнему представляют собой проблему. Особенно это актуально для 64-разрядных архитектур, где число виртуальных страниц очень велико. Одним из вариантов решения этой проблемы, позволяющей существенно уменьшить объем памяти, занятый под таблицы страниц, является применение *инвертированной таблицы страниц* (*inverted page table*). Этот подход применяется на машинах *PowerPC*, некоторых рабочих станциях *Hewlett-Packard*, *IBM RT*, *IBM AS/400* и других.

В этой таблице содержится по одной записи на каждый страничный кадр физической памяти. Достаточно одной таблицы для всех процессов. Для хранения функции отображения требуется фиксированная часть основной памяти, независимо от разрядности архитектуры, размера и количества процессов. Например, для компьютера *Pentium* с 256 Мбайт оперативной памяти нужна таблица размером 64 Кбайт строк.

Несмотря на экономию ОП, применение инвертированной таблицы имеет существенный минус – записи в ней (как и в ассоциативной памяти) не отсортированы по возрастанию номеров виртуальных страниц, что усложняет трансляцию адреса. Один из способов решения данной проблемы – использование *хеш-таблицы*¹² виртуальных адресов. Часть виртуального адреса, представляющая собой номер страницы, отображается в хеш-таблицу с использованием *функции хеширования*¹³. Каждой странице физической памяти здесь соответствует одна запись в хеш-таблице и инвертированной таблице страниц. Виртуальные адреса, имеющие одно значение хеш-функции, сцепляются друг с другом, при этом длина цепочки обычно не превышает двух записей.

¹² Хеш-таблица – структура данных вида ассоциативный массив, которая ассоциирует ключи со значениями. При этом *хэш* – число фиксированной длины, которое ставится в соответствие данным произвольной длины таким образом, чтобы вероятность появления различных данных с одинаковым хешем стремилась к нулю, а восстановить данные по их хешу было крайне трудно.

¹³ Хеш-функция – функция, выполняющая одностороннее преобразование (хеширование) входных данных.

4.2.9 Сегментно-страничное распределение

Как и в сегментном способе распределения памяти, программа разбивается на логически законченные части – сегменты – и виртуальный адрес содержит указание на номер соответствующего сегмента. Вторая составляющая виртуального адреса – смещение относительно начала сегмента – в свою очередь может быть представлена состоящей из двух полей: виртуальной страницы и индекса. Другими словами, получается, что виртуальный адрес теперь состоит из трех компонентов: сегмента, страницы и индекса. Получение физического адреса и извлечение из памяти необходимого элемента для этого способа иллюстрирует рис. 33. Очевидно, что этот способ организации доступа к памяти вносит еще большую временную задержку, т.к. необходимо сначала вычислить адрес дескриптора сегмента и прочесть его, затем определить адрес элемента таблицы страниц этого сегмента и извлечь из памяти необходимый элемент и уже только после этого можно приписать к номеру физической страницы номер ячейки в странице (индекс). Задержка доступа к искомой ячейке получается, по крайней мере, в три раза больше, чем при простой прямой адресации.

Чтобы избежать указанной неприятности вводится кэширование, причем кэш, как правило, строится по ассоциативному принципу. Другими словами, просмотры двух таблиц в памяти могут быть заменены одним обращением к ассоциативной памяти.

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

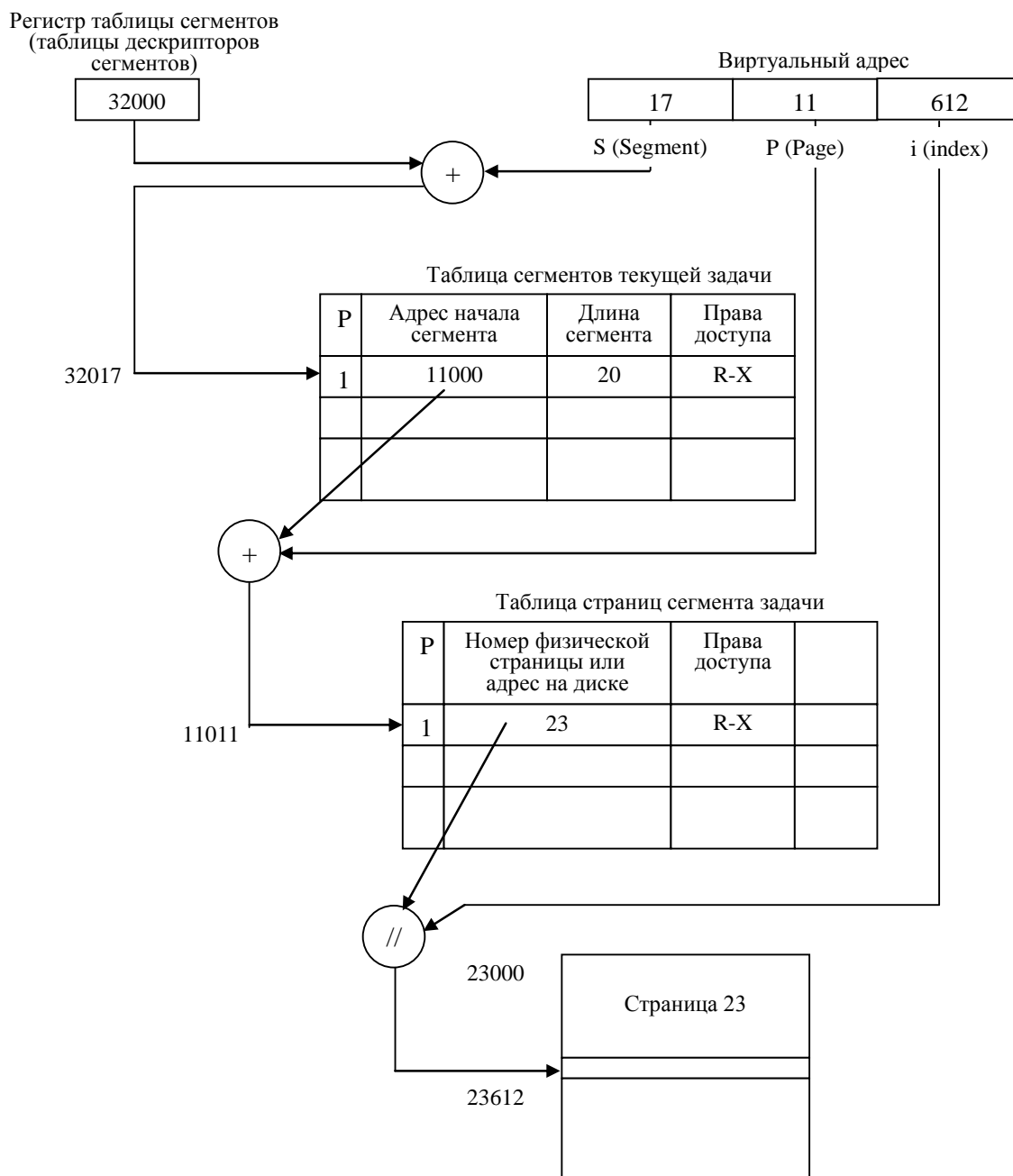


Рисунок 33 – Сегментно-страничный способ организации виртуальной памяти

Сегментно-страничный способ имеет целый ряд достоинств. Разбиение программы на сегменты позволяет размещать сегменты в памяти целиком. Сегменты разбиты на страницы, все страницы сегмента загружаются в память. Это позволяет сократить число обращений к отсутствующим страницам, поскольку вероятность выхода за пределы сегмента меньше вероятности выхода за пределы страницы. Страницы исполняемого сегмента находятся в памяти, но при этом они могут находиться не рядом друг с другом, а «россыпью», поскольку диспетчер па-

мемори манипулирует страницами. Наличие сегментов облегчает разделение программных модулей между параллельными процессами. Возможна и динамическая компоновка задачи. А выделение памяти страницами позволяет минимизировать фрагментацию.

Как отмечалось выше, этот способ распределения памяти требует значительных затрат вычислительных ресурсов и его не так просто реализовать, поэтому используется он сравнительно редко, причем в дорогих мощных вычислительных системах.

Возможность реализовать сегментно-страничное распределение памяти заложена и в семейство микропроцессоров i80x86, однако вследствие слабой аппаратной поддержки, трудностей при создании систем программирования и операционной системы практически в персональных компьютерах эта возможность не используется.

4.3 Вопросы для самопроверки

1. На какие виды разделяют запоминающие устройства компьютера?
2. Как выглядит иерархия запоминающих устройств по убыванию времени доступа, возрастанию цены и увеличению их емкости?
3. В чем заключается особая роль памяти компьютера?
4. Какие функции необходимы ОС для управления памятью в мультипрограммной системе?
5. Что подразумевается под пространством символьных имен программы?
6. Что подразумевается под виртуальным адресным пространством программы (процесса)?
7. Какова схема отображения пространства имен на физическую память компьютера?
8. Какие существуют варианты перехода (отображения) пространства символьных имен на физическую память? В чем суть каждого из вариантов?
9. Какие существуют виды (методы) распределения памяти, характерные для однопрограммных и мультипрограммных ОС?
10. В чем отличие сегмента кода и сегмента данных и как учитывается это отличие при распределении памяти?
11. Каковы особенности организации управления памятью в мультипрограммных ОС?
12. Чем характеризуется распределение памяти фиксированными разделами? Какие задачи при этом решает подсистема управления памятью ОС?

13. Чем характеризуется распределение памяти динамическими разделами? Какие задачи при этом решает подсистема управления памятью ОС?

14. В чем заключается распределение памяти перемещаемыми разделами? В какие моменты в данном случае можно выполнять сжатие памяти?

15. В чем заключается сегментное распределение памяти? Какова при этом схема получения физического адреса из виртуального? Какова структура таблицы сегментов?

16. Какие дисциплины замещения используются при определении того сегмента, который должен быть перемещен из оперативной во внешнюю память? Какие дисциплины требуют наличия дополнительных аппаратных средств процессора?

17. В чем заключается страничное распределение памяти? Какова при этом схема получения физического адреса из виртуального? Какова структура таблицы страниц?

18. Какие существуют варианты организации и использования таблиц страниц?

19. В чем заключается сегментно-сегментное распределение памяти? Какова при этом схема получения физического адреса из виртуального?

5. ФАЙЛОВЫЕ СИСТЕМЫ

История систем управления данными во внешней памяти начинается еще с магнитных лент, но современный облик они приобрели с появлением магнитных дисков. До этого каждая прикладная программа сама решала проблемы именования данных и их структуризации во внешней памяти.

Это затрудняло поддержание на внешнем носителе нескольких архивов долговременно хранящейся информации. Историческим шагом стал переход к использованию централизованных систем управления файлами. Система управления файлами берет на себя распределение внешней памяти, отображение имен файлов в адреса внешней памяти и обеспечение доступа к данным.

Рассмотрение принципов построения и особенностей функционирования файловых систем целесообразно начать с основ физической организации носителя большинства файловых систем – жесткого диска.

5.1 Физическая организация жесткого диска

Основным типом устройств, которые используются в современных вычислительных системах для хранения файлов, являются дисковые накопители. Эти устройства предназначены для считывания и записи данных на жесткие и гибкие магнитные диски. Жесткий диск состоит из одной или нескольких стеклянных или металлических пластин, каждая из которых покрыта с одной или двух сторон магнитным материалом. Таким образом, диск в общем случае состоит из пакета пластин (рис. 34).

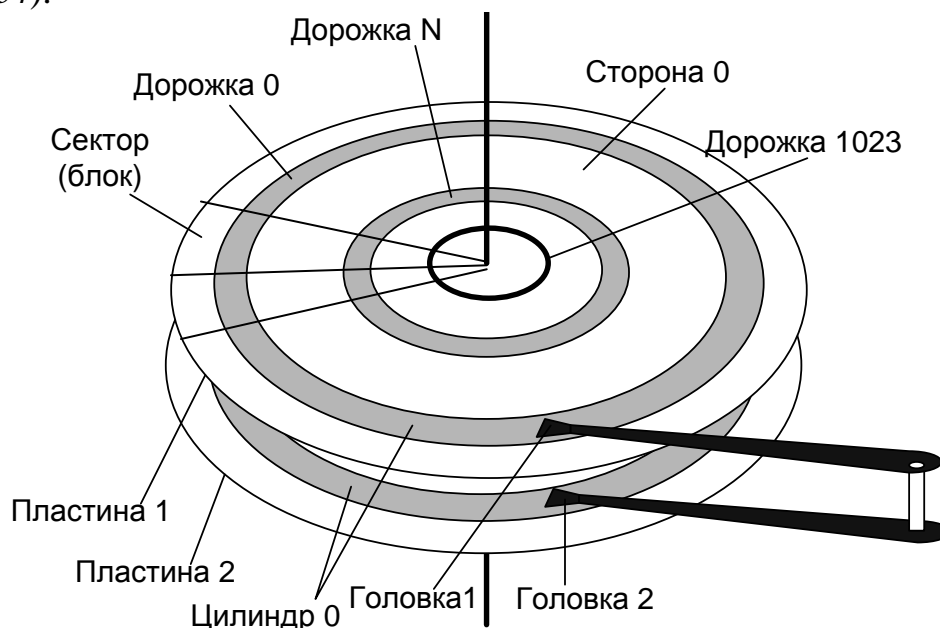


Рисунок 34 – Схема устройства жесткого диска

На каждой стороне каждой пластины размечены тонкие концентрические кольца – *дорожки* (англ. *traks*), на которых хранятся данные. Количество дорожек зависит от типа диска. Нумерация дорожек начинается с 0 от внешнего края к центру диска. Когда диск вращается, элемент, называемый головкой, считывает двоичные данные с магнитной дорожки или записывает их на нее.

Головка может позиционироваться над заданной дорожкой. Головки перемещаются над поверхностью диска дискретными шагами, каждый шаг соответствует сдвигу на одну дорожку. Запись на диск осуществляется благодаря способности головки изменять магнитные свойства дорожки. В некоторых дисках вдоль каждой поверхности перемещается одна головка, а в других – имеется по головке на каждую дорожку. В первом случае для поиска информации головка должна перемещаться по радиусу диска. Обычно все головки закреплены на едином перемещающем механизме и двигаются синхронно. Поэтому, когда головка фиксируется на заданной дорожке одной поверхности, все остальные головки останавливаются над дорожками с такими же номерами. В тех же случаях, когда на каждой дорожке имеется отдельная головка, никакого перемещения головок с одной дорожки на другую не требуется, за счет этого экономится время, затрачиваемое на поиск данных.

Совокупность дорожек одного радиуса на всех поверхностях всех пластин пакета называется *цилиндром* (англ. *cylinder*). Каждая дорожка разбивается на фрагменты, называемые *секторами* (англ. *sectors*) или *блоками* (англ. *blocks*). На первых жестких дисках все дорожки имели равное число секторов, в которые можно максимально записать одно и то же число байт. Сектор имел фиксированный для конкретной системы размер, выражающийся степенью двойки. Учитывая, что дорожки разного радиуса имели одинаковое число секторов, плотность записи становилась тем выше, чем ближе дорожка к центру. Однако, современные жесткие диски имеют постоянную плотность записи, поэтому на разных дорожках располагается различное количество секторов.

Сектор – наименьшая адресуемая единица обмена данными дискового устройства с ОП. Для того чтобы контроллер мог найти на диске нужный сектор, необходимо задать ему все составляющие адреса сектора: номер цилиндра, номер поверхности и номер сектора. В связи с тем, что прикладной программе в общем случае нужен не сектор, а некоторое количество байт, не обязательно кратное размеру сектора, то типичный запрос включает чтение нескольких секторов, содержащих требуемую информацию, и одного или двух секторов, содержащих наряду с требуемыми избыточные данные (рис. 35).

Операционная система при работе с диском использует, как правило, собственную единицу дискового пространства, называемую *класте*

ром. При создании файла место на диске ему выделяется кластерами. Например, если файл имеет размер 2560 байт, а размер кластера в файловой системе определен в 1024 байта, то файлу будет выделено на диске 3 кластера.

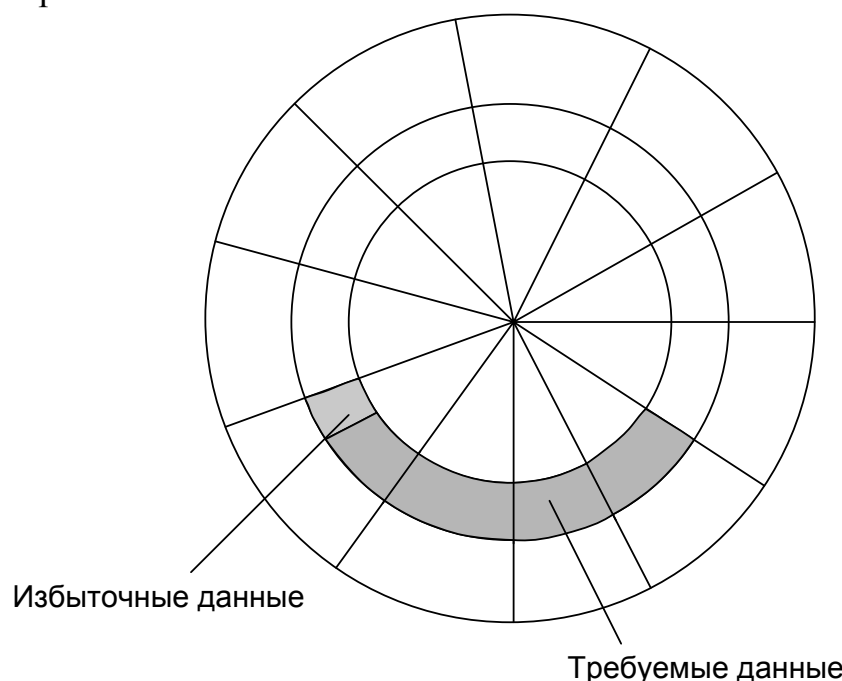


Рисунок 35 – Считывание избыточных данных при обмене с диском

Дорожки и секторы создаются в результате выполнения процедуры физического, или низкоуровневого форматирования диска, предшествующей использованию диска. Для определения границ блоков на диск записывается идентификационная информация. Низкоуровневый формат диска не зависит от типа ОС, которая этот диск будет использовать.

Разметку диска под конкретный тип файловой системы выполняют процедуры высокоуровневого, или логического, форматирования. При высокоуровневом форматировании определяется размер кластера и на диск записывается информация, необходимая для работы файловой системы, в том числе информация о доступном и неиспользуемом пространстве, о границах областей, отведенных под файлы и каталоги, информация о поврежденных областях.

Жесткий диск может содержать до четырех основных разделов. Это ограничение связано с характером организации данных на жестких дисках *IBM*-совместимых компьютеров.

В первом физическом секторе жесткого диска располагается головная запись загрузки и таблица разделов (табл. 1).

Головная запись загрузки (англ. *master boot record, MBR*) – первая часть данных на жестком диске. Она зарезервирована для программы начальной загрузки *BIOS (ROM Bootstrap routine)*, которая при загрузке

с жесткого диска считывает и загружает в память первый физический сектор на активном разделе диска, называемый *загрузочным сектором* (англ. *Boot Sector*). Программа, расположенная в *MBR*, носит название *вне-системного загрузчика* (*Non-System Bootstrap, NSB*).

Каждая запись в таблице разделов (англ. *partition table*) содержит начальную позицию и размер раздела на жестком диске, а также информацию о том, первый сектор какого раздела содержит загрузочный сектор.

Таблица 1. Структура таблицы разделов

Размер (байт)	Описание
446	Загрузочная запись (MBR)
16	Запись 1 раздела
16	Запись 2 раздела
16	Запись 3 раздела
16	Запись 4 раздела
2	Сигнатура $055AA_h$

Можно сказать, что таблица разделов – одна из наиболее важных структур данных на жестком диске. Если эта таблица повреждена, то не только не будет загружаться ни одна из установленных на компьютере ОС, но станут недоступными данные, расположенные в диске, особенно если жесткий диск был разбит на несколько разделов.

Последние два байта таблицы разделов имеют значение $055AA_h$, то есть чередующиеся значения 0 и 1 в двоичном представлении данных. Эта сигнатура выбрана для того, чтобы проверить работоспособность всех линий передачи данных. Значение $055AA_h$, присвоенное последним двум байтам, имеется во всех загрузочных секторах.

Многие ОС позволяют создавать, так называемый, расширенный (англ. *extended*) раздел, который по аналогии с разделами может разбиваться на несколько логических дисков (англ. *logical disks*). В этом смысле термин «*первичный*» можно признать не совсем удачным переводом слова «*primary*» – лучше было бы перевести «простейший», или «примитивный». В этом случае становится понятным и логичным термин «расширенный». Расширенный раздел содержит вторичную запись *MBR* (англ. *Secondary MBR, SMBR*), в состав которой вместо таблицы разделов входит аналогичная ей *таблица логических дисков* (англ. *Logical Disks Table, LDT*). Таблица логических дисков описывает размещение и характеристики раздела, содержащего единственный логический диск, а также может специфицировать следующую запись

SMBR. Следовательно, если в расширенном разделе создано K логических дисков, то он содержит K экземпляров *SMBR*, связанных в список. Каждый элемент этого списка описывает соответствующий логический диск и ссылается (кроме последнего) на следующий элемент списка. Загрузчик *NSB* служит для поиска с помощью таблицы разделов активного раздела, затем копирования в ОП компьютера системного загрузчика (англ. *System Bootstrap*, *SB*) из выбранного раздела и передачи на него управления, что позволяет осуществить загрузку ОС.

Вслед за сектором *MBR* размещаются собственно сами разделы (рис. 36). В процессе начальной загрузки сектора *MBR*, содержащего таблицу разделов, работают программные модули *BIOS*. Начальная загрузка считается выполненной корректно только в том случае, если таблица разделов содержит допустимую информацию.

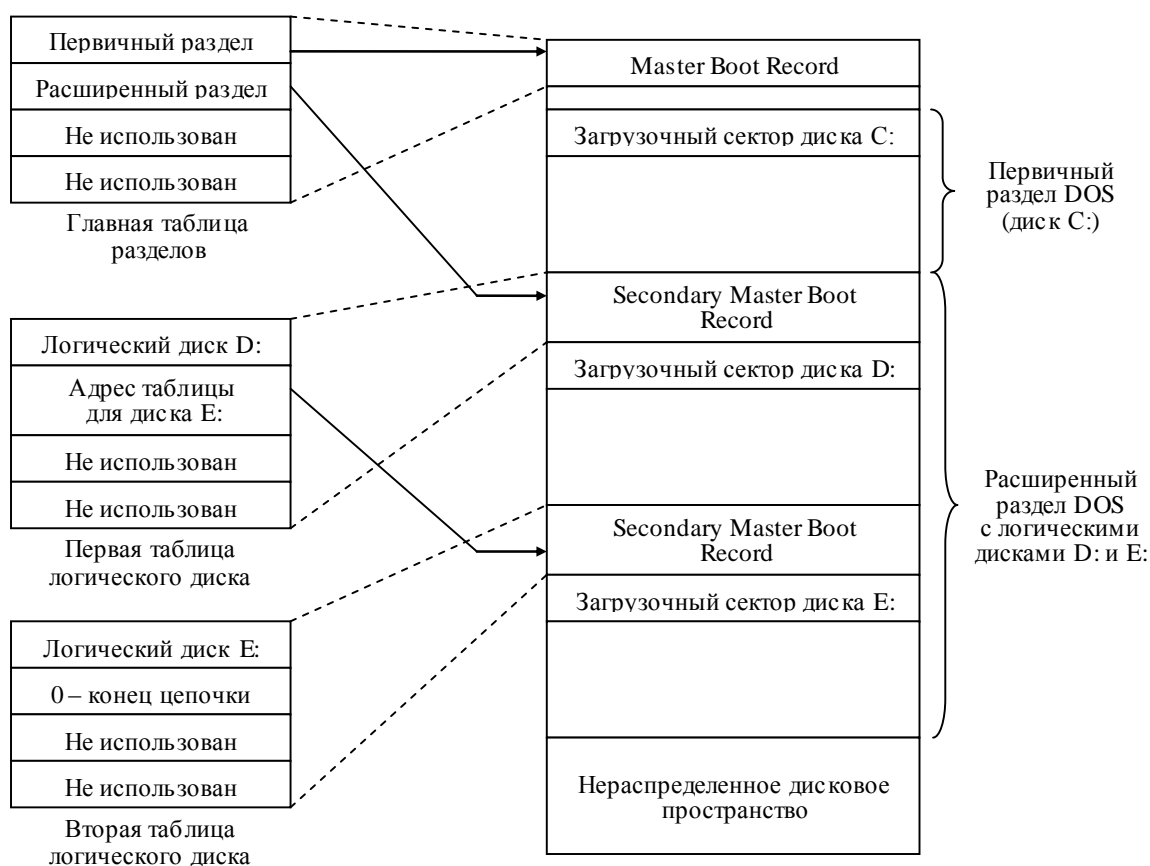


Рисунок 36 – Разбиение диска на разделы

5.2 Принципы построения файловой системы

5.2.1 Интерфейс файловой системы

В соответствии со своим предназначением файловая система должна организовать эффективную работу с данными, хранящимися во

внешней памяти и предоставить пользователю возможности для запоминания и выборки данных в ней.

Для организации хранения информации на диске пользователь вначале обычно выполняет его форматирование, выделяя на нем место для структур данных, которые описывают состояние файловой системы в целом. Затем создает нужную ему структуру каталогов (директорий), которые по существу являются списками вложенных каталогов и собственно файлов. И, наконец, заполняет дисковое пространство файлами, приписывая их тому или иному каталогу. Таким образом, ОС должна предоставить в распоряжение пользователя совокупность сервисов, традиционно реализованных через системные вызовы, которые обеспечивают:

- создание файловой системы на диске;
- необходимые операции для работы с каталогами;
- необходимые операции для работы с файлами.

Кроме того, файловые службы могут решать проблемы проверки и сохранения целостности файловой системы, проблемы повышения производительности и ряд других.

Прежде чем приступить к описанию работы отдельных файловых операций, необходимо рассмотреть ключевые алгоритмы и структуры данных, которые обеспечивают функционирование файловой системы.

5.2.2 Функциональная схема организации файловой системы

Система хранения данных на дисках может быть представлена в виде функциональной схемы (рис. 37). Рассмотрим ее подробнее.

Нижний уровень – оборудование. Это в первую очередь, магнитные диски с подвижными головками, особенности физической организации которых рассмотрены в п. 5.1.

Непосредственно с устройствами (дисками) взаимодействует часть ОС, называемая *системой ввода-вывода*. Система ввода-вывода (она состоит из драйверов устройств и обработчиков прерываний для передачи информации между памятью и дисковой системой) предоставляет в распоряжение более высокоуровневого компонента ОС – файловой системы используемое дисковое пространство в виде непрерывной последовательности блоков фиксированного размера. Система ввода-вывода имеет дело с физическими блоками диска, которые характеризуются адресом – номерами диска, цилиндра и сектора. Файловая система имеет дело с логическими блоками, каждый из которых имеет номер (от 0 или 1 до N). Размер этих логических блоков файла совпадает или кратен размеру физического блока диска и может быть задан равным размеру страницы виртуальной памяти, поддерживаемой аппаратурой компьютера совместно с ОС.

В структуре системы управления файлами можно выделить базисную подсистему, которая отвечает за выделение дискового пространства конкретным файлам, и более высокоуровневую логическую подсистему, которая использует структуру дерева директорий для предоставления модулю базисной подсистемы необходимой ей информации исходя из символического имени файла. Она также ответственна за авторизацию доступа к файлам.

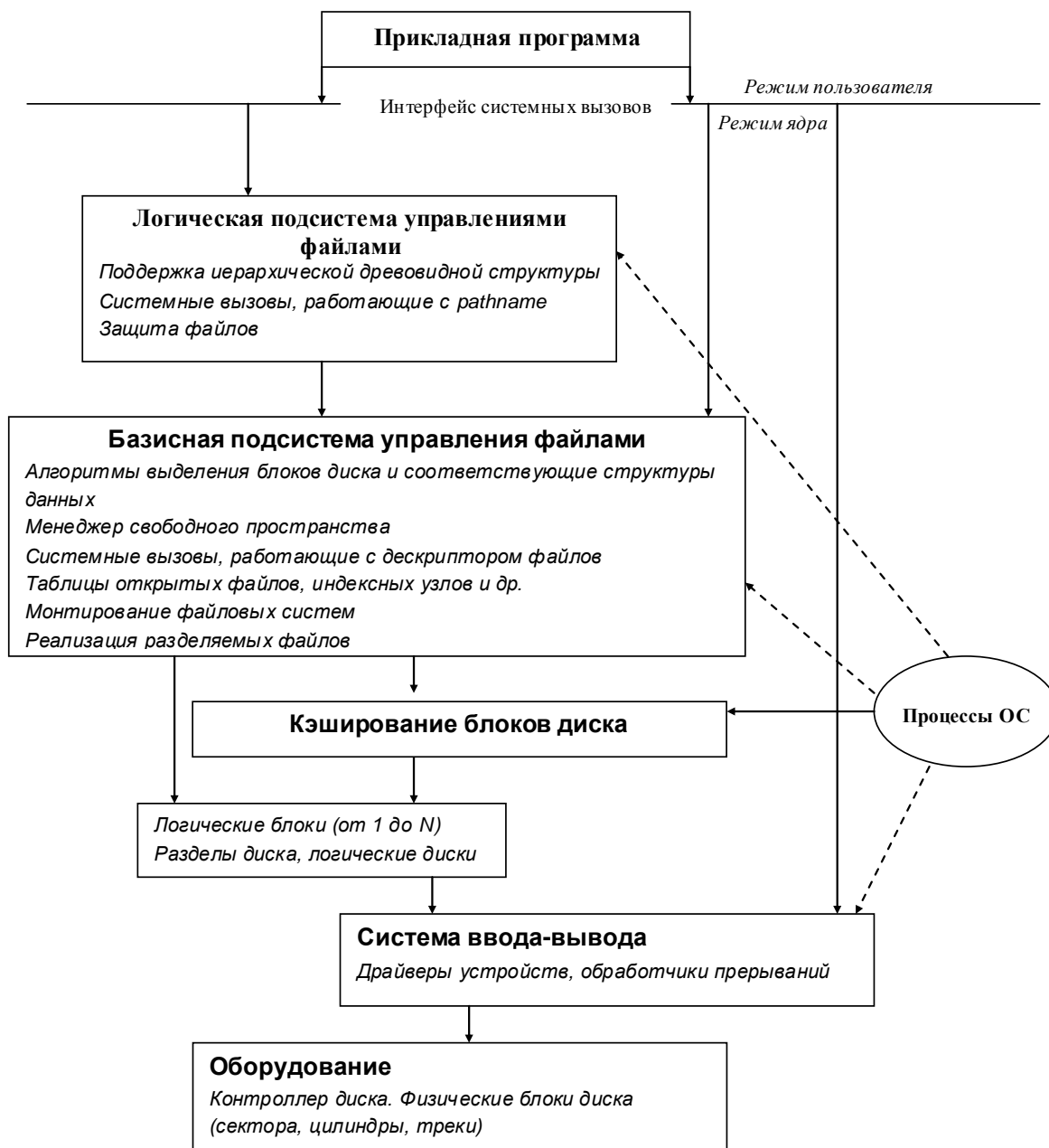


Рисунок 37 – Функциональная схема организации файловой системы

Информация на диске организована в виде иерархической древовидной структуры, состоящей из набора файлов, каждый из которых яв-

ляется хранилищем данных пользователя, и каталогов или директорий, которые необходимы для хранения информации о файлах системы.

Взаимодействие прикладной программы с файлом происходит следующим образом. От прикладной программы к логической подсистеме поступает запрос на открытие или создание файла. Логическая подсистема, используя структуру директорий, проверяет права доступа и вызывает базовую подсистему для получения доступа к блокам файла. После этого файл считается открытым, содержится в таблице открытых файлов, прикладная программа получает в свое распоряжение дескриптор (в системах *Microsoft* – *handle*) этого файла. Дескриптор файла является ссылкой на файл в таблице открытых файлов и используется в запросах прикладной программы на чтение-запись из этого файла. Запись в таблице открытых файлов указывает через систему *кэширования блоков диска* на блоки данного файла. Если к моменту открытия файл уже используется другим процессом, то есть содержится в таблице открытых файлов, то после проверки прав доступа к файлу может быть организован совместный доступ. При этом новому процессу также возвращается дескриптор файла.

Прежде чем рассмотреть структуру данных файловой системы на диске следует рассмотреть алгоритмы выделения дискового пространства и способы учета свободной и занятой дисковой памяти.

5.2.3 Типовая структура файловой системы на диске

Обзор способов работы с дисковым пространством дает общее представление о совокупности служебных данных, необходимых для описания файловой системы. Структуры данных типовой файловой системы, например *Unix*, на одном из разделов диска, может состоять из 4-х основных частей (рис. 38).

Суперблок	Структуры данных, описывающие свободное дисковое пространство и свободные индексные узлы	Массив индексных узлов	Блоки диска данных файлов
-----------	--	------------------------	---------------------------

Рисунок 38 – Пример структуры файловой системы на диске

В начале раздела находится *суперблок*, содержащий общее описание файловой системы, например:

- тип файловой системы;
- размер файловой системы в блоках;
- размер массива индексных узлов;
- размер логического блока;
- и т.д.

В файловых системах современных ОС для повышения устойчивости поддерживается несколько копий суперблока. В некоторых версиях ОС *Unix* суперблок включал также и структуры данных, управляющие распределением дискового пространства, в результате чего суперблок непрерывно подвергался модификации, что снижало надежность файловой системы в целом. Выделение структур данных, описывающих дисковое пространство, в отдельную часть является более правильным решением.

Описанные структуры данных создаются на диске в результате его форматирования специализированными утилитами ОС. Их наличие позволяет обращаться к данным на диске как к файловой системе, а не как к обычной последовательности блоков.

Массив индексных узлов (англ. *ilist*) содержит список индексов, соответствующих файлам данной файловой системы. Размер массива индексных узлов определяется администратором при установке системы. Максимальное число файлов, которые могут быть созданы в файловой системе, определяется числом доступных индексных узлов.

В блоках данных хранятся реальные данные файлов. Размер логического блока данных может задаваться при форматировании файловой системы. Заполнение диска содержательной информацией предполагает использование блоков хранения данных для файлов директорий и обычных файлов и имеет следствием модификацию массива индексных узлов и данных, описывающих пространство диска. Отдельно взятый блок данных может принадлежать одному и только одному файлу в файловой системе.

5.2.4 Способы выделения дискового пространства

Ключевой вопрос реализации файловой системы – способ связывания файлов с блоками диска. В ОС используется несколько способов выделения файлу дискового пространства, для каждого из которых сведения о локализации блоков данных файла можно извлечь из записи в директории, соответствующей символьному имени файла.

Выделение непрерывной последовательностью блоков. Простейший способ – хранить каждый файл, как непрерывную последовательность блоков диска. При непрерывном расположении файл характеризуется адресом и длиной (в блоках). Файл, стартующий с блока b , занимает затем блоки $b + 1$, $b + 2$, ... $b + n - 1$. Этот способ имеет два преимущества:

- легкая реализация, так как выяснение местонахождения файла сводится к вопросу, где находится первый блок;
- обеспечивает хорошую производительность, потому что целый файл может быть считан за одну дисковую операцию.

Например, непрерывное выделение использовано в ОС *IBM/CMS*, *RSX-11* (для исполняемых файлов).

Основная проблема, в связи с которой этот способ мало распространен – трудности в поиске места для нового файла. В процессе эксплуатации диск представляет собой некоторую совокупность свободных и занятых фрагментов. Проблема непрерывного расположения может рассматриваться как частный случай более общей проблемы выделения n блоков из списка свободных фрагментов. Наиболее распространенные стратегии решения этой проблемы – выбор первого подходящего по размеру блока (англ. *first fit*), наиболее подходящего, т.е. того, при размещении в котором наиболее тесно (англ. *best fit*), и наименее подходящего, т.е. выбирается наибольший блок (англ. *worst fit*).

Способ характеризуется наличием существенной *внешней фрагментации* (в большей или меньшей степени – в зависимости от размера диска и среднего размера файла). Кроме того, непрерывное распределение внешней памяти не применимо до тех пор, пока не известен максимальный размер файла. Иногда размер выходного файла оценить легко (при копировании), но чаще – трудно. Если места не достаточно, то пользовательская программа может быть приостановлена, предполагая выделение дополнительного места для файла при последующем перезапуске. Некоторые ОС используют модифицированный вариант непрерывного выделения: «основные блоки файла» + «резервные блоки». Однако с выделением блоков из резерва возникают те же проблемы, так как возникает задача выделения непрерывной последовательности блоков диска теперь уже из совокупности резервных блоков.

Выделение связным списком. Метод распределения блоков в виде связного списка решает основную проблему непрерывного выделения, то есть устраняет внешнюю фрагментацию (рис. 39). Каждый файл – связный список блоков диска. Запись в директории содержит указатель на первый и последний блоки файла. Каждый блок содержит указатель на следующий блок.

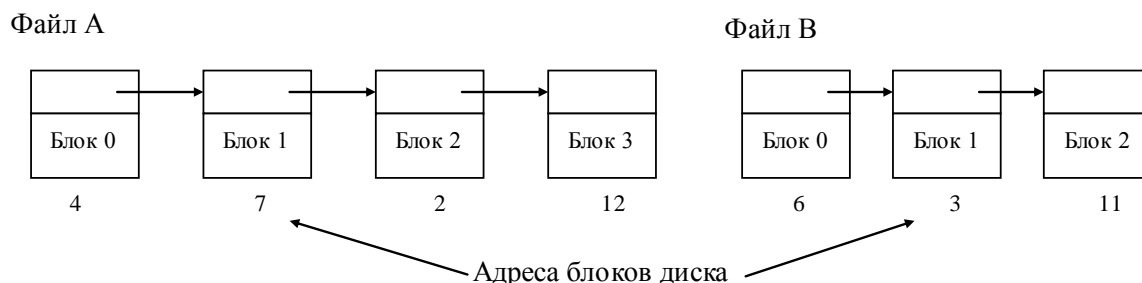


Рисунок 39 – Хранение файлов в связных списках дисковых блоков

Внешняя фрагментация для данного метода отсутствует. Любой свободный блок может быть использован для удовлетворения запроса.

Отметим, что нет необходимости декларировать размер файла в момент создания и, поэтому, файл может неограниченно расти. Связное выделение имеет несколько существенных недостатков:

- при прямом доступе к файлу для поиска i -го блока нужно осуществить несколько обращений к диску, последовательно считывая блоки от 1 до $i - 1$, то есть выборка логически смежных записей, которые занимают физически несмежные секторы, может требовать много времени;
- прямым следствием этого является низкая *надежность* – наличие дефектного блока в списке приводит к потере информации в остаточной части файла и, потенциально, к потере дискового пространства отведенного под этот файл;
- для указателя на следующий блок внутри блока нужно выделить место размером, традиционно определяемым степенью двойки (многие программы читают и записывают блоками по степеням двойки), который перестает быть таковым, так как указатель отбирает несколько байтов.

В связи с вышеизложенным, метод связного списка обычно не используется в чистом виде.

Распределение связным списком с использованием индекса. Недостатки предыдущего способа могут быть устранены путем изъятия указателя из каждого дискового блока и помещения его в индексную таблицу в памяти, которая называется *таблицей размещения файлов* (англ. *file allocation table – FAT*). Этой схемы придерживаются многие ОС (*MS-DOS, OS/2, MS Windows* и др.).

По-прежнему существенно, что запись в директории содержит только ссылку на первый блок. Далее при помощи таблицы *FAT* можно локализовать блоки файла независимо от его размера (значение равно адресу следующего блока этого файла). В тех строках таблицы, которые соответствуют последним блокам файлов, обычно записывается некоторое граничное значение, например *EOF* (рис. 40).

Номер блоков диска		
1	0	
2	10	
3	11	Начало файла F2
4	0	
5	EOF	
6	2	Начало файла F1
7	EOF	
8	0	
9	0	
10	7	
11	5	

Рисунок 40 – Способ выделения памяти с использованием связного списка в ОП

Главное достоинство данного подхода состоит в том, что по таблице отображения можно судить о физическом соседстве блоков, располагающихся на диске, и при выделении нового блока можно легко найти свободный блок диска, находящийся поблизости от других блоков данного файла. Минусом данной схемы может быть необходимость хранения в памяти этой довольно большой таблицы. Более подробно особенности использования таблицы размещения файлов рассмотрены в п. 5.5.1 при описании файловой системы FAT.

Индексные узлы. Четвертый и последний способ «выяснения принадлежности» блока к файлу – связать с каждым файлом маленькую таблицу, называемую *индексным узлом (i-node)*, которая перечисляет атрибуты и дисковые адреса блоков файла (рис. 41). Каждый файл имеет свой собственный индексный блок, который содержит адреса блоков данных. Запись в директории, относящаяся к файлу, содержит адрес индексного блока. По мере заполнения файла указатели на блоки диска в индексном узле принимают осмысленные значения. Индексирование поддерживает прямой доступ к файлу, без ущерба от внешней фрагментации.

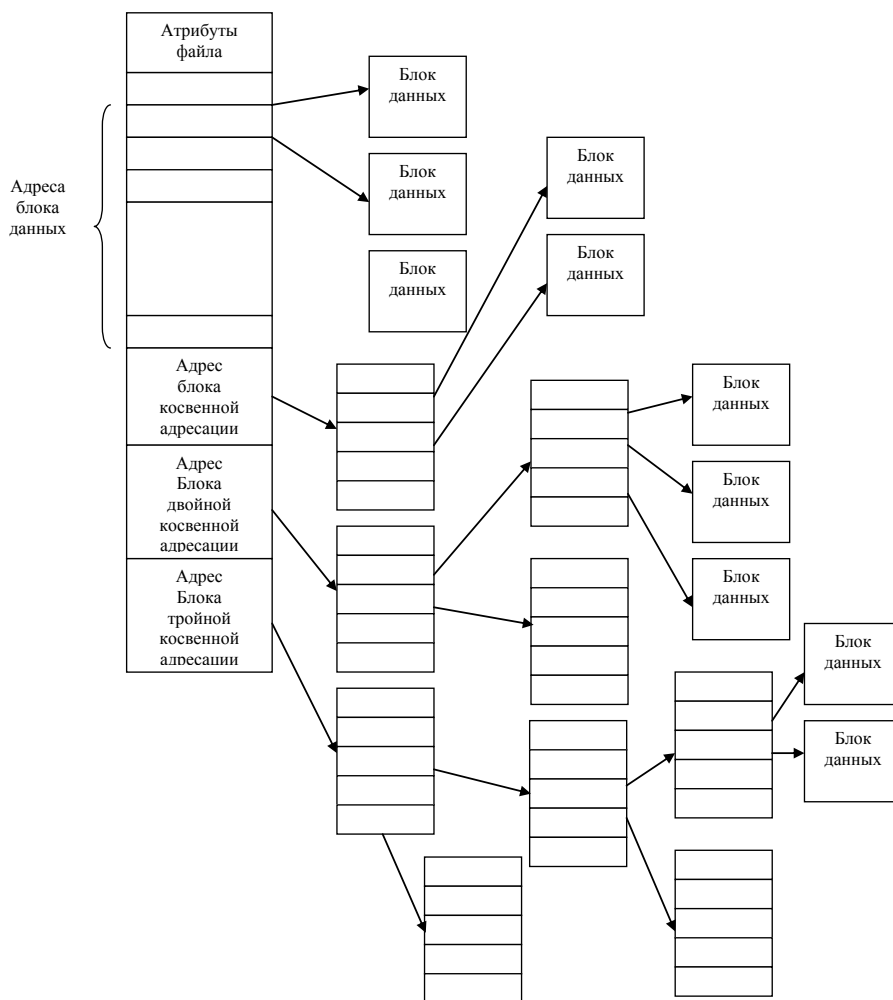


Рисунок 41 – Структура индексного узла

Первые несколько адресов блоков файла хранятся непосредственно в индексном узле, поэтому для маленьких файлов индексный узел хранит всю необходимую информацию, которая копируется с диска в память, в момент открытия файла. Для больших файлов один из адресов индексного узла указывает на блок *косвенной адресации*. Этот блок содержит адреса дополнительных блоков диска. Если этого недостаточно, используется блок *двойной косвенной адресации*, который содержит адреса блоков косвенной адресации. Если и этого недостаточно используют блок *тройной косвенной адресации*.

Эту схему распределения внешней памяти использует ОС *Unix*, а также файловые системы *HPFS*, *NTFS* и др. Такой подход позволяет при фиксированном, относительно небольшом размере индексного узла, поддерживать работу с файлами, размер которых может меняться от нескольких байт до нескольких гигабайт. Существенно то, что для маленьких файлов используется только прямая адресация, обеспечивающая высокую производительность.

5.2.5 Управление дисковым пространством

Одной из основных функций файловых систем является управление свободным и занятым пространством внешней памяти, включая учет используемого места на диске. Выделяют несколько способов такого учета. Рассмотрим наиболее распространенные из них.

Учет при помощи организации битового вектора. Часто список свободных блоков диска реализован в виде битового вектора (*bit map* или *bit vector*). Каждый блок представлен одним битом, принимающим значение 0 или 1, в зависимости от того занят он или свободен. Например, 00111100111100011000001... .

Главное преимущество этого подхода – относительная простота и эффективность при нахождении первого свободного блока или n последовательных блоков на диске. Многие компьютеры (например, *Intel* и *Motorola*) имеют специальные инструкции манипулирования битами, при помощи которых можно легко локализовать первый единичный бит в слове.

Описываемый способ учета свободных блоков используется в *Apple Macintosh*. К сожалению, он эффективен только если битовый вектор помещается в памяти целиком, что возможно только для относительно небольших дисков¹⁴.

Учет при помощи организации связного списка. Другой подход – связать в список все свободные блоки, поддерживая указатель на первый свободный блок в специальном месте диска, попутно кэшируя в

¹⁴ Например, диск размером 1.3 Гб с блоками по 512 байт нуждается в таблице размером 332Кбайт.

памяти эту информацию. Обычно необходим только первый свободный блок, но если это не так, то схема не будет эффективна, так как для трассирования списка нужно сделать достаточно много обращений к диску. Иногда прибегают к модификации подхода связного списка, организовав хранение адресов n свободных блоков в первом свободном блоке: первые $n - 1$ этих блоков используются для хранения данных, а последний – содержит адреса других n блоков, и т.д.

5.2.6 Размер логического блока

Размер логического блока является одним из ключевых параметров, влияющих на эффективность работы файловой системы в целом. В некоторых системах (например, *Unix*) он может быть задан при форматировании. Следует помнить, что небольшой размер блока приводит к тому, что каждый файл будет содержать много блоков. Чтение блока осуществляется с задержками на поиск и вращение, поэтому файл, содержащийся в многих блоках, будет считываться относительно медленно. Большие блоки обеспечивают более высокую скорость обмена с диском, но вследствие внутренней фрагментации (каждый файл занимает целое число блоков и в среднем половина последнего блока пропадает) снижается процент полезного дискового пространства.

Проведенные исследования показали, что большинство файлов имеет небольшой размер (в *Unix* приблизительно 85% файлов имеют размер менее 8 Кб и 48% – менее 1 Кб). На рис. 42 изображены две зависимости от размера блока: одна показывает убывающую степень утилизации (использования) диска, а вторая – возрастающую скорость обмена данными с диском. Зависимости имеют общую точку при размере блока 3 Кб. Обычный компромисс – выбор блока размером 512 б, 1 Кб, 2 Кб.

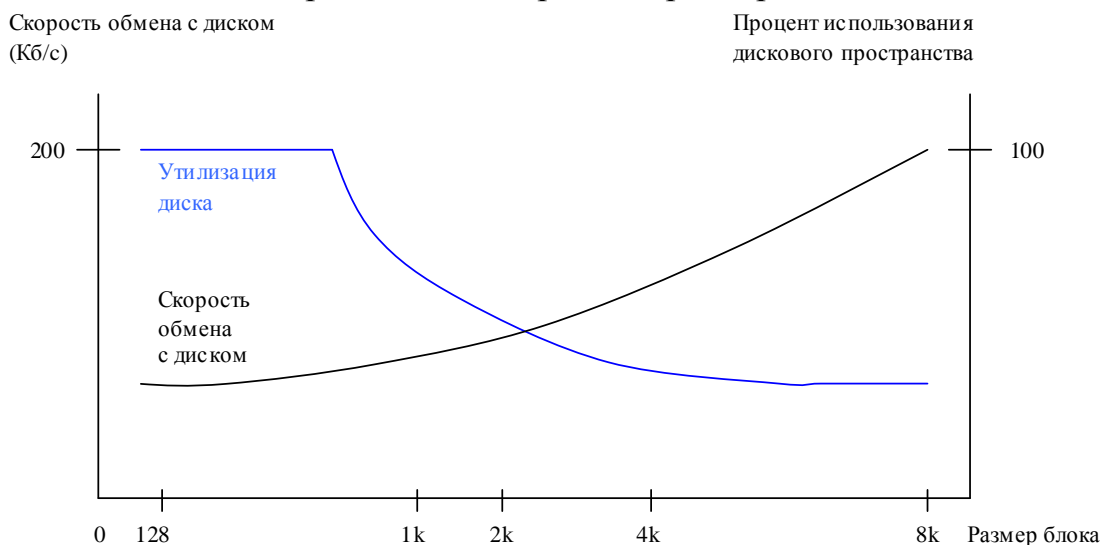


Рисунок 42 – Результаты исследований при определении оптимального размера блока

5.3 Особенности загрузки ОС

Рассмотрим подробнее процесс загрузки ОС как этап, предваряющий работу системы, сопряженный с взаимодействием с некоторым накопителем внешней памяти.

Процедура начальной загрузки (англ. *bootstrap loader*) вызывается как программное прерывание (*BIOS INT 19h*). Эта процедура определяет первое готовое устройство из списка разрешенных и доступных (гибкий или жесткий диск, а в современных компьютерах это могут быть еще и компакт-диск, привод *ZIP-drive*, сетевой адаптер или иное устройство) и пытается загрузить с него в ОП короткую главную *программу-загрузчик*. Для накопителей на жестких магнитных дисках – это известный главный или внесистемный загрузчик (*NSB*) из *MBR*, и ему передается управление.

Главный загрузчик определяет на диске активный раздел, загружает его собственный системный загрузчик и передает управление ему. Наконец, этот загрузчик находит и загружает необходимые файлы ОС и передает ей управление. Далее ОС выполняет инициализацию подведомственных ей программных и аппаратных средств – добавляет новые сервисы, вызываемые, как правило, тоже через механизм программных прерываний, и расширяет (или заменяет) некоторые сервисы *BIOS*.

Необходимо отметить, что в современных мультипрограммных ОС большинство сервисов *BIOS*, изначально расположенных в ПЗУ, как правило, заменяются собственными драйверами ОС, поскольку они должны работать в режиме прерываний, а не в режиме сканирования готовности.

Прежде чем форматировать диск под определенную файловую систему, он может быть поделен на *разделы* – непрерывные части физического диска, которую ОС представляет пользователю как *логические устройства* (*логические диски, логические разделы*)¹⁵. Логическое устройство функционирует так, как если бы это был отдельный физический диск. Именно с логическими устройствами работает пользователь, обращаясь к ним по символьным именам, используя, например, обозначения *A, B, C, SYS* и т.п. Операционные системы разного типа используют единое для всех них представление о разделах, но создают на его основе логические устройства, специфические для каждого типа ОС. Так же как файловая система, с которой работает одна ОС, в общем случае не может интерпретироваться ОС другого типа, логические устройства

¹⁵ Во многих ОС используется термин «том» (англ. *volume*), хотя толкование этого термина в разных ОС имеет свои нюансы.

не могут быть использованы ОС разного типа. На каждом логическом устройстве может создаваться только одна файловая система.

В частном случае, когда все дисковое пространство охватывается одним разделом, логическое устройство представляет физическое устройство в целом. Если диск разбит на несколько разделов, то для каждого из этих разделов может быть создано отдельное логическое устройство. Логическое устройство может быть создано и на базе нескольких разделов, причем эти разделы не обязательно должны принадлежать одному физическому устройству. Объединение нескольких разделов в единое логическое устройство может выполняться разными способами и преследовать разные цели, основные из которых: увеличение общего объема логического раздела, повышение производительности и отказоустойчивости. Примерами организации совместной работы нескольких дисковых разделов являются так называемые *RAID*¹⁶-массивы.

На разных логических устройствах одного и того же физического диска могут располагаться файловые системы разного типа. На рис. 43 показан пример диска, разбитого на три раздела, в которых установлены две файловых системы *NTFS* (разделы *C* и *E*) и одна файловая система *FAT* (раздел *D*).

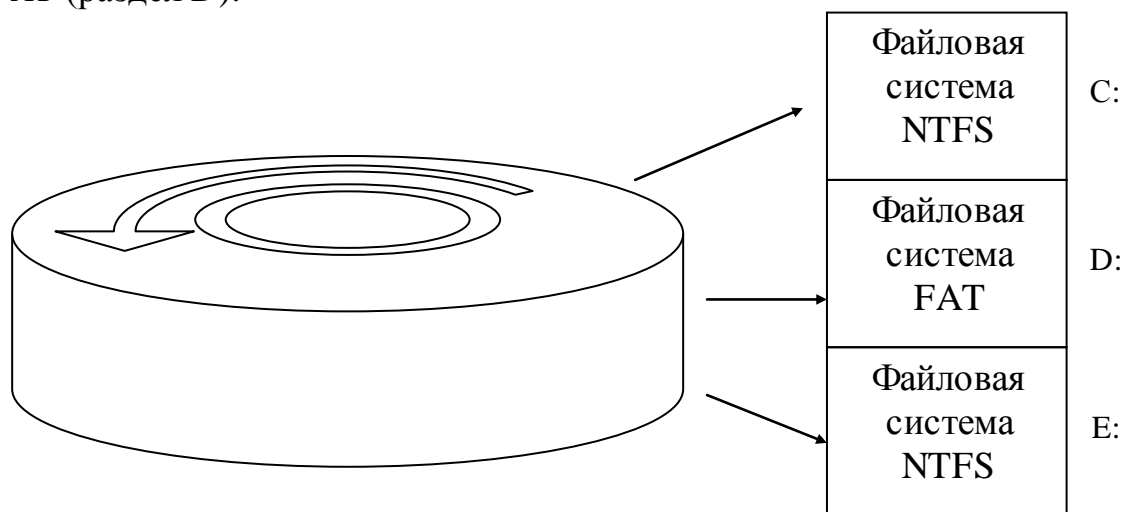


Рисунок 43 – Схематичное представление разбиения диска на разделы

Все разделы одного диска имеют одинаковый размер блока, определенный для данного диска в результате низкоуровневого форматирования. Однако в результате высокоуровневого форматирования в разных разделах одного и того же диска, представленных разными логическими устройствами, могут быть установлены файловые системы, в которых определены кластеры отличающихся размеров.

¹⁶ RAID (англ. *redundant array of independent/inexpensive disks*) – дисковый массив независимых дисков, предназначенный для повышения надёжности хранения данных и/или для повышения скорости чтения/записи информации.

Операционная система может поддерживать разные статусы разделов, особым образом отмечая разделы, которые могут быть использованы для загрузки модулей ОС, и разделы, в которых можно устанавливать только приложения и хранить файлы данных. Один из разделов диска помечается как загружаемый (или активный). Именно из этого раздела считывается загрузчик ОС.

5.4 Файлы и файловая система

Файловая система – это часть ОС, организующая работу с данными, хранящимися во внешней памяти, и обеспечивающей пользователю удобный интерфейс при работе с такими данными. Термин файловая система определяет, прежде всего, принципы доступа к данным, организованным в файлы. Говоря о файловых системах иногда употребляют термин *система управления файлами* – под которой следует понимать некую конкретную реализацию файловой системы, то есть комплекс программных модулей, обеспечивающих работу с файлами в конкретной ОС.

Непосредственное взаимодействие с диском при организации хранения информации на магнитном диске требует, например, знания устройства контроллера диска, особенностей работы с его регистрами. Очевидно, что такое взаимодействие – прерогатива системы ввода-вывода ОС (драйвера диска).

Для того чтобы избавить пользователя компьютера от сложностей взаимодействия с аппаратурой, была применена абстрактная (логическая) модель файловой системы, в которой операции записи или чтения файла концептуально проще, чем низкоуровневые операции работы с устройствами. Логическая модель файловой системы «материализуется» в виде дерева каталогов, выводимого на экран такими утилитами, как *Norton Commander* или *Windows Explorer*, в символьных составных именах файлов, в командах работы с файлами. Базовым элементом этой модели является *файл*, который так же, как и файловая система в целом, может характеризоваться как логической, так и физической структурой.

5.4.1 Цели и задачи файловой системы

Файл – это именованная область внешней памяти, в которую можно записывать и из которой можно считывать данные. Файлы хранятся в энергонезависимой памяти, обычно – на магнитных дисках. Основными целями использования файла являются:

- 1) Долговременное и надежное хранение информации. Долговременность достигается за счет использования запоминающих устройств, не зависящих от питания, а высокая надежность определяется средствами защиты доступа к файлам и общей организацией программного кода

ОС, при которой сбои аппаратуры чаще всего не разрушают информацию, хранящуюся в файлах.

2) Совместное использование информации. Файлы обеспечивают естественный и легкий способ разделения информации между приложениями и пользователями за счет наличия понятного человеку символического имени и постоянства хранимой информации и расположения файла. Пользователь должен иметь удобные средства работы с файлами, включая каталоги-справочники, объединяющие файлы в группы, средства поиска файлов по признакам, набор команд для создания, модификации и удаления файлов. Файл может быть создан одним пользователем, а затем использоваться другим пользователем, при этом создатель файла или администратор могут определить права доступа к нему других пользователей.

Файловая система, являющаяся неотъемлемой частью любой современной ОС, включает:

- совокупность всех файлов на диске;
- наборы структур данных, используемых для управления файлами (каталоги файлов, дескрипторы файлов, таблицы распределения свободного и занятого пространства на диске);
- комплекс системных программных средств, реализующих различные операции над файлами (создание, уничтожение, чтение, запись, именование и поиск файлов).

Файловая система позволяет программам обходиться набором относительно простых операций для выполнения действий над некоторым абстрактным объектом, представляющим файл. При этом программам не нужно иметь дело с деталями действительного расположения данных на диске, буферизацией данных и другими низкоуровневыми проблемами передачи данных с долговременного запоминающего устройства – все эти функции файловая система берет на себя. Файловая система распределяет дисковую память, поддерживает именование файлов, отображает имена файлов в соответствующие адреса во внешней памяти, обеспечивает доступ к данным, поддерживает разделение, защиту и восстановление файлов.

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

Задачи, решаемые файловой системой, зависят от способа организации вычислительного процесса в целом. Самый простой тип файловой системы реализуется в однопользовательских и однопрограммных ОС

(например, *MS-DOS*). Основные функции в такой файловой системе сведены к следующему перечню:

- именование файлов;
- программный интерфейс для приложений;
- отображение логической модели файловой системы на физическую организацию хранилища данных;
- устойчивость файловой системы к сбоям питания, ошибкам аппаратных и программных средств.

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

В многопользовательских ОС появляется еще одна задача – защита файлов одного пользователя от несанкционированного доступа другого пользователя. Таким образом, основными функциями файловой системы многопользовательской многозадачной ОС являются:

- *идентификация файлов* – связывание имени файла с выделенным ему пространством внешней памяти;
- *распределение внешней памяти между файлами* – для работы с конкретным файлом не требуется иметь информацию о местоположении этого файла на внешнем носителе информации (сторона магнитного диска, цилиндр, сектор);
- *обеспечение надежности и отказоустойчивости*;
- *обеспечение защиты от несанкционированного доступа*;
- *обеспечение совместного доступа к файлам* (пользователь не должен прилагать специальных усилий по обеспечению синхронизации доступа);
- *обеспечение высокой производительности*.

5.4.2 Типы файлов

Файловые системы поддерживают несколько функционально различных типов файлов, в число которых, как правило, входят:

- обычные файлы;
- файлы-каталоги;

- специальные файлы;
- отображаемые в память файлы;
- именованные конвейеры;
- другие.

Рассмотрим каждый из этих типов файлов.

Обычные файлы, или просто *файлы*, содержат информацию произвольного характера, которую заносит в них пользователь или которая образуется в результате работы системных и пользовательских программ. Большинство современных ОС (например, *Unix* или *MS Windows*) никак не ограничивает и не контролирует содержимое и структуру обычного файла. Содержание обычного файла определяется приложением, которое с ним работает. Обычные файлы бывают двух типов – *текстовые*¹⁷ и *двоичные*¹⁸. Обычно прикладные программы, работающие с файлами, распознают тип файла по его имени в соответствии с общепринятыми соглашениями. Например, файлы с расширениями *.c*, *.pas*, *.txt* – ASCII-файлы, файлы с расширениями *.exe* – исполняемые, файлы с расширениями *.obj*, *.zip* – бинарные и т.д. Все ОС должны уметь распознавать хотя бы один тип файлов – собственные исполняемые файлы.

Для пользователей файл обозначается с помощью идентификаторов – внешних имен (могут быть и внутренние имена файлов). Пользователи дают файлам символьные имена, при этом учитываются ограничения ОС как на используемые символы, так и на длину имени. До недавнего времени эти границы были весьма узкими. Так, в популярной файловой системе FAT длина имен ограничивается известной схемой 8.3 (8 символов – собственно имя, 3 символа – расширение имени), а в ОС *Unix System V* имя не может содержать более 14 символов. Однако пользователю гораздо удобнее работать с длинными именами, поскольку они позволяют дать файлу действительно мнемоническое название, по которому даже через достаточно большой промежуток времени можно будет вспомнить, что содержит этот файл. Поэтому современные файловые системы, как правило, поддерживают длинные символьные имена файлов. Например, файловая система *NTFS*, появившаяся в *Windows NT*, устанавливает, что имя файла может содержать до 255 символов, не считая завершающего нулевого символа.

Файлы-каталоги или просто *каталоги* – это особый тип файлов, которые содержат системную справочную информацию о наборе файлов, сгруппированных пользователями по какому-либо признаку

¹⁷ Текстовые файлы состоят из строк символов, представленных в ASCII-коде, их можно прочитать на экране и распечатать на принтере.

¹⁸ Двоичные файлы не используют ASCII-коды, часто имеют сложную внутреннюю структуру, например, объектный код программы или архивный файл.

(например, в одну группу объединяются файлы, содержащие документы одного договора, или файлы, составляющие один программный пакет), с другой стороны – это файл, содержащий системную информацию о группе файлов, его составляющих.

Во многих ОС в каталог могут входить файлы любых типов, в том числе другие каталоги, за счет чего может образовываться древовидная структура, удобная для поиска. Каталоги устанавливают соответствие между именами файлов и их характеристиками, используемыми файловой системой для управления файлами. В число таких характеристик входит, в частности, информация (или указатель на другую структуру, содержащую эти данные) о типе файла и расположении его на диске, правах доступа к файлу и датах его создания и модификации. Во всех остальных отношениях каталоги рассматриваются файловой системой как обычные файлы.

Специальные файлы – это фиктивные файлы, ассоциированные с устройствами ввода-вывода, которые используются для унификации механизма доступа к файлам и внешним устройствам. Специальные файлы позволяют пользователю выполнять операции ввода-вывода посредством обычных команд записи в файл или чтения из файла. Эти команды обрабатываются сначала программами файловой системы, а затем на некотором этапе выполнения запроса преобразуются ОС в команды управления соответствующим устройством.

Специальные файлы, так же как и устройства ввода-вывода, делятся на *блок-ориентированные*¹⁹ и *байт-ориентированные*²⁰. Следует помнить, что существуют некоторые внешние устройства, которые не относятся ни к одному из указанных классов, например, часы или таймеры, которые, с одной стороны, не адресуемы, а с другой стороны, не порождают потока байтов. Эти устройства только выдают сигналы прерывания в заданные моменты времени.

Отображаемые в память файлы (англ. *memory-mapped files*) – это мощная возможность ОС, позволяющая приложениям осуществлять доступ к файлам на диске тем же самым способом, каким осуществляется доступ к динамической памяти, то есть через указатели. Смысл отображения файла в память заключается в том, что содержимое файла (или часть содержимого) отображается в некоторый диапазон виртуального адресного пространства процесса, после чего обращение по какому-либо адресу из этого диапазона означает обращение к файлу на диске. Есте-

¹⁹ Используют механизм буфера, позволяющий увеличить эффективность передачи данных путем сохранения копии данных в блоках фиксированного размера в памяти (например, жесткий диск).

²⁰ Обеспечивают посимвольный небуферированный ввод/вывод, не адресуемы и не позволяют производить операцию поиска, генерируют или потребляют последовательность байтов (например, терминалы, строчные принтеры, сетевые адаптеры).

ственно, не каждое обращение к отображенному в память файлу вызывает операцию чтения/записи. Менеджер виртуальной памяти кэширует обращения к диску и тем самым обеспечивает высокую эффективность работы с отображенными файлами.

Именованные конвейеры (именованные каналы) – одно из средств межпроцессного взаимодействия, детали которого рассмотрены в п. 3.3.6.

5.4.3 Атрибуты файла

Кроме имени ОС часто связывают с каждым файлом и другую информацию, например дату модификации, размер и т.д. Эти характеристики файлов называют *атрибутами*. В разных файловых системах могут использоваться в качестве атрибутов разные характеристики. Например, такими характеристиками, могут быть следующие:

- информация о разрешенном доступе;
- пароль для доступа к файлу;
- владелец файла;
- создатель файла;
- признак «только для чтения»;
- признак «скрытый файл»;
- признак «системный файл»;
- признак «архивный файл»;
- признак «двоичный/символьный»;
- признак «временный» (удалить после завершения процесса);
- признак блокировки;
- длина записи;
- указатель на ключевое поле в записи;
- длина ключа;
- время создания, последнего доступа и последнего изменения;
- текущий размер файла;
- максимальный размер файла.

5.4.4 Доступ к файлам

С точки зрения внутренней структуры (логической организации) файл – это совокупность однотипных записей, каждая из которых информирует о свойствах одного объекта. Записи могут быть фиксированной длины, переменной длины или неопределенной длины. Записи переменной длины в своем составе содержат длину записи, а неопределенной длины – специальный символ конца записи. При этом каждая запись может иметь идентификатор, представляющий собой ключ, который может быть сложным и состоять из нескольких полей.

Существует несколько способов логической организации памяти:

- последовательный;
- индексно-последовательный;
- индексный;
- прямой;
- библиотечный.

Рассмотрим каждый из этих способов подробнее.

При *последовательном* способе организации памяти записи располагаются в физическом порядке и обеспечивают доступ в физической последовательности. Таким образом, для обработки записи с номером $N + 1$ необходимо последовательно обратиться к записям с номером $1, 2, \dots, N$. Это универсальный способ организации файла периферийного устройства, входного/выходного потока.

При *индексно-последовательном* способе записи располагаются в логической последовательности в соответствии со значением ключей записи, но физически записи располагаются в различных местах файла. Логическая последовательность файла фиксируется в специальной таблице индексов, в которой значение ключей связывается с физическим адресом записи. При такой организации доступ к записям осуществляется логически последовательно в порядке возрастания или убывания значения ключа или по значению ключа.

При *индексном* способе место записи в файле ее физический адрес определяется алгоритмом преобразования для ключа. Доступ к записям возможен только прямой, а алгоритм преобразования ключа называется хешированием. Ключ, использующий алгоритм хеширования, преобразуется в номер записи.

Прямой способ организации характеризуется тем, что в соответствии с ним осуществляется прямой доступ по порядковому номеру записи или по физическому адресу.

При библиотечном способе организации памяти файл состоит из последовательных подфайлов (разделов), первый из которых является оглавлением и содержит имена и адреса остальных подфайлов. При такой организации осуществляется комбинированный доступ – индексный прямой к разделу и последовательный в разделах.

В многопользовательских системах, независимо от рассмотренных выше способов логической организации памяти, существует проблема обеспечения раздельных прав доступа к ресурсам. В таких системах в качестве субъектов доступа могут выступать как отдельные пользователи, так и группы пользователей. Определение индивидуальных прав доступа для каждого пользователя позволяет максимально гибко задать политику расходования разделяемых ресурсов в вычислительной систе-

ме. Однако этот способ приводит в больших системах к чрезмерной загрузке администратора рутинной работой по повторению одних и тех же операций для пользователей с одинаковыми правами. Объединение таких пользователей в группу и задание прав доступа в целом для группы является одним из основных приемов администрирования в больших системах.

У каждого объекта доступа существует владелец. Владельцем может быть как отдельный пользователь, так и группа пользователей. Владелец объекта имеет право выполнять с ним любые допустимые для данного объекта операции. Во многих ОС существует особый пользователь (*superuser*, *root* или *administrator*), который имеет все права по отношению к любым объектам системы, не обязательно являясь их владельцем. Под таким именем работает администратор системы, которому необходим полный доступ ко всем файлам и устройствам для управления политикой доступа.

Различают два основных подхода к определению прав доступа:

1) Избирательный доступ (англ. *discretionary* – предоставленный на собственное усмотрение). Для каждого объекта сам владелец может определить допустимые операции с объектами. Между пользователями и группами пользователей в системах с избирательным доступом нет жестких иерархических взаимоотношений, то есть взаимоотношений, которые определены по умолчанию и которые нельзя изменить. Исключение делается только для администратора, по умолчанию наделяемого всеми правами.

2) Мандатный доступ (англ. *mandatory* – обязательный, принудительный). Система наделяет пользователя определенными правами по отношению к каждому разделяемому ресурсу (файлу) в зависимости от того, к какой группе пользователь отнесен. От имени системы выступает администратор, а владельцы объектов лишены возможности управлять доступом к ним по своему усмотрению. Все группы пользователей в такой системе образуют строгую иерархию, причем каждая группа пользуется всеми правами группы более низкого уровня иерархии, к которым добавляются права данного уровня. Членам какой-либо группы не разрешается предоставлять свои права членам групп более низких уровней иерархии.

Мандатные системы доступа считаются более надежными, но менее гибкими, обычно они применяются в специализированных вычислительных системах с повышенными требованиями к защите информации. В универсальных системах используются, как правило, избирательные методы доступа.

Несмотря на то, что здесь в основном рассматриваются механизмы контроля доступа к таким объектам, как файлы и каталоги, но следует

помнить, что эти же механизмы могут использоваться в современных ОС для контроля доступа к объектам любого типа – отличия заключаются лишь в наборе операций, характерных для того или иного класса объектов.

5.4.5 Операции над файлами

Определить *права доступа* к ресурсу – значит определить для каждого пользователя набор операций, которые ему разрешено применять к данному ресурсу. В разных ОС для одних и тех же типов ресурсов может быть определен свой список дифференцируемых операций доступа. Для файловых объектов этот список может включать следующие операции:

- *Создание файла*, не содержащего данных. Смысл данного вызова – объявить, что файл существует, и присвоить ему ряд атрибутов. При этом выделяется место для файла на диске и вносится запись в каталог.
- *Удаление файла* и освобождение занимаемого им дискового пространства.
- *Открытие файла*. Перед использованием файла процесс должен его открыть. Цель данного системного вызова – разрешить системе проанализировать атрибуты файла и проверить права доступа к нему, а также считать в ОП список адресов блоков файла для быстрого доступа к его данным.
- *Закрытие файла*. Если работа с файлом завершена, его атрибуты и адреса блоков на диске больше не нужны. В этом случае файл нужно закрыть, чтобы освободить место во внутренних таблицах файловой системы.
- *Позиционирование*. Возможность специфицировать текущую позицию в файле для считывания/записи данных.
- *Чтение данных из файла*. Обычно с текущей позиции. Пользователь должен задать объем считываемых данных и предоставить для них буфер в ОП.
- *Запись данных в файл с текущей позиции*. Если текущая позиция находится в конце файла, его размер увеличивается, в противном случае запись осуществляется на место имеющихся данных.

Наличие в системе многих пользователей предполагает организацию контролируемого доступа к файлам. Выполнение любой операции над файлом должно быть разрешено только в случае наличия у пользователя соответствующих привилегий. Обычно контролируются следующие операции: чтение, запись и выполнение. Другие операции, например копирование файлов или их переименование, также могут контро-

лироваться. Однако чаще эти операции реализуются через перечисленные. Так, операцию копирования файлов можно представить как операцию чтения и последующую операцию записи.

Наиболее общий подход к защите файлов от несанкционированного использования – сделать доступ зависящим от идентификатора пользователя, т.е. связать с каждым файлом или директорией *список прав доступа* (англ. *access control list*), где перечислены имена пользователей и типы разрешенных для них способов доступа к файлу.

Любой запрос на выполнение операции сверяется с таким списком. У такой техники есть два нежелательных следствия:

- конструирование подобного списка может оказаться сложной задачей, особенно если пользователи системы не известны заранее;
- запись в директории должна иметь переменный размер (включать список потенциальных пользователей).

Для решения этих проблем создают классификации пользователей, например, в ОС *Unix* все пользователи разделены на три группы:

- владелец (англ. *owner*);
- группа (англ. *group* – набор пользователей, разделяющих файл и нуждающихся в типовом способе доступа к нему);
- остальные (англ. *univers*).

В рамках такой ограниченной классификации задаются только три поля (по одному для каждой группы) для каждой контролируемой операции. В итоге в ОС *Unix* операции чтения, записи и исполнения контролируются при помощи 9 бит: *rw xrwxrwx*.

5.4.6 Иерархическая структура каталогов

Для решения проблем поиска и размещения файлов в системах управления файлами используются иерархические, многоуровневые каталоги файлов, двухуровневые имена файлов и средства фильтрации.

Простой одноуровневый каталог представляет собой оглавление тома (используется в однопользовательских ОС – рис. 44а). Иерархический, многоуровневый каталог (древовидный или сетевой) – это совокупность каталогов и дескрипторов файлов различной глубины (рис. 44б).

Каталоги образуют дерево, если файлу разрешено входить только в один каталог, и сеть – если файл может входить сразу в несколько каталогов. В ОС *MS-DOS* каталоги образуют древовидную структуру, а в ОС *Unix* – сетевую. Как и любой другой файл, каталог имеет символьное имя и однозначно идентифицируется составным именем, содержащим цепочку символьных имен всех каталогов, через которые проходит путь от корня до данного каталога. Каждый каталог группирует по определенным принципам файлы пользователей, которые могут быть защище-

ны паролем. Для упрощения работы имеется понятие текущего каталога, определяющего список доступных файлов и подкаталогов и позволяющего обращаться к файлам по собственным именам.

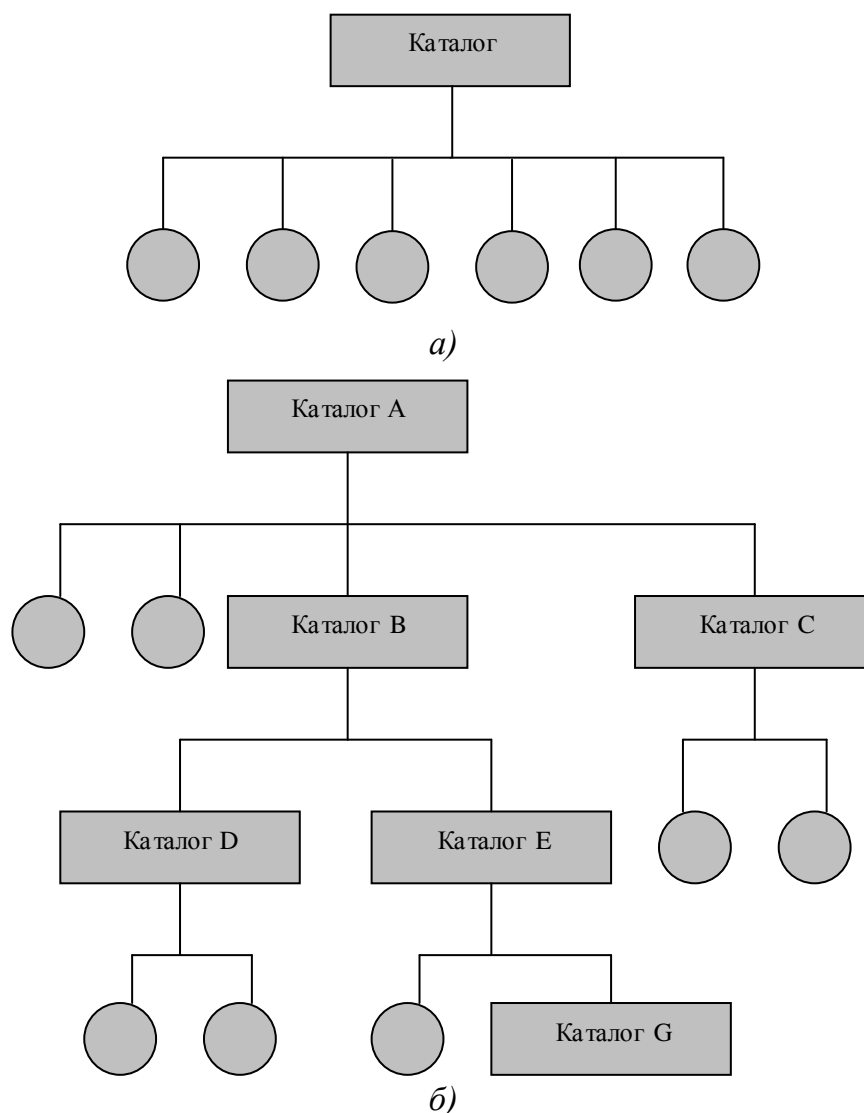


Рисунок 44 – Пример организации каталогов:

а) простой одноуровневый каталог; б) иерархический многоуровневый каталог

5.4.7 Операции над директориями

Как и в случае с файлами, система обязана обеспечить пользователя набором операций, необходимых для работы с директориями, реализованных через системные вызовы. Несмотря на то, что директории – это файлы, логика работы с ними несколько отличается от логики работы с обычными файлами и определяется природой этих объектов, предназначенных для поддержки структуры файлового архива. Рассмотрим в качестве примера некоторые системные вызовы, необходимые для работы с каталогами:

- *создание директории* – вновь созданная директория включает записи с именами «.» и «..», однако считается пустой.
- *удаление директории*;
- *открытие директории* для последующего чтения (например, чтобы перечислить файлы, входящие в директорию, процесс должен открыть директорию и считать имена всех файлов, которые она включает);
- *закрытие директории* после ее чтения для освобождения места во внутренних системных таблицах;
- *поиск* – данный системный вызов возвращает содержимое текущей записи в открытой директории;
- *получение списка файлов* в каталоге;
- *переименование* – имена директорий можно менять, как и имена файлов;
- *создание файла* – при создании нового файла необходимо добавить в каталог соответствующий элемент;
- *удаление файла* – удаление из каталога соответствующего элемента.

5.5 Особенности организации некоторых файловых систем

В предыдущих разделах рассмотрены общие концептуальные основы построения файловых систем и их логической и физической организации. Очевидно, что конкретные файловые системы могут обладать различными особенностями – иметь различные ограничения на имена файлов (количество и допустимые типы символов) и значения их расширения (позволяют определять тип файла или просто являются частью имени файла), поддерживать различный набор атрибутов и их предназначение и т.п.

Учитывая это, рассмотрим подробнее особенности функционирования конкретных файловых систем – как традиционных (например, *FAT 16* или *FAT 32*), так и новых, широко используемых в настоящее время (например, *NTFS* или *UFS*).

5.5.1 FAT

Файловая система *FAT* (от англ. *File Allocation Table*) была разработана *Биллом Гейтсом* (англ. *Bill Gates*) и *Марком МакДональдом* (англ. *Mark McDonald*) в 1977 году и первоначально использовалась в ОС *86-DOS*. Чтобы добиться переносимости программ из ОС *CP/M* в *86-DOS*, в ней были сохранены ранее принятые ограничения на имена файлов. В дальнейшем *86-DOS* была приобретена *Microsoft* и стала основой для

ОС *MS-DOS 1.0*, выпущенной в августе 1981 года. Файловая система *FAT* была предназначена для работы с гибкими дисками размером менее 1 Мбайт, и вначале не предусматривала поддержки жестких дисков. В настоящее время *FAT* поддерживает файлы и разделы размером до 2 Гбайт.

В *FAT* применяются следующие соглашения по именам файлов:

- имя должно начинаться с буквы или цифры и может содержать любой символ ASCII, за исключением пробела и символов « " \ [] : ; | = , ^ * ? »;
- длина имени не превышает 8 символов, за ним следует точка и необязательное расширение длиной до 3 символов;
- регистр символов в именах файлов не различается и не сохраняется.

Логический раздел, отформатированный под файловую систему *FAT*, имеет следующую структуру (на рис. 45).



Рисунок 45 – Структура раздела *FAT*

В блоке параметров *BIOS* содержится необходимая *BIOS* информация о физических характеристиках жесткого диска.

Таблица *FAT* (как основная копия, так и резервная) состоит из массива индексных указателей, количество которых равно количеству кластеров области данных (рис. 46).

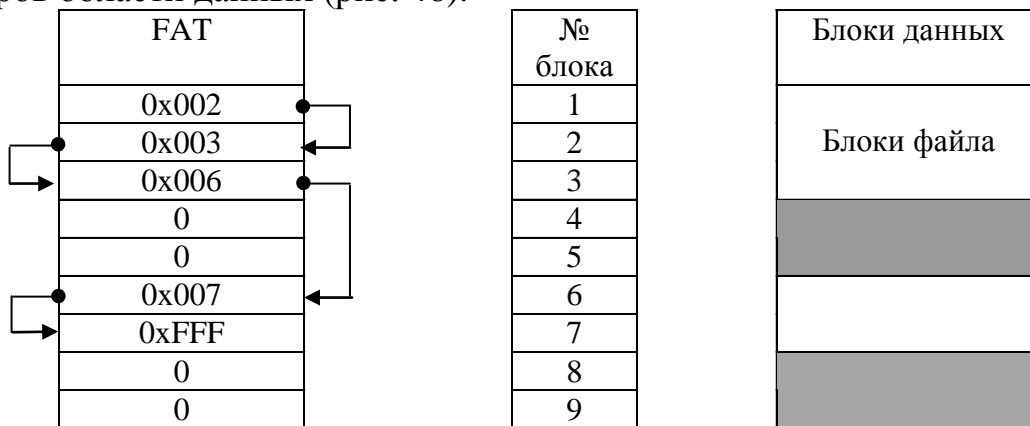


Рисунок 46 – Таблица *FAT* для связи данных индексных указателей и блоков данных файлов

Между кластерами и индексными указателями имеется взаимно однозначное соответствие – нулевой указатель соответствует нулевому кластеру и т.д. Индексный указатель может принимать следующие значения, характеризующие состояние связанного с ним кластера:

- кластер свободен (не используется);
- кластер используется файлом и не является последним кластером файла (в этом случае индексный указатель содержит номер следующего кластера файла);
- последний кластер файла;
- дефектный кластер;
- резервный кластер.

Файловая система *FAT* не может контролировать отдельно каждый сектор, поэтому она объединяет смежные секторы в *кластеры* (англ. *clusters*), тем самым уменьшая общее количество единиц хранения, за которыми должна следить файловая система. Размер кластера в *FAT* является степенью двойки и определяется размером тома при форматировании диска (табл. 2).

Таблица 2. Размеры разделов и кластеров в *FAT*

Размер раздела	Размер кластера	Тип <i>FAT</i>
< 16 Мб	4 Кб	FAT12
16 Мб – 127 Мб	2 Кб	FAT16
128 Мб – 255 Мб	4 Кб	FAT16
256 Мб – 511 Мб	8 Кб	FAT16
512 Мб – 1023 Мб	16 Кб	FAT16
1 Гб – 2 Гб	32 Кб	FAT16

Таблица *FAT* является общей для всех файлов раздела. В исходном состоянии (после форматирования) все кластеры раздела свободны и все индексные указатели (кроме тех, которые соответствуют резервным и дефектным блокам) принимают значение «свободен». При размещении файла ОС просматривает *FAT*, начиная с начала, и ищет первый свободный индексный указатель. После его обнаружения в поле записи каталога «номер первого кластера» фиксируется номер этого указателя. В кластер с этим номером записываются данные файла, он становится первым кластером файла. Если файл уместается в одном кластере, то в указатель, соответствующий данному кластеру, заносится специальное значение «последний кластер файла». Если же размер файла больше одного кластера, то ОС продолжает просмотр *FAT* и ищет следующий указатель на свободный кластер. После его обнаружения в предыдущий указатель заносится номер этого кластера, который теперь становится следующим кластером файла. Процесс повторяется до тех пор, пока не будут размещены все данные файла. Таким образом создается связный список всех кластеров файла. Из-за этого *FAT* называют файловой системой со связанными списками (пример представлен выше на рис. 39).

Оригинальная версия *FAT*, разработанная для *DOS 1.00*, использовала 12-битную таблицу размещения файлов и поддерживала разделы объемом до 16 Мб (в *DOS* можно создать не более двух разделов *FAT*). Для поддержки жестких дисков размером более 32 Мб разрядность *FAT* была увеличена до 16 бит (при этом старая система получила название *FAT 12*, а новая – *FAT 16*), а размер кластера – до 64 секторов (32 Кб). В связи с тем, что каждому кластеру может быть присвоен уникальный 16-разрядный номер, то *FAT* поддерживает максимально 2^{16} , или 65536 кластеров на одном томе.

Поскольку загрузочная запись слишком мала для хранения алгоритма поиска системных файлов на диске, то системные файлы должны находиться в определенном месте, чтобы загрузочная запись могла их найти. Фиксированное положение системных файлов в начале области данных накладывает жесткое ограничение на размеры корневого каталога и таблицы размещения файлов. Вследствие этого общее число файлов и подкаталогов в корневом каталоге на диске *FAT* ограничено 512.

Каждому файлу и подкаталогу в *FAT* соответствует 32-байтный элемент каталога (англ. *directory entry*), содержащий ряд параметров и атрибутов (табл. 3).

Таблица 3. Характеристики элемента каталога в *FAT*

Содержание	Размер (байт)
Имя файла	8
Расширение	3
Байт атрибутов	1
Зарезервировано	10
Время	2
Дата	2
Номер начального кластера с данными	2
Размер файла	4

Файловая система *FAT* всегда заполняет свободное место на диске последовательно от начала к концу. При создании нового файла или увеличении уже существующего она ищет самый первый свободный кластер в таблице размещения файлов. Если в процессе работы одни файлы были удалены, а другие изменились в размере, то появляющиеся в результате пустые кластеры будут рассеяны по диску. Если кластеры, содержащие данные файла, расположены не подряд, то файл оказывается фрагментированным. Сильно фрагментированные файлы значительно снижают эффективность работы, так как головки чтения/записи при поиске очередной записи файла должны будут перемещаться от одной области диска к другой. В состав ОС, поддерживающих *FAT*, обычно вхо-

дят специальные утилиты дефрагментации диска, предназначенные повысить производительность файловых операций.

Еще один недостаток *FAT* заключается в том, что ее производительность сильно зависит от количества файлов, хранящихся в одном каталоге. При большом количестве файлов (~1000), выполнение операции считывания списка файлов в каталоге может занять несколько минут. Это обусловлено тем, что в *FAT* каталог имеет линейную неупорядоченную структуру, и имена файлов в каталогах идут в порядке их создания. В результате, чем больше в каталоге записей, тем медленнее работают программы, так как при поиске файла требуется просмотреть последовательно все записи в каталоге.

Кроме того, следует отметить, что в *FAT* отсутствуют средства разграничения доступа, а также существует возможность потери информации о размещении всех файлов после разрушения таблицы *FAT* и ее копии.

Поскольку *FAT* изначально проектировалась для однопользовательской ОС *DOS*, то она не предусматривает хранения такой информации, как сведения о владельце или полномочия доступа к файлу/каталогу.

Система *FAT* является наиболее распространенной файловой системой и ее в той или иной степени поддерживают большинство современных ОС. Благодаря своей универсальности *FAT* может применяться на томах, с которыми работают разные ОС.

Хотя нет никаких препятствий для использования любой другой файловой системы при форматировании дисков, большинство ОС для совместимости используют *FAT*. Отчасти это можно объяснить тем, что простая структура *FAT* требует меньше места для хранения служебных данных, чем другие системы, преимущества которых заметны только при использовании их на носителях объемом более 100 Мб.

Очередное поколение жестких дисков характеризовалось большими объемами дискового пространства, в то время как возможности *FAT* уже достигли своего предела (*FAT* может поддерживать разделы размером до 2 Гб).

Ответом на подобное увеличение стала система *FAT 32* – усовершенствованная версия файловой системы *VFAT*, поддерживающая жесткие диски объемом до 2 терабайт. Впервые файловая система *FAT 32* была включена в состав ОС *Windows 95 OSR 2*. В *FAT 32* были расширены атрибуты файлов, позволяющие теперь хранить время и дату создания, модификации и последнего доступа к файлу или каталогу.

Из-за требования совместимости с ранее созданными программами структура *FAT 32* содержит минимальные изменения. Главные отличия от предыдущих версий *FAT* состоят в следующем. Блок начальной загрузки на разделах с *FAT 32* был увеличен до 2 секторов и включает в

себя резервную копию загрузочного сектора, что позволяет системе быть более устойчивой к возможным сбоям на диске. Объем, занимаемый таблицей размещения файлов, увеличился, поскольку теперь каждая запись в ней занимает 32 байта, и общее число кластеров на разделе *FAT 32* больше, чем на разделах *FAT*. Соответственно, выросло и количество зарезервированных секторов.

Необходимо отметить, что официально *Microsoft* не поддерживает разделы *FAT 32* объемом менее 512 Мб. Однако в версии утилиты *FDISK*, поставляемой вместе с *OSR2*, был представлен недокументированный флаг */FPRMT*, позволяющий отформатировать под *FAT 32* разделы объемом менее 512 Мб. *Microsoft* также не поддерживает *FAT 32* разделы с размером кластера меньше 4 Кб. Размеры кластера, предлагаемые по умолчанию при форматировании *FAT 32* дисков, приведены в табл. 4. Параметр */Z* утилиты *FORMAT* позволяет самостоятельно установить размер кластера на разделе *FAT 32*:

FORMAT <диск> */Z:n*,

где *n* – число секторов в кластере.

Таблица 4. Структура размеров раздела и кластера в *FAT32*

Размер раздела	Размер кластера
< 260 Мб	512 байт
260 Мб – 8 Гб	4 Кб
8 Гб – 16 Гб	8 Кб
16 Гб – 32 Гб	16 Кб
> 32 Гб	32 Кб

Корневой каталог в *FAT 32* больше не располагается в определенном месте, вместо этого в блоке *BPB* хранится указатель на начальный кластер корневого каталога. В результате снимается ранее существовавшее ограничение на число записей в корневом каталоге.

Кроме того, для учета свободных кластеров, в зарезервированной области на разделе *FAT 32* имеется сектор, содержащий число свободных кластеров и номер самого последнего использованного кластера. Это позволяет системе при выделении следующего кластера не перечитывать заново всю таблицу размещения файла.

Распространение файловая система *FAT 32* получила в следующих ОС: *Windows 95 OSR2*, *Windows 98* и *Windows NT 5.0*.

5.5.2 VFAT

Файловая система *Virtual FAT (VFAT)*, реализованная в *Windows NT 3.5*, *Windows 95 (DOS 7.0)* – это файловая система *FAT*, включающая поддержку длинных имен файлов (*Long File Name, LFN*) в кодировке *UNICODE* (каждый символ имени кодируется 2 байтами). Си-

стема *VFAT* использует ту же самую схему распределения дискового пространства, что и файловая система *FAT*, поэтому размер кластера определяется величиной раздела. В *VFAT* ослаблены ограничения, устанавливаемые соглашениями по именам файлов *FAT*:

- имя может быть длиной до 255 символов;
- в имя можно включать несколько пробелов и точек, однако, текст после последней точки рассматривается как расширение;
- регистр символов в именах не различается, но сохраняется.

Основной задачей при разработке *VFAT* была необходимость корректной работы старых программ, не поддерживающих длинные имена файлов. Как правило, прикладные программы для доступа к файлам используют функции ОС. Если у элемента каталога установить комбинацию битов атрибутов «только для чтения», «скрытый», «системный», «метка тома» – то любые файловые функции старых версий ОС *DOS* и *Windows* не заметят такого элемента каталога. Поэтому для каждого файла и подкаталога в *VFAT* хранится два имени: длинное и короткое в формате 8.3 для совместимости со старыми приложениями. Длинные имена хранятся в специальных записях каталога, байт атрибутов, у которых равен *0Fh*. Для любого файла или подкаталога непосредственно перед единственной записью каталога с его именем в формате 8.3 находится группа из одной или нескольких записей, представляющих длинное имя. Каждая такая запись содержит часть длинного имени файла не более 13 символов, из всех таких записей ОС составляет полное имя файла. Поскольку одно длинное имя файла может занимать до 21 записи, а корневой каталог *FAT* ограничен 512 записями, желательно ограничить использование длинных имен в корневом каталоге. Структура элемента каталога для длинного имени файла представлена в табл. 5.

Таблица 5. Пример элемента каталога для длинного имени в *VFAT*

Содержание	Размер (байт)
Порядок следования	1
Первые пять символов <i>LFN</i>	10
Байт атрибутов (<i>0Fh</i>)	1
Указатель типа (всегда 0)	1
Контрольная сумма части имени	1
Следующие шесть символов <i>LFN</i>	12
Номер начального кластера (всегда 0)	2
Следующие два символа <i>LFN</i>	4

Короткое имя генерируется файловой системой автоматически в формате 8.3. Для создания коротких имен (псевдонимов) файлов используется следующий алгоритм:

1) Из длинного имени удалить все не допустимые в именах *FAT* символы. Удалить точки в конце и начале имени. Затем удалить все точки, находящиеся внутри имени, кроме последней.

2) Обрезать строку, расположенную перед точкой, до 6 символов и добавить в ее конец «~1». Обрезать строку за точкой до 3 символов.

3) Полученные буквы преобразовать в прописные. Если сгенерированное имя совпадает с уже существующим, то увеличить число в строке «~1».

Данный алгоритм зависит от версии ОС и в будущих версиях может быть модифицирован.

Редактирование файлов программами, не поддерживающими длинные имена файлов, может приводить к потере длинных имен. Система *Windows* обнаруживает подобные элементы каталога, так как их контрольная сумма не соответствует больше тому, что записано в последующей записи каталога в формате 8.3. Однако такие записи не удаляются системой автоматически, они занимают дисковое пространство, до тех пор, пока не запущена программа *ScanDisk*, входящую в состав ОС. Следует помнить, что большинство старых дисковых утилит воспримут записи, соответствующие длинным именам, как ошибки логической структуры диска. Попытки использовать данные утилиты в лучшем случае приведут к потере длинных имен, а в худшем – к потере информации на диске.

5.5.3 NTFS

Файловая система *NTFS* (*New Technology File System*) – наиболее предпочтительная файловая система при работе с ОС *Windows NT*, поскольку она была специально для нее разработана.

Диск *NTFS* условно делится на две части. Первые 12% диска отводятся под так называемую *MFT* зону (*Master File Table* или *главная файловая таблица*) – пространство, в котором происходит рост метафайла *MFT*.

Заголовок	Стандартная информация	Имя файла или каталога	Дескриптор безопасности	Данные или указатели на них	...
-----------	------------------------	------------------------	-------------------------	-----------------------------	-----

Рисунок 47 – Структура таблицы *MFT* в файловой системе *NTFS*

Запись каких-либо данных в эту область невозможна. *MFT*-зона всегда держится пустой – это делается для того, чтобы самый главный, служебный файл не фрагментировался при своем росте. Остальные 88% диска представляют собой обычное пространство для хранения файлов, причем свободное место диска включает в себя всё физически свободное место, включая незаполненные фрагменты *MFT*-зоны.

В случае, если свободное пространство для записи файлов отсутствует, *MFT*-зона просто сокращается (как правило, в два раза), освобождая таким образом место для записи файлов. При освобождении места в обычной области *MFT*-зона может снова расшириться. При этом не исключена ситуация, когда в этой зоне остались и обычные файлы. В этом случае метафайл *MFT* все-таки будет фрагментирован.

Каждый файл на томе *NTFS* представлен записью в *MFT*. Система *NTFS* резервирует первые 16 записей таблицы размером около 1 Мб для специальной информации. Первая запись таблицы описывает непосредственно саму главную файловую таблицу. За ней следует зеркальная запись *MFT*. Если первая запись *MFT* разрушена, *NTFS* считывает вторую запись, чтобы отыскать зеркальный файл *MFT*, первая запись которого идентична первой записи *MFT*. Местоположение сегментов данных *MFT* и зеркального файла *MFT* хранится в секторе начальной загрузки. Копия сектора начальной загрузки находится в логическом центре диска. Третья запись *MFT* содержит файл регистрации, применяемый для восстановления файлов. Семнадцатая и последующие записи главной файловой таблицы используются собственно файлами и каталогами на томе.

В *NTFS* значительно расширены возможности по управлению доступом к отдельным файлам и каталогам, введено большое число атрибутов, реализована отказоустойчивость, средства динамического сжатия файлов, поддержка требований стандарта *POSIX*²¹. Система *NTFS* позволяет использовать имена файлов длиной до 255 символов, при этом она использует тот же алгоритм для генерации короткого имени, что и *VFAT* (п. 5.5.2).

В случае сбоя ОС или оборудования *NTFS* обладает возможностью самостоятельного восстановления так, что дисковый том остается доступным, а структура каталогов не нарушается. Эта возможность реализована путем использования журнала транзакций (содержащегося в специальном файле – *log file*), в котором регистрируются все операции, влияющие на структуру тома, включая создание файла и любые команды, изменяющие структуру каталогов. Каждая операция ввода-вывода, изменяющая файл на томе *NTFS*, рассматривается системой как транзакция и может выполняться как неделимый блок. При модификации файла пользователем сервис файла регистрации фиксирует всю информацию необходимую для повторения или отката транзакции. Если транзакция завершена успешно, производится модификация файла, а если нет, *NTFS* производит откат транзакции.

²¹ POSIX® (Portable Operating System Interface for Unix – интерфейс переносимой операционной системы *Unix*) — набор стандартов, описывающих интерфейсы между ОС и прикладной программой. Стандарт создан для обеспечения совместимости различных *Unix*-подобных ОС и переносимости прикладных программ на уровне исходного кода.

Схема распределения пространства на томе хранится в файле битовой карты (англ. *bitmap file*). Атрибут данных этого файла содержит битовую карту, каждый бит которой представляет один кластер тома и указывает, свободен ли данный кластер или занят некоторым файлом. В загрузочном файле (англ. *boot file*) хранится код начального загрузчика *Windows NT*. Кроме того, существует поддержка файлов «плохих» кластеров (англ. *bad cluster file*) для регистрации поврежденных участков на томе и файл тома (англ. *volume file*), содержащий имя тома, версию *NTFS* и бит, который устанавливается при повреждении тома. Наконец, имеется файл, содержащий таблицу определения атрибутов (англ. *attribute definition table*), которая задает типы атрибутов, поддерживаемые на томе, и указывает можно ли их индексировать, восстанавливать операцией восстановления системы и т.д.

Система *NTFS* распределяет пространство кластерами и использует для их нумерации 64 разряда, что дает возможность иметь 2^{64} кластеров, каждый размером до 64 Кбайт. Как и в *FAT* размер кластера может меняться, но необязательно возрастает пропорционально размеру диска (табл. 6).

Таблица 6. Размеры разделов и кластеров, устанавливаемые при форматировании раздела

Размер раздела	Размер кластера
< 512 Мб	512 байт
513 Мб - 1024 Мб (1 Гб)	1 Кб
1 Гб - 2 Гб	2 Кб
2 Гб - 4 Гб	4 Кб
8 Гб - 16 Гб	16 Кб
16 Гб - 32 Гб	32 Кб
> 32 Гб	64 Кб

Система *NTFS* позволяет хранить файлы размером до 16 эксабайт (2^{64} байт) и располагает встроенным средством уплотнения файлов в реальном времени. Сжатие является одним из атрибутов файла или каталога и подобно любому атрибуту может быть снято или установлено в любой момент (сжатие возможно на разделах с размером кластера не более 4 Кб). При уплотнении файла, в отличие от схем уплотнения, используемых в *FAT*, применяется пофайловое уплотнение. Это позволяет избежать при порче небольшого участка диска, в котором расположен некоторый файл, потери информации в прочих файлах.

Для уменьшения фрагментации *NTFS* всегда пытается сохранить файлы в непрерывных блоках. Эта система использует структуру каталогов в виде *B*-дерева, аналогичную высокопроизводительной файловой

системе *HPFS* (п. 5.5.4), а не структуре со связанным списком, применяемой в *FAT*. Благодаря этому поиск файлов в каталоге осуществляется быстрее, поскольку имена файлов хранятся отсортированными в лексикографическом порядке.

Несмотря на наличие защиты от несанкционированного доступа к данным *NTFS* не обеспечивает необходимую конфиденциальность хранимой информации. Для получения доступа к файлам достаточно загрузить компьютер в *DOS* с дискеты и воспользоваться каким-нибудь сторонним драйвером *NTFS* для этой системы.

Начиная с версии *Windows NT 5.0* (новое название *Windows 2000*) *Microsoft* поддерживает новую файловую систему *NTFS 5.0*. В новой версии *NTFS* были введены дополнительные атрибуты файлов, наряду с правом доступа введено понятие запрета доступа, позволяющее, например, при наследовании пользователем прав группы на какой-нибудь файл, запретить ему возможность изменять его содержимое. Новая система также позволяет:

- вводить ограничения (квоты) на размер дискового пространства, предоставленного пользователям;
- проецировать любой каталог (как на локальном, так и на удаленном компьютере) в подкаталог на локальном диске.

Интересной возможностью новой версии файловой системы является динамическое шифрование файлов и каталогов, повышающее надежность хранения информации. В состав *Windows NT 5.0* входит файловая система с шифрованием (*Encrypting File System, EFS*), использующая алгоритмы шифрования с общим ключом. Если для файла установлен атрибут шифрования, то при обращении пользовательской программы к файлу для записи или чтения происходит прозрачное для программы кодирование и декодирование файла.

5.5.4 HPFS

Высокопроизводительная файловая система (англ. *High Performance File System – HPFS*) была представлена фирмой *IBM* в 1989 году вместе с ОС *OS/2 1.20*. Файловая система *HPFS* также поддерживалась ОС *Windows NT* до версии 3.51 включительно. По производительности эта система существенно опережает *FAT*. Система *HPFS* позволяет использовать жесткие диски объемом до 2 Терабайт (первоначально до 4 Гбайт). Кроме того, она поддерживает разделы диска размером до 512 Гб и позволяет использовать имена файлов длиной до 255 символов (на каждый символ отводится 2 байта). В *HPFS* по сравнению с *FAT* уменьшено время доступа к файлам в больших каталогах.

Файловая система *HPFS* распределяет пространство на диске не кластерами как в *FAT*, а физическими секторами по 512 байт, что не

позволяет ее использовать на жестких дисках, имеющих другой размер сектора (блока). Чтобы уменьшить фрагментацию диска, при распределении пространства под файл система *HPFS* стремится, по возможности, размещать файлы в последовательных смежных секторах. Фрагмент файла, располагающийся в смежных секторах, называется *экстендом*.

Для нумерации единиц распределения пространства диска *HPFS* использует 32 разряда, что дает 2^{32} , или более 4 миллиардов номеров. Однако *HPFS* использует числа со знаком, что сокращает число возможных номеров блоков до 2 миллиардов. Помимо стандартных атрибутов файла, *HPFS* поддерживает *расширенные атрибуты файла* (англ. *Extended Attributes – EA*), которые могут содержать до 64 Кб различных дополнительных сведений о файле.

Диск *HPFS* имеет следующие три базовые структуры (рис. 48): *загрузочный блок* (англ. *BootBlock*), *дополнительный блок* (англ. *SuperBlock*) и *резервный блок* (англ. *SpareBlock*).

Загрузочный блок	Дополнительный блок	Резервный блок	Группа 1	Битовая карта группы 1	Битовая карта группы 2	Группа 2	Группа 3	Битовая карта группы 3	Битовая карта группы 4	Группа 4
------------------	---------------------	----------------	----------	------------------------	------------------------	----------	----------	------------------------	------------------------	----------

Рисунок 48 – Структура дискового раздела *HPFS*

Загрузочный блок в *HPFS* аналогичен загрузочному блоку в *FAT*. Он располагается в секторах с 0 по 15 и занимает на диске 8 Кб. Системные файлы, также как и в *FAT*, располагаются в корневом каталоге, но при этом физически могут находиться в любом месте на диске.

Дополнительный блок размещается в 16 секторе и содержит указатель на *список блоков битовых карт* (англ. *bitmap block list*). В этом списке перечислены все блоки на диске, в которых расположены битовые карты, используемые для обнаружения свободных секторов. Также в дополнительном блоке хранится указатель на *список дефектных блоков* (англ. *bad block list*), указатель на группу каталогов (англ. *directory band*), указатель на файловый узел корневого каталога и дата последней проверки диска. Файловый узел (англ. *fnode*) – это структура диска *HPFS*, которая содержит информацию о расположении файла и о его расширенных атрибутах.

Резервный блок размещен в следующем секторе и содержит *карту аварийного замещения* (англ. *hotfix map*), указатель на *список свободных запасных блоков* (англ. *directory emergency free block list*) и ряд системных флагов. Резервный блок обеспечивает высокую отказоустойчивость *HPFS* и позволяет восстанавливать поврежденные данные на диске.

Остальное пространство диска разделено на *группы* (англ. *band*) хранения данных. Каждая группа занимает 8 Мб и имеет свою собственную битовую карту свободного пространства, которая похожа на таблицу размещения файлов *FAT*. Каждому сектору группы соответствует один бит к ее битовой карте, показывающий занят ли соответствующий сектор. Битовые карты двух групп располагаются на диске рядом, также как располагаются и сами группы. Это дает возможность непрерывно разместить на жестком диске файл размером до 16 Мб.

Одна из групп данных размером 8 Мб, расположенная в середине жесткого диска и называемая группой каталогов, хранит информацию о каталогах диска. В ней наряду с остальными каталогами располагается и корневой каталог. Расположение группы каталогов в центре диска значительно сокращает время позиционирования головок чтения/записи.

В отличие от линейной структуры *FAT*, структура каталога в *HPFS* представляет собой сбалансированное дерево (так называемое *B-дерево*) с записями, расположенными в алфавитном порядке. Сбалансированное дерево состоит из *корневого* (англ. *root block*) и *оконечных блоков* (англ. *leaf block*). Блоки занимают 4 последовательных сектора и в среднем могут содержать 40 записей. Каждая запись корневого блока указывает на один из оконечных блоков (если только в каталоге не меньше 40 файлов). В свою очередь, каждая запись в оконечном блоке указывает на файловый узел файла или на оконечный блок следующего уровня. Таким образом, двухуровневая структура может содержать 40 оконечных блоков по 40 записей в каждом и описывать до 1600 файлов. При поиске файловая система *HPFS* просматривает только необходимые ветви дерева.

Файловый узел имеет размер 512 байт и всегда по возможности располагается непосредственно перед первым блоком своего файла. Каждый файл и каталог диска *HPFS* имеет свой файловый узел. Информация, хранящаяся в файловом узле, включает в себя расширенные атрибуты файла, если они достаточно малы, чтобы поместится в один сектор диска, и сокращенное имя файла в формате 8.3. Если расширенные атрибуты не помещаются в файловый узел, то в него записывается указатель на атрибуты.

Положение файла на диске описывается в файловом узле двумя 32-битными числами. Первое из чисел представляет собой указатель на первый блок файла, а второе – длину экстенда. Если файл фрагментирован, то его размещение описывается дополнительными парами 32-битных чисел. В файловом узле можно хранить информацию максимум о 8 экстендах файла. Если файл имеет большее число экстендов, то в его файловый узел записывается указатель на *блок размещения* (англ.

allocation block), который может содержать до 40 указателей на экстен-
ты или на другие блоки размещения.

5.5.5 UFS

Рассмотрим особенности организации файловой системы для ОС *Unix* – *Unix file system (UFS)*.

Файл в ОС *Unix* представляет собой множество символов с произ-
вольным доступом. В файле могут содержаться любые данные, поме-
щенные туда пользователем, и файл не имеет никакой иной структуры,
кроме той, какую создаст в нем пользователь.

Информация на дисках размещается блоками. В первой версии
файловой системы размер блока – 512 байт. Во многих современных
файловых системах, разработанных для конкретной версии *Unix*-
подобных систем, размер блока больше. Это позволяет повысить быст-
родействие файловых операций. Например, в системе *FFS (Fast File Sys-
tem* – быстродействующая файловая система) размер блока равен 8192
байт.

Раздел диска файловой системы в ОС *Unix* разбивается на следую-
щие области (рис. 49):

- неиспользуемый блок;
- управляющий блок, или суперблок, в котором хранится размер
логического диска и границы других областей;
- *i*-список, состоящий из описаний файлов, называемых *i*-узлами;
- область для хранения содержимого файлов.



Рисунок 49 – Принцип организация файловой системы в ОС *Unix*

Каждый *i*-узел содержит:

- идентификатор владельца;

- идентификатор группы владельца;
- биты защиты;
- физические адреса на диске или ленте, где находится содержимое файла;
- размер файла;
- время создания файла;
- время последнего изменения (*modification time*) файла;
- время последнего изменения атрибутов (*change time*) файла;
- число связей-ссылок, указывающих на файл;
- индикатор типа файла (каталог, обычный файл или специальный файл).

Следом за *i*-списком идут блоки, предназначенные для хранения содержимого файлов. Пространство на диске, оставшееся свободным от файлов, образует связанный список свободных блоков.

Таким образом, *UFS* представляет собой структуру данных, размещенную на диске и содержащую управляющий суперблок с описанием файловой системы в целом, массив *i*-узлов, в котором определены все файлы в файловой системе, сами файлы и, наконец, совокупность свободных блоков. Выделение пространства под данные осуществляется блоками фиксированного размера.

Каждый файл однозначно идентифицируется *старшим номером устройства*, *младшим номером устройства* и *i-номером* (индексом *i*-узла данного файла в массиве *i*-узлов). Когда вызывается драйвер устройства, по старшему номеру индексируется массив входных точек в драйверы. По младшему номеру драйвер выбирает одно устройство из группы идентичных физических устройств.

Файл-каталог, в котором перечислены имена файлов, позволяет установить соответствие между именами и самими файлами. Каталоги образуют древовидную структуру. На каждый обычный файл или файл устройства могут иметься ссылки в различных узлах этой структуры. В непривилегированных программах запись в каталог не разрешена, но при наличии соответствующих разрешений они могут читать их. Дополнительных связей между каталогами нет.

Большое число системных каталогов *Unix* использует для собственных нужд. Один из них, корневой каталог, является базой для всей структуры каталогов, и, «отталкиваясь» от него, можно найти все файлы. В других системных каталогах содержатся программы и команды, предоставляемые пользователям, а также файлы устройств.

Имена файлов задаются последовательностью имен каталогов, разделенных косой чертой (/) и приводящих к конечному узлу (листу) некоторого дерева. Если имя файла начинается с косой черты, то поиск по

дереву начинается в корневом каталоге. Если же имя файла не имеет в начале косой черты, то поиск начинается с текущего каталога. Имена файлов, начинающиеся с группы символов «../» (две точки и косая черта), подразумевают начало поиска в каталоге, родительском по отношению к текущему.

Файл, не являющийся каталогом, может встречаться в различных каталогах, возможно, под разными именами. Это называется *связыванием*. Элемент в каталоге, относящийся к одному файлу, называется *связью*. В *Unix*-системах все такие связи имеют равный статус. Файлы не принадлежат каталогам. Скорее, файлы существуют независимо от элементов каталогов, а связи в каталогах указывают на реальные (физические) файлы. Файл «исчезает», когда удаляется последняя связь, указывающая на него. Биты защиты, заданные в связях, могут отличаться от битов в исходном файле. Так решается проблема избирательного ограничения на доступ к файлам.

С каждым поддерживаемым системой устройством ассоциируется один или большее число специальных файлов. Операции ввода-вывода для специальных файлов осуществляются так же, как и для обычных дисковых файлов с той лишь разницей, что эти операции активизируют соответствующие устройства. Специальные файлы обычно находятся в каталоге */dev*. На специальные файлы могут указывать связи точно так же, как на обычные файлы.

Монтирование. В случае необходимости объединения файловых систем, находящихся на разных устройствах (например, как в случае, представленном на рис. 43), в единую файловую систему и описания единым деревом каталогов, ОС *Unix* применяет операцию *монтирования*.

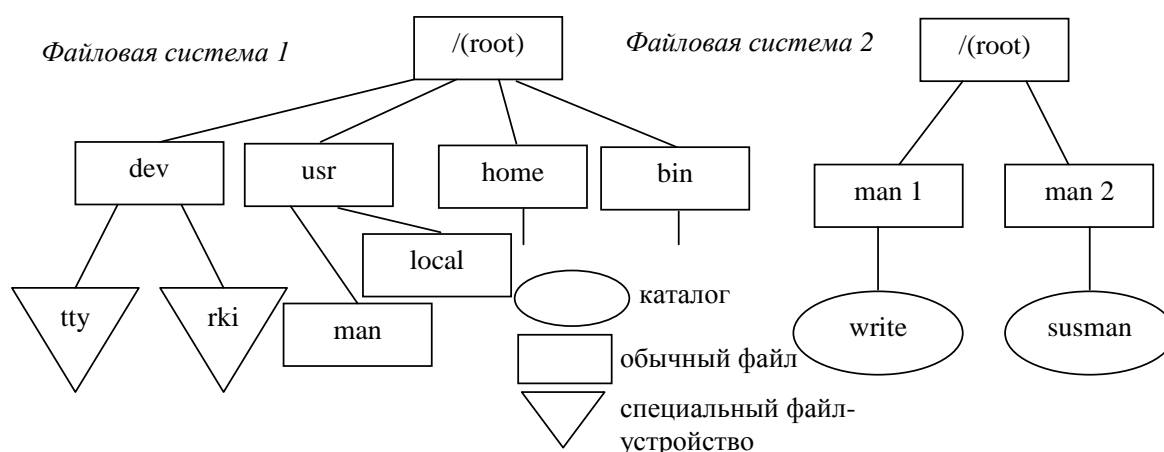


Рисунок 50 – Две файловые системы до монтирования

Среди всех имеющихся в системе логических дисковых устройств ОС выделяет одно устройство, называемое *системным*. Пусть имеются

две файловые системы, расположенные на разных логических дисках (рис. 50), причем один из дисков является системным.

Файловая система, расположенная на системном диске, назначается корневой. Для связи иерархий файлов в корневой файловой системе выбирается некоторый существующий каталог, в данном примере – каталог *man*. После выполнения монтирования выбранный каталог *man* становится корневым каталогом второй файловой системы. Через этот каталог монтируемая файловая система подсоединяется как поддерево к общему дереву (рис. 51).

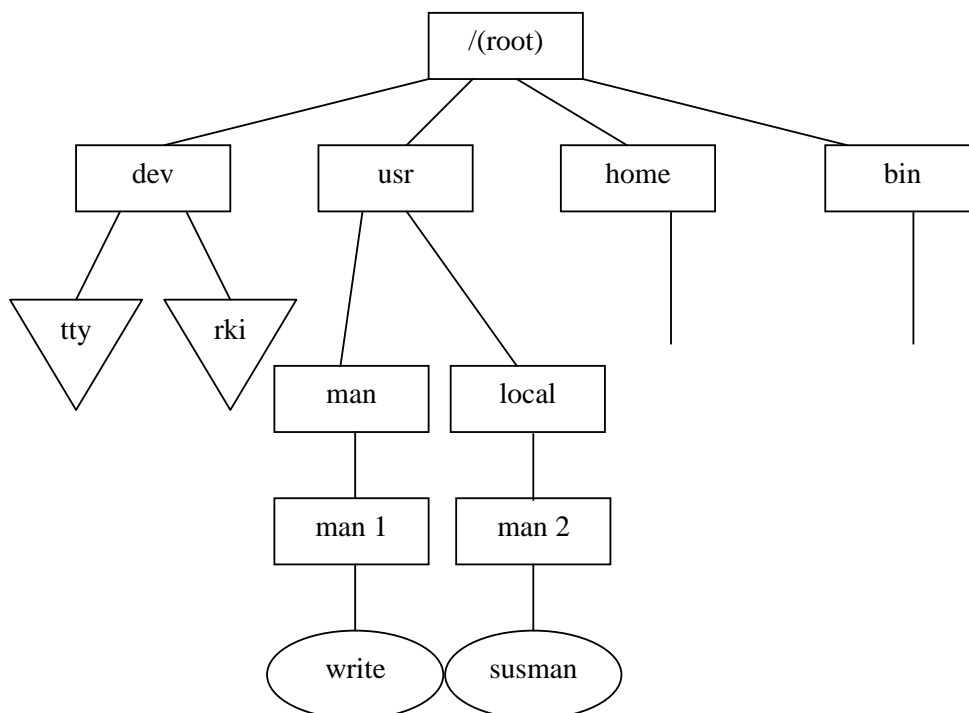


Рисунок 51 – Общая файловая система после монтирования

После монтирования общей файловой системы для пользователя нет логической разницы между корневой и смонтированной файловыми системами, в частности именование файлов производится так же, как если бы она с самого начала была единой.

От файловой системы не требуется, чтобы она целиком размещалась на том устройстве, где находится корень. Запрос от системы *mount* (на установку носителей и т.п.) позволяет встраивать в иерархию файлов файлы на сменных томах. Команда *mount* имеет несколько аргументов, но обязательных аргументов у стандартного варианта ее использования два: имя файла блочного устройства и имя каталога. В результате выполнения этой команды файловая подсистема, расположенная на указанном устройстве, подключается к системе таким образом, что ее содержимое заменяет собой содержимое заданного в команде каталога. Поэтому для *монтирования* соответствующего тома обычно используют

пустой каталог. Команда *umount* выполняет обратную операцию – «отсоединяет» файловую систему, после чего диск с данными можно физически извлечь из системы. Например, для записи данных на дискету необходимо ее «подмонтировать», а после работы – «размонтировать».

Монтирование файловых систем позволяет получить единое логическое файловое пространство, в то время как реально отдельные каталоги с файлами могут находиться в разных разделах одного жесткого диска и даже на разных жестких дисках. Причем, как отмечено выше, сами файловые системы для монтируемых разделов могут быть разными. Например, при работе в ОС *Linux* можно иметь часть разделов с файловой системой *EXT2FS*, а часть разделов – с файловой системой *EXT3FS*.

5.6 **Дисковые массивы RAID**

Дисковый массив *RAID*²² – это консолидированная серверная система для хранения данных большого объема, в которой для размещения информации используют несколько жестких дисков. Поддержка различных уровней избыточности, производительности и способов восстановления после сбоя осуществляется посредством целого ряда разнообразных методик хранения. В массивах *RAID* значительное число дисков относительно малой емкости используется для хранения крупных объемов данных, а также для обеспечения более высокой надежности и избыточности.

Дисковый массив *RAID* может быть образован несколькими способами. Некоторые типы массивов *RAID* предназначены в первую очередь для повышения производительности, гарантии высокого уровня надежности, обеспечения отказоустойчивости и коррекции ошибок. Общей функцией для массивов *RAID* является функция «горячей замены». Иными словами, пользователь имеет возможность удалить выбранный дисковод, установив на его место другой. Для большинства типов дисковых массивов *RAID* данные на замененном диске могут быть восстановлены автоматически, без отключения сервера.

Очевидно, что не только *RAID* поддерживает функции защиты данных большого объема. Однако традиционно применяемое в таких случаях программное обеспечение резервного копирования или зеркалирования действует намного медленнее и зачастую предусматривает отключение системы в случае возникновения ошибки на дисковом диске.

Вне зависимости от того, произошла ли замена дисководов из-за сбоя системы или вызвана какой-либо другой причиной, серверы для замены дисководов отключать не нужно – *RAID* позволяет восстановить

²² *Redundant array of independent disks (RAID)* – избыточный массив недорогих дисков.

данные с других дисководов, используя зеркальные копии или информацию о четности, и не требует отключения компьютера.

На практике, дисковые массивы *RAID* имеют различные варианты реализации, называемые системами *RAID* различного уровня – 0, 3 и 5.

RAID уровня 0. Представляет собой простейший вариант организации дискового массива. На обычном жестком диске данные хранятся в последовательных секторах одного и того же диска. Система использует как минимум два диска и разделяет данные на блоки, которые имеют размер от 512 байт до нескольких мегабайт и поочередно записываются на разные диски. Сегмент 1 записывается на диск 1, сегмент 2 – на диск 2 и т.д. Когда система доходит до последнего диска в массиве, то следующий сегмент она записывает на диск 1 и т. д.

Благодаря сегментации данных нагрузка ввода/вывода распределяется между всеми дисковыми. А поскольку чтение и запись на диски может осуществляться одновременно, производительность возрастает весьма существенно. Но такой подход не обеспечивает защиты данных. Если на диске возникает ошибка, то данные безвозвратно теряются. Системы *RAID* уровня 0 не предназначены для критически важных приложений, но хорошо подходят для таких задач, как создание и редактирование видео и изображений.

RAID уровня 1. На этом уровне работы *RAID* обеспечивается зеркалирование дисков. Данные тождественно дублируются на два идентичных диска. В случае отказа одного из дисков система продолжает работать с другим диском.

RAID уровня 2. Режим работы системы *RAID*, на котором обеспечивается чередование дисков и выделяется контрольный диск. При каждой операции чтение происходит со всех дисков.

RAID уровня 3. Предусматривают расщепление данных, выбирают один из дисков для хранения контрольной суммы по разделам. Такой подход позволяет обеспечить определенную отказоустойчивость и особенно полезен в средах, предполагающих интенсивное использование данных, либо в однопользовательских средах для доступа к длинным последовательным записям. Системы *RAID* уровня 3 не поддерживают перекрытия ввода/вывода и требуют применения синхронизованных дисководов для того, чтобы предотвратить снижение производительности при работе с короткими записями.

RAID уровня 4. Режим работы системы *RAID*, на котором обеспечивается разбиение данных по дискам с чередованием битов и контрольной суммой. *RAID 4* похож на *RAID 3*, но отличается от него тем, что данные разбиваются на блоки, а не на байты. Таким образом, удалось преодолеть проблему низкой скорости передачи данных небольшого объема. Запись же производится медленно из-за того, что четность

для блока генерируется при записи и записывается на единственный диск.

RAID уровня 5. Аналогичны системам уровня 3. Однако контрольная информация на диски заноситься не на отдельный диск, а равномерно по очереди распределяется по всем дискам, что позволяет избежать «узких» мест при работе с контрольным диском. В системах RAID уровня 5 перекрываются все операции ввода/вывода. Для организации подобных систем требуется от трех до пяти дисков. Они лучше подходят для многопользовательских комплексов, для которых производительность не имеет критически важного значения или которые выполняют небольшое количество операций записи.

RAID уровня 10. Зеркалированный массив, который затем записывается последовательно на несколько дисков, как RAID 0. Эта архитектура представляет собой массив типа RAID 0, сегментами которого являются массивы RAID 1. Он объединяет в себе высокую отказоустойчивость и производительность. Современные контроллеры используют этот режим по умолчанию для RAID 1. То есть, 1 диск основной, 2-й диск – зеркало, причем чтение производится с них поочередно, как для RAID 0. Собственно, сейчас можно считать что RAID 10 и RAID 1+0 – это просто разное название одного и того же метода аппаратного зеркалирования дисков. Но не стоит забывать, что полноценный RAID 1+0 должен содержать как минимум 4 диска.

5.7 Вопросы для самопроверки

1. Какова физическая организация жесткого диска?
2. Какова структура таблицы разделов жесткого диска? Где она располагается?
3. На какие уровни можно разделить логическую схему организации файловой системы? Каково основное назначение каждого из уровней?
4. Какова типовая структура файловой системы на диске?
5. Какие существуют способы выделения дискового пространства? Какой из них наиболее распространен в современных операционных системах?
6. Каковы основные недостатки при связном выделении блоков?
7. Какие существуют способы управления свободным и занятым пространством внешней памяти?
8. Что такое «логический блок диска»? Каким размером он должен обладать?
9. Каковы особенности загрузки ОС в основную память компьютера?
10. Какие типы файлов присутствуют в файловых системах?

11. Какие отличия в организации файловых систем *FAT12*, *FAT16*, *FAT32*?
12. Каковы основные особенности файловой системы *NTFS*? Что такое *MFT*-зона?
13. Каковы основные особенности организации файловой системы *UFS*?
14. Каково основное назначение дисковых массивов *RAID*?
15. Каковы отличия различных уровней системам *RAID*?

6. ПРАКТИЧЕСКОЕ ЗАНЯТИЕ № 1. ЗНАКОМСТВО С ОПЕРАЦИОННОЙ СИСТЕМОЙ UNIX

6.1 *Цель работы*

Ознакомиться с операционной системой *Unix*, получить практические навыки работы в наиболее распространенном командном интерпретаторе *bash*, изучить принципы организации файловой системы *Unix* и базовых команд управления файлами.

6.2 *Задание*

Осуществить в локальной сети с помощью программы *PuTTY* через протокол *ssh* доступ к удаленному компьютеру (необходимые данные для доступа указывает преподаватель) под управлением ОС *Linux* в консольном режиме. Ознакомиться с перечнем основных команд, используемых пользователями ОС *Linux* при работе в системе.

6.3 *Основы работы в операционной системе UNIX*

6.3.1 Интерфейс командной строки в системах *Unix*

В *Unix*-подобных ОС базовый уровень общения с пользователем заключается во вводе с клавиатуры команд и просмотре выводимой текстовой информации на дисплее (такой способ общения часто называют «интерфейсом командной строки»). Понятия «клавиатура» и «дисплей» в данном случае во многом условны и не обязательно означают реальные устройства компьютера, на котором работает пользователь. Например, «дисплеем» может быть окно графической среды пользователя или интерфейс программы удаленного доступа, использующей протоколы *telnet* или *ssh*. Но при этом смысл от этого не изменяется: базовый интерфейс пользователя предполагает ввод команд и вывод текстовой информации. Для того, чтобы подчеркнуть «виртуальность» устройств ввода и вывода текста, их вместе называют терминалом.

Unix – многопользовательская ОС, следовательно, каждый компьютер под управлением этой ОС может иметь множество терминалов, что позволяет одновременно работать многим пользователям. Терминалы могут быть непосредственно подключены к компьютеру или существовать где-либо в сети (локальной или глобальной). Сетевые терминалы обычно представляют собой компьютеры с собственной ОС, на которых запущена программа удаленного доступа, подобная стандартной программе *telnet*, работающей с использованием сетевых протоколов на основе TCP.

В настоящее время вместо *telnet* обычно используют программы,

работающие по защищенному шифрованием протоколу *ssh*.

Интерфейс командной строки – не единственный способ общения с ОС *Unix* (например, существуют графические интерфейсы пользователя: *KDE*, *GNOME*, *Xfse* и т. д., различные файловые оболочки и т. п.), но именно в *Unix* умение работать с командной оболочкой очень важно. Во многом это обусловлено огромным набором базовых команд, их чрезвычайной гибкостью и возможностью совместного использования для автоматизации обработки данных.

Регистрация

Во всех *Unix*-подобных ОС, установленных на конкретном компьютере, имеется некоторая база данных пользователей, имеющих право использования ресурсов этого компьютера. Пользователей, не включенных в этот список, система к работе не допустит. База данных ведется администратором компьютера и содержит для каждого пользователя следующую информацию:

- регистрационное имя;
- зашифрованный пароль;
- идентификатор пользователя (*User ID – UID*);
- список групп, в которые включен пользователь;
- путь к командной оболочке;
- путь к домашнему каталогу;
- другая дополнительная информация.

Регистрационное имя и пароль необходимы для процедуры регистрации пользователя. Идентификатор *UID* востребован внутренними функциями системы и непосредственно используется редко. Механизм групп позволяет объединять пользователей по определенным полномочиям на доступ к файлам и программам. Путь к командной оболочке нужен для ее запуска после процедуры регистрации. И, наконец, домашний каталог – это обычно место в файловой системе, целиком принадлежащее данному пользователю и, конечно, администратору.

Начало работы

Чтобы начать работу с *Unix*, нужно получить доступ к терминалу и зарегистрироваться в системе. В случае удаленной работы подключение терминала можно осуществить запуском программы поддерживающей протокол *ssh* и соединением с тем компьютером, на котором предполагается вести работу. Когда компьютер готов зарегистрировать пользователя, на экране отображается приглашение к вводу его имени:

login: _

В ответ на это приглашение нужно ввести регистрационное имя, согласованное с администратором или владельцем компьютера (понятно, что на собственном компьютере пользователь сам вправе выбирать имена пользователей). Имя пользователя рекомендуется составлять из строчных латинских букв и цифр. Некоторые имена (например, *root*, *ftp* и т. п.) могут быть зарезервированы для системных целей и не могут быть отданы обычному пользователю. Особый частный случай в большинстве систем – имя *root*, которое принадлежит администратору или владельцу компьютера. Это имя дает практически неограниченные права по управлению системой. Пользователя с правами *root* часто называют привилегированным.

После ввода имени и нажатия клавиши Enter («Return», «Ret», «CR») система выведет на экран запрос на ввод пароля. Например:

```
login: alex
```

```
password: _
```

Здесь и далее в аналогичных примерах вводимую пользователем информацию будем выделять другим шрифтом, что позволит отличать ее от выводимых системой символов.

Ввод пароля также завершается нажатием клавиши *Enter*. Вводимый пароль не отображается на экране. Если пользователь с указанным именем существует и его пароль введен правильно, система сделает домашний каталог пользователя текущим и запустит командную оболочку, связанную с данным пользователем. Оболочка обычно выполняет некоторый начальный набор команд, который может вывести приветственное сообщение, указать на наличие или отсутствие новой почты, выполнить начальный набор команд (все эти действия зависят от особенностей настройки конкретной ОС). И, наконец, на экране появится приглашение к вводу команды:

```
$ _
```

Приглашение не обязательно имеет такой вид, как показано выше, и зависит от конкретной командной оболочки и ее конфигурации.

Командные оболочки ОС Unix

В системах Unix используются различные командные оболочки (command shells), называемые также командными процессорами или интерпретаторами команд. Среди них наиболее известны и распространены:

- *sh* (Bourne shell) – оболочка Борна (испытана временем, но не слишком удобна в работе);

- *cs*h (C-shell) – оболочка *C* (несколько более удобна по сравнению с *sh*, но несовместима с ней по командному языку);
- *ksh* (Korn shell) – оболочка Корна (включает мощный командный язык, основанный на языке *sh*, и развитые средства интерактивной работы);
- *bash* (Bourne-Again Shell) – «снова» оболочка «Борна» (удобна для интерактивной работы, создана на основе *sh* и во многом с ней совместима).

Тип оболочки, как правило, можно определить по последнему символу приглашения: знак доллара («\$») указывает на *sh*-совместимую оболочку (*sh*, *bash*, *ksh*), а знак амперсанда («&») соответствует оболочке *cs*h. Однако у привилегированного пользователя, независимо от используемого командного процессора, последним символом приглашения обычно бывает знак решетки («#»).

Основными функциями командных оболочек являются:

- организация диалога с пользователем (ввод команд);
- выполнение внутренних команд;
- запуск внешних программ;
- исполнение командных файлов.

Возможности командных языков в системе Unix являются гораздо более полными, чем в системе MS-DOS, и вполне могут быть названы полноценными языками программирования. Командные языки в разных оболочках различаются, а стандартным принято считать командный язык оболочки *bash*.

Команды Unix и запуск программ

Общий синтаксис команд в Unix-подобных ОС выглядит следующим образом:

имя_команды [ключи ...] [параметры ...]

Первый элемент обозначает конкретную команду, аргументы (ключи и параметры) могут сообщать дополнительную информацию. Ключи обычно начинаются со знака «минус». Например, команда

```
ls -l -a /home
```

состоит:

- из имени команды «*ls*», выводящей список файлов в заданном каталоге;
- ключа (модификатора) «*l*», указывающего, что нужно вывести подробный листинг;

- ключа «a», указывающего, что нужно выводить все файлы, включая служебные («дот-файлы»);
- параметра «/home», задающего путь к каталогу.

В командах ОС Unix, их ключах и параметрах регистр букв (строчные или заглавные) различается. Для большей части команд характерна запись строчными буквами. Ключи во многих случаях могут объединяться в одну группу. Например, команда

```
ls -la /home
```

полностью эквивалентна рассмотренной выше.

Команды разделяются на внутренние, которые выполняются командным процессором, и внешние. Внутренних команд обычно немного, а их состав и синтаксис могут зависеть от используемой командной оболочки. При использовании *bash* полный список и краткий синтаксис внутренних команд можно получить, набрав после приглашения команду «help».

Внешние команды представляют собой запуск программ, независимых от оболочки. Для запуска программы простым указанием ее имени необходимо, чтобы путь к этой программе был указан в переменной среды PATH (аналог одноименной переменной среды в MS-DOS). Если программа не найдена в каталогах, перечисленных в PATH, перед именем программы должен быть явно указан путь, даже если программа находится в текущем каталоге (хотя в современных Unix системах это уже не требуется). Например, запуск программы *hello* из текущего каталога может выглядеть так:

```
$ ./hello
```

В этом примере знак доллара в начале строки представляет собой приглашение к вводу, формируемое системой, а остаток строки – информацию, введенную пользователем.

Пути в переменной среды PATH отделяются друг от друга знаками двоеточия без окаймляющих пробелов. Если вывести на экран листинги всех каталогов, входящих в PATH, можно таким образом получить полный список путей внешних команд системы, с которой осуществляется работа. Следует отметить, что значение любой переменной среды можно получить, указав в требуемом контексте ее имя с предшествующим знаком доллара. Например, в команде

```
$ echo $PATH
```

выражение \$PATH будет заменено командным интерпретатором на содержимое переменной PATH. Учитывая, что действие команды echo за-

ключается в выводе в стандартный поток вывода своего аргумента, то на экран попадет именно содержимое переменной среды PATH.

Изменение пароля

Первая команда, которую следует выполнить при первом сеансе работы в системе — команда изменения собственного пароля: *passwd*. Эта команда вызывается без параметров. После ее запуска на экране появится приглашение ввести старый пароль (если пароля не было, этот шаг может быть пропущен). После правильного ввода старого пароля будет предложено ввести новый пароль, а затем ввести его еще раз для исключения случайной ошибки. Пароли при вводе отображаться не будут. Ниже представлен примерный протокол работы команды *passwd*:

```
$ passwd
Changing password for alex
Old password:
New password:
Re-type new password:
$ _
```

Для выбора паролей существуют определенные правила. Основное требование состоит в том, что пароль не должен быть угадываемым. Не надо писать свое имя в обратном порядке, не следует составлять пароль из одинаковых или ряда соседних на клавиатуре букв и т. п. Пароль должен содержать минимум шесть-семь символов и включать необычные сочетания букв, цифр, дефисов и подчеркиваний. В современных системах предпочтительно использование длинных паролей из 12-ти и более символов. Многие реализации команды *passwd* пытаются определить пригодность нового пароля и выводят предупреждающие сообщения, если пароль неудачен. Типичным случаем, когда *passwd* может проявить «недовольство» — ввод пароля только из строчных букв.

Не следует думать, что вышеуказанные правила чрезмерны. Даже личный домашний компьютер при входе в сеть Internet через модемное соединение может стать видимым другим людям, которые могут попытаться поменять пароль или узнать пароль соединения с провайдером. Поэтому простой пароль может существенно упростить задачу злоумышленников и стоить «нерадивому» пользователю очень дорого.

Получение справочной информации

Системы *Unix*, как правило, поставляются с огромным количеством справочной информации в электронном виде. Справочная информация

разбита на разделы по тематике. Нумерация разделов в разных системах может быть разной. Пожалуй, самая часто используемая информация содержится в разделе 1, где рассматриваются команды и прикладные программы, доступные рядовым пользователям системы. В пределах раздела справочные материалы организованы по так называемым «страницам» (manual page). Каждая такая страница содержит документацию по конкретной команде, функции, интерфейсу, протоколу и т. п. и в реальности может быть многостраничным документом. Для получения справочной информации можно использовать команду

```
man [раздел] [ключ]
```

Для получения справки по использованию команды или программы аргумент «ключ» должен быть именем соответствующей команды или программы. Параметр «раздел» может представлять собой цифру (или букву) номера раздела справочных руководств, в котором находится нужная страница документации. Отметим, что номер раздела указывать необязательно, т. к. при его отсутствии будет найден первый подходящий раздел, где встречена нужная тема. Чтобы получить справку об использовании самой команды `man`, проще всего ввести

```
$ man man
```

Существуют и другие полезные команды для работы со справочными руководствами. В частности, команда

```
apropos [ключ]
```

позволяет найти и вывести перечень тех страниц руководств, которые содержат в строке краткого пояснения заданное ключевое слово ключ. Справочная информация `man` доступна только для внешних команд. Для получения подсказки по внутренним командам оболочки необходимо использовать команду `help`, например

```
$ help cd
```

Простейшие команды для работы с файловой системой

Команда изменения текущего каталога:

```
cd [имя каталога]
```

Если команда `cd` вызвана без аргументов, текущим каталогом станет домашний каталог пользователя. Чтобы вывести на экран полное

имя текущего каталога, нужно использовать команду *pwd* без аргументов. Команда

```
ls [имя_каталога]
```

позволяет получить листинг указанного каталога. Если имя каталога не указано, то будет выведен листинг текущего каталога. У команды *ls* есть несколько полезных ключей:

l – вывести полную информацию о каждом файле;

a – вывести листинг всех файлов, включая такие, имена которых начинаются с символа точки.

Команды *mkdir* и *rmdir* позволяют, соответственно, создать или удалить указанный каталог:

```
mkdir [имя каталога]
```

```
rmdir [имя каталога]
```

Команда просмотра файлов *less* позволяет просматривать файлы произвольного размера и перемещаться по их содержимому с помощью клавиш управления курсором (для выхода используется клавиша «q»):

```
less [имя файла]
```

Команда копирования файлов

```
cp [источник] [приемник]
```

Команда перемещения или переименования файлов

```
mv [источник] [приемник]
```

Команда удаления файлов

```
rm [имя файла]
```

С командами *cp* и *rm* может использоваться ключ «*r*», позволяющий копировать, перемещать или удалять каталоги со всем их содержимым рекурсивно.

Для полной информации о перечисленных командах, их аргументах и вариантах их использования можно обратиться к страницам руководства пользователя (команда *man*).

Стандартные потоки ввода-вывода

С каждой программой, запускаемой из командной строки Unix, связаны три стандартных потока данных:

- стандартный поток ввода (stdin);

- стандартный поток вывода (stdout);
- стандартный поток ошибок (stderr).

Программы, требующие входных данных, обычно читают информацию из стандартного потока ввода. Например, команда `wc` подсчитывает количество строк, слов и символов во входных данных. Если запустить эту команду без аргументов, то `wc` будет ожидать входных данных с терминала (чтобы закончить ввод данных, нужно нажать комбинацию клавиш *Ctrl-D*):

```
$ wc
two words
<Ctrl-D>
1    2   10
```

В данном примере программа `wc` прочитала введенный пользователем текст из стандартного потока ввода (куда пользователь ввел текст «two words»). По умолчанию этот поток соединен с терминалом (с клавиатурой) пользователя, но допускается его перенаправление. Чтобы связать данные стандартного входного потока с произвольным файлом, можно использовать операцию перенаправления «<», например:

```
$ wc < /etc/passwd
28   37  1052
```

В данном случае команда `wc` уже не требует ввода с клавиатуры, т. к. она уже получила входные данные из файла `/etc/passwd`. Заметим, что данная команда может иметь практическое применение – первая цифра означает количество строк в файле `/etc/passwd`, что соответствует количеству пользователей, зарегистрированных в системе.

Стандартный поток вывода – это поток, куда программы записывают выходные данные. В предыдущем примере команда `wc` выводила результат (три числа) именно в этот поток. Так же работают и большинство других неинтерактивных команд (включая *echo*, *pwd* и *ls*, рассмотренные выше). Подобно стандартному потоку ввода выходной поток изначально связан с терминалом и также допускает перенаправление. Для связывания стандартного потока вывода с файлом используется операция «>», например:

```
$ ls > filelist.txt
```


В этом примере команда *ls*, вместо того, чтобы вывести список файлов на экран, записала его в файл с именем «filelist.txt». При этом, если файл с таким именем не существовал, он будет создан, в противном случае его старое содержимое будет потеряно. Существует и другая возможность перенаправления вывода, когда новые выходные данные будут дописаны в конец существующего файла. Для этого используется операция «>>». В следующем примере текущие дата и время будут дописаны в конец файла с именем «dates.txt»:

```
$ date >> dates.txt
```

Сообщения об ошибках выводятся в стандартный поток ошибок. Например, пусть выполняется попытка получить список файлов в каталоге без соответствующих прав доступа:

```
$ ls -l /home/ftp/bin/
```

```
ls: /home/ftp/bin/: Access denied
```

В данном случае команда *ls* вывела сообщение в поток стандартной ошибки. Чтобы перенаправить его в указанный файл, можно использовать операции «2>» и «2>>» (по аналогии с «>» и «>>», только цифра 2 говорит о том, что нужно перенаправить поток ошибок), например:

```
$ ls -l /home/ftp/bin/ 2> last-error.txt
```

Операции перенаправления ввода-вывода можно комбинировать, например:

```
$ wc < /etc/passwd 2>> errors.txt > result.txt
```

Существует другой полезный способ перенаправления ввода-вывода – конвейеры команд. Операция «|» (знак вертикальной черты) позволяет перенаправить стандартный поток вывода одной команды на стандартный входной поток другой команды:

```
$ ls -l /etc | less
```

В этом примере команда *ls* выводит длинный список файлов в каталоге */etc*, эти данные попадают на вход программы *less*, которая позволяет пролистывать текст с помощью клавиш управления курсором. Так осуществляется «объединение» двух независимых команд в один «конвейер».

Рассмотрим более сложный пример формирования конвейера команд. Пусть нам требуется получить в файле «bash-users.txt» отсортиро-

ванный список пользователей в системе, пользующихся командной оболочкой *bash*. Этого можно было бы добиться использованием нескольких команд, сохраняя промежуточные данные во временных файлах (комментарии к командам оболочки приведенные после знака #)

```
$ grep 'bash' /etc/passwd > list1.tmp
# Поиск по заданному шаблону «bash» в файле /etc/passwd
$ sort < list1.tmp > list2.tmp
# Сортировка по алфавиту данных из файла list1.tmp и запись в list2.tmp
$ cut -f1 -d: < list2.tmp > bash-users.txt
#Выделение первых полей строк по разделителю :
# и запись в файл bash-users.txt
$ rm list1.tmp list2.tmp
# Удаление временных файлов
```

Конвейеризация команд позволяет обойтись одной составной командой без использования промежуточных файлов

```
$ grep 'bash' /etc/passwd | sort | cut -f1 -d: > bash-users.txt
```

Заметим, что команды типа *sort* или *cut* часто называют фильтрами. Фильтры получают данные из стандартного входного потока, преобразовывают их и выводят в стандартный поток вывода.

Завершение работы с *Unix*

Каждый сеанс работы с ОС *Unix* должен заканчиваться вводом команды *logout*. Также можно использовать комбинацию клавиш *Ctrl-D*, которая позволяет выполнить команду завершения работы с командной оболочкой, после чего система переходит в режим ожидания регистрации следующего пользователя. Если сеанс работы производился с удаленной машины с использованием протоколов *telnet* или *ssh*, то завершение работы командной оболочки вызывает разрыв соединения.

6.3.2 Основы интерактивной работы в оболочке *bash*

Оболочка (*shell*) или командный интерпретатор в *Unix*-системах обеспечивает два набора функций:

- интерпретация командного языка и исполнение команд, введенных пользователем или подготовленных заранее в текстовом файле;
- интерактивное взаимодействие с пользователем, т. е. предоставление пользователю возможности редактирования и ввода команд.

Ниже рассмотрены особенности работы второй группы из набора функций, т. е. интерактивные возможности командной оболочки *bash*, которая является стандартной для систем *GNU/Linux*, и может быть установлена в других *Unix*-подобных системах.

Оболочка *bash* предоставляет пользователю развитые средства интерактивной работы. В частности, она поддерживает редактирование командной строки, повтор символов, макросы, «карман» (буфер), а также историю команд (т. е. возможность повторить ранее введенную команду) и настраиваемое автоматическое дополнение.

Следует отметить, что умение пользоваться интерактивными возможностями оболочки значительно повышает эффективность работы в *Unix*-системе (особенно в сочетании с хорошим знанием командного языка). Более того, работа непосредственно в командной оболочке часто оказывается значительно более продуктивной по сравнению с использованием файловых менеджеров, таких как *Norton Commander*, *Far Manager* или *Windows Explorer*. Обратная сторона преимуществ работы в оболочке *Unix* заключается в длительном начальном периоде изучения.

Далее рассмотрим лишь некоторые наиболее используемые приемы интерактивной работы. Для более полного описания возможностей оболочки следует пользоваться руководством по использованию *bash* (команда *man bash*).

Редактирование командной строки

Классические оболочки *Unix* позволяли вводить команды как последовательность символов, завершая ввод нажатием клавиши *Enter*. Современные версии командных оболочек, такие как *bash*, включают развитые средства редактирования.

Для многих функций редактирования используются комбинации клавиш с модификаторами *CTRL* и *META*. Модификатор *CTRL* имеется на клавиатуре *IBM*-совместимых компьютеров, а в качестве *META* чаще всего используется клавиша *ALT*. Работоспособность модификатора *META* зависит от настройки терминала, графической среды или программы удаленного доступа. Если с помощью клавиши *ALT* не удастся добиться желаемого результата, можно использовать альтернативный

способ ввода *META*-комбинаций. Для этого перед символом нужно нажать (и отпустить) клавишу *Esc*. Таким образом, например, комбинацию клавиш *META-d* можно заменить последовательностью нажатий *Esc, d*. Для ввода комбинаций наподобие *META-_* (знак подчеркивания) или *META->* (знак «больше») необходимо нажимать и удерживать клавишу *Shift*.

В табл. 1.7–1.9 приведены основные команды для работы в командной строке. Одному действию соответствует, как правило, несколько разных комбинаций клавиш, т. к. их работоспособность может зависеть от типа терминала. Поэтому, если не работает какая-либо из клавиш (например, *Home*), вместо нее может быть использована альтернативная комбинация (например, *CTRL-a*). Также следует отметить, что многие из комбинаций клавиш имеют аналогичное или похожее назначение и в других программах, распространенных в *Unix*.

Таблица 1.7

Команды перемещения по командной строке

Комбинация клавиш	Описание действия
Вправо <i>CTRL-f</i>	Перемещение на один символ вправо
Влево <i>CTRL-b</i>	Перемещение на один символ влево
<i>META</i> -вправо <i>META-f</i>	Перемещение на одно слово вправо
<i>META</i> -влево <i>META-b</i>	Перемещение на одно слово влево
<i>Home CTRL-a</i>	Перемещение в начало строки
<i>End CTRL-e</i>	Перемещение в конец строки

Таблица 1.8

Удаление и вставка фрагментов команд

Комбинация клавиш	Описание действия
<i>Backspace CTRL-h</i>	Удалить символ слева от курсора
<i>Del CTRL-d</i>	Удалить символ в позиции курсора
<i>CTRL-u</i>	Вырезать часть строки слева от курсора
<i>CTRL-k</i>	Вырезать часть строки справа от курсора
<i>META-Backspace CTRL-w</i>	Вырезать слово слева от курсора
<i>META-d</i>	Вырезать слово справа от курсора
<i>CTRL-y</i>	Вставить последний вырезанный текст в позицию курсора
<i>CTRL-/ CTRL-_</i>	Отменить последнюю операцию редакти-

	рования
--	---------

Таблица 1.9

Прочие комбинации клавиш

Комбинация клавиш	Описание действия
<i>Enter</i>	Выполнить текущую команду (положение курсора не имеет значения)
<i>CTRL-L</i>	Очистить экран и поместить текущую команду в верхней строке экрана
<i>CTRL-d</i>	Выйти из оболочки <i>bash</i> , аналогично вводу команды <i>logout</i> (только при условии, что командная строка пуста)

Использование истории команд

Оболочка *bash* поддерживает историю команд, т. е. запоминает введенные ранее команды. Это позволяет вернуться к любой ранее введенной команде, а также использовать отдельные фрагменты команд из истории для ускорения ввода новых команд. История сохраняется при выходе из оболочки в файле с именем *.bash_history* в домашнем каталоге пользователя и загружается вновь при следующем запуске *bash*. Таким образом, история команд не пропадает в перерывах между сеансами работы. Впрочем, существует ограничение на количество запоминаемых команд (например, 1000), и при превышении этого ограничения самые ранние команды будут автоматически удаляться.

Чтобы просмотреть историю команд, можно использовать команду *history*. Если после имени этой команды указан числовой аргумент, то будет выведено соответствующее число последних введенных команд. Например:

```
$ history 5
4995 mkdir tmp/work
4996 cd tmp/work
4997 cp ~/work/log.txt.
4998 joe log.txt
4999 history 5
```

Как видно из вывода команды *history*, каждой команде поставлен в соответствии ее порядковый номер в истории. Чтобы выполнить одну из команд истории, можно ввести в командной строке заданный номер, предварив его восклицательным знаком. Например:

```
$ !4996
```

```
cd tmp/work
```

Очевидно, что вызов команд с использованием их номера непрактичен. Удобнее использовать похожий синтаксис, указывая вместо номера первые несколько символов команды. В этом случае будет произведен поиск команды совпадающими с первыми символами, начиная с конца истории, т. е. с недавно вводимых команд. Например:

```
$ !cd
```

```
cd tmp/work
```

Однако такой способ также имеет недостатки при практическом использовании из-за возможности легко ошибиться и выполнить неверную команду. Вместо этого чаще используют интерактивные операции навигации и поиска в истории. Наиболее употребительные комбинации клавиш, связанные с историей команд, приведены в табл. 1.10.

Таблица 1.10

Некоторые комбинации клавиш для навигации по истории команд

Комбинация клавиш	Описание действия
Вверх <i>CTRL-p</i>	Перейти к предыдущей команде
Вниз <i>CTRL-n</i>	Перейти к следующей команде
<i>META-<</i>	Перейти в начало истории команд
<i>META-></i>	Перейти в конец истории команд (т. е. к текущей команде)
<i>CTRL-r</i>	Осуществить обратный инкрементальный поиск в истории команд (см. описание ниже)
<i>META-.</i>	Вставить последнее слово предыдущей команды в текущую позицию курсора
<i>CTRL-o</i>	Аналогично <i>Enter</i> , но после выполнения команды показать следующую строку истории

Самый простой способ использования истории заключается в переходе на команду, подобную той, что требуется ввести, ее редактировании и нажатии клавиши *Enter*. Если же при этом вместо *Enter* нажать комбинацию *CTRL-o*, то это позволит повторить ввод серии последовательных команд, сохраненных в истории.

Отдельного внимания заслуживает возможность инкрементального поиска в истории (комбинация клавиш *CTRL-r*). Это, пожалуй, наиболее мощный способ использования истории команд. После нажатия комби-

нации клавиш *CTRL-r* обычное приглашение к вводу команд исчезает и появляется индикатор режима инкрементального поиска

```
(reverse-i-search)`': _
```

В этом режиме можно вводить символ за символом любую часть команды из истории и в процессе ввода постоянно видеть наиболее позднюю из совпадающих команд. Например, если происходит поиск команды, содержащей подстроку «web», то после нажатия *CTRL-r* вводим сначала букву «w»

```
(reverse-i-search) 'w': cd tmp/work
```

увидим, что поиск пока не дал нужного результата, и уточняем поиск, вводя следующую букву, «e»

```
(reverse-i-search) 'we': ./update-web.sh
```

Теперь видно, что найденная команда уже содержит фрагмент «web» и для ее нахождения было достаточно ввести лишь два символа. Если же найденная команда оказалась не той, что искали, можно использовать *CTRL-r* для перехода на более ранние команды, также содержащие строку поиска. Продолжая предыдущий пример, повторно нажимаем *CTRL-r*. При этом будет найдена другая, более ранняя команда, например

```
(reverse-i-search) 'we': cd work/web/homepage/
```

Теперь можно выйти из режима поиска несколькими способами. Чтобы перейти на найденную команду в истории, достаточно нажать *Esc* или комбинацию клавиш *CTRL-j*. Чтобы отменить поиск и вернуться в исходное состояние, можно нажать *CTRL-g*. И наконец, нажатие *Enter* приведет к немедленному исполнению найденной команды.

Использование автоматического дополнения в командной строке

Автоматическое дополнение (*completion*) позволяет значительно ускорить ввод команд, имен файлов, имен переменных и имен машин в командной строке. Например, пусть в системе установлена программа *bunzip2* и нет ни одной другой программы или команды, начинающейся буквами «bun». В таком случае в *bash* достаточно набрать в начале командной строки эти три буквы и нажать клавишу *Tab*. При этом остальные символы, формирующие имя команды, будут вставлены автоматически. В оболочке *bash* поддерживается несколько типов дополнения и множество комбинаций клавиш для их активизации. Рассмотрим лишь

две наиболее полезные возможности выполнять автоматическое дополнение (табл. 1.11).

Таблица 1.11

Возможности автоматического дополнения в командной строке

Комбинация клавиш	Описание действия
<i>Tab</i>	Дополнение наиболее подходящим окончанием
<i>META-Tab</i>	Дополнение на основе фраз из истории команд (поскольку роль модификатора <i>META</i> часто исполняет клавиша <i>ALT</i> , а комбинация <i>ALT-Tab</i> обычно используется графической средой для вызова этой команды рекомендуется использовать последовательность нажатий <i>Esc, Tab</i>)

Дополнение с помощью *Tab* может работать по-разному в зависимости от использования контекста. Табл. 1.12 в упрощенном виде показывает правила выбора типа дополнения.

Таблица 1.12

Возможности автоматического дополнения в командной строке

Контекст	Тип дополнения
Начало строки	Дополнение имени команды (поиск среди имен встроенных команд оболочки и программ в переменной среды <i>PATH</i>)
После символа <i>\$</i>	Дополнение имени переменной (поиск среди имен установленных переменных среды)
После символа <i>@</i>	Дополнение имени машины (поиск среди имен машин в файле <i>/etc/hosts</i>)
После символа <i>~</i>	Дополнение имени пользователя (поиск среди имен известных системе пользователей)
После шаблона имени файла	Замена шаблона, только если найден лишь один подходящий файл (в данном случае производится не дополнение, а замена шаблона на подходящее имя файла)
В остальных случаях	Дополнение имени файла (поиск среди имен файлов и каталогов)

Дополнение с помощью *META-Tab* всегда ищет дополнения в истории команд, выбирая фразы, начинающиеся с символов, стоящих перед текущей позицией курсора. Если однозначного варианта не найдено,

независимо от типа дополнения дописывается только часть, совпадающая во всех вариантах, и, в зависимости от конфигурации оболочки, может быть выведен список подходящих дополнений. Если список вариантов не выводится автоматически, его обычно можно вывести повторным нажатием *Tab* или *META-Tab*.

6.3.3 Файловая система

Особенности формирования файлового пространства

Файловое пространство *Unix*-систем представляет собой иерархию файлов, которая имеет единый общий корень – так называемый корневой каталог, обозначаемый знаком прямой косой черты «/». Чтобы однозначно идентифицировать любой файл, можно указать путь к этому файлу от корневого или текущего каталога. Все элементы пути отделяются друг от друга символом прямой косой черты. Если первый символ строки также косая черта, то путь берет начало в корневом каталоге, в противном случае – в текущем. Путь с единственным именем обозначает файл в текущем каталоге. Примеры:

- docs.ps – файл с именем docs.ps в текущем каталоге;
- /usr/doc/FAQ/README – файл с именем README в каталоге /usr/doc/FAQ;
- work/thesis.tex – файл thesis.tex в подкаталоге work текущего каталога.

Понятие текущего каталога несколько отличается от такового в системе *MS-DOS* или *Windows*. В *Unix* у каждого процесса собственный текущий каталог. Корневой каталог файлового дерева *Unix* обычно содержит следующие подкаталоги (в разных системах эта структура может отличаться):

- /bin – минимальный набор исполняемых файлов, необходимый для работоспособности системы;
- /etc – файлы конфигурации системы;
- /dev – файлы устройств;
- /home – домашние каталоги пользователей;
- /lib – основные системные библиотеки и модули;
- /root – каталог администратора системы root;
- /proc – файлы-образы выполняющихся процессов;
- /sbin – минимальный набор утилит администратора;
- /tmp – каталог для временных файлов;

- /usr – основной объем файлов системы: установленные программы, библиотеки, исходные тексты ядра, файлы данных и прочее;
- /var – каталог для изменяющейся информации (учетных данных, почтовых ящиков, очередей принтера, отформатированных страниц документации, логов и др.).

Следует отметить, что символ косой черты не является частью имен каталогов, а лишь указывает, что данные элементы находятся в корневом каталоге. В каждом каталоге также существует два особых «подкаталога» с именами «две точки» и «точка». Первый из них служит указателем на однозначно определенный родительский каталог (вышестоящий), а второй – на данный текущий каталог. Например, путь «../readme» указывает на файл «*readme*», который находится в родительском каталоге (на ступень выше), а путь «./readme.now» укажет на файл «*readme.now*», который находится в текущем каталоге.

Большая часть файлового дерева *Unix* обычно сосредоточена в каталоге /usr. Как правило, там можно найти следующие подкаталоги:

- /usr/bin – исполняемые файлы;
- /usr/doc – документация в различных форматах;
- /usr/etc – файлы конфигурации программного обеспечения, установленного дополнительно;
- /usr/include – включаемые файлы для программ, например на языке C;
- /usr/info – документация пользователя в гипертекстовом формате *info*;
- /usr/lib – разделяемые библиотеки;
- /usr/local – локальное программное обеспечение, файлы данных и библиотеки (этот каталог в некоторых системах может не использоваться);
- /usr/man – руководства пользователя (*manual pages*);
- /usr/sbin – утилиты администратора;
- /usr/share – данные, совместно используемые различными прикладными программами;
- /usr/src – исходные тексты различных компонент системы, включая ядро.

Описанная в данном случае структура каталогов относится к ОС *Red Hat Enterprise Linux 5.2*.

Формирование имен файлов

В связи с тем, что зачастую для одного языка существует несколько кодировок (например, для русского языка существуют следующие кодировки: *CP866*, *CP1251*, *KOI-8R* и т. д., хотя в последнее время с распро-

странением *UTF8* ситуация постепенно улучшается), то рекомендуется, чтобы имя файла или каталога составлялось из следующих символов:

- прописные и строчные латинские буквы;
- цифры;
- символ подчеркивания;
- символ точки;
- знак минуса (не должен быть первым символом имени);
- знак плюса (использовать не рекомендуется).

В каждой конкретной ОС в именах файлов могут быть допустимы и другие символы, но их использование может привести к некорректности работы некоторых программ и, кроме того, может затруднить перенос файлов между разными ОС. Не рекомендуется использовать названия файлов из локальных таблиц кодировок (например, имена файлов на русском языке), т. к. очень часто для одного языка существует несколько кодировок.

Максимальная длина имени файла варьируется в разных системах и зависит скорее от используемой файловой системы, чем от самой ОС. Обычно можно использовать достаточно длинные имена файлов (до 255 символов). Максимальный размер файла в файловой системе также зависит от ее типа. Для современных файловых систем размер файла более 4 Гбайт не является проблемой.

Как отмечалось выше, прописные и строчные буквы в системе *Unix* различаются, т. е. имена «*filename*», «*FILENAME*» и «*FileName*» являются разными. При этом файлы, отличающиеся только регистром букв, могут находиться в одном каталоге.

В отличие от системы MS-DOS, знак точки является обычным символом, допустимым в любом месте имени файла, а такого понятия, как расширение имени файла, строго говоря, в системе *Unix* нет. Тем не менее, последние части имен файлов, отделенные от остальной части именами точками, часто указывают на тип файла. В качестве примера имя файла «*my-photo.tiff.gz*» может означать, что файл представляет собой изображение в формате *TIFF*, сжатое программой сжатия *gzip*.

Точка, являющаяся первым символом имени файла или каталога, имеет особое значение: такие имена по умолчанию не выводятся в листинге содержимого каталогов (хотя к ним можно свободно обращаться), для получения полного списка файлов вместо *ls*, нужно ввести *ls -a*. Другими словами, чтобы сделать файл «скрытым», нужно начать его

имя с точки. Этим часто пользуются для именования служебных файлов, на которые не имеет смысла обращать особое внимание.

Просмотр и интерпретация прав доступа к файлам

ОС семейства *Unix* – традиционно многопользовательские системы. Чтобы начать работать, пользователь должен «войти» в систему, введя со свободного терминала свое регистрационное имя и пароль. Человек, зарегистрированный в учетных файлах системы и, следовательно, имеющий учетное имя, называется зарегистрированным пользователем системы. Регистрацию новых пользователей обычно выполняет администратор системы. Основными минимальными данными, требуемыми для регистрации пользователя в системе, являются:

- имя пользователя;
- название группы, к которой относится пользователь;
- пароль.

В *Unix* базовые права доступа к файлам включают три составляющие:

- разрешение чтения (обозначается буквой «*r*», от слова *Read*);
- разрешение записи (буква «*w*», от слова *Write*);
- разрешение выполнения (буква «*x*», от слова *eXecute*).

Разрешение на чтение позволяет пользователю читать содержимое файлов, а в случае каталогов – просматривать перечень имен файлов в каталоге (используя, например, команду *ls*).

Разрешение на запись позволяет пользователю писать в файл, т. е. изменять его содержимое. Для каталогов это дает право создавать в каталоге новые файлы и каталоги или удалять файлы в этом каталоге.

Наконец, разрешение на выполнение позволяет пользователю запускать файлы на исполнение (как программы в машинном коде, так и командные файлы). Если на файле стоит атрибут выполнения, то независимо от его имени он считается программой, которую можно запустить (в отличие от *DOS* или *Windows*, в *Unix* возможность исполнения файла не зависит от «расширения» имени файла, такого как *.exe*). Разрешение на выполнение применительно к каталогам означает возможность перехода в этот каталог (например, командой *cd*). Поэтому для каталогов право выполнения часто называют правом поиска. Отметим, что для каталогов биты чтения и выполнения (*r* и *x*) чаще всего используются в паре, т. е. либо присутствуют оба, либо отсутствуют.

В атрибутах доступа к файлам, перечисленные типы прав доступа могут быть предоставлены для трех классов пользователей:

- владельца (у каждого файла в *Unix* есть один владелец);
- группы (с каждым файлом связана группа пользователей этого файла);
- всех остальных пользователей.

Набор прав доступа для конкретных файлов можно просмотреть с помощью команды *ls -l*. Например:

```
$ ls -l tmp/
drwxrwxr-x 10 john  users    1024 Aug 30 2002 newdir
-rw-r----- 1 john  users    173727 Jan 13 23:48 archive-0113.zip
```

В этом примере видно, что владельцем файлов является пользователь *john*, а группой владельцев является группа *users*. Набор букв и прочерков в левой части определяет тип файла (первый символ) и права доступа к файлу (остальные девять символов). В приведенном примере первая запись относится к каталогу (первая буква *d*) и демонстрирует права доступа *gwxgwxr-x*. Вторая запись относится к обычному файлу (прочерк на месте первого символа) и показывает права *rw-r-----*. Девять символов прав доступа определяют возможность чтения (*r*), записи (*w*) и выполнения (*x*) для владельца файла (первые три символа), группы владельца (следующие три символа) и всех остальных (последние три символа). Прочерки означают отсутствие соответствующих прав. Следовательно, в приведенном примере

- *john* и все пользователи группы *users* могут просматривать и изменять содержимое каталога *newdir*, а также переходить в него, а остальные пользователи могут читать и переходить в этот каталог, но не могут создавать или удалять в нем новые файлы;
- *john* может читать и изменять файл *archive-0113.zip*, пользователи группы *users* могут только читать содержимое этого файла, а все остальные не имеют к нему никаких прав доступа.

Кроме символьного представления прав доступа часто используется цифровая форма. В цифровом представлении права доступа составляются из трех восьмеричных цифр, каждая из которых определяет набор из трех битов полномочий *gwx*. Чтобы перевести права доступа из символьного представления в числовое, следует:

- представить набор прав в двоичном виде (например, 110100000 для набора прав *rw-r-----*);

- перевести полученное двоичное число в восьмеричную систему счисления (например, восьмеричным представлением двоичного числа 110100000 будет 640).

Права доступа также можно в числовой форме задать путем суммирования восьмеричных значений отдельных битов прав доступа:

- 400 – владелец имеет право на чтение;
- 200 – владелец имеет право на запись;
- 100 – владелец имеет право на выполнение;
- 040 – группа имеет право на чтение;
- 020 – группа имеет право на запись;
- 010 – группа имеет право на выполнение;
- 004 – остальные имеют право на чтение;
- 002 – остальные имеют право на запись;
- 001 – остальные имеют право на выполнение.

Можно заметить, что для прав доступа `rw-r-----` получим: $400 + 200 + 040 = 640$.

Типы файлов

В ОС *Unix* имеются следующие основные типы файлов:

- обычные файлы (*regular files*);
- каталоги (*directories*);
- символичные ссылки (*symbolic links*);
- файлы физических устройств (*device files*);
- именованные каналы (*named pipes*);
- доменные гнезда (*sockets*).

Обычные файлы используются наиболее широко и представляют собой именованные наборы данных с возможностью произвольного доступа.

Каталоги – специальный тип файлов, позволяющий группировать вместе другие файлы и каталоги. Содержимое каталога представляет собой список находящихся в нем файлов.

Файлы устройств в *Unix* являются средством общения прикладных программ с драйверами оборудования компьютера. Для того чтобы передать данные драйверу какого-либо устройства, прикладная программа должна произвести запись в соответствующий специальный файл. По аналогии, операция чтения из файла устройства означает получение данных от его драйвера. Обычно каталог с файлами имеет имя «*/dev*».

Символьные ссылки подобны «ярлыкам» в *Windows*. Они позволяют создавать альтернативные имена файлов, причем могут указывать на файлы в других каталогах. При открытии программой символьной ссылки фактически открывается файл, на который она указывает. Символьные ссылки могут указывать как на обычные файлы, так и на каталоги и файлы других типов.

Именованные каналы еще называют буфером *FIFO* (*First In First Out*). Через файлы такого типа два независимых процесса могут обмениваться данными: все, что записано в файл одним процессом, может быть прочитано другим.

Гнезда – это абстрактные конечные точки сетевого соединения. Записывая данные в этот файл, процесс отправляет их в сеть. При этом процессы, установившие связь через пару гнезд, могут быть запущены как на разных компьютерах, так и на одном.

Монтирование сторонних файловых систем

Как было представлено выше, к файловой системе *Unix* могут быть подключены сторонние файловые системы, например файловые системы других ОС или файловые системы, расположенные на внешних носителях (флоппи-дисках, CD-ROM и др.). Чтобы сторонняя файловая система была доступна ОС *Unix*, необходимо осуществить операцию ее монтирования. Фактически, монтирование – это указание того, куда системе следует адресовываться при обращении к объектам сторонней файловой системы. Для системы это указание называется точкой монтирования.

Теоретически можно указать системе произвольное место точки монтирования, но на практике для монтирования сторонних файловых систем существует каталог */mnt*. В нем необходимо создать подкаталог, который будет служить точкой монтирования. Например файловую систему *Windows*, физически расположенную на первом логическом разделе того же винчестера (на диске *C*), можно сделать доступной для *Unix*, примонтировав ее с помощью команды *mount* (предварительно нужно создать каталог */mnt/windows*)

```
mount -t vfat /dev/hda1 /mnt/windows (пример для Linux),
```

```
mount -t msdos /dev/ad0s1 /mnt/windows (пример для FreeBSD),
```

где ключ *t* и аргумент *vfat* (*msdos*) означают тип монтируемой сторонней файловой системы (в данном случае *FAT*); */dev/hda1* – первый раздел первого жесткого диска, к которому система обращается с помо-

щью файла устройства; `/mnt/windows` – каталог, представляющий собой точку монтирования.

Системный файл регистрации пользователей

В *Unix*-системах регистрация пользователей ведется в файле `/etc/passwd`. Содержимое этого файла представляет собой последовательность текстовых строк. Каждая строка соответствует одному зарегистрированному в системе пользователю и содержит семь полей, разделенных символами двоеточия. Эти поля таковы:

- регистрационное имя пользователя;
- зашифрованный пароль;
- значение *UID*;
- значение *GID* основной группы;
- комментарий (может содержать расширенную информацию о пользователе, например имя, должность, телефоны и т. п.);
- домашний каталог;
- командная оболочка пользователя.

Файл `/etc/passwd` должен быть доступен для чтения всем пользователям, т. к. к нему должны обращаться многие программы, запускаемые от имени рядового пользователя (например, чтобы узнать соответствие *UID* регистрационному имени). Но доступность для чтения всех зашифрованных паролей серьезно уменьшает безопасность системы, потому что современные вычислительные мощности позволяют сравнительно быстро подбирать пароли (в особенности, неудачно выбранные некоторыми пользователями). Поэтому часто используется схема теневого паролей (*shadow passwords*), при которой поле пароля в `/etc/passwd` игнорируется, а реальный пароль берется из другого файла (например, `/etc/shadow`), доступного для чтения только привилегированному пользователю. Файл теневого паролей часто содержит и другую важную информацию: срок, в течение которого допускается использование неизменного пароля, дата последнего изменения пароля и т. п. При использовании теневого паролей второе поле в `/etc/passwd` обычно содержит символ звездочки или любой другой произвольный символ. Пустым полем пароля в `/etc/passwd` оставлять нельзя, так как в этом случае система может посчитать, что данному пользователю пароль не требуется. Пример строки из файла `/etc/passwd` (заметьте, что поле комментария в данном случае отсутствует):


```
john:*:1004:101::/home/john:/bin/sh
```

Дополнительную информацию о формате файла */etc/passwd* можно получить, набрав команду *man* по теме *passwd* в разделе 5 (форматы файлов). Эта команда будет выглядеть так:

```
man 5 passwd
```

Файл регистрации групп пользователей

Информация о группах, известных системе, содержится в файле */etc/group*. Подобно файлу регистрации пользователей, информация в */etc/group* представляет собой набор строк, по одной для каждой зарегистрированной группы пользователей. Каждая строка содержит четыре поля, разделенных двоеточиями:

- регистрационное имя группы;
- пароль группы (обычно это поле пустое, так как группам обычно не назначают пароли);
- значение *GID*, соответствующее данной группе;
- разделенный запятыми список пользователей, входящих в группу (может быть пустым).

Заметим, что пустой список пользователей в записи */etc/group* не означает, что в этой группе нет ни одного пользователя, так как *GID* основной группы пользователя определяется в файле */etc/passwd*.

Определение идентификаторов пользователей и групп

Чтобы определить *UID* пользователя, *GID* и имя его основной группы, а также список прочих групп, в который включен пользователь, можно использовать команду *id*. В случае ее использования без аргументов, команда выведет информацию о текущем пользователе. Если же указать в качестве аргумента имя зарегистрированного пользователя, вывод команды будет соответствовать указанному пользователю.

Частным случаем команды *id* является команда *groups*. Она выдает список имен всех групп, в которые включен текущий или указанный пользователь.

Ввод команды *who* без аргументов позволяет получить список пользователей, работающих в данный момент в системе. Если же набрать *whoami*, система выведет информацию о текущем пользователе.

Как обычно, дополнительную информацию о всех перечисленных командах можно получить с помощью команды *man*, например

```
$ man who
```

Изменение владельцев файлов

Владельцем файла становится пользователь, создавший этот файл. Группой владельца по умолчанию становится основная группа регистрации пользователя. Для изменения владельцев предназначена стандартная команда *chown* (*change owner*). Однако в современных системах владельца файлов может изменять только привилегированный пользователь (*root*). У обычного пользователя существует возможность изменения только группы владельцев, и то лишь в пределах тех групп, в которые входит сам пользователь. Для изменения группы владельцев удобно использовать команду *chgrp* (*change group*). Например, чтобы сделать группой владельцев каталога *newdir* группу *student*, можно ввести

```
chgrp student newdir
```

Существует возможность рекурсивного изменения владельцев для всех файлов и подкаталогов заданного каталога. Для этого следует использовать ключ *R*, например

```
chgrp -R student newdir
```

Заметим, что вместо одного имени файла или каталога в приведенных командах можно использовать множество имен, разделенных пробелами, или шаблоны имен файлов (это относится и к большей части других команд *Unix*, принимающих имена файлов в качестве последних аргументов).

Изменение прав доступа к файлам

Изменить права доступа к файлу может либо его владелец, либо привилегированный пользователь (*root*). Делается это командой *chmod* (*change mode*). Существует два формата использования этой команды: с использованием символьного и числового представления прав доступа. Использование числового представления позволяет одной командой изменить полный набор прав доступа, например

```
chmod 770 newdir
```

Данная команда установит права доступа в числовое значение 770, т. е. *gwxrwx---*, что даст полные права владельцу и группе владельца и никаких прав всем остальным.

Использование символьного представления прав доступа в команде *chmod* может показаться несколько сложнее, но позволяет манипулиро-

вать отдельными битами прав доступа. Например, чтобы снять бит записи для группы владельца каталога *newdir*, достаточно ввести

```
chmod g-w newdir
```

Условный синтаксис этой команды таков:

```
chmod {u,g,o,a}{+,-,=}{r,w,x} файлы ...
```

В качестве аргументов команда принимает указание классов пользователей:

- «*u*» – владелец-пользователь (*user*),
- «*g*» – владелец-группа (*group*),
- «*o*» – остальные пользователи (*others*),
- «*a*» – все вышеперечисленные группы вместе (*all*).

Операции, которые можно произвести с правами доступа, следующие:

- «*+*» – добавить,
- «*-*» – убрать,
- «*=*» – присвоить.

Права доступа («*r*», «*w*», «*x*») назначаемы каталогам и файлам.

Как и в команде *chgrp*, в *chmod* может использоваться ключ *R*, позволяющий рекурсивно обрабатывать содержимое подкаталогов.

Права доступа по умолчанию, команда *umask*

Очевидно, что при создании новых файлов и каталогов они уже будут обладать определенным набором прав доступа. Эти права доступа, устанавливаемые по умолчанию, определяются значением маски прав доступа, которая устанавливается командой *umask*. При вводе команды *umask* без аргументов она выведет текущее значение маски, при использовании восьмеричного числа в качестве аргумента будет установлено новое значение.

Маска прав доступа определяет, какие права должны быть удалены из полного набора прав, т. е. маска прав доступа является в некотором роде обратным значением прав доступа по умолчанию. Например, маска 022 приведет к сбросу битов записи для группы владельца и остальных пользователей. Заметим, что для обычных файлов (не каталогов) все биты выполнения (x) в правах по умолчанию будут сброшены независимо от текущей маски.

Пример, демонстрирующий эффект команды *umask*:

```

$ umask
002
$ mkdir dir1
$ ls -l
drwxrwxr-x  2 john  users    1024 Apr 21 07:29 dir1
$ umask 072
$ umask
072
$ mkdir dir2
$ ls -l
drwxrwxr-x  2 john  users    1024 Apr 21 07:29 dir1
drwx---r-x  2 john  users    1024 Apr 21 07:30 dir2

```

6.4 Последовательность выполнения работы

1. Ознакомиться с теоретическим материалом.
2. Зарегистрироваться в системе под именем, выданным преподавателем.
3. Ознакомиться со следующими командами для пользовательской работы в ОС *Unix*: *man*, *apropos*, *ls*, *cd*, *pwd*, *mkdir*, *rmdir*, *cp*, *mv*, *rm*, *cat*, *echo*, *less*, *touch*, *grep*, *date*, *history*. Определить параметры, которые следует считать основными при использовании данных команд.
4. Определить абсолютный путь своего домашнего каталога.
5. Определить значения следующих переменных окружения: *PATH*, *MANPATH*, *PAGER*.
6. Определить границы файлового пространства, где система позволяет создавать собственные файлы и каталоги (возможное использование автоматического скрипта).
7. Проверить, возможно ли вмешательство в личное файловое пространство другого пользователя.
8. Ознакомиться с командами определения прав доступа к файлам и их изменения (команды *id*, *groups*, *ls*, *stat*, *chmod*, *chown*, *chgrp*, *umask*).
9. Найти запись в файле */etc/passwd*, соответствующую вашему регистрационному имени.
10. Определить свой *UID*, узнать, к каким группам относится ваше регистрационное имя, объяснить вывод команд *id*, *groups*.
11. Определить список групп, в которые входит пользователь *root*.

12. Узнать, какими правами доступа обладают вновь создаваемые файлы и каталоги (т. е. создать новый файл и новый каталог, и просмотреть для них права доступа).

13. Определить значение *umask*, при котором создаваемые файлы и каталоги будут недоступны для чтения, записи и исполнения никому, кроме владельца.

14. Сделать свой домашний каталог видимым для всех пользователей группы *users*.

15. Создать в домашнем каталоге подкаталог *tmp*, файлы в котором сможет создавать, удалять и переименовывать любой, входящий в группу *users*, при этом содержимое этого подкаталога не должно быть видимым всем прочим пользователям.

7. ПРАКТИЧЕСКОЕ ЗАНЯТИЕ № 2. ЗНАКОМСТВО СО СТАНДАРТНОЙ УТИЛИТОЙ *GNU MAKE* ДЛЯ ПОСТРОЕНИЯ ПРОЕКТОВ В ОС *UNIX*

7.1 *Цель работы*

Ознакомиться с техникой компиляции программ на языке программирования *C* (*C++*) в среде ОС семейства *Unix*, а также получить практические навыки использования утилиты *GNU make* для сборки проекта.

7.2 *Задание*

Изучить особенности работы с утилитой *make* при создании проекта на языке *C* (*C++*) в ОС *Unix*, а также получить практические навыки использования утилиты *GNU make* при создании и сборке проекта.

7.3 *Основы использования утилиты построения проектов Make*

«Сборка» большинства программ для ОС семейства *Unix* производится с использованием утилиты *make*. Эта утилита считывает файл (обычно носящий имя «*makefile*» или «*makefile*»; далее, упоминая имя этого файла, будем использовать *makefile*), в котором содержатся инструкции, и выполняет в соответствии с ними действия, необходимые для сборки программы. Во многих случаях *makefile* полностью генерируется специальной программой. Например, для разработки процедур сборки используются программы *autoconf/automake*. Однако в некоторых программах может потребоваться непосредственное создание файла *makefile* без использования процедур автоматической генерации.

Следует отметить, что существует, как минимум, три различных наиболее распространенных варианта утилиты *make*: *GNU make*, *System V make* и *Berkeley make*.

7.3.1 *Основные правила работы с утилитой make*

Основными составляющими любого *make*-файла являются правила (*rules*). В общем виде правило выглядит так:

```
<цель_1> ... <цель_n>: <зависимость_1> ... <зависимость_n>  
<команда_1>  
...  
<команда_n>
```

Цель (target) – это некоторый желаемый результат, способ достижения которого описан в правиле. Цель может представлять собой имя файла. В этом случае правило описывает, каким образом можно получить новую версию этого файла.

В следующем примере целью является файл *iEdit* (исполняемый файл программы некоторого гипотетического проекта текстового редактора с главным файлом проекта *main.cpp* и дополнительными – *Editor.cpp*, *TextLine.cpp*). Правило описывает, каким образом можно получить новую версию бинарного файла *iEdit* (скомпоновать из перечисленных объектных файлов):

```
iEdit: main.o Editor.o TextLine.o
gcc main.o Editor.o TextLine.o -o iEdit
```

Если необходимо скомпилировать проект, написанный на C++, то можно использовать компилятор g++. Следует также отметить, что ключ *o* компилятора *gcc* указывает имя конечно бинарного файла.

Цель также может быть именем некоторого действия. В таком случае правило описывает, каким образом совершается указанное действие. В следующем примере целью является действие *clean* (очистка, удаление):

```
clean:
rm *.o iEdit
```

Подобного рода цели называют псевдоцелями (*pseudo targets*) или абстрактными целями (*phony targets*).

Зависимость (*dependency*) – это некие «исходные данные», необходимые для достижения указанной в правиле цели, некоторое «предварительное условие» для достижения цели. Зависимость может представлять собой имя файла. Для того чтобы успешно достичь указанной цели, этот файл должен существовать.

В следующем правиле файлы *main.o*, *Editor.o* и *TextLine.o* являются зависимостями. Эти файлы должны существовать для того, чтобы стало возможным достижение цели – построение файла *iEdit*:

```
iEdit: main.o Editor.o TextLine.o
gcc main.o Editor.o TextLine.o -o iEdit
```

Зависимость также может быть именем некоторого действия. Это действие должно быть предварительно выполнено перед достижением цели, указанной в правиле. В следующем примере зависимость *clean_obj* является именем действия (удалить объектные файлы программы):

```
clean_all: clean_obj
rm iEdit
clean_obj:
rm *.o
```

Для того, чтобы достичь цели *clean_all*, необходимо сначала выполнить действие (достигнуть цели) *clean_obj*.

Команды – это действия, которые необходимо выполнить для обновления либо достижения цели. В следующем примере командой является вызов компилятора *gcc*. Утилита *make* отличает строки, содержащие команды, от прочих строк *make*-файла по наличию символа табуляции (символа с кодом 9) в начале строки:

```
iEdit: main.o Editor.o TextLine.o
gcc main.o Editor.o TextLine.o -o iEdit
```

В приведенном выше примере строка *gcc main.o Editor.o TextLine.o -o iEdit* должна начинаться с символа табуляции.

Общий алгоритм работы *make*

Типичный *make*-файл проекта содержит несколько правил. Каждое из правил имеет некоторую цель и некоторые зависимости. Смыслом работы *make* является достижение цели, которую она выбрала в качестве главной цели (*default goal*). Если главная цель является именем действия (т. е. абстрактной целью), то смысл работы *make* заключается в выполнении соответствующего действия. Если же главная цель является именем файла, то программа *make* должна построить самую «свежую» версию указанного файла.

Выбор главной цели. Главная цель может быть прямо указана в командной строке при запуске *make*. В следующем примере *make* будет стремиться достичь цели *iEdit* (получить новую версию файла *iEdit*)

```
make iEdit
```

В этом примере *make* должна достичь цели *clean* (очистить директорию от объектных файлов проекта)

```
make clean
```


Если не указывать какой-либо цели в командной строке, то *make* выбирает в качестве главной первую встреченную в *make*-файле цель. В следующем примере из четырех перечисленных в *make*-файле целей (*iEdit*, *main.o*, *Editor.o*, *TextLine.o*, *clean*) по умолчанию в качестве главной будет выбрана цель *iEdit*:

```
iEdit: main.o Editor.o TextLine.o
gcc main.o Editor.o TextLine.o -o iEdit
main.o: main.cpp
gcc -c main.cpp
Editor.o: Editor.cpp
gcc -c Editor.cpp
TextLine.o: TextLine.cpp
gcc -c TextLine.cpp
clean:
rm *.o
```

Схематично «верхний уровень» алгоритма работы *make* можно представить так:

```
make()
{
    главная_цель = ВыбратьГлавнуюЦель ()
    ДостичьЦели (главная_цель)
}
```

Достижение цели. После того как главная цель выбрана, *make* запускает «стандартную» процедуру достижения цели. Сначала в *make*-файле выполняется поиск правила, которое описывает способ достижения этой цели (функция «НайтиПравило»). Затем к найденному правилу применяется обычный алгоритм обработки правил (функция «ОбработатьПравило»):

```
ДостичьЦели (Цель)
{
    правило = НайтиПравило (Цель)
    ОбработатьПравило (правило)
}
```

Обработка правил. Обработка правила разделяется на два основных этапа. На первом этапе обрабатываются все зависимости, перечисленные в правиле (функция «ОбработатьЗависимости»). На втором этапе принимается решение о том, нужно ли выполнять указанные в правиле команды (функция «НужноВыполнятьКоманды»). При необходимости перечисленные в правиле команды выполняются (функция «ВыполнитьКоманды»):

```
ОбработатьПравило(Правило)
{
    ОбработатьЗависимости (Правило)
    если НужноВыполнятьКоманды (Правило)
    {
        ВыполнитьКоманды (Правило)
    }
}
```

Обработка зависимостей. Функция «ОбработатьЗависимости» поочередно проверяет все перечисленные в правиле зависимости. Некоторые из них могут оказаться целями каких-нибудь правил. Для этих зависимостей выполняется обычная процедура достижения цели (функция «ДостичьЦели»). Те зависимости, которые не являются целями, считаются именами файлов. Для таких файлов проверяется факт их наличия. При их отсутствии *take* аварийно завершает работу с сообщением об ошибке.

```
ОбработатьЗависимости (Правило)
{
    цикл от i=1 до Правило.число_зависимостей
    {
        если ЕстьТакаяЦель (Правило.зависимость[ i ])
        {
            ДостичьЦели (Правило.зависимость[ i ])
        }
        иначе
        {
            ПроверитьНаличиеФайла (Правило.зависимость[ i ])
        }
    }
}
```

```

    }
  }
}

```

Обработка команд. На стадии обработки команд решается вопрос о том, следует ли выполнять описанные в правиле команды или нет. Считается, что нужно выполнять команды в таких случаях, как:

- цель является именем действия (абстрактной целью);
- цель является именем файла и этого файла не существует;
- какая-либо из зависимостей является абстрактной целью;
- цель является именем файла и какая-либо из зависимостей, являющихся именем файла, имеет более позднее время модификации, чем цель.

В противном случае (т. е. ни одно из вышеприведенных условий не выполняется) описанные в правиле команды не выполняются. Алгоритм принятия решения о выполнении команд схематично можно представить так:

```

НужноВыполнятьКоманды (Правило)
{
    если Правило.Цель.ЯвляетсяАбстрактной ()
        return true
    // цель является именем файла
    если ФайлНеСуществует (Правило.Цель)
        return true
    цикл от i=1 до Правило.Число_зависимостей
    {
        если Правило.Зависимость[ i ].ЯвляетсяАбстрактной ()
            return true
        иначе
            // зависимость является именем файла
            {
                если ВремяМодификации(Правило.Зависимость[ i ]) >
                    ВремяМодификации (Правило.Цель)
                    return true
            }
    }
}

```

```

    }
    return false
}

```

Абстрактные цели и имена файлов. Имена действий от имен файлов утилита *make* отличает следующим образом: сначала выполняется поиск файла с указанным именем, и если файл найден, то считается, что цель или зависимость являются именем файла; в противном случае считается, что данное имя является либо именем несуществующего файла, либо именем действия (различия между этими двумя вариантами не делается, поскольку они обрабатываются одинаково).

Следует отметить, что подобный подход имеет ряд недостатков. Во-первых, утилита *make* не рационально расходует время, выполняя поиск несуществующих имен файлов, которые на самом деле являются именами действий. Во-вторых, при подобном подходе имена действий не должны совпадать с именами каких-либо файлов или директорий, иначе *make*-файл будет выполняться ошибочно.

Некоторые версии *make* предлагают свои варианты решения этой проблемы. Так, например, в утилите *GNU make* имеется механизм (специальная цель *.PHONY*), с помощью которого можно указать, что данное имя является именем действия.

7.3.2 Пример практического использования утилиты *make*

Пример создания простейшего *make*-файла

Рассмотрим, как утилита *make* будет обрабатывать такой *make*-файл (makefile):

```

iEdit: main.o Editor.o TextLine.o
gcc main.o Editor.o TextLine.o -o iEdit
main.o: main.cpp
gcc -c main.cpp
Editor.o: Editor.cpp
gcc -c Editor.cpp
TextLine.o: TextLine.cpp
gcc -c TextLine.cpp
clean:
rm *.o

```

Предположим, что в директории с проектом находятся следующие файлы:

main.cpp

Editor.cpp

TextLine.cpp

Предположим также, что программа *make* была вызвана следующим образом:

make

Цель не указана в командной строке, поэтому запускается алгоритм выбора цели (функция «ВыбратьГлавнуюЦель»). Главной целью становится файл *iEdit* (первая цель из первого правила). Цель *iEdit* передается функции «ДостичьЦели». Эта функция выполняет поиск правила, которое описывает обрабатываемую цель. В данном случае это первое правило *make*-файла. Для найденного правила запускается процедура обработки (функция «ОбработатьПравило»).

Сначала поочередно обрабатываются описанные в правиле зависимости (функция «ОбработатьЗависимости»). Первая зависимость – объектный файл *main.o*. Поскольку в *make*-файле есть правило с такой целью (функция «ЕстьТакаяЦель» возвращает *true*), то для цели *main.o* запускается процедура «ДостичьЦели».

Функция «ДостичьЦели» ищет правило, где описана цель *main.o*. Эта цель описана во втором правиле *make*-файла. Для этого правила запускается функция «ОбработатьПравило», которая запускает процесс обработки зависимостей (функция «ОбработатьЗависимости»). Во втором правиле указана единственная зависимость – *main.cpp*. Такой цели в *make*-файле не существует, поэтому считается, что зависимость *main.cpp* является именем файла. Далее, проверяется наличие этого файла на диске (функция «ПроверитьНаличиеФайла») – такой файл существует. На этом процесс обработки зависимостей завершается.

После обработки зависимостей функция «ОбработатьПравило» принимает решение о том, следует ли выполнять указанные в правиле команды (функция «НужноВыполнятьКоманды»). Цели правила (файла *main.o*) не существует, поэтому команды выполнять следует. Функция «ВыполнитьКоманды» запускает указанную в правиле команду (компилятор *gcc*), в результате чего создается файл *main.o*, так называемый объектный файл (*object file*).

Цель *main.o* достигнута (объектный файл *main.o* построен). Теперь *make* возвращается к обработке остальных зависимостей первого правила. Зависимости *Editor.o* и *TextLine.o* обрабатываются аналогично. Для них выполняются те же действия, что и для зависимости *main.o*.

После того, как все зависимости (*main.o*, *Editor.o* и *TextLine.o*) обработаны, решается вопрос о необходимости выполнения указанных в правиле команд (функция «НужноВыполнятьКоманды»).

Поскольку цель (*iEdit*) является именем файла, который в данный момент не существует, то принимается решение выполнить описанную в правиле команду (функция «ВыполнитьКоманды»).

Содержащаяся в правиле команда запускает компилятор *gcc*, в результате чего создается исполняемый (бинарный) файл *iEdit*. Таким образом, главная цель (*iEdit*) достигнута. На этом программа *make* завершает свою работу.

Рассмотрим еще один пример работы утилиты *make* в условиях, когда для обработки описанного выше *make*-файла будет выполнена команда

```
make clean
```

Цель явно указана в командной строке, поэтому главной целью становится абстрактная цель *clean*. Цель *clean* передается функции «ДостичьЦели». Эта функция ищет правило, которое описывает обрабатываемую цель. Это будет пятое правило *make*-файла. Для найденного правила запускается процедура обработки (функция «ОбработатьПравило»).

Поскольку в правиле не указано каких-либо зависимостей, *make* сразу переходит к этапу обработки указанных в правиле команд. Цель является именем действия, поэтому команды нужно выполнять. Указанные в правиле команды выполняются, и цель *clean*, таким образом, считается достигнутой. На этом программа *make* завершает работу.

Использование переменных

Возможность использования переменных внутри *make*-файла — очень удобное и часто используемое свойство *make*. В традиционных версиях утилиты, переменные ведут себя подобно макросам языка C. Для задания значения переменной используется оператор присваивания. Например, выражение

```
obj_list := main.o Editor.o TextLine.o
```

присваивает переменной *obj_list* значение «*main.o Editor.o TextLine.o*» (без кавычек). Пробелы между символом «:=» и началом первого слова игнорируются. Следующие за последним словом пробелы также. Значение переменной можно использовать с помощью конструкции

```
$(имя_переменной)
```

Например, при обработке такого *make*-файла

```
dir_list := . .. src/include
```

```
all:
```

```
echo $(dir_list)
```

на экран будет выведена строка

```
. .. src/include
```

Переменные могут не только содержать текстовые строки, но и «ссылаться» на другие переменные. Например, в результате обработки *make*-файла

```
optimize_flags := -O3
```

```
compile_flags := $(optimize_flags) -pipe
```

```
all:
```

```
echo $(compile_flags)
```

на экран будет выведено

```
-O3 -pipe
```

Во многих случаях использование переменных позволяет упростить *make*-файл и повысить его наглядность. Для того чтобы облегчить модификацию *make*-файла, можно разместить «ключевые» имена и списки в отдельных переменных и поместить их в начало *make*-файла:

```
program_name := iEdit
```

```
obj_list := main.o Editor.o TextLine.o
```

```
$(program_name): $(obj_list)
```

```
gcc $(obj_list) -o $(program_name)
```

```
...
```

Адаптация такого *make*-файла для сборки другой программы сведется к изменению нескольких начальных строк.

Использование автоматических переменных

Автоматические переменные – это переменные со специальными именами, которые «автоматически» принимают определенные значения перед выполнением описанных в правиле команд. Автоматические переменные можно использовать для «упрощения» записи правил. Такое, например, правило

```
iEdit: main.o Editor.o TextLine.o  
gcc main.o Editor.o TextLine.o -o iEdit
```

с использованием автоматических переменных можно записать следующим образом:

```
iEdit: main.o Editor.o TextLine.o  
gcc $^ -o $@
```

Здесь `$^` и `$@` являются автоматическими переменными. Переменная `$^` означает «список зависимостей». В данном случае при вызове компилятора `gcc` она будет ссылаться на строку «*main.o Editor.o TextLine.o*». Переменная `$@` означает «имя цели» и будет в этом примере ссылаться на имя «*iEdit*». Если бы в примере была использована следующая автоматическая переменная `$<`, то она указывала бы на первое имя зависимости, т. е. в данном случае на файл *main.o*.

Иногда использование автоматических переменных совершенно необходимо, например в шаблонных правилах.

Шаблонные правила

Шаблонные правила (*implicit rules* или *pattern rules*) – это правила, которые могут быть применены к целой группе файлов. В этом их отличие от обычных правил, описывающих отношения между конкретными файлами. Традиционные реализации *make* поддерживают так называемую «суффиксную» форму записи шаблонных правил:

```
.<расширение_файлов_зависимостей>.<расширение_файлов_целей>:  
    <команда_1>  
    <команда_2>  
    ...  
    <команда_n>
```

Например, следующее правило гласит, что все файлы с расширением «*о*» зависят от соответствующих файлов с расширением «*срр*»:

```
.срр.о:  
gcc -с $^
```


Для современной реализации *make* более предпочтительная следующая запись данной цели:

```
%o: %.cpp  
gcc -c $^ -o $@
```

Следует обратить внимание на использование автоматической переменной *\$^* для передачи компилятору имени файла-зависимости. Поскольку шаблонное правило может применяться к разным файлам, использование автоматических переменных – это единственный способ узнать для каких файлов задействуется правило в данный момент. Шаблоны правил позволяют упростить *make*-файл и сделать его более универсальным. Рассмотрим простой проектный файл:

```
iEdit: main.o Editor.o TextLine.o  
gcc $^ -o $@  
main.o: main.cpp  
gcc -c $^  
Editor.o: Editor.cpp  
gcc -c $^  
TextLine.o: TextLine.cpp  
gcc -c $^
```

Все исходные тексты программы обрабатываются одинаково: для них вызывается компилятор *gcc*. С использованием шаблонных правил этот пример можно переписать так:

```
iEdit: main.o Editor.o TextLine.o  
gcc $^ -o $@  
%.o: %.cpp  
gcc -c $^
```

Когда *make* ищет в файле проекта правило, описывающее способ достижения искомой цели (см. п. «Достижение цели», функция «Найти-Правило»), то в расчет принимаются и шаблонные правила. Для каждого из них проверяется, нельзя ли задействовать это правило для достижения искомой цели.

Пример создания более сложного *make*-файла

Предыдущие два примера создания *make*-файлов существенно упрощают создание проектов. Следует отметить, что работа по перечис-

лению всех объектных файлов, составляющих программу, может быть также автоматизирована. При этом вариант создания бинарного файла типа

```
iEdit: *.o
```

```
gcc $< -o $@
```

может не сработать, т. к. в указанном случае будут учтены только существующие в данный момент объектные файлы. Особенно это актуально в случае наличия сложных заголовочных файлов (*.h), определяющих зависимости частей проекта. Для того чтобы избежать подобного затруднения, следует использовать более сложный способ, который основан на предположении, что все файлы, содержащие исходный текст, должны быть скомпилированы и скомпонованы в результирующую программу. Такой вариант методики сборки состоит из двух шагов:

1. Получить список всех файлов с исходным текстом программы (всех файлов с расширением «сpp»). Для этого следует использовать функции обработки строк, в данном случае функцию *wildcard*, которая получает список файлов с заданным шаблоном в указанном каталоге.

2. Преобразовать список исходных файлов в список объектных файлов (заменить расширение «сpp» на расширение объектных файлов «о»). Для этого следует воспользоваться функцией *patsubst*, которая заменяет заданную подстроку в заданной строке.

Следующий пример содержит модифицированную версию *make*-файла с использованием указанных двух шагов:

```
iEdit: $(patsubst %.cpp,%.o,$(wildcard *.cpp))
```

```
gcc $^ -o $@
```

```
%.o: %.cpp
```

```
gcc -c $<
```

```
main.o: main.h Editor.h TextLine.h
```

```
Editor.o: Editor.h TextLine.h
```

```
TextLine.o: TextLine.h
```

Список объектных файлов программы строится автоматически (цель *iEdit*). Сначала с помощью функции *wildcard* получается список всех файлов с расширением «сpp», находящихся в директории проекта. Затем с помощью функции *patsubst* полученный таким образом список исходных файлов преобразуется в список объектных файлов (расширение файлов меняется с «сpp» на «о»). *Make*-файл теперь стал более универсальным.

Автоматическое построение зависимостей от заголовочных файлов

Автоматизировав перечисление объектных файлов, остается проблема перечисления зависимостей объектных файлов от заголовочных файлов. Например:

....

main.o: main.h Editor.h TextLine.h

Editor.o: Editor.h TextLine.h

TextLine.o: TextLine.h

Перечисление зависимостей «вручную» может потребовать существенного объема работы. Не всегда достаточно просто открыть файл с исходным текстом и перечислить имена всех заголовочных файлов, подключаемых с помощью директивы «*#include*»: заголовочные файлы могут включать в себя другие заголовочные файлы и подобная «цепочка зависимостей» может быть достаточно длинной. Построение списка зависимостей можно реализовать с использованием утилиты *make* и компилятора *gcc*.

Для совместной работы с *make* компилятор *gcc* имеет несколько опций:

- Ключ компиляции *M*. Для каждого файла с исходным текстом препроцессор будет выдавать на стандартный выход список зависимостей в виде правила для программы *make*. В список зависимостей попадает сам исходный файл, а также все файлы, включаемые с помощью директив «*#include <имя_файла>*» и «*#include "имя_файла"*». После запуска препроцессора компилятор останавливает работу и генерации объектных файлов не происходит.

- Ключ компиляции *MM*. Аналогичен ключу *M*, но в список зависимостей попадает только сам исходный файл и файлы, включаемые с помощью директивы «*#include "имя_файла"*».

- Ключ компиляции *MD*. Аналогичен ключу *M*, но список зависимостей выдается не на стандартный выход, а записывается в отдельный файл зависимостей. Имя этого файла формируется из имени исходного файла путем замены его расширения на расширение «*d*». Например, файл зависимостей для файла *main.cpp* будет называться *main.d*. В отличие от ключа *M* компиляция проходит обычным образом, а не прерывается после фазы запуска препроцессора.

- Ключ компиляции *MMD*. Аналогичен ключу *-MD*, но в список зависимостей попадает только сам исходный файл, и файлы, включаемые с помощью директивы «*#include "имя_файла"*».

Как видно, компилятор может работать двумя способами – в одном случае он выдает только список зависимостей и заканчивает работу (оп-

ции *M* и *MM*). В другом случае компиляция происходит как обычно, только в дополнении к объектному файлу генерируется еще и файл зависимостей (опции *MD* и *MMD*). Предпочтительней использовать второй вариант, т. к.

1) при изменении какого-либо из исходных файлов будет построен заново лишь один соответствующий ему файл зависимостей;

2) построение файлов зависимостей происходит «параллельно» с основной работой компилятора и практически не отражается на времени компиляции.

Из двух возможных опций, *MD* и *MMD*, предпочтительней первая, т. к. с помощью директивы «*#include <имя_файла>*» часто включаются не только «стандартные» (например «*#include <iostream.h>*»), но и свои собственные заголовочные файлы, которые могут иногда меняться (например, «*#include «myclass.h»*»).

После того как файлы зависимостей сформированы, они имеют расширение «*d*». Для того, чтобы сделать их доступными утилите *make*, следует использовать директиву *#include*:

```
include $(wildcard *.d)
```

Следует обратить внимание на использование функции *wildcard*, т. к. конструкция

```
include *.d
```

будет правильно работать только в том случае, если в каталоге будет находиться хотя бы один файл с расширением «*d*». Если таких файлов нет, то *make* аварийно завершится, т. к. потерпит неудачу при попытке «построить» эти файлы. Если же использовать функцию *wildcard*, то при отсутствии искомых файлов эта функция просто вернет пустую строку. Далее, директива *include* с аргументом в виде пустой строки, будет проигнорирована, не вызывая ошибки. Теперь новый вариант *makefile* из этого примера выглядит следующим образом:

```
iEdit: $(patsubst %.cpp,%.o,$(wildcard *.cpp))
```

```
gcc $^ -o $@
```

```
%.o: %.cpp
```

```
gcc -c -MD $<
```

```
include $(wildcard *.d)
```

Файлы с расширением «*d*» – это сгенерированные компилятором *gcc* файлы зависимостей. Вот, например, как выглядит файл *Editor.d*, в котором перечислены зависимости для файла *Editor.cpp*:

```
Editor.o: Editor.cpp Editor.h TextLine.h
```

Теперь при изменении любого из файлов, *Editor.cpp*, *Editor.h* или *TextLine.h*, файл *Editor.cpp* будет перекомпилирован для получения новой версии файла *Editor.o*.

Размещение файлов с исходными текстами по директориям

Приведенный выше *make*-файл вполне работоспособен и с успехом может быть использован для сборки небольших программ. Однако с увеличением размера программы, становится не очень удобным хранить все файлы с исходными текстами в одном каталоге. В таком случае предпочтительно размещать эти файлы по разным директориям, отражающим логическую структуру проекта. Для этого нужно модифицировать *make*-файл, чтобы неявное правило

```
%.o: %.cpp
```

```
gcc -c $<
```

осталось работоспособным. Для этого используют переменную *VPATH*, в которой перечисляются все директории, где могут располагаться исходные тексты. В следующем примере файлы *Editor.cpp* и *Editor.h* расположены в каталоге *Editor*, а файлы *TextLine.cpp* и *TextLine.h* в каталоге *TextLine*:

```
main.cpp
```

```
main.h
```

```
Editor /
```

```
Editor.cpp
```

```
Editor.h
```

```
TextLine /
```

```
TextLine.cpp
```

```
TextLine.h
```

```
makefile
```

Вот как выглядит *makefile* для этого примера:

```
source_dirs := Editor TextLine
```

```
search_wildcard s := $(addsuffix /*.cpp,$(source_dirs))
```

```
iEdit: $(notdir $(patsubst %.cpp,%.o,$(wildcard $(search_wildcard s))))
```

```
gcc $^ -o $@
```

```
VPATH := $(source_dirs)
```

```
%.o: %.cpp
```

```
gcc -c -MD $(addprefix -I ,$(source_dirs)) $<
```

```
include $(wildcard *.d)
```

По сравнению с предыдущим вариантом *make*-файла он претерпел следующие изменения:

- для хранения списка директорий с исходными текстами, который нужно будет указывать в нескольких местах, заведена отдельная переменная *source_dirs*;
- шаблон поиска для функции *wildcard* (переменная *search_wildcard s*) строится «динамически», исходя из списка директорий *source_dirs*;
- переменная *VPATH* используется для поиска файлов с исходными текстами;
- компилятору разрешается искать заголовочные файлы во всех директориях с исходными текстами; для этого используется функция *addprefix*, добавляющая префикс-флаг «*I*» к строке компилятора *gcc*;
- при формировании списка объектных файлов из имен исходных файлов «убирается» имя каталога, где они расположены (с помощью функции *notdir*);
- при формировании списка исходных файлов с расширением «*crr*» была использована функция *addsuffix*, добавляющая суффикс «*/*.**crr*» к названиям каталогов с исходными файлами, указанными в переменной *source_dirs*.

Сборка программы с разными параметрами компиляции

Часто возникает необходимость в получении нескольких вариантов программы, скомпилированных с использованием различным параметров. Типичным примером использования двух различных вариантов является использование отладочной и рабочей версии программы. В таких случаях следует использовать подход, при котором

1) все варианты программы собираются с помощью одного и того же *make*-файла;

2) необходимые настройки компилятора осуществляются в *make*-файле с использованием параметров, передаваемых программе *make* в командной строке (например, флаги компиляции)

```
make compile_flags="-O3 -funroll-loops -fomit-frame-  
pointer"
```

Таким образом, наиболее простым способом задания параметров компиляции будет внесение дополнительной переменной *compile_flags* в *makefile*. Если параметров компиляции несколько (строка с параметрами

содержит пробелы), то строка со значением переменной *compile_flags* должна быть заключена в кавычки. Командный файл *makefile* с использованием параметров может выглядеть следующим образом:

```
override compile_flags := -pipe
source_dirs := Editor TextLine
search_wildcard s := $(addsuffix /*.cpp,$(source_dirs))
iEdit: $(notdir $(patsubst %.cpp,%.o,$(wildcard $(search_wildcard s))))
gcc $^ $(compile_flags) -o $@
VPATH := $(source_dirs)
%.o: %.cpp
gcc -c -MD $(addprefix -I,$(source_dirs)) $(compile_flags) $<
include $(wildcard *.d)
```

Переменная *compile_flags* получает свое значение из командной строки, если оно задано, в противном случае используется значение по умолчанию, т. е. «*pipe*». Для ускорения работы компилятора к параметрам компиляции добавляется флажок *pipe*. Следует обратить внимание на необходимость использования в примере директивы *override*, использованной для изменения переменной *compile_flags* внутри *make*-файла. В противном случае переданные флаги компиляции из командной строки не будут размещены в переменной *compile_flags*.

7.4 Последовательность выполнения работы

1. Ознакомиться с теоретическим материалом.
2. Используя любой текстовый редактор, создать простейшую программу на языке C (C++) с использованием, как минимум, двух исходных файлов (с программным кодом).
3. Для автоматизации сборки проекта утилитой *make* создать *make*-файл (см. п. «Пример создания более сложного *make*-файла»).
4. Выполнить программу (скомпилировать, при необходимости отладить).
5. Показать, что при изменении одного исходного файла и последующем вызове *make* будут исполнены только необходимые команды компиляции (неизмененные файлы перекомпилированы не будут) и изменены атрибуты и/или размер объектных файлов (файлы с расширением *.o*).
6. Создать *make*-файл с высоким уровнем автоматизированной обработки исходных файлов программы согласно следующим условиям:

- имя скомпилированной программы (выполняемый или бинарный файл), флаги компиляции и имена каталогов с исходными файлами и бинарными файлами (каталоги *src*, *bin* и т. п.) задаются с помощью переменных в *makefile*;

- зависимости исходных файлов на языке *C* (*C++*) и цели в *make*-файле должны формироваться динамически;

- наличие цели *clean*, удаляющей временные файлы;

- каталог проекта должен быть структурирован следующим образом:

- *src* – каталог с исходными файлами;
- *bin* – каталог с бинарными файлами (скомпилированными);
- *makefile*.

8. ПРАКТИЧЕСКОЕ ЗАНЯТИЕ № 3. ЗНАКОМСТВО С ПОТОКАМИ И ИХ СИНХРОНИЗАЦИЕЙ В ОС *UNIX*

8.1 *Цель работы*

Ознакомиться с подсистемой управления потоками в операционной системе *Unix* и основными программными средствами для создания, управления и удаления потоков.

8.1 *Задание*

Изучить основные программные средства управления потоками ОС *Unix*, а также способы синхронизации потоков. Разработать приложения для многопоточных вычислений с использованием синхронизации посредством мьютексов, семафоров и условных переменных.

8.2 *Управление потоками*

8.2.1 Программирование потоков

В *Linux* каждый поток на самом деле является процессом, и для того, чтобы создать новый поток, нужно создать новый процесс. Однако для создания дополнительных потоков используются процессы особого типа. Эти процессы представляют собой обычные дочерние процессы главного процесса, но они разделяют с главным процессом адресное пространство, файловые дескрипторы и обработчики сигналов. Для обозначения процессов этого типа применяется специальный термин – легкие процессы (*lightweight processes*). Поскольку для потоков не требуется создавать собственную копию адресного пространства (и других ресурсов) своего процесса-родителя, создание нового легкого процесса требует значительно меньших затрат, чем создание полноценного дочернего процесса.

Спецификация POSIX 1003.1c требует, чтобы все потоки многопоточного приложения имели один идентификатор, однако в *Linux* у каждого процесса, в том числе и у процессов-потоков, есть свой идентификатор.

8.2.2 Основные функции для работы с потоками

Для работы с потоками используются следующие основные функции:

- *pthread_create* – создание потока;
- *pthread_join* – блокирование работы вызвавшего функцию процесса или потока в ожидании завершения потока;

- *pthread_cancel* – досрочное завершение потока из другого потока или процесса;
- *pthread_exit* – завершает поток, код завершения передается функции *pthread_join*. Данная функция подобна функции *exit*, однако вызов *exit* в «основном» процессе программы приведет к завершению всей программы.

8.2.3 Запуск и завершение потока

Потоки создаются функцией *pthread_create*, имеющей следующую сигнатуру:

```
int pthread_create (pthread_t* tid, pthread_attr_t* attr,
void*(*function)(void*), void* arg)
```

Данная функция определена в заголовочном файле `<pthread.h>`. Первый параметр этой функции представляет собой указатель на переменную типа *pthread_t*, которая служит идентификатором создаваемого потока. Второй параметр – указатель на переменную типа *pthread_attr_t* – используется для установки атрибутов потока. Третьим параметром функции *pthread_create* должен быть адрес функции потока. Эта функция играет для потока ту же роль, что функция *main* для главной программы. Четвертый параметр функции *pthread_create* имеет тип *void**. Этот параметр может использоваться для передачи значения в функцию потока. Вскоре после вызова *pthread_create* функция потока будет запущена на выполнение параллельно с другими потоками программы.

Новый поток запускается не сразу после вызова *pthread_create*, потому что перед тем, как запустить новую функцию потока, нужно выполнить некоторые подготовительные действия, а поток-родитель при этом продолжает выполняться. Необходимо учитывать данное обстоятельство при разработке многопоточного приложения, в противном случае возможны серьезные ошибки при выполнении программы. Если при создании потока возникла ошибка, то функция *pthread_create* возвращает ненулевое значение, соответствующее номеру ошибки.

Функция потока должна иметь сигнатуру вида

```
void* func_name(void* arg)
```

Имя функции может быть любым. Аргумент *arg* является указателем, который передается в последнем параметре функции *pthread_create*. Функция потока может вернуть значение, которое затем может быть обработано другим потоком или процессом.

Функция, вызываемая из функции потока, должна обладать свойством реентерабельности (этим же свойством должны обладать рекурсивные функции). Реентерабельная функция – это функция, которая мо-

жет быть вызвана повторно в то время, когда она уже вызвана. Такие функции используют локальные переменные (и локально выделенную память) в тех случаях, когда их нереентерабельные аналоги могут воспользоваться глобальными переменными.

Завершение функции потока происходит в следующих случаях:

- функция потока вызвала функцию *pthread_exit*;
- функция потока достигла точки выхода;
- поток был досрочно завершён другим потоком или процессом.

Функция *pthread_exit* объявлена в заголовочном файле *<pthread.h>* и ее сигнатура имеет вид:

```
void pthread_exit(void *retval)
```

Аргументом функции является указатель на возвращаемый объект. Нельзя возвращать указатель на стековый (нединамический) объект, объявленный в теле функции потока, либо на динамический объект, создаваемый и удаляемый в теле функции, т. к. после завершения потока все стековые объекты его функции удаляются. В итоге указатель будет содержать адрес памяти с неопределённым содержимым, что может привести к серьёзной ошибке.

В случае, если необходимо дождаться завершения потока в теле родительского процесса, вызывается функция *pthread_join* следующего вида:

```
int pthread_join(pthread_t th, void** thread_return)
```

Первый аргумент *th* необходим для указания ожидаемого потока, значение этого аргумента определяется в результате выполнения функции *pthread_create*. В качестве второго аргумента *thread_return* выступает указатель на аргумент функции *pthread_exit* либо *NULL*, если поток ничего не возвращает.

8.2.4 Досрочное завершение потока

Функции потоков можно рассматривать как вспомогательные программы, находящиеся под управлением функции *main*. Точно так же, как при управлении процессами иногда возникает необходимость досрочно завершить процесс, многопоточной программе может понадобиться досрочно завершить один из потоков. Для досрочного завершения потока можно воспользоваться функцией *pthread_cancel*:

```
int pthread_cancel(pthread_t thread)
```

Единственным аргументом этой функции является идентификатор потока – *thread*. Функция *pthread_cancel* возвращает 0 в случае успеха и ненулевое значение (код ошибки) в случае ошибки.

Несмотря на то, что *pthread_cancel* может завершить поток досрочно, ее нельзя назвать средством принудительного завершения потоков. В теле функции потока можно не только самостоятельно выбрать порядок завершения в ответ на вызов *pthread_cancel*, но и вовсе игнорировать этот вызов. Поэтому вызов функции *pthread_cancel* следует рассматривать как запрос на выполнение досрочного завершения потока.

Функция *pthread_setcancelstate* определяет, будет ли поток реагировать на обращение к нему с помощью *pthread_cancel* или не будет. Сигнатура функции имеет вид

```
int pthread_setcancelstate(int state, int* oldstate)
```

Аргумент *state* может принимать два значения:

- *PTHREAD_CANCEL_DISABLE* – запрет завершения потока;
- *PTHREAD_CANCEL_ENABLE* – разрешение на завершение потока.

Во второй аргумент *oldstate* записывается указатель на предыдущее значение аргумента *state*. С помощью функции *pthread_setcancelstate* можно указывать участки кода потока, во время исполнения которых поток нельзя завершить вызовом функции *pthread_cancel*:

```
//...
// участок функции, который можно досрочно завершить
//...
pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);
//...
// участок функции, который нельзя досрочно завершить
//...
pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
//...
// участок функции, который можно досрочно завершить
//...
```

Функция *pthread_testcancel* создает точку возможного досрочного завершения потока (точку отмены). Такие точки необходимы для корректного завершения потока, т. к. даже если досрочное завершение разрешено, поток, получивший запрос на досрочное завершение, часто может завершить работу не сразу. Если поток находится в режиме отложенного досрочного завершения (именно этот режим установлен по умолчанию), он выполнит запрос на досрочное завершение, только достигнув одной из точек отмены. Сигнатура функции *pthread_testcancel*

```
void pthread_testcancel()
```

В соответствии со стандартом *POSIX* точками отмены являются вызовы многих «обычных» функций, например *open*, *pause* и *write*.

Тем не менее, выполнение потока может быть прервано принудительно, не дожидаясь точек отмены. Для этого необходимо перевести поток в режим немедленного завершения, что делается с помощью вызова функции

```
pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, NULL)
```

Вызов функции

```
pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, NULL)
```

снова переводит поток в режим отложенного досрочного завершения.

8.2.5 Синхронизация потоков

При выполнении нескольких потоков во многих случаях необходимо синхронизировать их взаимодействие. Существует несколько способов синхронизации потоков:

- взаимные исключения – мьютексы (Mutex – MUTual EXclusion);
- переменные состояния;
- семафоры.

Механизм использования переменных состояния и семафоров в многопоточных приложениях аналогичен механизму использования этих методов синхронизации в многопроцессных приложениях.

Механизм мьютексов представляет общий метод синхронизации выполнения потоков. Мьютекс можно определить как объект синхронизации, который устанавливается в особое сигнальное состояние, когда не занят каким-либо потоком. В любой момент мьютексом может владеть только один поток.

Использование мьютексов гарантирует, что только один поток в некоторый момент времени выполняет критическую секцию кода. Мьютексы можно использовать и в однопоточном коде.

Доступны следующие действия с мьютексом: инициализация, удаление, захват или открытие, попытка захвата.

Объекты синхронизации потоков являются переменными в памяти процесса и обладают живучестью объектов процесса. Потоки в различных процессах могут связаться друг с другом через объекты синхронизации, помещенные в разделяемую память потоков, даже в случае, когда потоки в различных процессах невидимы друг для друга.

Объекты синхронизации можно также разместить в файлах, где они будут существовать независимо от создавшего их процесса.

Необходимость в синхронизации потоков возникает в следующих случаях:

1. Если синхронизация – это единственный способ гарантировать последовательность разделяемых (общих) данных.

2. Если потоки в двух или более процессах могут использовать единственный объект синхронизации совместно. При этом объект синхронизации должен инициализироваться только одним из взаимодействующих процессов, потому что повторная инициализация объекта синхронизации устанавливает его в открытое состояние.

3. Если синхронизация может гарантировать достоверность изменяющихся данных.

4. Если процесс может отобразить файл и существует поток в этом процессе, который получает уникальный доступ к записям. Как только установлена блокировка, любой другой поток в любом процессе, отображающем файл, который пытается установить блокировку, блокируется, пока запись в файл не будет закончена.

8.3 Мьютексы

8.3.1 Функции синхронизации потоков с использованием мьютексов

Для синхронизации потоков с использованием мьютексов используются следующие основные функции:

- *pthread_mutex_init* – инициализирует взаимоисключающую блокировку;
- *pthread_mutex_destroy* – удаляет взаимоисключающую блокировку;
- *pthread_mutex_lock* – устанавливает блокировку. В случае, если блокировка была установлена другим потоком, текущий поток останавливается до снятия блокировки другим процессом;
- *pthread_mutex_unlock* – снимает блокировку.

8.3.2 Инициализация и удаление объекта атрибутов мьютекса

Атрибуты мьютекса могут быть связаны с каждым потоком. Чтобы изменить атрибуты мьютекса по умолчанию, можно объявить и инициализировать объект атрибутов мьютекса, а затем изменить определенные значения. Часто атрибуты мьютекса устанавливаются в одном месте в

начале приложения, чтобы быстро найти и изменить их. После того, как сформированы атрибуты мьютекса, можно его инициализировать.

Функция

```
int pthread_mutexattr_init (pthread_mutexattr_t* mattr)
```

используется, чтобы инициализировать объект атрибутов *mattr* значениями по умолчанию. Память для каждого объекта атрибутов выделяется системой поддержки потоков во время выполнения.

Пример вызова функции *pthread_mutexattr_init*:

```
#include <pthread.h>
pthread_mutexattr_t mattr;
int ret;
/* инициализация атрибутов значениями по умолчанию */
ret = pthread_mutexattr_init(&mattr);
```

Для корректного удаления объекта атрибутов, созданного с помощью функции *pthread_mutexattr_init*, необходимо вызвать функцию *pthread_mutexattr_destroy*. В противном случае возможна утечка памяти, так как тип *pthread_mutexattr_t* является закрытым. Она возвращает 0 после успешного завершения или другое значение, если произошла ошибка. Пример вызова функции *pthread_mutexattr_destroy*:

```
#include <pthread.h>
pthread_mutexattr_t mattr;
int ret;
/* удаление атрибутов */
ret = pthread_mutexattr_destroy(&mattr);
```

8.3.3 Область видимости мьютекса

Областью видимости мьютекса может быть либо некоторый процесс, либо вся система. В первом случае оперировать мьютексом могут только потоки, созданные процессом, в котором создан и мьютекс. Во втором случае мьютекс существует в разделяемой памяти и может быть разделен среди потоков нескольких процессов. По умолчанию мьютекс создается в области видимости процесса и обладает живучестью процесса.

Чтобы установить область видимости атрибутов мьютекса используется функция

```
int pthread_mutexattr_setpshared(pthread_mutexattr_t* mattr, int)
```

Первый аргумент *mutexattr* является указателем на объект атрибутов, для которого устанавливается область видимости. Вторым аргументом является константой, обозначающей устанавливаемую область видимости: *PTHREAD_PROCESS_PRIVATE* для области видимости процесса и *PTHREAD_PROCESS_SHARED* для области видимости системы. Мьютекс, созданный в области видимости системы, должен существовать в разделяемой памяти.

Пример вызова функции *pthread_mutexattr_setpshared*:

```
#include <pthread.h>

pthread_mutexattr_t mutexattr;

int ret;

ret = pthread_mutexattr_init(&mutexattr);

/* переустановка на значение по умолчанию: private */
ret = pthread_mutexattr_setpshared
(&mutexattr, PTHREAD_PROCESS_PRIVATE);
```

Функция

*pthread_mutexattr_getpshared(pthread_mutexattr_t *mutexattr, int *pshared)*

используется для получения текущей области видимости мьютекса потока:

```
#include <pthread.h>

pthread_mutexattr_t mutexattr;

int pshared, ret;

/* получить атрибут pshared для мьютекса */
ret = pthread_mutexattr_getpshared(&mutexattr, &pshared);
```

8.3.4 Инициализация мьютекса

Функция *pthread_mutex_init* предназначена для инициализации мьютекса

```
int pthread_mutex_init(pthread_mutex_t *mp, const pthread_mutexattr_t *mutexattr).
```

Мьютекс, на который указывает первый аргумент *mp*, инициализируется значением по умолчанию, если вторым аргументом *mutexattr* равен *NULL*, или определенными атрибутами, которые уже установлены с помощью *pthread_mutexattr_init*.

Функция *pthread_mutex_init* возвращает 0 после успешного завершения или другое значение, если произошла ошибка. Пример использования функции *pthread_mutexattr_init*:


```
#include <pthread.h>
pthread_mutex_t mp = PTHREAD_MUTEX_INITIALIZER;
pthread_mutexattr_t mattr;
int ret;
/* инициализация мьютекса значением по умолчанию */
ret = pthread_mutex_init(&mp, NULL);
```

Когда мьютекс инициализируется, он находится в открытом (разблокированном) состоянии. Статически определенные мьютексы могут инициализироваться непосредственно значениями по умолчанию с помощью константы *PTHREAD_MUTEX_INITIALIZER*. Пример инициализации:

```
/* инициализация атрибутов мьютекса по умолчанию*/
ret = pthread_mutexattr_init(&mattr);
/* смена значений mattr с помощью функций */
ret = pthread_mutexattr_setpshared
(&mattr, PTHREAD_PROCESS_SHARED);
/* инициализация мьютекса произвольными значениями */
ret = pthread_mutex_init(&mp, &mattr);
```

8.3.5 Запирание мьютекса

Функция *pthread_mutex_lock* используется для запирания или захвата мьютекса. Аргументом функции является адрес запираемого мьютекса. Если мьютекс уже заперт, вызывающий поток блокируется и мьютекс ставится в очередь приоритетов. Когда происходит возврат из *pthread_mutex_lock*, мьютекс запирается, а вызывающий поток становится его владельцем. Функция *pthread_mutex_lock* возвращает 0 после успешного завершения, или другое значение, если произошла ошибка. Пример вызова:

```
#include <pthread.h>
pthread_mutex_t mp;
int ret;
ret = pthread_mutex_lock(&mp);
```

Для открытия (разблокировки) мьютекса используется функция *pthread_mutex_unlock*. При этом мьютекс должен быть закрыт, а вызывающий поток должен быть владельцем мьютекса, то есть тем, кто его запер. Пока любые другие потоки ждут доступа к мьютексу, его по-

ток-владелец, находящийся в начале очереди, не блокирован. Функция *pthread_mutex_unlock* возвращает 0 после успешного завершения или другое значение, если произошла ошибка. Пример вызова:

```
#include <pthread.h>

pthread_mutex_t mp;

int ret;

ret = pthread_mutex_unlock(&mp);
```

Существует способ захвата мьютекса без блокирования потока. Функция *pthread_mutex_trylock* пытается провести записание мьютекса. Она является неблокирующей версией *pthread_mutex_lock*. Если мьютекс уже закрыт, вызов возвращает ошибку, однако поток, вызвавший эту функцию, не блокируется. В противном случае мьютекс закрывается, а вызывающий процесс становится его владельцем. Функция *pthread_mutex_trylock* возвращает 0 после успешного завершения или другое значение, если произошла ошибка. Пример вызова:

```
#include <pthread.h>

pthread_mutex_t mp;

int ret;

ret = pthread_mutex_trylock(&mp);
```

Захват через мьютекс не должен повторно инициализироваться или удаляться, пока другие потоки могут его использовать. Если мьютекс инициализируется повторно или удаляется, приложение должно убедиться, что в настоящее время этот мьютекс не используется.

8.3.6 Удаление мьютекса

Функция *pthread_mutex_destroy* используется для удаления мьютекса в любом состоянии. Функция *pthread_mutex_destroy* возвращает 0 после успешного завершения или другое значение, если произошла ошибка. Пример вызова:

```
#include <pthread.h>

pthread_mutex_t mp;

int ret;

ret = pthread_mutex_destroy(&mp);
```

Иерархия блокировок

Иногда может возникнуть необходимость одновременного доступа к нескольким ресурсам. При этом возникает проблема, заключающаяся в

том, что два потока пытаются захватить оба ресурса, но запирают соответствующие мьютексы в различном порядке.

В приведенном ниже примере два потока запирают мьютексы 1 и 2 и возникает тупик при попытке запереть один из мьютексов.

Таблица 3.2

Пример «тупика» для двух потоков

Поток 1	Поток 2
<pre>/* использует ресурс 1 */ pthread_mutex_lock(&m1); /* теперь захватывает ресурсы 2+1 */ pthread_mutex_lock(&m2);</pre>	<pre>/* использует ресурс 2 */ pthread_mutex_lock(&m2); /* теперь захватывает ресурсы 1+2 */ pthread_mutex_lock(&m1);</pre>

Наилучшим способом избежать проблем является записание нескольких мьютексов в одном и том же порядке во всех потоках. Эта техника называется иерархией блокировок: мьютексы упорядочиваются путем назначения каждому своего номера. После этого придерживаются правила – если мьютекс с номером n уже заперт, то нельзя запираить мьютекс с номером меньше n .

Если блокировка всегда выполняется в указанном порядке, тупик не возникнет. Однако эта техника может использоваться не всегда, поскольку иногда требуется запираить мьютексы в порядке, отличном от порядка их номеров.

Чтобы предотвратить тупик в этой ситуации, лучше использовать функцию `pthread_mutex_trylock`. Один из потоков должен освободить свой мьютекс, если он обнаруживает, что может возникнуть тупик.

Ниже проиллюстрировано использование условной блокировки:

```
// Поток 1:
pthread_mutex_lock(&m1);
pthread_mutex_lock(&m2);
/* обработка */
/* нет обработки */
pthread_mutex_unlock(&m2);
pthread_mutex_unlock(&m1);

// Поток 2:
for (; ;) {
pthread_mutex_lock(&m2);
```

```

if (pthread_mutex_trylock(&m1)==0)
/* захват! */
break;
/* мьютекс уже заперт */
pthread_mutex_unlock(&m2);
}
/* нет обработки */
pthread_mutex_unlock(&m1);
pthread_mutex_unlock(&m2);

```

В примере выше поток 1 запирает мьютексы в нужном порядке, а поток 2 пытается закрыть их по-своему. Чтобы убедиться, что тупик не возникнет, поток 2 должен аккуратно обращаться с мьютексом 1; если поток блокировался, ожидая мьютекс, который будет освобожден, он, вероятно, только что вызвал тупик с потоком 1. Чтобы гарантировать, что это не случится, поток 2 вызывает *pthread_mutex_trylock*, который запирает мьютекс, если тот свободен. Если мьютекс уже заперт, поток 2 получает сообщение об ошибке. В этом случае поток 2 должен освободить мьютекс 2, чтобы поток 1 мог запереть его, а затем освободить оба мьютекса.

Синхронизация с использованием семафора

Семафор предназначен для синхронизации потоков по действиям и данным, и в общем случае способ использования семафора сходен со способом использования мьютексов.

Семафор (S) – это защищенная переменная, значения которой можно опрашивать и менять только при помощи специальных операций P(S) и V(S) и операции инициализации. Семафор может принимать целое неотрицательное значение. При выполнении потоком операции P над семафором S значение семафора уменьшается на 1 при $S > 0$ или поток блокируется, «ожидая на семафоре», при $S = 0$. При выполнении операции V(S) происходит пробуждение одного из потоков, ожидающих на семафоре S, а если таковых нет, то значение семафора увеличивается на 1. В простом случае, когда семафор работает в режиме 2-х состояний ($S > 0$ и $S = 0$), его алгоритм работы полностью совпадает с алгоритмом мьютекса.

Как следует из вышесказанного, при входе в критическую секцию поток должен выполнять операцию P(S), а при выходе из критической секции операцию V(S).

Прототипы функций для манипуляции с семафорами описываются в файле `<semaphore.h>`. Ниже приводятся прототипы функций вместе с пояснением их синтаксиса и выполняемых ими действий:

- `int sem_init(sem_t* sem, int pshared, unsigned int value)` – инициализация семафора `sem` значением `value`. В качестве `pshared` всегда необходимо указывать 0;
- `int sem_wait(sem_t* sem)` – «ожидание на семафоре». Выполнение потока блокируется до тех пор, пока значение семафора не станет положительным. При этом значение семафора уменьшается на 1;
- `int sem_post(sem_t* sem)` – увеличивает значение семафора `sem` на 1;
- `int sem_destroy(sem_t* sem)` – уничтожает семафор `sem`;
- `int sem_trywait(sem_t* sem)` – неблокирующий вариант функции `sem_wait`. При этом вместо блокировки вызвавшего потока функция возвращает управление с кодом ошибки в качестве результата работы.

Синхронизация с использованием условной переменной

Условная переменная позволяет потокам ожидать выполнения некоторого условия (события), связанного с разделяемыми данными. Над условными переменными определены две основные операции: информирование о наступлении события и ожидание события. При выполнении операции «информирование» один из потоков, ожидающих значения условной переменной, возобновляет свою работу.

Условная переменная всегда используется совместно с мьютексом. Перед выполнением операции «ожидание» поток должен заблокировать мьютекс. При выполнении операции «ожидание» указанный мьютекс автоматически разблокируется. Перед возобновлением ожидающего потока выполняется автоматическая блокировка мьютекса, позволяющая потоку войти в критическую секцию, после критической секции рекомендуется разблокировать мьютекс. При подаче сигнала другим потокам рекомендуется функцию «сигнализации» так же защитить мьютексом.

Прототипы функций для работы с условными переменными содержатся в файле `pthread.h`. Ниже приводятся прототипы функций вместе с пояснением их синтаксиса и выполняемых ими действий:

- `pthread_cond_init(pthread_cond_t* cond, const pthread_condattr_t* attr)` – инициализирует условную переменную `cond` с указанными атрибутами `attr` или с атрибутами по умолчанию (при указании 0 в качестве `attr`);
- `int pthread_cond_destroy(pthread_cond_t* cond)` – уничтожает условную переменную `cond`;

- *int pthread_cond_signal(pthread_cond_t* cond)* – информирование о наступлении события потоков, ожидающих на условной переменной *cond*;
- *int pthread_cond_broadcast(pthread_cond_t* cond)* – информирование о наступлении события потоков, ожидающих на условной переменной *cond*. При этом возобновлены будут все ожидающие потоки;
- *int pthread_cond_wait(pthread_cond_t* cond, pthread_mutex_t* mutex)* – ожидание события на условной переменной *cond*.

Рассмотренных средств достаточно для решения разнообразных задач синхронизации потоков. Вместе с тем они обеспечивают взаимоисключение на низком уровне и не наполнены семантическим смыслом. При непосредственном их использовании легко допустить ошибки различного рода: забыть выйти из критической секции, использовать примитив не по назначению, реализовать вложенное использование примитива и т. д. При этом операции с мьютексами, семафорам и условными переменными оказываются разбросанными по всему программному коду приложения, что повышает вероятность появления ошибки и усложняет ее поиск и устранение.

8.3.7 Компиляция многопоточной программы

Для компиляции и сборки многопоточной программы необходимо иметь следующее:

- стандартный компилятор C (cc, gcc, g++ и т. д.);
- файлы заголовков: `<thread.h>`, `<pthread.h>`, `<errno.h>`, `<limits.h>`, `<signal.h>`, `<unistd.h>`;
- библиотеку реализации потоков (*libpthread*);
- другие библиотеки, совместимые с многопоточными приложениями (*libc*, *libm*, *libw*, *libintl*, *libnsl*, *libsocket*, *libmalloc*, *libmapmalloc* и др.).

Файл заголовка `<pthread.h>`, используемый с библиотекой *pthread*, компилирует код, который является совместимым с интерфейсами многопоточности, определенными стандартом POSIX 1003.1c.

Для компиляции программы, использующей потоки и реентерабельные системные функции, необходимо дополнительно указать в строке вызова компилятора следующие аргументы:

`-D_REENTRANT -lpthread`.

Команда компиляции `D` включает макрос `_REENTRANT`. Этот макрос указывает, что вместо обычных функций стандартной библиотеки к программе должны быть подключены их реентерабельные аналоги. Реентерабельный вариант библиотеки *glibc* написан таким образом, чтобы реализованные в ней реентерабельные функции как можно меньше отличались от их обычных аналогов. Также в строке вызова компилятора

ра могут дополнительно указываться пути для поиска заголовочных файлов (ключ «I») и путь для поиска библиотек (ключ L). Для компилятора указывается «/», что программа должна быть связана с библиотекой `libpthread`, которая содержит все специальные функции, необходимые для работы с потоками.

8.3.8 Особенности отладки многопоточной программы

Отладка многопоточной программы сложнее, чем отладка однопоточной. Ниже приведены наиболее типичные ошибки исходного кода, которые могут вызвать ошибки исполнения в многопоточных программах.

1. Доступ к глобальной памяти без использования механизмов синхронизации.

2. Создание тупиков, вызванных двумя потоками, пробующими получить права на одну и ту же пару глобальных ресурсов в различном порядке (один поток управляет одним ресурсом, а второй управляет другим ресурсом и ни один не может продолжать выполнение до освобождения нужного им ресурса).

3. Попытка повторно получить доступ к уже захваченному ресурсу (рекурсивный тупик).

4. Создание скрытого промежутка при синхронизации. Это происходит, когда сегмент кода, защищенный механизмом синхронизации, содержит вызов функции, которая освобождает и затем повторно создает механизм синхронизации прежде, чем управление возвращается к вызывающему потоку. В результате вызывающему «кажется», что глобальные данные были защищены, хотя это не так.

5. Невнимание к тому факту, что потоки по умолчанию создаются с типом `PTHREAD_CREATE_JOINABLE` и ресурсы таких потоков должны быть утилизированы (возвращены родительскому процессу) посредством вызова функции `pthread_join`. Обратите внимание, что `pthread_exit` не освобождает выделенную память.

6. Создание глубоко вложенных, рекурсивных обращений и использование больших автоматических массивов может вызвать проблемы, потому что многопоточные программы имеют более ограниченный размер стека, чем однопоточные.

7. Определение неадекватного размера стека или использование стека не по умолчанию.

8.3.9 Примеры практической реализации

Пример программы с использованием потока:

```
#include <stdlib.h>
```

```

#include <stdio.h>
#include <pthread.h>
int i = 0;
void* thread_func(void *arg) {
    pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);
    for (i=0; i < 4; i++) {
        printf("I'm still running!\n");
        sleep(1);
    }
    pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
    pthread_testcancel();
    printf("YOU WILL NOT STOP ME!!!\n");
}
int main(int argc, char * argv[]) {
    pthread_t thread;
    pthread_create(&thread, NULL, thread_func, NULL);
    while (i < 1) sleep(1);
    pthread_cancel(thread);
    printf("Requested to cancel the thread\n");
    pthread_join(thread, NULL);
    printf("The thread is stopped.\n");
    return EXIT_SUCCESS;
}

```

Пример реализации многопоточной программы:

```

#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>
#define NUM_THREADS 6
void *thread_function(void *arg);
int main() {
    int res;

```



```

int lots_of_threads;
pthread_t a_thread[NUM_THREADS];
void *thread_result;
srand ( (insigned)time(NULL) );
for (lots_of_threads = 0; lots_of_threads < NUM_THREADS;
lots_of_threads++)
{
    res = pthread_create (&a_thread[lots_of_threads],NULL,
        thread_function, (void *)&lots_of_threads);
    if (res != 0) {
        perror("Thread creation failed");
        exit(EXIT_FAILURE);
    }
    sleep(1);
}
printf("Waiting for threads to finish...\n");
for (lots_of_threads = NUM_THREADS - 1; lots_of_threads >= 0;
lots_of_threads--)
{
    res = pthread_join (a_thread[lots_of_threads],&thread_result);
    if (res == 0) printf("Picked up a thread\n");
    else perror("pthread_join failed");
}
printf("All done\n");
exit(EXIT_SUCCESS);
}

void *thread_function(void *arg) {
    int my_number = *(int *)arg;
    int rand_num;
    printf ("thread_function is running. Argument was %d\n",
my_number);
    rand_num=1+(int)(9.0*rand()/(RAND_MAX+1.0));

```

```

        sleep(rand_num);
        printf ("Bye from %d\n", my_number);
        pthread_exit(NULL);
    }

```

Пример использования мьютекса для контроля доступа к переменной. В приведенном ниже коде функция *increment_count* использует мьютекс, чтобы гарантировать атомарность (целостность) модификации разделяемой переменной *count*. Функция *get_count()* использует мьютекс, чтобы гарантировать, что переменная *count* атомарно считывается.

```

#include <pthread.h>

pthread_mutex_t count_mutex;
long count;

void increment_count() {
    pthread_mutex_lock(&count_mutex);
    count = count + 1;
    pthread_mutex_unlock(&count_mutex);
}

long get_count() {
    long c;
    pthread_mutex_lock(&count_mutex);
    c = count;
    pthread_mutex_unlock(&count_mutex);
    return (c);
}

```

Пример многопоточной программы с синхронизацией с использованием мьютексов:

```

#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
#include <math.h>
#define SIZE_I 2
#define SIZE_J 2

```

```

float X[SIZE_I][SIZE_J];
float S[SIZE_I][SIZE_J];
int count = 0; // глобальный счетчик
struct DATA_ {
    double x;
    int i;
    int z;
};
typedef struct DATA_ DATA;
pthread_mutex_t lock; //Исключающая блокировка
// Функция для вычислений
double f(double x) { return x*x; }
// Потокосная функция для вычислений
void *calc_thr (void *arg) {
    DATA* a = (DATA*) arg;
    X[a->i][a->z] = f(a->x);
    // установка блокировки
    pthread_mutex_lock(&lock);
    // изменение глобальной переменной
    count ++;
    // снятие блокировки
    pthread_mutex_unlock(&lock);
    delete a;
    return NULL;
}
// Потокосная функция для ввода
void *input_thr(void *arg) {
    DATA* a = (DATA*) arg;
    printf("S[%d][%d]:", a->i, a->z);
    scanf("%f", &S[a->i][a->z]);
    delete a;
    return NULL;
}

```

```

int main() {
//массив идентификаторов потоков
pthread_t thr[ SIZE_I * SIZE_J ];
// инициализация мьютекса
pthread_mutex_init(&lock, NULL);
DATA *arg;
// Ввод данных для обработки
for (int i=0; i<SIZE_I; i++) {
    for (int z=0; z<SIZE_J; z++) {
        arg = new DATA;
        arg->i = i;
        arg->z = z;
// создание потока для ввода элементов матрицы
pthread_create (&thr[ i* SIZE_J + z ], NULL, input_thr, (void *)arg);
    } // for (int z=0; z<SIZE_J; P ++z)
} // for (int i=0; i<SIZE_I; P ++i)
// Ожидание завершения всех потоков ввода данных
// идентификаторы потоков хранятся в массиве thr
for(int i = 0; i < SIZE_I*SIZE_J; i++) pthread_join (thr[i], NULL);
// Вычисление элементов матрицы
pthread_t thread;
printf("Start calculation\n");
for (int i=0; i<SIZE_I; i++) {
    for (int z=0; z<SIZE_J; z++) {
        arg = new DATA;
        arg->i = i;
        arg->z = z;
        arg->x = S[i][z];
// создание потока для вычислений
pthread_create (&thread, NULL, calc_thr, (void *)arg);
// перевод потока в отсоединенный режим
pthread_detach(thread);
// for (int z=0; z<SIZE_J; z++)

```

```

}    // for (int i=0;i<SIZE_I; i++)
do {
// Основной процесс "засыпает" на 1с
sleep(1);
// Проверка состояния вычислений
printf("finished %d threads.\n", count);
} while ( count < SIZE_I*SIZE_J);
// Вывод результатов
for (int i=0;i<SIZE_I; i++) {
    for (int z=0; z<SIZE_J; z++) {
        printf("X[%d][%d] = %f\t", i, z, X[i][z]);
    }
    printf("\n");
}
// удаление мьютекса
pthread_mutex_destroy(&lock);
return 0;
}

```

Пример многопоточной программы с синхронизацией семафорами:

```

#include "main.h"
#include <iostream.h>
#include <semaphore.h>
#include <fstream.h>
#include <stdio.h>
#include <error.h>
void* WriteToFile(void*);
int errno;
sem_t psem;
ofstream qfwrite;

int main(int argc, char **argv) {
pthread_t tidA,tidB;
int n;

```

```

char filename[] = "./result.txt";
qfwrite.open(&filename[0]);
sem_init(&psem,0,0);
sem_post(&psem))
    pthread_create(&tidA,NULL,&WriteToFile,(void*)100));
    pthread_create(&tidB,NULL,&WriteToFile,(void*)100));
    pthread_join(tidA,NULL));
    pthread_join(tidB,NULL));
sem_destroy(&psem));
qfwrite.close();
}
void* WriteToFile(void *f){
int max = (int)f;
for (int i=0; i<=max; i++)
{
    sem_wait(&psem);
    qfwrite<<pthread_self()<<"-writetofilecounter i="<<i<<endl;
    qfwrite<<flush;
    sem_post(&psem);
}
return NULL;
}

```

Пример многопоточной программы с синхронизацией с использованием условных переменных. Ниже приведен фрагмент программы, использующей семафоры для синхронизации записи (*writer*) и чтения (*reader*) данных в буфер *data* и из него, емкость буфера – одна запись.

```

#include "main.h"
#include <iostream.h>
#include <semaphore.h>
#include <fstream.h>
#include <stdio.h>
#include <error.h>
...

```

```

int full;
pthread_mutex_t mx;
pthread_cond_t cond;
int data;
void *writer(void *)
{
    while(1)
    {
        int t= write_to_buffer ();
        pthread_mutex_lock(&mx)
        while(full) {
            pthread_cond_wait(&cond, &mx);
        }
        data=t;
        full=1;
        pthread_cond_signal(&mx);
        pthread_mutex_unlock(&mx);
    }
    return NULL;
}

```

```

void * reader(void *)
{
    while (1)
    {
        int t;
        pthread_mutex_lock(&mx);
        while (!full)
        {
            pthread_cond_wait(&cond, &mx);
        }
        t=data;
        full=0;
    }
}

```

```

        pthread_cond_signal(&mx);
        pthread_mutex_unlock(&mx);
        read_from_buffer();
    }
    return NULL;
}
...

```

8.4 Последовательность выполнения работы

1. Ознакомиться с теоретическим материалом.
2. Разработать три многопоточные программы с использованием минимум двух потоков и различных средств синхронизации. Например: два потока записывают и читают информацию из одного файла; два потока увеличивают значение общей переменной; два потока с различной частотой считывают и записывают данные в общий буфер памяти.
3. Обеспечить синхронизированную работу потоков в критической секции с использованием:
 - мьютексов;
 - семафоров;
 - условных переменных.
4. Убедиться в результативности применения средств синхронизации потоков, сравнив результаты работы программ с использованием и без использования средств синхронизации.

9. ПРАКТИЧЕСКОЕ ЗАНЯТИЕ № 4. ЗНАКОМСТВО С ПРОЦЕССАМИ, ПЕРЕДАЧЕЙ ДАННЫХ МЕЖДУ ПРОЦЕССАМИ И ИХ СИНХРОНИЗАЦИЕЙ

9.1 *Цель работы*

Практическое знакомство с объектом процесс, основными механизмами передачи данных между процессами, а также синхронизацией взаимодействующих процессов в ОС *Unix*.

9.2 *Задание*

Изучить базовые возможности оболочки *bash* ОС *Unix* по управлению процессами (заданиями). Разработать приложения реализующие схему «клиент» – «сервер» с использованием средств межпроцессного взаимодействия: семафоров, разделяемой памяти, программных каналов и одной очереди сообщений.

9.3 *Основы оперирования процессами в оболочке bash*

9.3.1 *Задания и процессы*

Многие командные оболочки (включая *bash* и *tcsh*) имеют функции управления заданиями (*job control*). Управление заданиями позволяет запускать одновременно несколько команд или заданий (*jobs*) и осуществлять управление ими. Прежде чем говорить об этом более подробно, следует рассмотреть понятие процесс (*process*).

Каждый раз при запуске программы стартует некоторый процесс. Вывести список протекающих в настоящее время процессов можно командой *ps*, например, следующим образом:

```
/home/larry# ps
PID TT STAT TIME COMMAND
 24 3 S   0:03 (bash)
161 3 R   0:00 ps
/home/larry#
```

Номера процессов (*process ID*, или *PID*), указанные в первой колонке, являются уникальными номерами, которые система присваивает каждому работающему процессу. Последняя колонка, озаглавленная *COMMAND*, указывает имя работающей команды. В данном случае в списке указаны процессы, которые запустил сам пользователь *larry*. В системе работает еще много других процессов, их полный список можно

выдать командой *ps -aux*. Однако среди команд, запущенных пользователем *larry*, есть только *bash* (командная оболочка для пользователя *larry*) и сама команда *ps*. Видно, что оболочка *bash* работает одновременно с командой *ps*. Когда пользователь ввел команду *ps*, оболочка *bash* начала ее исполнять. После того, как команда *ps* закончила свою работу (таблица процессов выведена на экран), управление возвращается процессу *bash*.

Работающий процесс также называют заданием (*job*). Здесь и далее не будем делать различия между этими понятиями. Следует отметить, что обычно процесс называют «заданием», когда имеют ввиду управление заданием (*job control*) – это функция командной оболочки, которая предоставляет пользователю возможность переключаться между несколькими заданиями.

В большинстве случаев пользователи запускают только одно задание – это та команда, которую они ввели последней в командной оболочке. Однако, используя свойство управления заданиями, можно запустить сразу несколько заданий и, по мере надобности, переключаться между ними.

Управление заданиями может быть полезно, если, например, вы редактируете большой текстовый файл и хотите временно прервать редактирование, чтобы сделать какую-нибудь другую операцию. С помощью функций управления заданиями можно временно покинуть редактор, вернуться к приглашению командной оболочки и выполнить какие-либо другие действия. Когда они будут сделаны, можно вернуться обратно к работе с редактором и обнаружить его в том же состоянии, в котором он был покинут. У функций управления заданиями есть еще много полезных применений.

9.3.2 Передний план и фоновый режим

Задания могут быть либо на переднем плане (*foreground*), либо фоновыми (*background*). На переднем плане в любой момент времени может быть только одно задание. Задание на переднем плане – это то задание, с которым происходит взаимодействие пользователя; оно получает ввод с клавиатуры и посылает вывод на экран (если ввод или вывод не перенаправили куда-либо еще). Напротив, фоновые задания не получают ввода с терминала; как правило, такие задания не нуждаются во взаимодействии с пользователем.

Некоторые задания исполняются очень долго и во время их работы не происходит ничего интересного. Пример таких заданий – компилирование программ, а также сжатие больших файлов. Нет никаких причин смотреть на экран и ждать, когда эти задания выполнятся. Такие задания

следует пускать в фоновом режиме. В это время можно работать с другими программами.

Задания также можно (временно) приостанавливать (*suspend*). Затем приостановленному заданию можно дать указание продолжать работу на переднем плане или в фоновом режиме. При возобновлении исполнения приостановленного задания его состояние не изменяется – задание продолжает выполняться с того места, где его остановили.

Прерывание задания – действие отличное от приостановки задания. При прерывании (*interrupt*) задания процесс погибает. Прерывание заданий обычно осуществляется нажатием соответствующей комбинации клавиш, обычно это *Ctrl-C*. Восстановить прерванное задание никаким образом невозможно. Следует также знать, что некоторые программы перехватывают команду прерывания, так что нажатие комбинации клавиш *Ctrl-C* может не прервать процесс немедленно. Это сделано для того, чтобы программа могла уничтожить следы своей работы прежде, чем она будет завершена. На практике некоторые программы прервать таким способом нельзя.

9.3.3 Перевод заданий в фоновый режим и уничтожение заданий

Начнем с простого примера. Рассмотрим команду *yes*, которая на первый взгляд покажется бесполезной. Эта команда посылает бесконечный поток строк, состоящих из символа «у», в стандартный вывод:

```
/home/larry# yes
```

```
у
```

```
у
```

```
у
```

```
у
```

```
у
```

Последовательность таких строк будет бесконечно продолжаться. Уничтожить этот процесс можно нажатием клавиши прерывания, которая обычно является комбинацией *Ctrl-C*. Поступим теперь иначе. Чтобы на экран не выводилась эта бесконечная последовательность, перенаправим стандартный вывод команды *yes* на */dev/null*. Как отмечалось выше, устройство */dev/null* действует как «черная дыра»: все данные, посланные в это устройство, пропадают. С помощью этого устройства очень удобно избавляться от слишком обильного вывода некоторых программ:

```
/home/larry# yes > /dev/null
```

Теперь на экран ничего не выводится. Однако и приглашение командной оболочки также не возвращается. Это происходит потому, что команда *yes* все еще работает и посылает свои сообщения, состоящие из букв *y* на */dev/null*. Уничтожить это задание также можно нажатием клавиш прерывания.

Допустим теперь, что вы хотите, чтобы команда *yes* продолжала работать, но при этом и приглашение командной оболочки должно вернуться на экран. Для этого можно команду *yes* перевести в фоновый режим, и она будет там работать, не общаясь с вами.

Один способ перевести процесс в фоновый режим – приписать символ «&» к концу команды. Пример:

```
/home/larry# yes > /dev/null &  
\verb+[1] 164+  
/home/larry#
```

Как видно, приглашение командной оболочки опять появилось. Однако, что означает «*[1] 164*»? И действительно ли команда *yes* продолжает работать?

Сообщение «*[1]*» представляет собой номер задания (*job number*) для процесса *yes*. Командная оболочка присваивает номер задания каждому исполняемому заданию. Поскольку *yes* является единственным исполняемым заданием, ему присваивается номер 1. Число «*164*» является идентификационным номером, соответствующим данному процессу (*PID*), и этот номер также дан процессу системой. Как мы увидим дальше, к процессу можно обращаться, указывая оба этих номера.

Итак, теперь у нас есть процесс команды *yes*, работающий в фоне и непрерывно посылающий поток из букв *y* на устройство */dev/null*. Для того, чтобы узнать статус этого процесса, нужно исполнить команду *jobs*, которая является внутренней командой оболочки

```
/home/larry# jobs  
[1]+  Running                  yes >/dev/null &-  
/home/larry#
```

Мы видим, что эта программа действительно работает. Для того, чтобы узнать статус задания, можно также воспользоваться командой *ps*, как это было показано выше.

Для того, чтобы прервать работу задания, используется команда *kill*. В качестве аргумента этой команде дается либо номер задания, либо *PID*. В рассмотренном выше случае номер задания был 1, так что команда

```
/home/larry# kill %1
```

прервет работу задания. Когда к заданию обращаются по его номеру, а не *PID*, тогда перед этим номером в командной строке нужно поставить символ процента.

Теперь введем команду *jobs* снова, чтобы проверить результат предыдущего действия:

```
/home/larry# jobs
```

```
[1]+  Terminated          yes >/dev/null
```

```
/home/larry#
```

Фактически задание уничтожено, и при вводе команды *jobs* в следующий раз на экране о нем не будет никакой информации.

Уничтожить задание можно также, используя идентификационный номер процесса (*PID*). Этот номер, наряду с идентификационным номером задания, указывается во время старта задания. В нашем примере значение *PID* было 164, так что команда

```
/home/larry# kill 164
```

была бы эквивалентна команде

```
/home/larry# kill %1
```

При использовании *PID* в качестве аргумента команды *kill* вводить символ «%» не требуется.

9.3.4 Приостановка и продолжение работы заданий

Предложим еще один метод, с помощью которого процесс можно перевести в фоновый режим. Процесс запускается обычным образом (на переднем плане), затем приостанавливается командой *stop*, а потом запускается повторно в фоновом режиме.

Запустим сначала процесс командой *yes* на переднем плане, как это делалось раньше

```
/home/larry# yes > /dev/null
```

Как и ранее, поскольку процесс работает на переднем плане, приглашение командной оболочки на экран не возвращается.

Теперь вместо того, чтобы прервать задание комбинацией клавиш *Ctrl-C*, задание можно приостановить (*suspend*, буквально — «подвесить»). «Подвешенное» задание не будет уничтожено, его выполнение будет временно остановлено до тех пор, пока оно не будет возобновлено. Для приостановки задания надо нажать соответствующую комбинацию клавиш, обычно это *Ctrl-Z*

```
/home/larry# yes > /dev/null
```

```
{ctrl-Z}  
[1]+ Stopped          yes >/dev/null  
/home/larry#
```

Приостановленный процесс попросту не выполняется. На него не тратятся вычислительные ресурсы процессора. Приостановленное задание можно запустить с той же точки, как будто бы оно и не было приостановлено.

Для возобновления выполнения задания на переднем плане можно использовать команду *fg* (от слова «*foreground*» – передний план):

```
/home/larry# fg  
yes >/dev/null
```

Командная оболочка еще раз выведет на экран название команды, так что пользователь будет знать, какое именно задание он в данный момент запустил на переднем плане. Приостановим это задание еще раз нажатием клавиш *Ctrl-Z*, но в этот раз запустим его в фоновый режим командой *bg* (от слова «*background*» – фон). Это приведет к тому, что данный процесс будет работать так, как если бы при его запуске использовалась команда с символом «&» в конце (как это делалось в предыдущем разделе)

```
/home/larry# bg  
[1]+ yes &>/dev/null &  
/home/larry#
```

При этом приглашение командной оболочки возвращается. Сейчас команда *jobs* должна показывать, что процесс *yes* действительно в данный момент работает; этот процесс можно уничтожить командой *kill*, как это делалось раньше.

Для того чтобы приостановить задание, работающее в фоновом режиме, нельзя пользоваться комбинацией клавиш *Ctrl-Z*. Прежде чем приостанавливать задание, его нужно перевести на передний план командой *fg* и лишь потом приостановить. Таким образом, команду *fg* можно применять либо к приостановленным заданиям, либо к заданию, работающему в фоновом режиме.

Между заданиями в фоновом режиме и приостановленными заданиями есть большая разница. Приостановленное задание не работает и на него не тратятся вычислительные мощности процессора. Это задание не выполняет никаких действий. Приостановленное задание занимает некоторый объем оперативной памяти компьютера, хотя оно может быть перенесено в «своп». Напротив, задание в фоновом режиме выполняется, использует память и совершает некоторые действия, которые,

возможно, требуются пользователю, но он в это время может работать с другими программами.

Задания, работающие в фоновом режиме, могут пытаться выводить некоторый текст на экран. Это будет мешать работать над другими задачами. Например, если ввести команду

```
/home/larry# yes &
```

(стандартный вывод не был перенаправлен на устройство `/dev/null`), то на экран будет выводиться бесконечный поток символов `y`. Этот поток невозможно будет остановить, поскольку комбинация клавиш *Ctrl-C* не воздействует на задания в фоновом режиме. Чтобы остановить эту выдачу, надо использовать команду *fg*, а затем уничтожить задание комбинацией клавиш *Ctrl-C*.

Сделаем еще одно замечание. Обычно командой *fg* и командой *bg* воздействуют на те задания, которые были приостановлены последними (эти задания будут помечены символом «+» рядом с номером задания, если ввести команду *jobs*). Если в одно и то же время работает одно или несколько заданий, задания можно помещать на передний план или в фоновый режим, задавая в качестве аргументов команды *fg* или команды *bg* их идентификационный номер (job ID). Например, команда

```
/home/larry# fg %2
```

помещает задание номер 2 на передний план, а команда

```
/home/larry# bg %3
```

помещает задание номер 3 в фоновый режим. Использовать *PID* в качестве аргументов команд *fg* и *bg* нельзя. Более того, для перевода задания на передний план можно просто указать его номер. Так, команда

```
/home/larry# %2
```

будет эквивалентна команде

```
/home/larry# fg %2
```

Важно помнить, что функция управления заданием принадлежит оболочке. Команды *fg*, *bg* и *jobs* являются внутренними командами оболочки.

9.4 Механизмы межпроцессного взаимодействия в ОС Unix

При решении задачи синхронизации процессов и их взаимодействия посредством различных механизмов, предоставляемых ОС, может потребоваться использование следующих системных вызовов:

- создание, завершение процесса, получение информации о процессе: *fork*, *exit*, *getpid*, *getppid* и т. д.;
- синхронизация процессов: *signal*, *kill*, *sleep*, *alarm*, *wait*, *pause*, *semop*, *semctl*, *semcreate* и т. д.;
- создание информационного канала, разделяемой памяти, очереди сообщений и работа с ними: *pipe*, *mkfifo*, *read*, *write*, *msgget*, *shmget*, *msgctl*, *shmctl* и т. д.

Механизм межпроцессного взаимодействия (Inter-Process Communication Facilities – *IPC*) включает

- средства, обеспечивающие возможность синхронизации процессов при доступе к совместно используемым ресурсам, – *семафоры* (*semaphores*);
- средства, обеспечивающие возможность послыки процессом сообщений другому произвольному процессу, – очереди сообщений (*message queues*);
- средства, обеспечивающие возможность наличия общей для процессов памяти, – сегменты разделяемой памяти (*shared memory segments*);
- средства, обеспечивающие возможность «общения» процессов, как родственных, так и нет, через пайпы или каналы (*pipes*).

Наиболее общим понятием *IPC* является ключ, хранимый в общесистемной таблице и обозначающий объект межпроцессного взаимодействия, доступный нескольким процессам. Обозначаемый ключом объект может быть очередью сообщений, набором семафоров или сегментом разделяемой памяти. Ключ имеет тип *key_t*, состав которого зависит от реализации и определяется в файле *<sys/types.h>*. Ключ используется для создания объекта межпроцессного взаимодействия или получения доступа к существующему объекту.

Семафоры

Для работы с семафорами поддерживаются три системных вызова:

- *semget* – для создания и получения доступа к набору семафоров;
- *semop* – для манипулирования значениями семафоров (системный вызов, который позволяет процессам синхронизоваться на основе использования семафоров);
- *semctl* – для выполнения разнообразных управляющих операций над набором семафоров.

Прототипы перечисленных системных вызовов описаны в файлах

```
#include <sys/ipc.h>
#include <sys/sem.h>
```

Системный вызов *semget* имеет синтаксис


```
int semid = semget (key_t key, int count, int flag)
```

параметрами которого является ключ или уникальное имя сегмента (*key*), набора семафоров и дополнительные флаги (*flag*), определенные в `<sys/ipc.h>`, число семафоров в наборе семафоров (*count*), обладающих одним и тем же ключом. Системный вызов возвращает идентификатор набора семафоров *semid*. Живучесть такого семафора определяется живучестью ядра, т. е. объект семафор будет уничтожен тогда и только тогда, когда произойдет перезагрузка ядра либо его принудительно удалят. После вызова *semget* индивидуальный семафор идентифицируется идентификатором набора семафоров и номером семафора в этом наборе. Флаги системного вызова *semget* приведены ниже в таблице.

Таблица 4.1

Флаги системного вызова semge

Флаг	Описание
<i>IPC_CREAT</i>	Вызов <i>semget</i> создает новый семафор для данного ключа. Если флаг <i>IPC_CREAT</i> не задан, а набор семафоров с указанным ключом уже существует, то обращающийся процесс получит идентификатор существующего набора семафоров
<i>IPC_EXLC</i>	Флаг <i>IPC_EXLC</i> вместе с флагом <i>IPC_CREAT</i> предназначен для создания (и только для создания) набора семафоров. Если набор семафоров уже существует, <i>semget</i> возвратит -1, а системная переменная <i>errno</i> будет содержать значение <i>EEXIST</i>

Младшие 9 бит флага задают права доступа к набору семафоров (табл. 4.2).

Таблица 4.2

Константы режима доступа при создании нового объекта IPC

Константа	Описание
<i>S_IRUSR</i>	Владелец – чтение
<i>S_IWUSR</i>	Владелец – запись
<i>S_IRGRP</i>	Группа – чтение
<i>S_IWGRP</i>	Группа – запись
<i>S_IROTH</i>	Прочие – чтение
<i>S_IWOTH</i>	Прочие – запись

Таким образом, флаг создания семафора можно указать так:

```
int flag = S_IRUSR | S_IWUSR | S_IRGRP | IPC_CREAT;
```

Системный вызов *semctl* имеет формат

```
int semctl (int semid, int sem_num, int command, union semun arg)
```

где *semid* – это идентификатор набора семафоров; *sem_num* – номер семафора в группе; *command* – код операции; *arg* – указатель на структуру, содержимое которой интерпретируется по-разному, в зависимости от операции.

Объединение имеет вид

```
union semun
{
    int val; /* устанавливает значение семафора только для SETVAL */
    struct semid_ds *buf;
    /* используется командами IPC_STAT и IPC_SET */
    unsigned short *array; /* используется командами SETALL и GETALL */
};
```

Объединение *semun* всегда должен быть переопределен в глобальной секции программы. Структура *semid_ds* выглядит следующим образом:

```
struct semid_ds {
    struct ipc_perm sem_perm; /* разрешения на операции */
    struct sem *sem_base; /* указатель на массив семафоров в наборе */
    ushort sem_nsems; /* количество семафоров */
    time_t sem_otime; /* время последнего вызова semop() */
    time_t sem_ctime; /* время создания последнего IPC_SET */
};
```

Вызов *semctl* позволяет:

- уничтожить набор семафоров или индивидуальный семафор в указанной группе (*IPC_RMID*);
- вернуть значение отдельного семафора (*GETVAL*) или всех семафоров (*GETALL*);
- установить значение отдельного семафора (*SETVAL*) или всех семафоров (*SETALL*);
- вернуть число семафоров в наборе семафоров (*GETPID*).

Основным системным вызовом для манипулирования семафором является

```
int semop (int semid, struct sembuf *op_array, int count)
```

где *semid* – это ранее полученный дескриптор группы семафоров; *op_array* – массив структур *sembuf*

```
struct sembuf {
```

```

short sem_num; /* номер семафора: 0,1,2..n */
short sem_op;  /* операция с семафором */
short sem_flg; /* флаги операции: 0, IPC_NOWAIT, SEM_UNDO */
};

```

определенных в файле `<sys/sem.h>` и содержащих описания операций над семафорами группы, а *count* – количество элементов массива. Значение, возвращаемое системным вызовом, является значением последнего обработанного семафора.

Если указанные в массиве *op_array* номера семафоров не выходят за пределы общего размера набора семафоров, то системный вызов последовательно меняет значение семафора (если это возможно) в соответствии со значением поля «операция». Возможны три случая:

1. Отрицательное значение *sem_op*.
2. Положительное значение *sem_op*.
3. Нулевое значение *sem_op*.

Если значение поля операции *sem_op* отрицательно и его абсолютное значение меньше или равно значению семафора *semval*, то ядро прибавляет это отрицательное значение к значению семафора. Если в результате значение семафора стало нулевым, то ядро активизирует все процессы, ожидающие нулевого значения этого семафора. Если же значение поля операции *sem_op* по абсолютной величине больше семафора *semval*, то ядро увеличивает на единицу число процессов, ожидающих увеличения значения семафора и усыпляет текущий процесс до наступления этого события.

Если значение поля операции *sem_op* положительно, то оно прибавляется к значению семафора *semval*, а все процессы, ожидающие увеличения значения семафора, активизируются (пробуждаются в терминологии Unix).

Если значение поля операции *sem_op* равно нулю и значение семафора *semval* также равно нулю, выбирается следующий элемент массива *op_array*. Если же значение семафора *semval* отлично от нуля, то ядро увеличивает на единицу число процессов, ожидающих нулевого значения семафора, а обратившийся процесс переводится в состояние ожидания. При использовании флага *IPCNOWAIT* ядро ОС Unix не блокирует текущий процесс, а лишь сообщает в ответных параметрах о возникновении ситуации, приведшей к блокированию процесса при отсутствии флага *IPCNOWAIT*.

Именованные и неименованные каналы (пайпы)

Операционные системы семейства *Unix* всегда поддерживают два типа однонаправленных каналов:

- неименованные каналы;
- именованные каналы FIFO.

Неименованные каналы – это самая первая форма IPC в *Unix* (1973), главным недостатком которых является отсутствие имени, вследствие чего они могут использоваться для взаимодействия только родственными процессами. В *Unix System* третьей редакции (1982) были добавлены каналы FIFO, которые называются именованными каналами. Аббревиатура FIFO расшифровывается как «first in, first out» – «первым вошел, первым вышел», то есть эти каналы работают как очереди. Именованные каналы в *Unix* функционируют подобно неименованным – позволяют передавать данные только в одну сторону. Однако в отличие от программных каналов каждому каналу FIFO сопоставляется полное имя в файловой системе, что позволяет двум неродственным процессам обратиться к одному и тому же FIFO. Доступ и к именованным каналам, и к неименованным организуется с помощью обычных функций *read* и *write*.

FIFO создается вызовом *mkfifo*:

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(const char *pathname, mode_t mode);
/* возвращает 0 при успешном выполнении, -1 при ошибке */
```

Здесь *pathname* – обычное для *Unix* полное имя файла, которое и будет именем FIFO.

Аргумент *mode* указывает битовую маску разрешений доступа к файлу (табл. 4.2), аналогично второму аргументу команды *open*.

Функция *mkfifo* действует как *open*, вызванная с аргументом *mode = O_CREAT | O_EXCL*. Это означает, что создается новый канал FIFO или возвращается ошибка *EEXIST* в случае, если канал с заданным полным именем уже существует. Если не требуется создавать новый канал, вызывайте *open* вместо *mkfifo*. Для открытия существующего канала или создания нового, в том случае, если его еще не существует, вызовите *mkfifo*, проверьте, не возвращена ли ошибка *EEXIST*, и если такое случится, вызовите функцию *open*.

Команда *mkfifo* также создает канал FIFO. Ею можно пользоваться в сценариях интерпретатора или из командной строки.

Живучесть каналов определяется живучестью процессов, т. е. канал будет существовать до тех пор, пока он не будет принудительно закрыт либо не останется ни одного процесса работающего с каналом.

После создания канал FIFO должен быть открыт на чтение или запись с помощью либо функции `open`, либо одной из стандартных функций открытия файлов из библиотеки ввода-вывода (например, `fdopen`). FIFO может быть открыт либо только на чтение, либо только на запись. Нельзя открывать канал на чтение и запись, поскольку именованные каналы могут быть только односторонними (рис. 4.1).

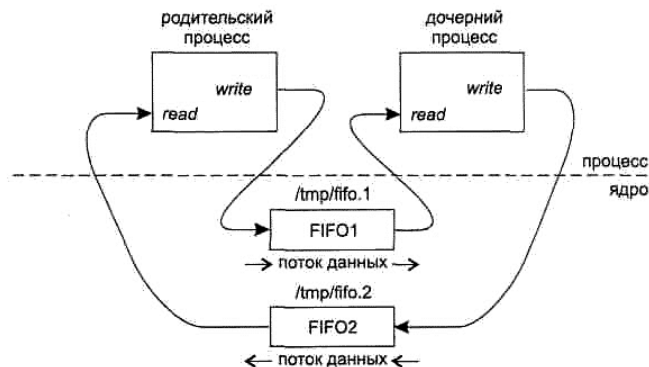


Рис. 4.1. Взаимодействие двух процессов посредством каналов FIFO

При записи в программный канал или канал FIFO вызовом `write` данные всегда добавляются к уже имеющимся, а вызов `read` считывает данные, помещенные в программный канал или FIFO первыми. При вызове функции `lseek` для программного канала или FIFO будет возвращена ошибка `ESPIPE`.

Неименованные каналы создаются вызовом `pipe()` и предоставляют возможность только однонаправленной (односторонней) передачи данных:

```
#include <unistd.h>

int fd[2];

pipe(fd);

/* возвращает 0 в случае успешного завершения, -1 - в случае ошибки;*/
```

Функция возвращает два файловых дескриптора: `fd[0]` и `fd[1]`, причем первый открыт для чтения, а второй – для записи.

Хотя канал создается одним процессом (рис. 4.2), он редко используется только этим процессом, каналы обычно используются для связи между двумя процессами (родительским и дочерним) следующим образом: процесс создает канал, а затем вызывает `fork`, создавая свою копию – дочерний процесс (рис. 4.3); затем родительский процесс закрывает открытый для чтения конец канала, а дочерний – открытый на запись конец канала (рис. 4.4). Это обеспечивает одностороннюю передачу данных между процессами (рис. 4.5)

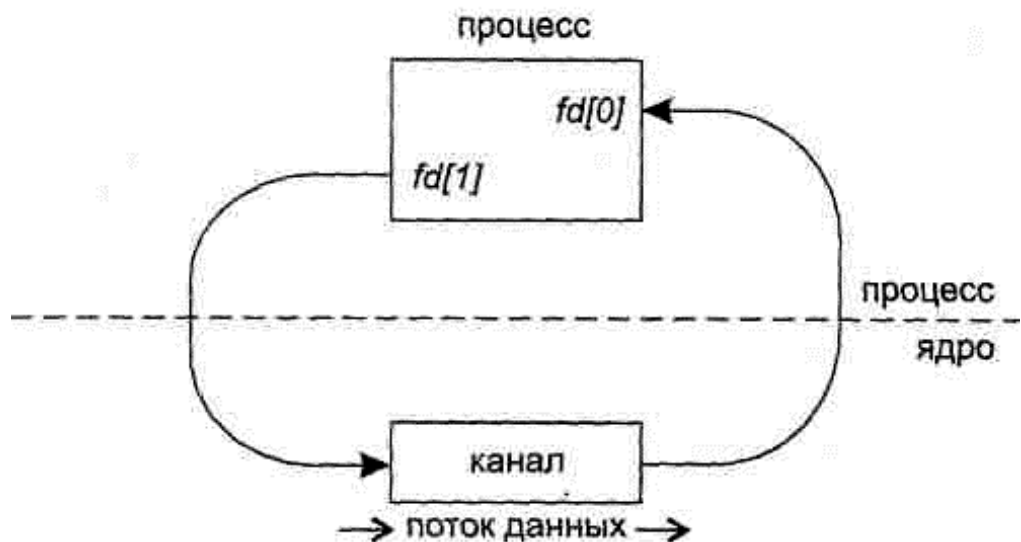


Рис. 4.2. Функционирование канала для случая одиночного процесса

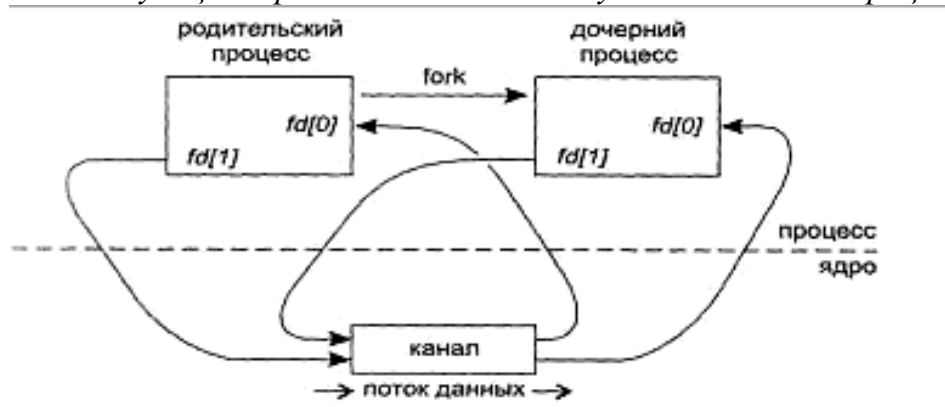


Рис. 4.3. Функционирование канала после создания дочернего процесса (после вызова *fork*)

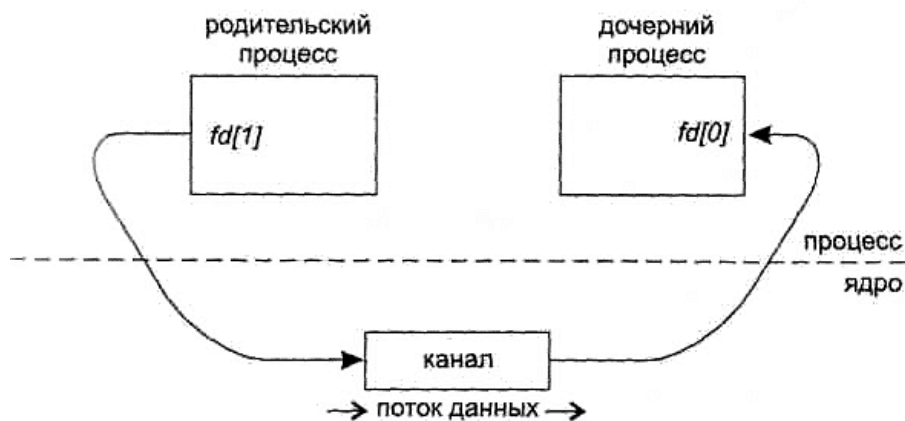


Рис. 4.4. Функционирование канала между двумя процессами

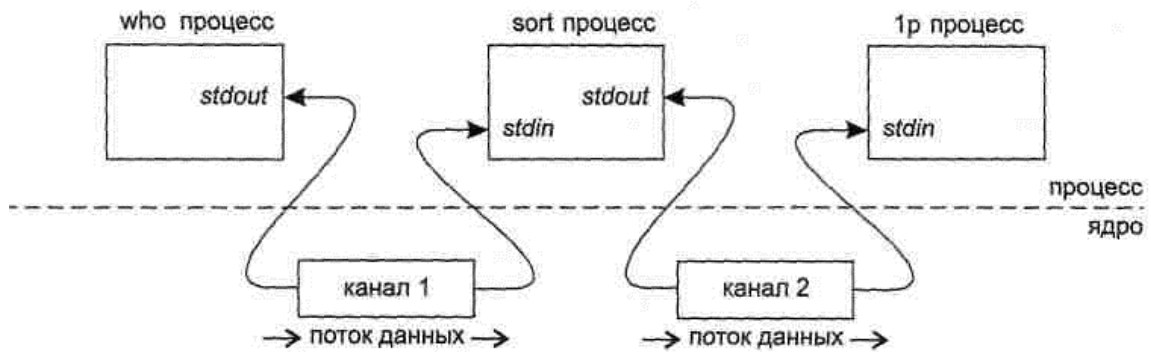


Рис. 4.5. Функционирование каналов между тремя процессами в конвейерной обработке

При вводе команды типа

`who | sort | 1p`

интерпретатор команд Unix выполняет вышеописанные действия для создания трех процессов с двумя каналами между ними (рис. 4.5).

Интерпретатор также подключает открытый для чтения конец каждого канала к стандартному потоку ввода, а открытый на запись — к стандартному потоку вывода.

Все рассмотренные выше неименованные каналы были однонаправленными (односторонними), то есть позволяли передавать данные только в одну сторону. При необходимости передачи данных в обе стороны нужно создавать пару каналов и использовать каждый из них для передачи данных в одну сторону. Этапы создания двунаправленного неименованного канала IPC следующие:

- создаются каналы 1 ($fd1[0]$ и $fd1[1]$) и 2 ($fd2[0]$ и $fd2[1]$);
- вызов `fork`;
- родительский процесс закрывает доступный для чтения конец канала 1 ($fd1[0]$);
- родительский процесс закрывает доступный для записи конец канала 2 ($fd2[1]$);
- дочерний процесс закрывает доступный для записи конец канала 1 ($fd1[1]$);
- дочерний процесс закрывает доступный для чтения конец канала 2 ($fd2[0]$).

9.4.1 Очереди сообщений

Для обеспечения возможности обмена сообщениями между процессами механизм очередей поддерживается следующими системными вызовами:

- `msgget` для образования новой очереди сообщений или получения дескриптора существующей очереди;

- *rnsgrcv* для постановки сообщения в указанную очередь сообщений;
- *rnsgrcv* для выборки сообщения из очереди сообщений;
- *rmsgctl* для выполнения ряда управляющих действий.

Прототипы перечисленных системных вызовов описаны в файлах

```
#include <sys/ipc.h>
```

```
#include <sys/msg.h>
```

По системному вызову *msgget* в ответ на ключ (*key*) определяется уникальное имя очереди и набор флагов (полностью аналогичны флагам в системном вызове *semget*). Вызовом *msgget* ядро либо создает новую очередь сообщений в ядре и возвращает пользователю идентификатор созданной очереди, либо находит элемент таблицы очередей сообщений ядра, содержащий указанный ключ, и возвращает соответствующий идентификатор очереди:

```
int msgqid = msgget(key_t key, int flag)
```

Таким образом, очередь сообщения обладает живучестью ядра. Для помещения сообщения в очередь служит системный вызов *msgsnd*:

```
int rnsgrcv (int msgqid, void *rmsg, size_t size, int flag)
```

где *msg* — это указатель на структуру длиной *size*, содержащую определяемый пользователем целочисленный тип сообщения и символьный массив-сообщение, причем размер пользовательских данных вычисляется следующим образом: $size = sizeof(msg) - sizeof(long)$.

Структура *msg* всегда имеет вид

```
struct rnsq {
    long rntype; /* тип сообщения */
    char mtext[SOMEVALUE]; /* текст сообщения */
};
```

Поле типа *long* всегда должно быть первым в структуре, далее могут следовать в любом порядке пользовательские данные, в этом случае ядро не накладывает ограничение на тип данных, а только на их длину (зависящую от реализации системы). Параметр *flag* определяет действия ядра для вызвавшего потока при чтении очереди или выходе за пределы допустимых размеров внутренней буферной памяти. Если *flag* = 0, то при отсутствии сообщения в очереди поток блокируется. Если *flag* = *IPCNOWAIT*, то поток не блокируется и при отсутствии сообщения возвращается ошибка *ENOMSG*.

Условиями успешной постановки сообщения в очередь являются:

- наличие прав процесса по записи в данную очередь сообщений;

- непревышение длиной сообщения, заданного системой верхнего предела;
- положительное значение типа сообщения.

Если же оказывается, что новое сообщение невозможно буферизовать в ядре по причине превышения верхнего предела суммарной длины сообщений, находящихся в данной очереди сообщений (флаг *IPCNOWAIT* при этом отсутствует), то обратившийся процесс откладывается (усыпляется) до тех пор, пока очередь сообщений не разгрузится процессами, ожидающими получения сообщений, или очередь не будет удалена, или вызвавший поток не будет прерван перехватываемым сигналом.

Для приема сообщения используется системный вызов *msgrcv*

```
int rmsgrcv (int msgqid, void *msg, size_t size, long
rmsg_type, int flag)
```

Аргумент *rmsg_type* задает тип сообщения, которое нужно считать из очереди

- если значение равно 0, то возвращается первое сообщение в очереди, т. е. самое старое сообщение;
- если тип больше 0, то возвращается первое сообщение, тип которого равен указанному числу;
- если тип меньше нуля, возвращается первое сообщение с наименьшим типом, значение которого меньше, либо равно модулю указанного числа.

Значение *size* в данном случае указывает ядру, что возвращаемые данные не должны превышать размера указанного в *size*.

Системный вызов *msgctl* позволяет управлять очередями сообщений

```
int msgctl (int msgqid, int command, struct msgid_ds *msg_stat)
```

и используется:

- для опроса состояния описателя очереди сообщений (*command* = *IPCSTAT*) и помещения его в структуру *msgstat*;
- изменения его состояния (*command* = *IPCSET*), например изменения прав доступа к очереди;
- для уничтожения указанной очереди сообщений (*command* = *IPCRMID*).

9.4.2 Работа с разделяемой памятью

Для работы с разделяемой памятью используются системные вызовы:

- *shmget* создает новый сегмент разделяемой памяти или находит существующий сегмент с тем же ключом;
- *shmat* подключает сегмент с указанным описателем к виртуальной памяти обращающегося процесса;
- *shmdt* отключает от виртуальной памяти ранее подключенный к ней сегмент с указанным виртуальным адресом начала;
- *shmctl* служит для управления разнообразными параметрами, связанными с существующим сегментом.

Прототипы перечисленных системных вызовов описаны в файлах

```
#include <sys/ipc.h>
```

```
#include <sys/shm.h>
```

После того как сегмент разделяемой памяти подключен к виртуальной памяти процесса, этот процесс может обращаться к соответствующим элементам памяти с использованием обычных машинных команд чтения и записи.

Системный вызов

```
int shmid = shmget (key_t key, size_t size, int flag)
```

на основании параметра *size* определяет желаемый размер сегмента в байтах. Если в таблице разделяемой памяти находится элемент, содержащий заданный ключ, и права доступа не противоречат текущим характеристикам обращающегося процесса, то значением системного вызова является идентификатор существующего сегмента, причем параметр *size* должен быть в этом случае равен 0. В противном случае создается новый сегмент с размером не меньше установленного в системе минимального размера сегмента разделяемой памяти и не больше установленного максимального размера. Живучесть объектов разделяемой памяти определяется живучестью ядра. Создание сегмента не означает немедленного выделения под него основной памяти, и это действие откладывается до выполнения первого системного вызова подключения сегмента к виртуальной памяти некоторого процесса. Флаги *IPC_CREAT* и *IPC_EXCL* аналогичны рассмотренным выше.

Подключение сегмента к виртуальной памяти выполняется путем обращения к системному вызову *shmat*:

```
void *virtaddr = shmat(int shmid, void *daddr, int flags).
```

Параметр *shmid* – это ранее полученный идентификатор сегмента, а *daddr* – желаемый процессом виртуальный адрес, который должен соответствовать началу сегмента в виртуальной памяти. Значением системного вызова является фактический виртуальный адрес начала сегмента. Если значением *daddr* является *NULL*, ядро выбирает наиболее удобный

виртуальный адрес начала сегмента. Флаги системного вызова *shmat* приведены ниже в таблице.

Таблица 4.3

Флаги системного вызова *shmat*

Флаг	Описание
<i>SHM_RDONLY</i>	Ядро подключает участок памяти только для чтения
<i>SHM_RND</i>	Определяет, если возможно, способ обработки ненулевого значения <i>daddr</i> .

Для отключения сегмента от виртуальной памяти используется системный вызов *shmdt*:

```
int shmdt (*daddr)
```

где *daddr* – это виртуальный адрес начала сегмента в виртуальной памяти, ранее полученный от системного вызова *shmat*.

Системный вызов *shmctl*

```
int shmctl (int shmid, int command, struct shmid_ds *shrn_stat)
```

по синтаксису и назначению аналогичен *msgctl*.

9.4.3 Примеры практической реализации

Семафоры

Программа *semsyn*, исходный код которой приведен ниже, создает семафор и два процесса, синхронизирующихся с помощью созданного семафора. В программе дочерний процесс является главным, он блокирует и разблокирует семафор, родительский процесс ждет освобождения семафора.

```
#include <unistd.h>
#include <stdio.h>
#include <error.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/sem.h>
#include <sys/ipc.h>
#include <fcntl.h>
#include <time.h>
#include <iostream.h>
#define MAXLINE 128
```

```

#define SVSEM_MODE (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)
#define SKEY 1234L // идентификатор семафора
union semun {
    int val;
    struct semid_ds *buf;
    ushort *array;
};
int var;
int main(int argc, char **argv) {
    char filename[] = "./rezult.txt";
    pid_t pid; // идентификатор дочернего процесса
    time_t ctime; // переменная времени
    int oflag, c, semid;
    struct tm *ctm;
    union semun arg;
    struct semid_ds seminfo;
    struct sembuf psmb;
    unsigned short *prt = NULL;

    var = 0;
    oflag = SVSEM_MODE | IPC_CREAT; // флаг семафора
    printf("Parent: Creating semaphore...\n");
    semid = semget(SKEY, 1, oflag); // создание семафора
    arg.buf = &seminfo;
    printf("Parent: Getting info about semaphore (not required, for exam-
ple)...\n");
    semctl(semid, 0, IPC_STAT, arg); //получение инф. о семафоре
    arg.buf->sem_ctime;
    ctm = localtime(&ctime);
    printf("%s %d %s %d %s %d %s", "Parent: Creating time - ",
    ctm->tm_hour, ":", ctm->tm_min, ":", ctm->tm_sec, "\n");
    arg.val = 5;
    printf("%s %d %s", "Parent: Setting value \'", arg.val, "\' to sema-
phores...\n");

```

```

semctl(semid, 0, SETVAL, arg); // установка значения семафора
printf("Parent: Creating child process...\n");
if ((pid = fork()) == 0) { // child process ;
    printf("  Child: Child process was created...\n");
    struct sembuf csmb;
    unsigned short semval;
    union semun carg;
    int oflag = SVSEM_MODE | IPC_EXCL;
    printf("  Child: Opening semaphore...\n");
    int smd = semget(SKEY, 1, oflag); // открытие семафора
    csmb.sem_num = 0;
    csmb.sem_flg = 0;
    csmb.sem_op = -1;
    printf("  Child: Locking semaphore...\n");
    semop(smd,&csmb,1); // блокировка семафора
    printf("  Child: Do something...\n");
    // работа процесса в защищенном режиме
    sleep(2);
    // работа процесса в защищенном режиме закончена
    printf("  Child: Done something...\n");
    carg.buf = NULL;
    carg.array = &semval;
    semctl(smd,0,GETALL,carg); // получение значения семафора
    semval = *carg.array;
    printf("%s %d %s", "  Child: Semaphore value = ",semval,"\n");
    csmb.sem_num = csmb.sem_flg = 0;
    csmb.sem_op = -semval;
    printf("  Child: Unlocking semaphore...\n");
    semop(smd,&csmb,1);
    printf("  Child: Terminating child process...\n");
    exit(0);
}

```

```

printf("Parent: Waiting for unlocking semaphore...\n");
psmb.sem_num = psmb.sem_flg = psmb.sem_op = 0;
semop(semid,&psmb,1);
printf("Parent: Semaphore is unlocked...\n");
printf("Parent: Waiting for SIGCHLD...\n");
waitpid(pid,NULL,0);
printf("Parent: Deleting semaphore...\n");
semctl(semid, 0, IPC_RMID);
exit(0);
}

```

Запуск приведенной выше программы происходит следующим образом:

semsyn

Parent: Creating semaphore...

Parent: Getting info about semaphore (not required, for example)...

Parent: Creating time - 13 : 14 : 6

Parent: Setting value " 5 " to semaphore...

Parent: Creating child process...

Child: Child process was created...

Child: Opening semaphore...

Child: Locking semaphore...

Child: Do something...

Parent: Waiting for unlocking semaphore...

Child: Done something...

Child: Semaphore value = 4

Child: Unlocking semaphore...

Parent: Semaphore is unlocked...

Parent: Waiting for SIGCHLD...

Child: Terminating child process...

Parent: Deleting semaphore...

Во время работы программы создается семафор с живучестью ядра

ipcs -s

----- Semaphore Arrays -----

```
key      semid  owner  perms  nsems
0x000004d2 425986  root   644    1
```

Разделяемая память

Программа *shmget* создает сегмент разделяемой памяти, принимая из командной строки полное имя произвольного файла и длину сегмента.

```
#include <stdio.h>
#include <error.h>
#include <fcntl.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <unistd.h>
#include <stdlib.h>
#define SVSHM_MODE (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)
int main(int argc, char **argv)
{
    int c, id, oflag;
    char *ptr;
    size_t length;
    oflag = SVSHM_MODE | IPC_CREAT; // флаг создания семафора
    while ( (c = getopt(argc, argv, "e")) != -1) {
        switch (c) { // просмотр ключей командной строки
            case 'e':
                oflag |= IPC_EXCL;
                break;
        }
    }
    if (optind != argc - 2)
    {
        printf("usage: shmget [ -e ] <path_to_file> <length>");
        return 0;
    }
    length = atoi(argv[optind + 1]);
    id = shmget(ftok(argv[optind], 0), length, oflag);
```

```

ptr = (char*) shmat(id, NULL, 0);
return 0;
}

```

Вызов *shmget* создает сегмент разделяемой памяти указанного размера. Полное имя, передаваемое в качестве аргумента командной строки, преобразуется в ключ IPC System V вызовом функции *ftok*. Если указан параметр *e* командной строки и в системе существует сегмент с тем же именем, запуски программы завершатся по ошибке. Если известно, что сегмент уже существует, то в командной строке должна быть указана нулевая длина сегмента памяти.

Вызов *shmat* подключает сегмент к адресному пространству процесса, после чего программа завершает работу. В связи с тем, что разделяемая память System V обладает «живучестью ядра», то сегмент разделяемой памяти при этом не удаляется.

Программа *shrmid* вызывает функцию *shmctl* с командой *IPC_RMID* для удаления сегмента разделяемой памяти из системы.

```

#include <stdio.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <error.h>
#include <fcntl.h>
#define SVSHM_MODE (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)
int main(int argc, char **argv)
{
    int id;
    if (argc != 2)
    {
        printf("usage: shrmid <path_to_file>");
        return 0;
    }
    id = shmget(ftok(argv[1], 0), 0, SVSHM_MODE);
    shmctl(id, IPC_RMID, NULL);
    return 0;
}

```


Программа *shmwrite* заполняет сегмент разделяемой памяти последовательностью значений 0, 1, 2, ..., 254, 255. Сегмент разделяемой памяти открывается вызовом *shmget* и подключается вызовом *shmat*. Его размер может быть получен вызовом *shmctl* с командой *IPC_STAT*.

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <error.h>
#include <fcntl.h>
#define SVSHM_MODE (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)
int main(int argc, char **argv)
{
    int i, id;
    struct shmid_ds buff;
    unsigned char *ptr;
    if (argc != 2)
    {
        printf("usage: shmwrite <path_to_file>");
        return 0;
    }
    id = shmget(ftok(argv[1], 0), 0, SVSHM_MODE);
    ptr = (unsigned char*) shmat(id, NULL, 0);
    shmctl(id, IPC_STAT, &buff);
    /* 4set: ptr[0] = 0, ptr[1] = 1, etc. */
    for (i = 0; i < buff.shm_segsz; i++) *ptr++ = i % 256;
    return 0;
}
```

Программа *shmread* проверяет последовательность значений, записанную в разделяемую память программой *shmwrite*.

```
#include <stdio.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/shm.h>
```

```

#include <error.h>
#include <fcntl.h>
#define SVSHM_MODE (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)
int main(int argc, char **argv)
{
    int i, id;
    struct shmid_ds buff;
    unsigned char c, *ptr;
    if (argc != 2)
    {
        printf("usage: shmread <path_to_file>");
        return 0;
    }
    id = shmget(ftok(argv[1], 0), 0, SVSHM_MODE);
    ptr = (unsigned char*) shmat(id, NULL, 0);
    shmctl(id, IPC_STAT, &buff);
    /* check that ptr[0] = 0, ptr[1] = 1, and so on. */
    for (i = 0; i < buff.shm_segsz; i++)
        if ( (c = *ptr++) != (i % 256)) printf("ptr[%d] = %d", i, c);
    return 0;
}

```

Рассмотрим результат запуска приведенных выше программ при работе с разделяемой памятью. Вначале создается сегмент разделяемой памяти длиной 1234 байта. Для идентификации сегмента используем полное имя исполняемого файла /tmp/test1. Это имя будет передано функции *ftok*:

```

shmget /tmp/test1 1234
ipcs -bmo
IPC status from <running system> as of Thu Jan 8 13:17:06 1998
T ID   KEY      MODE     OWNER  GROUP  NATTCH SEGSZ
Shared Memory:
m 1    0x0000f12a  --rw-r--r-- rstevens otherl 0      1234

```

Программа *ipcs* запускается для того, чтобы убедиться, что сегмент разделяемой памяти действительно был создан и не был удален по завершении программы *shmcreate*.

Запуская программу *shmwrite*, можно заполнить содержимое разделяемой памяти последовательностью значений. Затем с помощью программы *shmread* проверяется содержимое сегмента разделяемой памяти:

```
shmwrite shmget
```

```
shmread shmget
```

```
shmrmid shmget
```

```
ipcs -bmo
```

```
IPC status from <running system> as of Thu Jan 8 13:17:06 1998
```

```
T ID KEY MODE OWNER GROUP NATTCH SEGSZ
```

```
Shared Memory:
```

Удалить разделяемую память можно, вызвав

```
shmrmid /tmp/test1
```

Программные каналы

Программа *pipes* создает два процесса и обеспечивает двустороннюю связь между ними посредством неименованных каналов.

```
#include <unistd.h>
```

```
#include <stdio.h>
```

```
#include <sys/types.h>
```

```
#include <sys/wait.h>
```

```
#include <iostream.h>
```

```
#include <strings.h>
```

```
#include <fstream.h>
```

```
#define MAXLINE 128
```

```
void server(int,int), client(int,int);
```

```
int main(int argc, char **argv) {
```

```
int pipe1[2],pipe2[2]; // идентификаторы каналов
```

```
pid_t childpid = 0;
```

```
printf("Parent: Creating pipes...\n");
```

```
pipe(pipe1);
```

```
pipe(pipe2);
```

```
printf("Parent: Pipes created...\n");
```

```

printf("Parent: Creating child process...\n");
if ((childpid = fork()) == 0) { // child process starts
    printf("Child:      Child process created...\n");
    close(pipe1[1]);
    close(pipe2[0]);
    printf("Child:      Starting server...\n");
    server(pipe1[0], pipe2[1]);
    printf("Child:      Terminating process...\n");
    exit(0);
}
// parent process
close(pipe1[0]);
close(pipe2[1]);
sleep(2);
printf("Parent:      Starting client...\n");
client(pipe2[0], pipe1[1]);
printf("Parent: Waiting for child process to terminate a zombie...\n");
waitpid(childpid, NULL, 0);
printf("Parent: Zombie terminated...\n");
return 0;

}

void server(int readfd, int writefd) {
    char str[MAXLINE];
    strcpy(str, "some string to transmit");
    ssize_t n = 0;
    printf("%s %s %s", "Child:  Server: Transferring string
to client - '", str, "'\n");
    write(writefd, str, strlen(str));
    sleep(1);
    printf("Child:  Server: Waiting for replay from client...");
    while ((n = read(readfd, str, MAXLINE)) > 0)
    {

```

```

        str[n] = 0;
        printf("%s %s %s", "Received OK from client - \'", str, "\\n");
        break;
    }

    printf("Child:  Server was terminated...\\n");
    return;
}

void client(int readfd, int writefd) {
    ssize_t n = 0;
    char buff[MAXLINE];
    while ((n = read(readfd, buff, MAXLINE)) > 0 )
    {
        buff[n] = 0;
        printf("%s %s %s", "Client: Recieved string from server: \'", buff, "\\n");
        break;
    }
    printf("Parent: Client: Sending OK to server\\n");
    sleep(1);
    strcpy(buff, "sends OK from client");
    write(writefd, buff, strlen(buff));
    return;
}

```

Далее приведены программы, организующие межпроцессное взаимодействие посредством именованных каналов.

Программа сервер

```

#include <unistd.h>
#include <stdio.h>
#include <error.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <iostream.h>

```

```

#include <strings.h>
#include <fstream.h>
#include <sys/stat.h>
#include <errno.h>
#include <fcntl.h>
#define MAXLINE 128
#define FILE_MODE (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)
#define FIFO1 "/tmp/fifo.1"
#define FIFO2 "/tmp/fifo.2"

int main(int argc, char **argv) {
    int      readfd = -1, writefd = -1;
    pid_t    childpid = 0;
    ssize_t  n;
    char str[MAXLINE];

    strcpy(str, "some string to transmit ");
    cout<<"Creating pipes..."<<endl;
    unlink(FIFO1);
    unlink(FIFO2);
    if (mkfifo(FIFO1, FILE_MODE) == EEXIST) cout<<"\n Pipes is ex-
ists"<<endl;
    if (mkfifo(FIFO2, FILE_MODE) == EEXIST) cout<<"\n Pipes is ex-
ists"<<endl;
    cout<<"Pipes created..."<<endl;
    writefd = open(FIFO2, O_WRONLY);
    if ((writefd != -1)) {
        cout<<"Transmitting the string..."<<endl;
        write(writefd, str, strlen(str));
        readfd = open(FIFO1, O_RDONLY);
        cout<<"Waiting for respond..."<<endl;
        while ((n = read(readfd, str, MAXLINE)) > 0) {
            str[n] = 0;
            cout<<"Received string - \""<<str<<"\"<<endl;

```

```

        break;
    }
    close(readfd);
    close(writefd);
    unlink(FIFO1);
    unlink(FIFO2);
} else cout<<"Can't open pipes..."<<endl;
return 1;
}

```

Программа клиент

```

#include <unistd.h>
#include <stdio.h>
#include <error.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <iostream.h>
#include <strings.h>
#include <fstream.h>
#include <sys/stat.h>
#include <errno.h>
#include <fcntl.h>
#define MAXLINE 128
#define FILE_MODE (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)
#define FIFO1 "/tmp/fifo.1"
#define FIFO2 "/tmp/fifo.2"

int main(int argc, char **argv) {
    int readfd = -1, writefd = -1;
    pid_t childpid = 0;
    ssize_t n = 0;
    char str[MAXLINE];
    ofstream fsw("./result.txt");

```

```

    fsw<<"Opening pipes..."<<endl;
while (1)
{
    readfd = open(FIFO2, O_RDONLY, 0);
    if (readfd != -1) {
        fsw<<"Pipes opened..."<<endl;
        fsw<<"Waiting for respond..."<<endl;
        while ((n = read(readfd,str, MAXLINE)) > 0) {
            str[n] = 0;
            fsw<<"Received string - \""<<str<<"\""<<endl;
            break;
        }
        strcpy(str,"Ok from other process");
        writefd = open(FIFO1, O_WRONLY, 0);
        fsw<<"Transmitting the string - \""<<str<<"\""<<endl;
        write(writefd,str,strlen(str));
        close(readfd);
        close(writefd);
        break;
    }
    sleep(1);
}
fsw.close();
return 1;
}

```

Рассмотрим результат запуска приведенных выше программ, использующих неименованные каналы.

```

pipes
Parent: Creating pipes...
Parent: Pipes created...
Parent: Creating child process...
Child:  Child process created...
Child:  Starting server...

```



```

Child:  Server: Tranfering string to client - " some string to transmit "
Child:  Server: Waiting for replay from client...Received OK from client - "
sends OK from client "
Child:  Server was terminated...
Child:  Terminating process...
Parent: Creating pipes...
Parent: Pipes created...
Parent: Creating child process...
Parent: Starting client...
Client: Recieved string from server: " some string to transmit "
Parent: Client: Sending OK to server
Parent: Waiting for child porecess to terminate a zombie...
Parent: Zombie terminated...

```

Программы, взаимодействующие через каналы FIFO, запускаются следующим образом:

```

client &
Opening pipes...
Pipes opened...
Waiting for respond...
Received string - " some string to transmit "
Transmitting the string - "Ok from other process"
server
Creating pipes...
Pipes created...
Transmitting the string...
Waiting for respond...
Received string - "Ok from other process"
[1]+  Exit 1                  ./pn (wd: ~/makegnu/ipc/pipe_name/2/bin)
(wd now: ~/makegnu/ipc/pipe_name/1/bin)

```

Очереди сообщений

Программа *msgcreate* создает очередь сообщений. Параметр командной строки *e* позволяет указать флаг *IPC_EXCL*. Полное имя файла, являющееся обязательным аргументом командной строки, передается функ-

ции *ftok*. Получаемый ключ преобразуется в идентификатор функцией *msgget*.

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/types.h>
#include <sys/msg.h>
#include <error.h>
#include <unistd.h>
#include <fcntl.h>
#define SVMSG_MODE (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)
int main(int argc, char **argv)
{
    int c, oflag, mqid;
    oflag = SVMSG_MODE | IPC_CREAT;
    while ( (c = getopt(argc, argv, "e")) != -1) {
        switch (c) {
            case 'e':
                oflag |= IPC_EXCL;
                break;
        }
    }
    if (optind != argc - 1)
    {
        printf("usage: msgcreate [ -e ] <path_to_file>");
        return 0;
    }
    mqid = msgget(ftok(argv[optind], 0), oflag);
    return 0;
}
```

Программа *msgsnd* помещает в очередь одно сообщение заданной длины и типа. В программе создается указатель на структуру *msgbuf* общего вида, а затем путем вызова *calloc* выделяется место под реальную структуру (буфер записи) соответствующего размера.

```

#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <unistd.h>
#include <error.h>
#include <fcntl.h>
#define MSG_W (S_IWUSR)
int main(int argc, char **argv)
{
    int mqid;
    size_t len;
    long type;
    struct msgbuf *ptr;
    if (argc != 4)
    {
        printf("usage: msgsnd <path_to_file><#bytes><type>");
        return 0;
    }
    len = atoi(argv[2]);
    type = atoi(argv[3]);
    mqid = msgget(ftok(argv[1], 0), MSG_W);
    ptr = (struct msgbuf*) calloc(sizeof(long) + len, sizeof(char));
    ptr->mtype = type;
    msgsnd(mqid, ptr, len, 0);
    return 0;
}

```

Программа *msgrcv* считывает сообщение из очереди. В командной строке может быть указан параметр *n*, отключающий блокировку, а параметр *t* может быть использован для указания типа сообщения в функции *msgrcv*.

```

#include <stdio.h>
#include <fcntl.h>

```

```

#include <unistd.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdlib.h>
#define MSG_R (S_IRUSR | S_IRGRP | S_IROTH)
#define MAXMSG (8192 + sizeof(long))
int main(int argc, char **argv)
{
    int c, flag, mqid;
    long type;
    ssize_t n;
    struct msgbuf *buff;
    type = flag = 0;
    while ( (c = getopt(argc, argv, "nt:")) != -1) {
        switch (c) {
            case 'n':
                flag |= IPC_NOWAIT;
                break;
            case 't':
                type = atol(optarg);
                break;
        }
    }
    if (optind != argc - 1)
    {
        printf("usage: msgrcv [ -n ][ -t type ]<path_to_file>");
        return 0;
    }
    mqid = msgget(ftok(argv[optind], 0), MSG_R);
    buff = (msgbuf*) malloc(MAXMSG);
    n = msgrcv(mqid, buff, MAXMSG, type, flag);
    printf("read %d bytes, type = %ld\n", n, buff->mtype);
    return 0;
}

```

```
}
```

Программа *msgctl* удаляет очередь.

```
#include <stdio.h>
```

```
#include <sys/ipc.h>
```

```
#include <sys/msg.h>
```

```
#include <fcntl.h>
```

```
#include <error.h>
```

```
int main(int argc, char **argv)
```

```
{
```

```
int mqid;
```

```
if (argc != 2)
```

```
{
```

```
    printf("usage: msgrmid <path_to_file>");
```

```
    return 0;
```

```
}
```

```
mqid = msgget(ftok(argv[1], 0), 0);
```

```
msgctl(mqid, IPC_RMID, NULL);
```

```
return 0;
```

```
}
```

Результат запуска приведенных выше программ для случая с тремя сообщениями в очереди:

```
msgcreate /tmp/no/such/file
```

```
ftok error for pathname "tmp/no/such/file" and id 0: No such file or directory
```

```
touch /tmp/testl
```

```
msgcreate /tmp/testl
```

```
msgsnd /tmp/testl 1 100
```

```
msgsnd /tmp/testl 2 200
```

```
msgsnd /tmp/testl 3 300
```

```
ipcs -qo
```

```
IPC status from <running system> as of Sat Jan 10 11:25:45 1998
```

```
T  ID   KEY      MODE    OWNER  GROUP  CBYTES QNUM
```

```
Message Queues:
```

```
q  100  0x0000003e  --rw-r--r-  rstevens  otherl  6      3
```

Сначала происходит попытка создания очереди с помощью имени несуществующего файла. Пример показывает, что файл, указываемый в качестве аргумента функции *ftok*, обязательно должен существовать. Затем создается файл */tmp/test1* и используется его имя при создании очереди сообщений. После этого в очередь помещаются три сообщения длиной 1, 2 и 3 байта со значениями типа 100, 200 и 300. Программа *ipcs* показывает, что в очереди находятся 3 сообщения общим объемом 6 байт.

Теперь с помощью аргумента *type* при вызове *msgrcv* считываются сообщения в произвольном порядке:

```
msgrcv -t 200 /tmp/test1
read 2 bytes, type = 200
msgrcv -t -300 /tmp/test1
read 1 bytes, type = 100
msgrcv /tmp/test1
read 3 bytes, type = 300
msgrcv -n /tmp/test1
msgrcv error: No message of desired type
```

В первом примере запрашивается сообщение с типом 200, во втором примере – сообщение с наименьшим значением типа, не превышающим 300, а в третьем – первое сообщение в очереди. Последний запуск *msgrcv* иллюстрирует действие флага *IPC_NOWAIT*.

Удалить очередь можно, вызвав

```
msgrmid /tmp/test1
```

9.5 Последовательность выполнения работы

1. Ознакомиться с теоретическим материалом.
2. Запустить несколько заданий (например, команд просмотра файлов *less*), возвращаясь в командную строку комбинацией клавиш *Ctrl-Z* и изучить действие команд *ps*, *jobs*, *fg*, *bg*, *kill*, *killall*.
3. Обеспечить синхронизацию процессов и передачу данных между ними на примере двух приложений «клиент» и «сервер», создав два процесса (два исполняемых файла) – процесс «клиент» (первый исполняемый файл) и процесс «сервер» (второй исполняемый файл). С помощью механизмов межпроцессного взаимодействия обеспечить передачу информации от «клиента» к «серверу» и наоборот. В качестве типа передаваемой информации можно использовать: данные, вводимые

с клавиатуры; данные, считываемые из файла; данные, генерируемые случайным образом и т. п.

4. Обмен данными между процессами «клиент»-«сервер» осуществить следующим образом:

- с использованием программных каналов (именованных либо неименованных, по указанию преподавателя);
- с использованием (по указанию преподавателя) одного из перечисленных вариантов:
 - разделяемая память (обязательна синхронизация процессов, например с помощью семафоров);
 - очередь сообщений.

Литература

1. Бэкон Дж., Харрис Т. Операционные системы. – СПб.: Изд-во «БХВ-Петербург», 2004.– 800 с.
2. Гордеев А. В. Операционные системы : учебник / А. В. Гордеев. – 2-е изд. – СПб. : Питер, 2004. – 416 с.
3. Дейтел Х.М., Дейтел П.Дж., Чорнес Д.Р. Операционные системы. Основы и принципы. – Изд-во «Бином-пресс», 2006. – 1204 с.
4. Иртегов. Д. Введение в операционные системы. – СПб.: Изд-во «БХВ-Петербург», 2008.– 1040 с.
5. Карпов В.Е. , Коньков К.А. Основы операционных систем. Курс лекций. Учебное пособие. – Изд-во «Интернет-университет информационных технологий», 2005. – 632 с.
6. Кастер Х. Основы Windows NT и NTFS. – М.: Изд-во «Русская редакция», 1996.– 440 с.
7. Курячий Г.В., Маслинский К. А. Учебный курс «Операционная система Linux» [Электронный ресурс]. – режим доступа: <http://www.intuit.ru/department/os/linux/> (1.03.2010).
8. Олифер В.Г. Сетевые операционные системы: учебное пособие / В.Г. Олифер, Н.А. Олифер. – СПб.: Питер, 2003. – 538 с.
9. Программирование для Linux. Поток. [Электронный ресурс].– режим доступа: <http://www.citforum.ru/programming/unix/threads/> (1.03.2010).
- 10.Робачевский А.М. Операционная система Unix. – СПб.: БХВ-Санкт-Петербург, 1999.– 528 с.
- 11.Средства параллельного программирования для ОС Linux [Электронный ресурс]. – режим доступа: http://www.opennet.ru/docs/RUS/linux_parallel/ (1.03.2010).
- 12.Стен Келли-Бутл. Введение в Unix. – М.: «Лори», 1995. – 600 с.
- 13.Стивенс У. Unix: взаимодействие процессов. – СПб.: Питер, 2003. – 576 с.
- 14.Стивенс У.Р., Феннер Б., Рудофф Э.М. Unix: разработка сетевых приложений. 3-е изд. – СПб.: Питер, 2007. – 1039 с.
- 15.Таненбаум Э., Вудхалл А.. Операционные системы. Разработка и реализация (+CD). Классика CS. 3-е изд. – СПб.: Питер, 2007. – 704 с.
- 16.QNX Realtime operating system (RTOS) software, development tools, and services for embedded applications. [Электронный ресурс].– режим доступа: <http://www.qnx.com/> (1.03.2010).
- 17.Cisco IOS Software - Products & Services - Cisco Systems. [Электронный ресурс].– режим доступа: http://www.cisco.com/en/US/products/sw/iosswrel/products_ios_cisco_ios_software_category_home.html (1.03.2010)

Учебное издание

ЗАМЯТИН Александр Владимирович

ОПЕРАЦИОННЫЕ СИСТЕМЫ. Теория и практика

Учебное пособие

Научный редактор
доктор технических наук

В.А. Силич

Подписано к печати __.__._____. Формат 60х84/16. Бумага «Снегурочка».

Печать XEROX. Усл.печ.л. __, ___. Уч.-изд.л. __, __.


Заказ _____. Тираж _____ экз.



Томский политехнический университет
Система менеджмента качества
Томского политехнического университета сертифициро-
вана

NATIONAL QUALITY ASSURANCE по стандарту ISO
9001:2000



ИЗДАТЕЛЬСТВО  ТПУ. 634050, г. Томск, пр. Ленина, 30.