МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
им. М.В.ЛОМОНОСОВА
ФАКУЛЬТЕТ ВЫЧИСЛИМТЕЛЬНОЙ МАТЕМАТИКИ И КИБЕРНЕТИКИ
КАФЕДРА ИНТЕЛЛЕКТУАЛЬНЫХ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ

ОПЕРАЦИОННЫЕ СИСТЕМЫ

Конспект лекций

МОСКВА
2026

Содержание

СОДЕРЖАНИЕ....1

1 ВВЕДЕНИЕ....5
1.1 Основы архитектуры вычислительной системы....10
1.1.1 Структура ВС....10
1.1.2 Аппаратный уровень ВС....11
1.1.3 Управление физическими ресурсами ВС....12
1.1.4 Управление логическими/виртуальными ресурсами....14
1.1.5 Системы программирования....16
1.1.6 Прикладные системы....22
1.1.7 Выводы....27
1.2 Основы компьютерной архитектуры....30
1.2.1 Структура, основные компоненты....30
1.2.2 Оперативное запоминающее устройство....33
1.2.3 Центральный процессор....38
1.2.3.1 Регистровая память....38
1.2.3.2 Устройство управления. Арифметико-логическое устройство....39
1.2.3.3 КЭШ-память....40
1.2.3.4 Аппарат прерываний....42
1.2.4 Внешние устройства....47
1.2.4.1 Внешние запоминающие устройства....48
1.2.4.2 Модели синхронизации при обмене с внешними устройствами....51
1.2.4.3 Потоки данных. Организация управления внешними устройствами....52
1.2.5 Иерархия памяти....54
1.2.6 Аппаратная поддержка операционной системы и систем программирования....55
1.2.6.1 Требования к аппаратуре для поддержки мультипрограммного режима..56
1.2.6.2 Проблемы, возникающие при исполнении программ....60
1.2.6.3 Регистровые окна....62
1.2.6.4 Системный стек....65
1.2.6.5 Виртуальная память....66
1.2.7 Многомашинные, многопроцессорные ассоциации....71
1.2.8 Терминальные комплексы (ТК)....75
1.2.9 Компьютерные сети....77
1.2.10 Организация сетевого взаимодействия. Эталонная модель ISO/OSI....79
1.2.11 Семейство протоколов TCP/IP. Соответствие модели ISO/OSI....83
1.3 Основы архитектуры операционных систем....89
1.3.1 Структура ОС....90
1.3.2 Логические функции ОС....94
1.3.3 Типы операционных систем....96

2 УПРАВЛЕНИЕ ПРОЦЕССАМИ....99
2.1 Основные концепции....99
2.1.1 Модели операционных систем....99
2.1.2 Типы процессов....102
2.1.3 Контекст процесса....103
2.2 Реализация полновесных процессов в ОС Unix....103
2.2.1 Процесс ОС Unix....103
2.2.2 Базовые средства управления процессами в ОС Unix....106
2.2.2.1 fork() – создание нового процесса....106

2.2.2.2 exec() – семейство системных вызовов смены тела процесса.....109
2.2.2.3 Завершение процесса 113
2.2.2.4 wait() – ждем «завершения» процесса 114
2.2.3 Жизненный цикл процесса. Состояния процесса.....116
2.2.4 Формирование процессов 0 и 1 117
2.3 Реализация легковесных процессов Unix (нити).....121
2.3.1 Создание/запуск нити.....121
2.3.2 Операции с идентификатором нити 121
2.3.3 Завершение нити.....122
2.3.3.1 Завершение нити и всего процесса. 122
2.3.3.2 Завершение отдельной нити 122
2.3.4 Примеры программирования нитей.....123
2.4 Планирование 126
2.5 Взаимодействие процессов.....126
2.5.1 Разделяемые ресурсы и синхронизация доступа к ним 126
2.5.2 Способы организации взаимного исключения 128
2.5.3 Классические задачи синхронизации процессов.....131
2.5.3.1 «Обедающие философы».....131
2.5.3.2 «Читатели и писатели».....134
2.5.3.3 «Спящий парикмахер». 139

3 МЕЖПРОЦЕССНОЕ ВЗАИМОДЕЙСТВИЯ В ОС UNIX 141
3.1 Базовые средства реализации взаимодействия процессов в ОС Unix.....141
3.1.1 Сигналы 143
3.1.1.1 Пример. Двухпроцессный вариант программы “Будильник”. 149
3.1.2 Надежные сигналы. 150
3.1.2.1 Пример. Работа с сигнальной маской. 151
3.1.2.2 Пример. Использование надежных сигналов. 153
3.1.3 Неименованные каналы 154
3.1.4 Именованные каналы 163
3.1.5 Нелокальные переходы 165
3.1.5.1 Пример. Использование нелокальных переходов. 166
3.1.6 Трассировка процессов – модель межпроцессного взаимодействия «главный-подчиненный» 167
3.1.6.1 Пример. Общая схема использования механизма трассировки. 169
3.1.6.2 Пример. Трассировка процессов. 172
3.2 Система межпроцессного взаимодействия IPC (Inter-Process Communication) 173
3.2.1 Очередь сообщений IPC.....175
3.2.2 Разделяемая память IPC 182
3.2.3 Массив семафоров IPC.....185
3.2.3.1 Доступ к семафору 185
3.2.3.2 Операции над семафором 185
3.2.3.3 Управление состоянием массива семафоров 187
3.2.3.4 Пример - работа с разделяемой памятью и с семафорами. 187
3.3 Сокеты 190
3.3.1 Типы сокетов. Коммуникационный домен 191
3.3.2 Создание и конфигурирование сокета.....192
3.3.2.1 Создание сокета 192
3.3.2.2 Связывание.....193
3.3.3 Предварительное установление соединения. 195
3.3.3.1 Сокеты с установлением соединения. Запрос на соединение. 195
3.3.3.2 Сервер: прослушивание сокета и подтверждение соединения. 195

3.3.4 Прием и передача данных 196
3.3.5 Завершение работы с сокетом 197
3.3.6 Резюме: общая схема работы с сокетами 198
3.3.7 Примеры использования сокетов 200
3.3.7.1 Сетевое взаимодействие через сокеты, с установлением соединения. 200
3.3.7.2 Взаимодействие без предварительного установления соединения. 204
3.4 Взаимодействие нитей в ОС Unix 206
3.4.1 Создание, инициализация, концепции мьютекса 207
3.4.2 Операции над мьютексами 207
3.4.3 Простой пример синхронизации нитей 208
3.4.4 Читатели и писатели с использованием нитей 209

4 ФАЙЛОВЫЕ СИСТЕМЫ 213
4.1 Основные концепции 213
4.1.1 Структурная организация файлов 214
4.1.2 Атрибуты файлов 215
4.1.3 Основные правила работы с файлами. Типовые программные интерфейсы 216
4.1.4 Подходы в практической реализации файловой системы 219
4.1.5 Модели реализации файлов 221
4.1.6 Модели реализации каталогов 223
4.1.7 Соответствие имени файла и его содержимого 224
4.1.8 Координация использования пространства внешней памяти 225
4.1.9 Квотирование пространства файловой системы 226
4.1.10 Надежность файловой системы 227
4.1.11 Проверка целостности файловой системы 228
4.2 Примеры реализаций файловых систем 230
4.2.1 Организация файловой системы ОС Unix. Виды файлов. Права доступа 231
4.2.2 Логическая структура каталогов 232
4.2.3 Внутренняя организация файловой системы: модель версии System V 233
4.2.3.1 Работа с массивами номеров свободных блоков 234
4.2.3.2 Работа с массивом свободных индексных дескрипторов 235
4.2.3.3 Индексные дескрипторы. Адресация блоков файла 235
4.2.3.4 Файл-каталог 237
4.2.3.5 Достоинства и недостатки файловой системы модели System V 238
4.2.4 Внутренняя организация файловой системы: модель версии Fast File System (FFS) BSD 239
4.2.4.1 Стратегии размещения 239
4.2.4.2 Внутренняя организация блоков 240
4.2.4.3 Выделение пространства для файла 241
4.2.4.4 Структура каталога FFS 242
4.2.4.5 Блокировка доступа к содержимому файла 242
4.2.5 Организация обмена данными с файлами 243
4.2.6 Предопределенные номера индексных дескрипторов 245
4.2.6.1 Алгоритм поиска ИД файла при его открытии 245

5 УПРАВЛЕНИЕ ОПЕРАТИВНОЙ ПАМЯТЬЮ 246
5.1 Одиночное непрерывное распределение.....246
5.2 Распределение неперемещаемыми разделами.....247
5.3 Распределение перемещаемыми разделами.....249
5.4 Страничное распределение.....250
5.5 Сегментное распределение.....257
5.6 Сегментно-страничное распределение.....259

6 УПРАВЛЕНИЕ ВНЕШНИМИ УСТРОЙСТВАМИ.....261
6.1 Общие концепции.....261
6.1.1 Архитектура организации управления внешними устройствами.....261
6.1.2 Программное управление внешними устройствами.....262
6.1.3 Планирование дисковых обменов.....263
6.1.4 RAID-системы. Уровни RAID.....266
6.2 Работа с внешними устройствами в ОС Unix.....269
6.2.1 Файлы устройств, драйверы.....269
6.2.2 Системные таблицы драйверов устройств.....270
6.2.3 Ситуации, вызывающие обращение к функциям драйвера.....271
6.2.4 Включение, удаление драйверов из системы.....271
6.2.5 Буферизация при блок-ориентированном обмене.....272
6.2.6 Борьба со сбоями.....272

1 Введение

Настоящая книга основывается на многолетнем опыте чтения авторами курсов лекций и проведении семинарских занятий по операционным системам на факультете вычислительной математики и кибернетики Московского государственного университета им. М.В.Ломоносова.

Операционная система является одним из ключевых понятий, связанных с функционированием компьютеров и их программного обеспечения. В существующей литературе многие понятия, связанные с вычислительной техникой, определяется неоднозначно, что иногда вносит путаницу в представление полной картины того, что и как функционирует в современном компьютере. Неоднозначностью определений страдает и понятие **операционная система**. В каких-то источниках операционная система определяется, «*как система интерфейсов, предназначенная для обеспечения удобства работы пользователя с компьютером*», в каких-то — это «*посредник между программами пользователя и аппаратными средствами*», кто-то сопоставляет это понятие с «*возможностями и интерфейсами, предоставляемыми инструментальными средствами программирования и/или прикладными системами*». В целом, каждая из перечисленных интерпретаций понятия **операционная система** имеет право на существование, и природа появления того или иного представления ясна. Однако многие из используемых трактовок термина операционная система ориентированы на конкретную категорию пользователей (программист, пользователь прикладной системы, системный программист и т.п.) и не формируют целостной картины функций свойств и взаимосвязей с другими компонентами программного обеспечения и аппаратуры компьютера.

Авторы настоящей книги ставили перед собой цель выстроить систему определений и рассмотреть основные свойства и примеры реализации тех или иных аппаратных и программных компонентов, функционирующих в компьютере, в их взаимосвязи, акцентировав основное внимание на понятии **операционная система**, на основах ее построения, примерах организации тех или иных частей наиболее распространенных на сегодняшний день ОС.

История появления и развития операционных систем целиком и полностью связана с развитием и становлением аппаратных возможностей компьютеров. Рассмотрим ключевые этапы этого процесса.

Первое поколение компьютеров: середина 40-х — начало 50-х годов XX века. Компьютеры этого поколения строились на электронно-вакуумных лампах. В 1946 г. в Пенсильванском университете США была разработана вычислительная машина ENIAC (Electronic Numerical Integrator and Computer), которая считается одной из первых электронных вычислительных машин (ЭВМ). Данная машина была разработана по заказу министерства обороны США и применялась для решения задач энергетики и баллистики. Производительность таких компьютеров измерялась от сотен до тысяч команд (операций) в секунду. Компьютер состоял из процессора, оперативного запоминающего устройства и достаточно примитивных внешних устройств: устройства вывода (вывод цифровой информации на бумажную ленту), внешних запоминающих устройств (ВЗУ) — аппаратных средств хранения готовых к исполнению программы и данных (магнитные ленты), и устройства ввода, позволявшего вводить в оперативную память компьютера предварительно подготовленные на специальных носителях (перфокартах, перфоленте и пр.) программы и данные.

Изначально компьютеры первого поколения использовались в однопользовательском, персональном режиме, т.е. вся система монопольно предоставлялась одному пользователю, при этом программа и необходимые данные, представленные в машинных кодах в двоичном представлении, вводились в оперативную память, а затем запускалась на исполнение. Пользователь (программист) использовал

аппаратную консоль (или пульт управления) компьютера для ввода и запуска программы чтения данных через устройства ввода. Результат выполнения программы выводился на устройство печати. В случае возникновения ошибки работа компьютера по выполнению команд программы прерывалась, и возникшая ситуация отображалась на индикаторах пульта управления, содержание которых анализировалось программистом. Программирование в машинных кодах привносило ряд проблем, связанных с техническими сложностями написания, модификации и отладки программы. Кроме того, в задачу программиста входило кодирование всех необходимых операций ввода/вывода с помощью специальных машинных команд управления внешними устройствами. В этот период от пользователя компьютера требовались не только алгоритмические знания и навыки решения конкретной прикладной задачи, но и достаточно хорошие знания организации и использования аппаратуры компьютера, доскональное знание системы команд и кодировки, используемой в данном компьютере, знание особенностей программирования устройств ввода/вывода, т.е. программист должен был быть, отчасти, и инженером-электронщиком. На этапе существования компьютеров первого поколения появился класс программ, обеспечивающих определенные сервисные функции программирования — это ассемблеры, первые языки высокого уровня и трансляторы для этих языков, а также простейшие средства организации и использования библиотек программ. Кроме того, можно говорить о зарождении класса сервисных, **управляющих программ**: представителями этого класса являлись программы чтения и загрузки в оперативную память программ и данных с внешних устройств. Эти программы были предопределены фирмой-разработчиком компьютеров и вводились в оперативную память с использованием аппаратной консоли. Так при помощи консоли было возможно вручную ввести в оперативную память последовательность команд, составляющих управляющую программу, и осуществить ее запуск. В случае если управляющая программа размещалась на внешнем запоминающем устройстве, через аппаратную консоль вводилась последовательность команд, обеспечивающих чтение кода управляющей программы в оперативную память и передачу управления на ее точку входа.

Компьютеры второго поколения: конец 50-х годов — вторая половина 60-х годов XX века. Традиционно, с компьютерами второго поколения связывается использование полупроводниковых приборов — диодов и транзисторов, которые по функциональной емкости, размеру, энергопотреблению в десятки раз превосходили возможности электронно-вакуумных ламп. В результате компьютеры второго поколения стали обладать существенно более развитыми логическими возможностями и в тысячи раз превосходили компьютеры первого поколения по производительности. Широкое распространение получили новые высокопроизводительные внешние устройства ЭВМ. Все это определило и активное совершенствование программного обеспечения и способов использования компьютеров. Можно с уверенностью утверждать, что реальное зарождение понятия операционной системы связано именно с появлением и совершенствованием архитектуры компьютеров второго поколения.

Этапом в развитии форм использования компьютеров стала **пакетная обработка заданий**, суть которой состояла в следующем. В компьютере работала специальная управляющая программа, в функции которой входила последовательная загрузка в оперативную память и запуск на выполнение программ из заранее подготовленного пакета программ. Пакет программ физически может быть представлен в виде «большой» стопки перфокарт, в которой программы находятся последовательно. В этом случае управляющая программа по завершении выполнения текущей программы осуществляет чтение очередной программы через устройство считывания перфокарт, загрузку ее в оперативную память и передача управления на фиксированную точку входа в программу (адрес памяти с которого должно начинаться выполнение программы). По завершении программы или после возникновения в программе ошибки, которая вызывает аварийную остановку выполнения программы, управление передается в управляющую программу (Рис. 1).

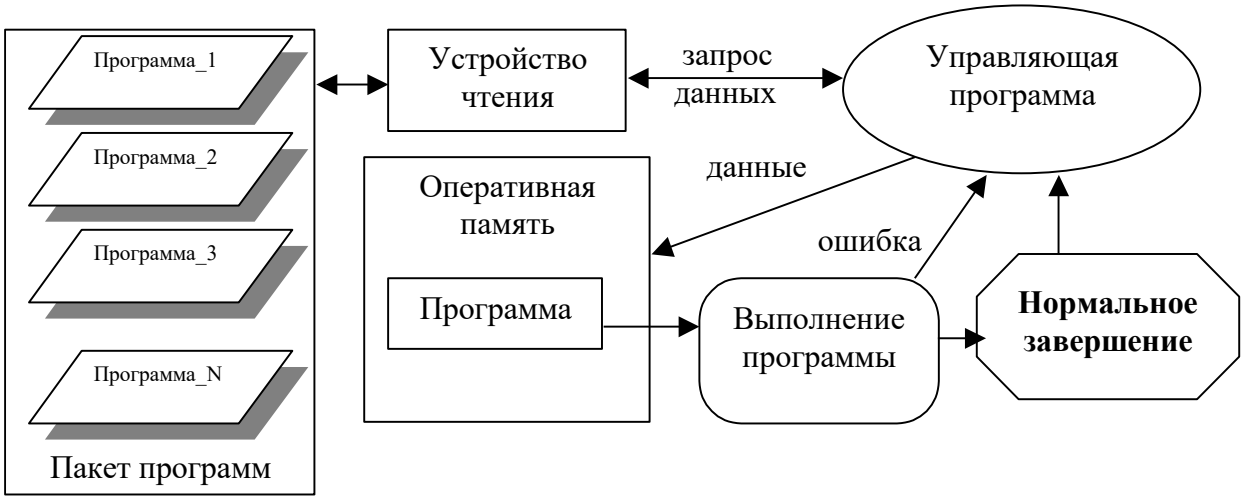


Рис. 1. Пакетная обработка заданий.

Такой процесс продолжался до тех пор, пока все программы из пакета не будут выполнены. Развитие подобных управляющих программ послужило основой появлению первых прообразов операционных систем, которые в разных случаях назывались **мониторными системами**, **супервизорами** или **диспетчерами**.

Следующим этапом развития понятия операционная система стало появление компьютеров второго поколения, имевших аппаратную поддержку режима **мультипрограммирования** — режима, при котором одновременно находилась в обработке не одна, а несколько программ. При этом в каждый момент времени команды одной из обрабатываемых программ выполнялись процессором, другие выполняли обмен данными с внешними устройствами, третьи были готовы к выполнению процессором и ожидали своей очереди. В СССР представителем машин второго поколения, обеспечивавших поддержку мультипрограммной обработки, была вычислительная машина БЭСМ-6, созданная под руководством академика С.А.Лебедева. Для данного компьютера была разработана серия операционных систем, которые по своей структуре и основным функциям были достаточно близки к современным ОС (НД-69, НД-70, ОС ДУБНА, ДИСПАК, ОС ИПМ и др.). Прародительницей подавляющего большинства этих операционных систем была система под названием Д-68 (Диспетчер–68), разработанная под руководством Л.Н.Королева.

Развитие мультипрограммных систем, расширение спектра решаемых задач и существенное увеличение количества пользователей компьютеров потребовало развития «дружественности» интерфейсов между пользователем и системой. С точки зрения инструментальных средств программирования это развитие языков программирования и систем программирования, которое представимо в следующей эволюционной последовательности: *система команд компьютера* ➔ *автокоды и ассемблеры* ➔ *языки программирования высокого уровня* ➔ *проблемно-ориентированные языки программирования* (Рис. 2).

Операционные системы также получили свое развитие в этот период времени: появились **языки управления заданиями**, которые позволяли пользователю до начала выполнения его программы сформировать набор требований по организации выполнения программы. Появились первые прообразы **современных файловых систем** — систем, позволяющих систематизировать и упростить способы хранения и доступа пользователей к данным, размещенным на внешних запоминающих устройствах, что позволило пользователю работать с данными во внешней памяти не в терминах физических устройств и координат местоположения данных на этих физических устройствах, а в терминах имен или адресов некоторых наборов данных. В связи с этим у пользователя появилась возможность абстрагироваться от знания особенностей и способов организации хранения и

доступа к данным конкретных физических устройств, что во многом послужило основой для появления **виртуальных устройств**.

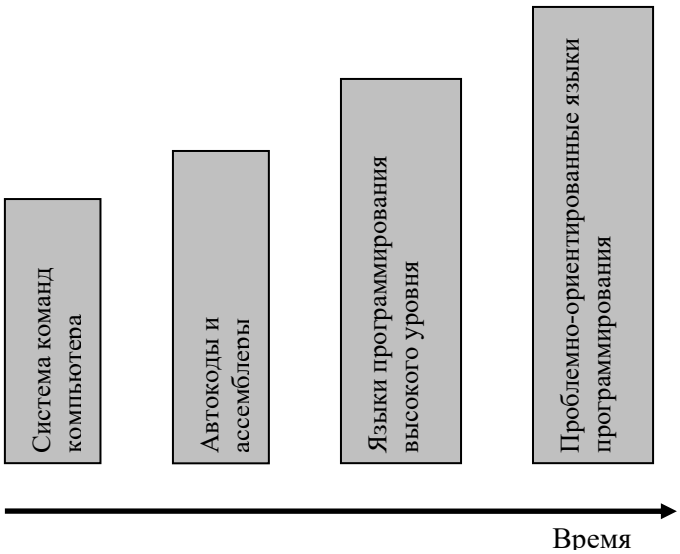


Рис. 2. Развитие языков и систем программирования.

Компьютеры третьего поколения: конец 60-х — начало 70-х годов XX века. Основным отличием компьютеров этого поколения было использование в качестве элементной базы интегральных схем, что определило увеличение производительности компьютеров, существенное снижение их размеров, веса, появление новых, высокопроизводительных внешних устройств. И, наверное, главной особенностью архитектуры компьютеров третьего поколения было начало аппаратной унификации их узлов и устройств, позволившей стимулировать создание семейств компьютеров, аппаратная комплектация которых могла достаточно просто варьироваться владельцем компьютера. Наиболее яркими представителями таких семейств были компьютеры серий IBM-360 фирмы IBM и семейство малых компьютеров PDP-11 фирмы DEC. Компьютеры первых двух поколений строились, как единые, аппаратно-целостные устройства, комплектация и возможности которых были существенно предопределены на этапе их производства. Их аппаратная модификация, обычно, была крайне затруднительна. Третье поколение компьютеров строилось на модульном принципе, что позволяло, при необходимости, осуществлять замену и расширение состава внешних устройств, увеличивать размеры оперативной памяти, заменять процессор на более производительный. Все это повлияло и на развитие и структуру операционных систем, которые вслед за аппаратурой приобрели модульную организацию с унификацией межмодульных интерфейсов. В операционных системах появились специальные программы управления устройствами — **драйверы устройств**, которые имели стандартные интерфейсы, позволявшие при аппаратной модификации компьютера достаточно просто обеспечивать программный доступ к новым или модифицированным устройствам. Кроме того, для обеспечения простоты и «дружественности» общения пользователя с различными устройствами компьютера появились виртуальные устройства, драйверы которых предоставляли пользователю набор единых правил работы с группой внешних устройств, что позволило создавать программы, не зависящие от типов используемых внешних устройств. Операционные системы компьютеров третьего поколения предоставляли новые режимы использования компьютеров, одним из таких режимов был диалоговый режим доступа к компьютеру. Вершиной идей, заложенных в операционные системы компьютеров третьего поколения, стала операционная система Unix, которая открыла направление развития комплексной стандартизации

пользовательских интерфейсов, как на уровне интерфейсов командных языков, так и на различных уровнях программных интерфейсов от правил взаимодействия с драйверами устройств до интерфейсов с прикладными системами.

Завершение формирования сегодняшнего понятия операционной системы может быть связано с появлением **четвертого и последующих поколений** компьютеров, в построении которых использовалась элементная база, основанная на больших интегральных схемах. Компьютеры четвертого поколения, в первую очередь, ассоциируются с персональными компьютерами, совершившими в полном смысле слова революцию в массовом распространении информационных технологий. Компьютер из инструмента прикладного программиста стал повседневным, массово распространенным и доступным оборудованием. В связи с этим возник целый ряд проблем, решение которых потребовалось в операционных системах. В первую очередь это совершенствование «дружественности» пользовательских интерфейсов, упрощающих взаимодействие пользователя и операционной системы. Здесь лидирующую позицию занимают операционные системы компании Microsoft, которые в полном смысле слова совершили революцию в обеспечении массовости освоения компьютера. Активное развитие получили сетевые технологии, что привело к появлению сетевых и распределенных операционных систем. В этот период времени наибольшее развитие получила всемирная сеть Internet. В свою очередь возникли задачи обеспечения операционными системами безопасности хранения и передачи данных.

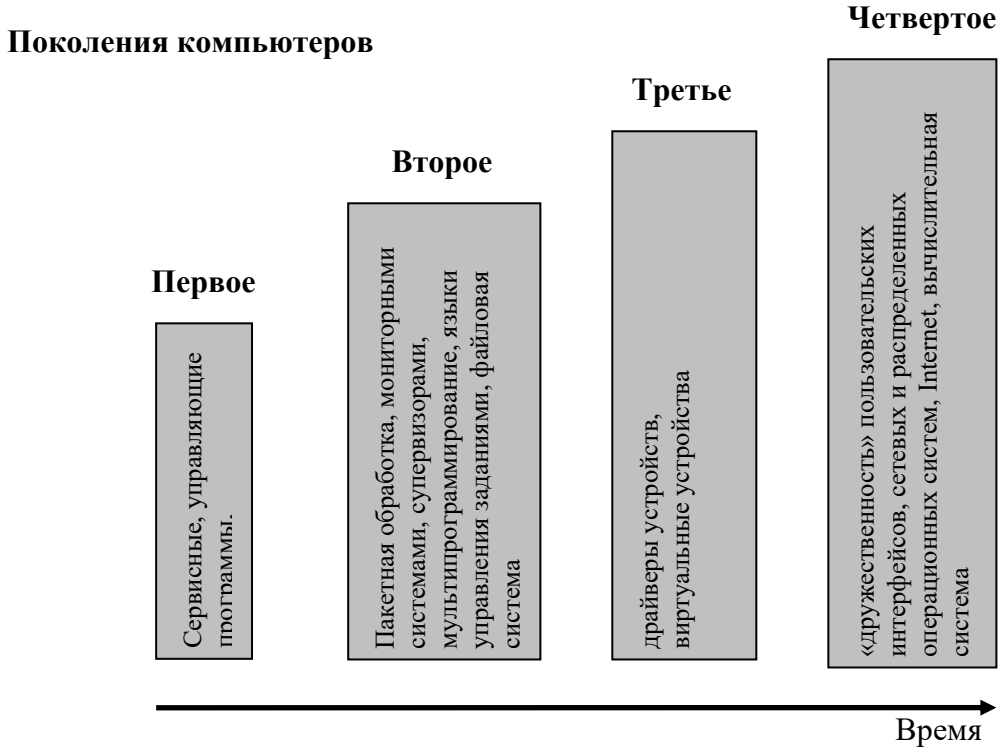


Рис. 3. Этапы эволюции.

На данном этапе в результате эволюции понятий образовалось достаточно полное и однозначное определение того, что называется операционной системой, определена типовая структура операционной системы и функции ее основных компонентов. Сформировались принципиально новые разновидности операционных систем и режимов использования компьютеров.

Следует отметить тот факт, что развитие компьютеров, системного программного обеспечения, методов применения вычислительной техники показали, что единственным периодом истории, когда аппаратная часть разрабатывалась исключительно в качестве

вычислителя без учета потребностей поддержки решения задач организации вычислительного процесса был период создания и производства компьютеров первого поколения. На сегодняшний день аппаратура и программное обеспечение современных компьютеров представляют единую взаимозависимую **вычислительную систему**, в которой многие функции операционной системы нельзя рассматривать вне контекста аппаратной поддержки компьютера, а многие аппаратные возможности сложно рассматривать вне контекста операционных систем (Рис. 3).

1.1 Основы архитектуры вычислительной системы

Современный компьютер и его программное обеспечение невозможно рассматривать в отдельности друг от друга. Рассматривая функционирование компьютера, мы всегда имеем в виду функционирование системы, в которой интегрированы аппаратура компьютера и его программное обеспечение. Результатом этой интеграции является **вычислительная система** (**ВС**), возможности и эксплуатационные качества которой определяются как аппаратурой компьютера, так и функционирующим на нем программным обеспечением. ***Вычислительную систему*** можно определить, как совокупность аппаратных и программных средств, функционирующих в единой системе и предназначенных для решения задач определенного класса. Рассмотрим структурную организацию вычислительной системы.

1.1.1 Структура ВС

Традиционным представлением структуры вычислительной системы является пирамида (Рис. 4). Каждый из уровней пирамиды определяет свой уровень абстракции свойств вычислительной системы. Основанием является **аппаратный уровень** вычислительной системы — это основа всей иерархии, так как многие характеристики и функциональные возможности последующих программных уровней существенно определяются свойствами аппаратуры компьютера, находящегося в основе вычислительной системы.

Представление о возможностях и свойствах конкретной вычислительной системы формируются с позиций каждого из уровней структурной организации. Так вычислительная система представляется пользователю прикладной системы, работающей на компьютере, в виде совокупности возможностей этой прикладной системы. Примером может служить игровой автомат, являющийся компьютером, на котором функционирует операционная система, а также игровая программа, которая в данном случае является прикладной системой. Пользователю данной специализированной вычислительной системы ее свойства представляются в виде функциональных возможностей игровой программы и итоговой производительностью системы (если процессор компьютера маломощный, до динамика игры может быть недостаточной).

Другой пример — компьютер, используемый для обучения школьников языку программирования. Это означает, что для школьника или его учителя свойства данной вычислительной системы будут представляться с позиций уровня системы программирования, построенной на основе транслятора языка программирования, на котором идет обучение. Представление свойств вычислительной системы в данном случае будет формироваться из пользовательского интерфейса системы программирования в сочетании со свойствами и производительностью аппаратных компонентов компьютера.

Рис. 4. Структура вычислительной системы.

Взаимосвязи уровней иерархической структуры вычислительной системы, их характеристики могут проявляться как в виде непосредственных межуровневых интерфейсов, определенных однозначным набором правил использования объектов одного уровня другим, так и косвенным влиянием одного уровня на другой. Примером подобного косвенного взаимодействия может служить влияние, оказываемое на характеристики функционирования всей вычислительной системы в целом, производительности или емкости аппаратных компонентов компьютера (внешних устройств, процессора, оперативной памяти, линий связи и пр.). В качестве иллюстрации рассмотрим вычислительную систему, имеющую канал связи, обеспечивающий доступ в Интернет со скоростью 64 Kbps. Данная система сможет обеспечить достаточно комфортные условия для интенсивной работы в Интернете одного–двух пользователей. Если количество пользователей возрастет до 10, будут возникать задержки при обработке запросов, что снизит качество работы пользователей. При росте числа пользователей до 100, организовать их интенсивную работу в Интернете на данной вычислительной системе не представляется возможным, т.к. пропускная способность канала связи не справится с потоком запросов, поступающих от пользователей. Таким образом, проявляется косвенное влияние пропускной способности канала связи на эксплуатационные характеристики (или качества) вычислительной системы.

Рассмотрим основные характеристики и суть взаимосвязи уровней пирамиды, представляющей структуру вычислительной системы.

1.1.2 Аппаратный уровень ВС

Итак, аппаратный уровень вычислительной системы определяется набором аппаратных компонентов и их характеристик, используемых вышестоящими уровнями иерархии и оказывающих влияние на эти уровни. С позиций уровней, расположенных выше, аппаратный уровень предоставляют т.н. **физические ресурсы**, или **физические устройства** вычислительной системы. Каждому физическому ресурсу соответствуют определенные аппаратные компоненты компьютера и их характеристики. Физическими ресурсами являются процессор компьютера, оперативная память, внешние устройства, входящие в состав компьютера. Каждому физическому ресурсу вычислительной системы обычно соответствуют следующие характеристики:

– **правила программного использования**, определяющие возможность корректного использования данного ресурса в программе (для процессора компьютера эти правила описывают машинный язык — систему команд

данного компьютера, на основании которой возможно построение работающей программы, для внешнего устройства компьютера подобные правила описывают способы программного управления данным устройством, к примеру, это могут быть специальные команды ввода-вывода процессора);

- параметры физического ресурса, характеризующие его **объемные характеристики и/или производительность** (для процессора компьютера таким параметром может служить его тактовая частота, а для внешнего запоминающего устройства — объем информации, которая может храниться на данном устройстве и скорость доступа);
- степень использования данного физического ресурса в вычислительной системе — это параметры, которые характеризуют **степень занятости или используемости** данного физического ресурса (для процессора компьютера такой характеристикой является время его работы, затраченное на выполнение программ пользователей, для оперативного запоминающего устройства это будет объем используемой памяти, для линий связи — это ее загруженность).

В принципе нет единого правила формирования этих характеристик для любого физического ресурса: они зависят от конкретного устройства компьютера, от архитектуры компьютера, от стратегии использования данного ресурса. Так, например, для одного и того же внешнего устройства правила его программного использования могут существенно отличаться от того, каким образом данное устройство подключено к компьютеру. Об этом более подробно будет рассказано несколько позднее, в пункте, посвященном внешним устройствам компьютера (см. раздел 2). Тем не менее, данные характеристики служат для обеспечения взаимосвязи аппаратного уровня вычислительно системы с последующими уровнями иерархии.

Если мы будем рассматривать уровни организации вычислительной системы с точки зрения возможностей и средств программирования, то на аппаратном уровне пользователю вычислительной системы предоставлены в качестве средств программирования система команд компьютера и аппаратные интерфейсы программного взаимодействия с физическими ресурсами, что на самом деле практически полностью совпадает со средствами программирования, которые были доступны программистам на ранних этапах освоения компьютеров первого поколения.

1.1.3 Управление физическими ресурсами ВС

Уровень управления физическими ресурсами — это первый уровень **системного программного обеспечения** вычислительной системы. Его назначение — систематизация и стандартизация правил программного использования физических ресурсов. Для иллюстрации проблемы вернемся во времени к компьютерам первого поколения. Начальный этап зарождения вычислительной техники был этапом структурного «хаоса»: вычислительная система представлялась двухуровневой моделью, состоящей из уровня аппаратуры компьютера и уровня всего программного обеспечения. Программа пользователя включала в себя как кодовую часть, реализующую решение конкретной прикладной задачи, так и часть, которая обеспечивала взаимодействие с физическими устройствами компьютера (в большинстве случаев речь шла об управлении внешними устройствами компьютера). Программирование управления физическими устройствами — достаточно кропотливая работа, при которой необходимо учитывать сложную логику организации взаимодействия с конкретным устройством компьютера. Для адаптации возможности программы для работы с другими типами устройств требовалась существенная модификация кода программы в части, обеспечивающей это взаимодействие, что приводило к существенным трудозатратам, а также снижало надежность программы из-за роста риска внесения ошибок в логику ее работы.

Частичным решением этих проблем стало появление специальных стандартных программ — **драйверов физических ресурсов** (или **драйверов физических устройств**). ***Драйвер физического устройства*** — программа, основанная на использовании команд управления конкретного физического устройства и предназначенная для организации работы с данным устройством. Драйвер физического устройства скрывает от пользователя детальные элементы управления конкретным физическим устройством и предоставляет пользователю упрощенный программный интерфейс работы с устройством. Интерфейс драйвера физического устройства ориентирован на конкретные свойства устройства. Таким образом, в вычислительной системе, способной одновременно работать со значительным количеством устройств (устройства печати, устройства внешней памяти и т.п.), пользователю становится доступным спектр драйверов физических устройств, каждый из которых имеет свои особенности использования. Драйвер физического устройства стал неотъемлемой частью самого физического устройства и в большинстве случаев разрабатывался производителем устройства вместе с сами устройством.

Совокупность драйверов физических устройств составляет уровень **управления физическими устройствами** вычислительной системы. Уровень управления физическими устройствами стандартизует правила, по которым возможно внесение в систему новых драйверов устройств. Следует отметить, что в системе для одного и того же физического устройства возможно наличие нескольких различных драйверов, которые имеют различные пользовательские интерфейсы, а также предоставляют различные возможности. Примером может служить устройство магнитной лены, которое в зависимости от драйвера может сохранять информацию либо в виде последовательности блоков одинакового размера, либо в виде логических записей произвольного размера (Рис. 5).

Рис. 5. Пример зависимости от драйвера.

Таким образом, на уровне управления физическими ресурсами (устройствами) вычислительной системы пользователю доступна система команд компьютера, а также интерфейсы драйверов физических устройств компьютера.

Появление уровня управления физическими устройствами упростило процесс адаптации программы для работы с различными типами и разновидностями устройств, а также существенно повысило надежность программирования и снизило уровень требований к программисту о знании специфики управления конкретными устройствами. Однако использование исключительно уровня драйверов физических устройств оставило ряд специфических проблем:

- программист должен быть «знаком» со всеми интерфейсами драйверов используемых физических устройств;
- программы пользователей, использующие конкретные драйверы физических устройств, должны модифицироваться каждый раз, когда возникает необходимость

использовать другие физические устройства данного типа (это работа несоизмеримо проще той, которая выполнялась, когда внешнее устройство непосредственно программировалось в программе пользователя, но, тем не менее, в программу необходимо внести изменения, позволяющие использовать другой драйвер с другими интерфейсами).

1.1.4 Управление логическими/виртуальными ресурсами

Развитием системного программного обеспечения стало появление уровня **управления логическими**, или **виртуальными**, **ресурсами** (или устройствами). В основу этого уровня легло обобщение особенностей физических устройств одного вида и создание драйверов, имеющих единые интерфейсы, посредством которых осуществляется доступ к различным физическим устройствам одного типа. Для этих целей в современных вычислительных системах предусмотрена возможность программного создания и использования т.н. ***логических***, или ***виртуальных, ресурсов*** (виртуальное —нечто реально не существующее, не имеющее реальной, физической организации). ***Логическое/виртуальное устройство*** (***ресурс***) — это устройство/ресурс, некоторые эксплутационные характеристики которого (возможно все) реализованы программным образом. Современные вычислительные системы позволяют создавать разнообразные логические/виртуальные устройства и соответствующие им драйверы. ***Драйвер логического/виртуального ресурса*** — это программа, обеспечивающая существование и использование соответствующего ресурса. Для этих целей при его реализации возможно использование существующих драйверов физических и виртуальных устройств. Возможно построение достаточно развитой иерархии логических устройств. Например, на Рис. 6 изображена упрощенная схема организации ввода-вывода в системе. Она включает в себя многоуровневую иерархию виртуальных и физических устройств и соответствующих им драйверов, по степени обобщения которых можно выделить следующие группы.

A. Драйверы физических устройств — обеспечивают доступ к конкретным физическим устройствам. Например, драйвер жесткого диска фирмы IBM модели Deskstar или драйвер жесткого диска фирмы Seagate модели Barracuda 3. Каждый из данных драйверов имеет особенности, характеризующие конкретное устройство, отраженные в соответствующем интерфейсе.
B. Драйверы виртуальных устройств определенного типа (например, драйвер виртуального диска), предоставляют обобщенные интерфейсы доступа к разнообразным физическим устройствам данного типа. Данные драйверы имеют связи с драйверами конкретных физических устройств данного типа. Запрос к данному драйверу виртуального устройства обычно транслируется драйверу конкретного физического устройства и, в конечном итоге, управляющие команды получит само устройство. Кроме того, возможна «реализация» виртуального устройства определенного типа на устройствах других типов, например, возможна организация работы с виртуальным диском, реализованном на пространстве оперативной памяти, в этом случае драйвер виртуального устройства имеет связь с драйверами физических устройств других типов.
C. Драйверы виртуальных устройств, которым затруднительно поставить в соответствие физическое устройство или группу физических устройств определенного типа. Примером могут служить драйверы различных файловых систем (файловая система — программный компонент вычислительной системы, обеспечивающий именованное хранение и доступ к данным).

Основным результатом появления уровня управления виртуальными устройствами вычислительной системы стала многоуровневая унификация интерфейсов доступа к ресурсам вычислительной системы, что существенно упростило проблему программирование устройств компьютера, а также предоставило качественно новые возможности в функционировании вычислительных систем и в создании их программного

обеспечения. Примером могут служить файловые системы, которые обеспечивают простые и надежные интерфейсы именованного хранения и использования данных, полностью скрывая от пользователя проблемы ее внутренней организации. К примеру, пользователь современной вычислительной системы может не только не знать, на каком внешнем запоминающем устройстве размещены данные его файлов, он может не знать и территориальное расположение и тип компьютера, на котором хранятся его данные. Существенное развитие получили средства **управления виртуальными устройствами** (ресурсами), которые обеспечивают контроль за созданием и использованием ресурсов вычислительной системы.

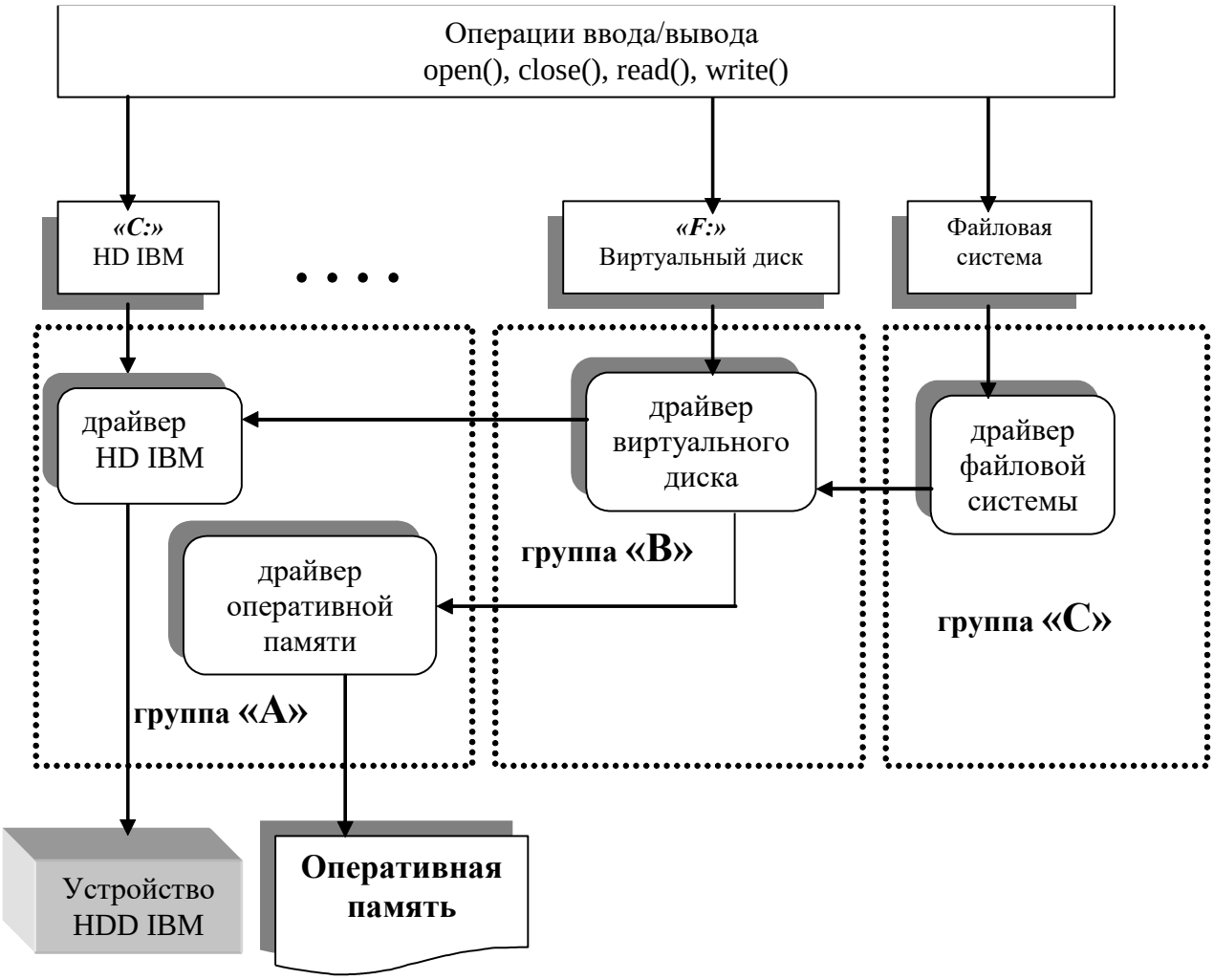


Рис. 6. Схема организации ввода-вывода в системе.

Итак, мы рассмотрели два первых программных уровня структуры вычислительной системы — это уровни, обеспечивающие функционирование ресурсов в вычислительной системе. Под ***ресурсами вычислительной системы*** мы будем понимать совокупность всех физических и виртуальных ресурсов. Одной из характеристик ресурсов вычислительной системы является их конечность. То есть рано или поздно в системе возникает конкуренция за обладание ресурсом между его программными потребителями. При этом если речь идет о таком виртуальном ресурсе, как файловая система, то конечным является размер файловой системы на устройствах хранения данных, ограничения на предельное количество зарегистрированных в файловой системе файлов. Именно за эти параметры возможно возникновение конкуренции при использовании файловой системы. А теперь попытаемся вернуться к проблеме определения понятия операционной системы. ***Операционная система*** — это комплекс программ, обеспечивающий управление

ресурсами вычислительной системы. Это основная концепция данного понятия. Позднее мы будем уточнять это определение, рассматривать отдельные функции ОС. В структурной организации вычислительной системы операционная система представляется уровнями управления физическими и виртуальными ресурсами.

С точки зрения средств программирования, доступных на уровне управления виртуальными ресурсами, пользователю предоставляются система команд компьютера, а также интерфейсы, обеспечивающие доступ к устройствам компьютера (как физическим, так и виртуальным). Доступная пользователю совокупность интерфейсов устройств компьютера может включать в себя как аппаратные интерфейсы доступа к устройствам, так и драйверы физических и/или виртуальных устройств. Конкретный состав интерфейсов определяется свойствами вычислительной системы, соответствующими, уровнями управления ресурсами, а также привилегиями пользователя (об этом подробнее мы будем говорить несколько позднее).

1.1.5 Системы программирования

Прежде чем начать рассматривать следующий уровень структурной организации вычислительных систем, обратимся к последовательности этапов, традиционно связываемых с разработкой и внедрением программных систем. Совокупность этих этапов составляют ***жизненный цикл программы*** в вычислительной системе. Остановимся на основных задачах, решаемых на каждом из этапов жизненного цикла программы. Следует отметить, что мы будем рассматривать традиционное, неформальное определение этапов жизненного цикла программы, которые сформировались естественным образом в процессе появления и развития вычислительной техники и программного обеспечения. На сегодняшний день существуют международные стандарты, которые формализуют понимание жизненного цикла программы (например, ISO/IEC 12207: 1995 “Information Technology — Software Life Cycle Processes), но это стандарты, соответствующие исключительно сегодняшнему пониманию этого термина и связанные во многом с существующими на сегодня технологиями программирования.

Проектирование программной системы. На данном этапе принимаются решения, традиционно включающие в себя следующие шаги.

- Исследование решаемой задачи, формирование концептуальных требований к разрабатываемой программной системе.
- Определение характеристик **объектной вычислительной системы** — характеристик аппаратных и программных компонентов вычислительной системы, в рамках которой будет работать создаваемая программная система.
- Построение моделей функционирования автоматизируемого объекта.
- Определение характеристик **инструментальной вычислительной системы** — вычислительной системы, которая будет использоваться при создании программной системы. Зачастую характеристики объектной и инструментальной вычислительной системы совпадают: тип вычислительных систем, на которых в дальнейшем будет работать программная система, совпадает с типом вычислительной системы, которая использовалась при разработке. Однако, в общем случае это не совсем так. Тип и качества инструментальных вычислительных систем могут в корне отличаться от соответствующих характеристик объектных ВС. Примером может служить программирование специализированных вычислительных систем, предназначенных для управления технологическими процессами. Очевидно, что специализированная вычислительная система, которая управляет навигационной системой космического спутника, не должна обладать возможностями разработки на ней программного обеспечения. Специализация данной системы ориентирована на решение конкретных, достаточно специальных задач (например, обработки сигналов, поступающих от радаров). Программное обеспечение для подобной вычислительной системы может

разрабатываться отдельно, на вычислительной системе, предназначенной для этих целей.

- Выбор основных алгоритмов, инструментальных средств, которые будут использованы при программировании, а также разработка архитектуры программного решения, включающей разбиение программного решения на основные модули и определение информационных связей между модулями системы, а также правила взаимодействия с объектной вычислительной системой.
- Априорная оценка ожидаемых результатов. Один из важнейших шагов проектирования программной системы, заключающийся в предварительной оценке характеристик проектируемого решения до начала его практической реализации. Для этих целей используются различные методы моделирования. Наличие априорной оценки ожидаемых результатов проектирования программной системы позволяет существенно повысить качество программного продукта, который будет создан на основании результатов этапа проектирования, а также сократить затраты на его создание.

Данная последовательность шагов является достаточно укрупненной, и не всегда проектирование разбивается на линейную последовательность этих шагов. Часто проектирование представляет собою итерационный процесс, в котором возможны неоднократные возвраты к тем или иным шагам (Рис. 7).

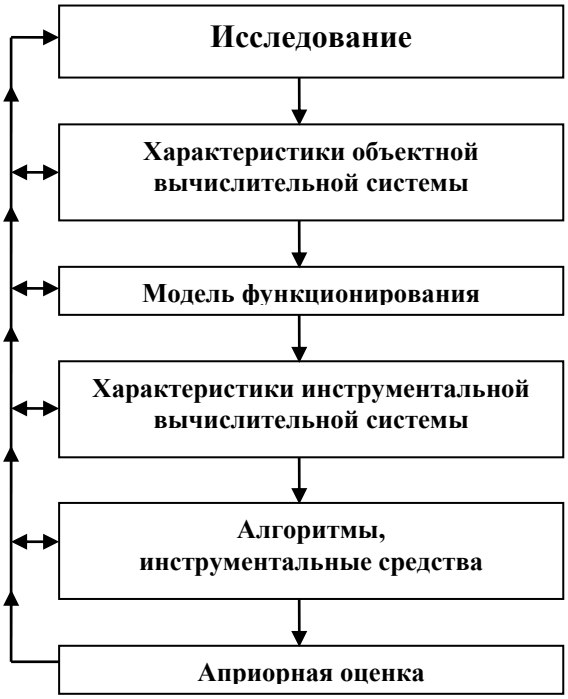


Рис. 7. Этапы проектирования.

Следующий этап жизненного цикла программы — **кодирование** (**программная реализация**, или **реализация**). Это этап построения кода программой системы на основании спецификаций, полученных при ее проектировании. На данном этапе используются инструментальные средства программирования:

- трансляторы языков программирования, средства поддержки и использования библиотек программ, формирования модулей, которые могут исполняться в вычислительной системе;
- средства управления разработкой программных продуктов коллективом разработчиков.

Результатом этапа кодирования является реализация программной системы, которая может представляться в виде совокупности исходных модулей программы, объектных или библиотечных модулей, а также модулей исполняемого кода разрабатываемой программной системы (Рис. 8). Большое значение для разработки больших, логически

сложных программных систем имеют средства управления разработкой программных продуктов, которые позволяют организовать эффективную коллективную работу над реализацией программного проекта. Традиционно они включают в себя следующие компоненты:

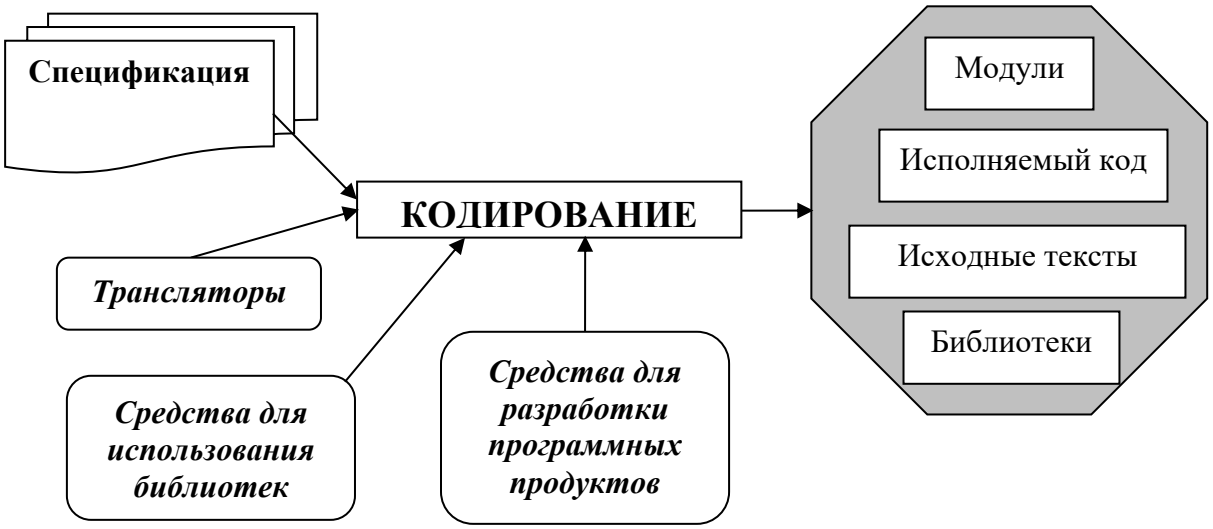


Рис. 8. Кодирование.

- средства автоматизации контроля использования межмодульных интерфейсов, которые обеспечивают контроль правильности использования в программе спецификаций, регламентирующих межмодульные связи (количество, тип, права доступа к параметрам, обеспечивающим межмодульной взаимодействие в программе);
- средства автоматизации получения объектных и исполняемых модулей программы, обеспечивающие автоматический контроль за соответствием исходных модулей объектным и исполняемым модулям (так, если в проекте появилась новая редакция некоторого исходного модуля, то при запуске этого средства автоматически произойдет последовательность действий, обновляющих объектные и исполняемые модули, зависящие от данного исходного модуля);
- система поддержки версий — система, позволяющая фиксировать состояние разработки программного проекта (создание версии проекта) и, при необходимости, возвращаться в разработке к той или иной версии проекта.

Этап **тестирования и отладки** программной системы. Можно представить программу в виде некоторого автомата, получающего на входе исходные данные, а на выходе формирующий результат (Рис. 9). Одной из задач проектирования программной системы является определение ее правил функционирования, точнее, правил, по которым для входных данных формируются выходные данные (или результаты). Тестирование программы — процесс проверки правильности функционирования программы на заранее определенных наборах входных данных — **тестах**, или **тестовых нагрузках**. В общем случае, говорить о "правильности" программы вообще не совсем корректно. Мы можем говорить о правильности функционирования программы на некоторых наборах тестов. Таким образом, при тестировании выявляется работоспособность программы на данном тесте (или на наборе тестов) или имеющаяся в программе ошибка. Понятно, что для любой программы абсолютно полным тестом является перебор всевозможных входных данных программы, но множество таких тестов настолько велико, что обработать их не представляется возможным. Поэтому актуальной задачей в тестировании является решение проблемы формирования минимального набора тестов или тестовых нагрузок, наиболее полно проверяющих функциональность программы (**тестовое покрытие**).

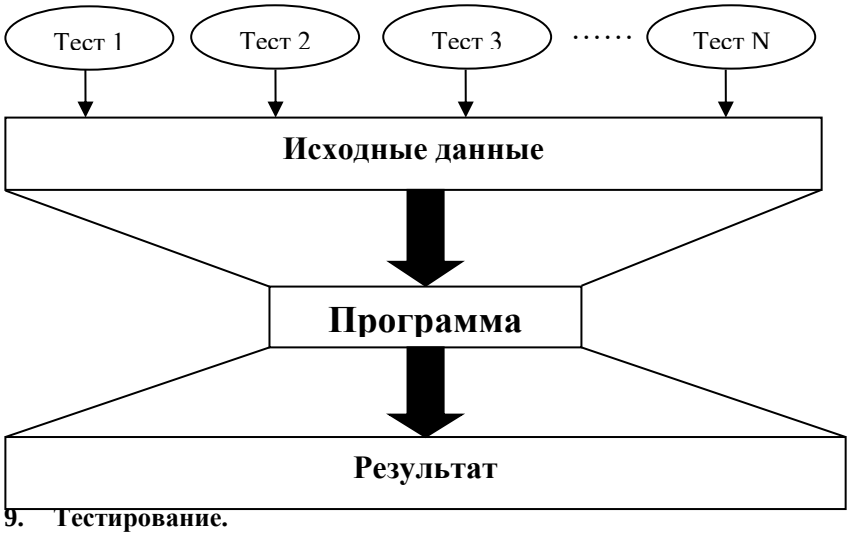


Рис. 9. Тестирование.

Другим компонентом данного этапа является **отладка**. Отладка — это поиск, локализация и исправление зафиксированных при тестировании или в процессе эксплуатации ошибок. Для обеспечения процесса отладки используются специальные программные средства — отладчики. Средства отладки существенно зависят от типа и назначения создаваемой программной системы.

Этап **ввода программной системы в эксплуатацию (внедрение) и сопровождения**. Немаловажным этапом жизненного цикла программы в вычислительной системе является этап, связанный с представлением разрабатываемой программной системы в качестве программного продукта. Одним из основных требований, предъявляемых к программному продукту, является возможность эксплуатации соответствующей программной системы без постоянного участия разработчика программы. Это достигается, с одной стороны, соответствующей надежностью программы (для этого программа должна быть максимально полно протестирована и устойчива к всевозможным комбинациям входных данных), а с другой стороны — это наличие подробной и адекватной программе документации, необходимой для всех категорий пользователей данной программной системы (пользователь, системный программист, администратор, оператор и т.п.).

Итак, мы рассмотрели основные этапы жизненного цикла программы в вычислительной системе. При создании различных программных систем, при использовании различных технологий разработки данные этапы могут выполняться как линейно, так и итерационно, с возвратами от одного этапа к другому, последовательными уточнениями спецификаций и расширением реализации программной системы. Современные технологии разработки программного обеспечения специфицируют различные модели организации жизненного цикла программной системы. Традиционная модель — **каскадная модель** (Рис. 10) — представляет разработку в виде строго линейной последовательности этапов, каждый из которых заканчивается фиксацией результата, и только после этого начинается последующий.

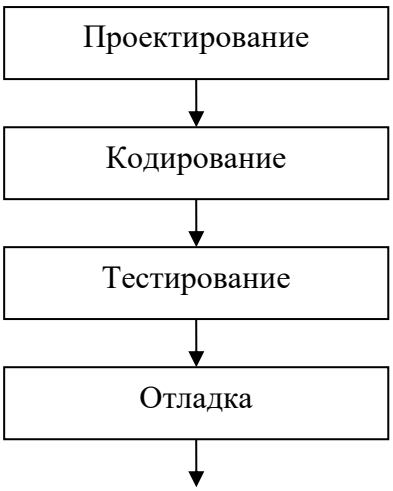


Рис. 10. Каскадная модель.

В определенном смысле эта модель является вырожденной, т.к. соблюсти эти правила на практике достаточно сложно. Примером может служить связка этапа тестирования и отладки с предшествующими этапами, которая по своей сути итерационна (после обнаружения и локализации ошибки зачастую необходимо вернуться к этапу кодирования, а возможно и проектирования). Прагматическим развитием каскадной модели является **каскадная итерационная модель** (Рис. 11), которая в общем случае, предоставляет возможность осуществления анализа полученных на этапе результатов и возврат к любому предшествующему этапу.

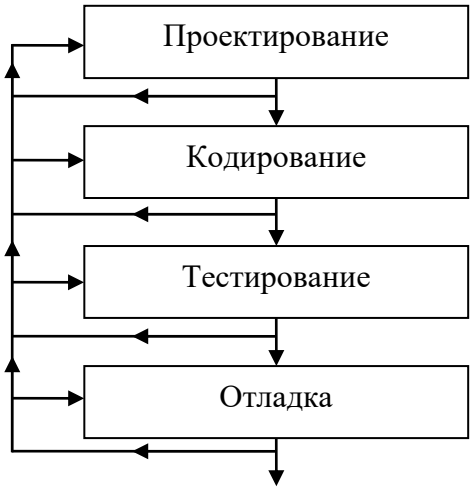


Рис. 11. Каскадная итерационная модель.

Современные технологии разработки программного обеспечения помимо каскадной модели используют и другие модели организации жизненного цикла программных систем. В частности, популярной является **спиральная модель** организации жизненного цикла (Рис. 12).

Данная модель основана на том, что процесс разработки программной системы складывается из последовательности "спиралей", каждая из которых включает этапы проектирования, кодирования, тестирования и получения результата. Под результатом понимается очередная детализация проекта и получение последовательности программ — **прототипов**. Прототип — программа, реализующая частичную функциональность и внешние интерфейсы разрабатываемой системы. Последовательность прототипов, в

конечном счете, сходится к реализации программной системы. А детализации проекта, в итоге, превращаются в полный проект системы.

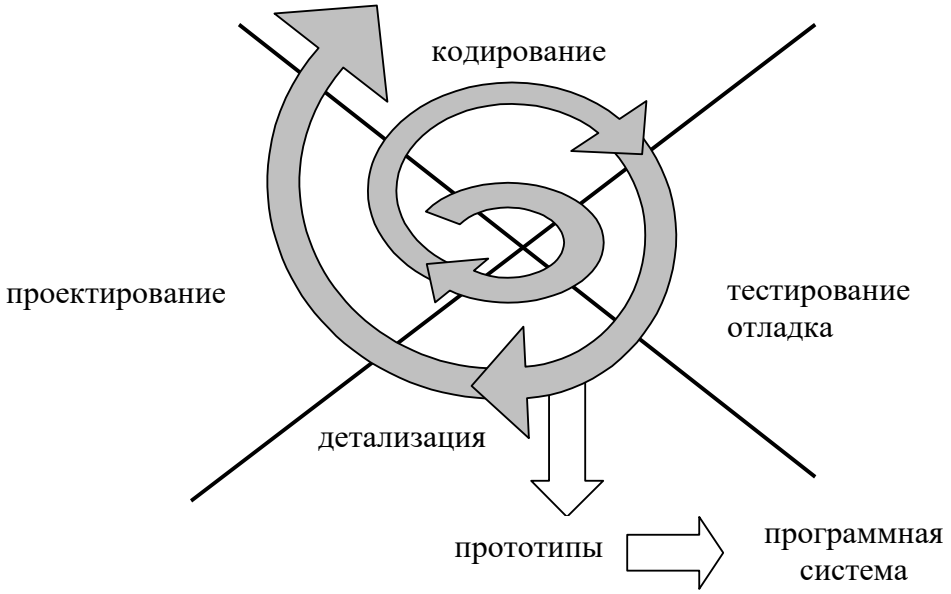


Рис. 12. Спиральная модель.

Вернемся к рассмотрению следующего уровня иерархической организации вычислительных систем — к **уровню систем программирования**. ***Система программирования*** — комплекс программ, обеспечивающий поддержание этапов **жизненного цикла программы в вычислительной системе**. Этапы жизненного цикла программы оставались в той или иной мере неизменными с момента зарождения вычислительных систем, т.к. всегда были и решались проблемы проектирования программной системы, кодирования, тестирования и отладки, подготовки эксплуатационной документации и сопровождения. В тоже время, определение системы программирования как комплекса программных средств, предназначенных для автоматизации этапов жизненного цикла программы, изменялось постоянно вместе с появлением и развитием данных средств. Рассмотрим развитие состава и основных функций понятия система программирования в хронологии развития вычислительных систем.

Начало 50-х годов XX века. Первые системы автоматизации программирования. Система программирования или система автоматизации программирования включала в себя ассемблер (или автокод) и загрузчик. Несколько позднее появились библиотеки стандартных программ и макрогенераторы. Основная функция первых систем программирования — предоставление программисту системы мнемонического обозначения компьютерных команд и данных, используемых в программах, а также предоставление возможности создавать и использовать библиотеки программ.

Середина 50-х — начало 60-х годов XX века. Появление и распространение языков программирования высокого уровня (Фортран, Алгол-60, Кобол и др.). Формирование концепций модульного программирования. Система программирования: макроассемблеры, трансляторы языков высокого уровня, редакторы внешних связей, загрузчики.

Середина 60-х — начало 90-х годов XX века. Развитие интерактивных и персональных систем, появление и развитие языков объектно-ориентированного программирования. Система программирования: трансляторы языков программирования, редакторы внешних связей, загрузчики, средства поддержания библиотек программ, интерактивные и пакетные средства отладки программ, системы контроля версий, средства поддержки проектов.

90-е годы XX века — настоящее время. Появление промышленных средств автоматизации проектирования программного обеспечения, CASE-средств (Computer-Aided Software/System Engineering), унифицированного языка моделирования UML. Системы программирования: интегрированные системы, предоставляющие комплексные решения в автоматизации проектирования, кодирования, тестирования, отладки и сопровождения программного обеспечения.

Мы видим, что интерпретация термина **система программирование** претерпела изменение от самого примитивного: «*система программирования — это транслятор языка программирования и средства редактирования связей*», — до современного: «*система программирования — это комплекс программ, обеспечивающий технологию автоматизации проектирования, кодирования, тестирования, отладки и сопровождения программного обеспечения*». Функции конкретной системы программирования определяются составом программных компонентов, которые могут использоваться для поддержания этапов жизненного цикла программы, и степенью интеграции этих компонентов. Таким образом, системой программирования будет являться как система, включающая только транслятор языка Си, ассемблер, редактор связей и интерактивный отладчик, так и, например, система **Rational Rose —** набор объектно-ориентированных CASE-средств, предназначенных для автоматизации процессов анализа, моделирования и проектирования с использованием UML, а также для автоматической генерации кодов программ на различных языках (C++, Java и пр.), разработки проектной документации и реверсного инжиниринга программ. На сегодняшний день выбор конкретной системы программирования во многом зависит как от масштабности и сложности решаемой задачи автоматизации, так и от квалификации программистов.

Уровень системы программирования основывается на доступе к виртуальным и физическим ресурсам, предоставляемым операционной системой (или уровнями управления физическими и виртуальными ресурсами), и предоставляет программистам инструментальные средства разработки программных систем, каждая из которых предназначена для решения своего круга задач.

1.1.6 Прикладные системы

Итак, мы переходим к вершине структурной организации вычислительных систем — к уровню **прикладного программного обеспечения**. ***Прикладная система —*** это программная система, ориентированная на решение или автоматизацию решения задач из конкретной предметной области. Прикладная система является прагматической основой всей вычислительной системы, так как, в конечном счете, именно для решения конкретных прикладных задач создавались все те уровни вычислительной системы, которые мы рассмотрели к настоящему времени.

В истории развития прикладных систем можно выделить четыре этапа. Первый — прикладные системы компьютеров первого поколения. Основной характеристикой данных систем являлось то, что для автоматизации решения каждой конкретной задачи создавалась уникальная программная система, которая не предполагала возможность модификации функциональности, переноса с одной вычислительной системы на другую (Рис. 13). Пользовательского интерфейса не было, как такового. Подавляющее большинство решаемых прикладных задач было связано с моделированием физических процессов, и, в свою очередь, результаты моделирования представлялись в виде последовательностей чисел и числовых таблиц. Уровень инструментальных средств программирования, доступных для решения прикладных задач, накладывал достаточно жесткие требования к квалификации специалистов, занимающихся автоматизацией решения прикладных задач. Кроме знания предметной области, алгоритмов и методов решения соответствующих прикладных задач программист должен был владеть средствами программирования компьютеров первого поколения — уметь использовать для этих целей систему команд или ассемблер компьютера.

Рис. 13. Первый этап развития прикладных систем.

Второй этап — развитие систем программирования и появление средств создания и использования библиотек программ (Рис. 14).

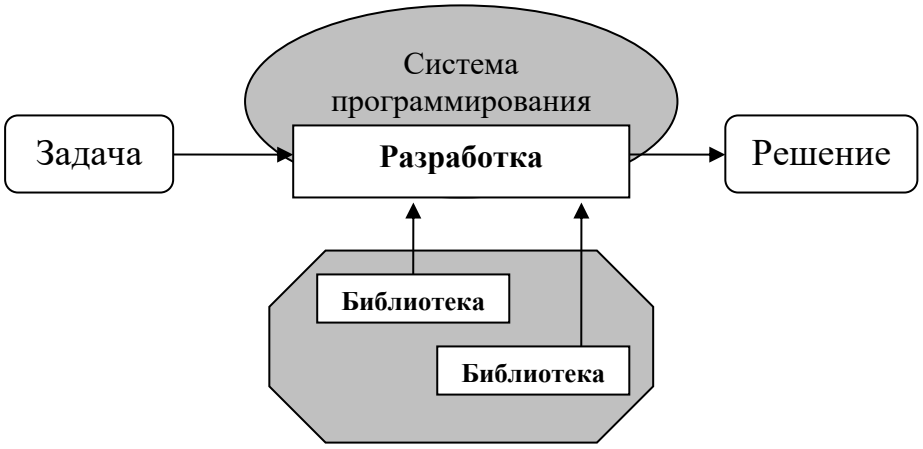


Рис. 14. Второй этап развития прикладных систем.

Библиотеки прикладных программ позволили аккумулировать и многократно использовать практический опыт численного решения типовых задач из конкретных предметных областей. Составляющие библиотеку подпрограммы служили "строительными блоками", которые в интеграции с системами программирования использовались для разработки прикладных систем. Библиотеки прикладных программ стали одними из первых программных систем, которые могли относиться к категории программных продуктов — документированных, прошедших детальное тестирование, распространенное в пользовательской среде. Библиотеки прикладных программ, наверное, были одними из первых коммерческих программных продуктов, т.е. они являлись интеллектуальным товаром, который можно было продать и купить. Примером может служить библиотека программ численного интегрирования, включающая в свой состав подпрограммы, реализующие всевозможные методы численного нахождения значений определенных интегралов. Библиотеки прикладных программ существенно упростили процесс разработки прикладных систем, однако требования к квалификации прикладного программиста оставались достаточно высокими. Прикладные системы этого этапа создавались с использованием стандартных систем программирования и в большей части были уникальны: создавались для решения конкретной задачи в конкретных условиях.

Третий этап характеризуется появлением **пакетов прикладных программ (ППП)**, которые включали в себя программные продукты (Рис. 15), предназначенные для решения широкого комплекса задач из конкретной прикладной области и обладающие следующими свойствами:

– программные продукты имели развитые, стандартизованные пользовательские интерфейсы, не требующие высокой программисткой квалификации от прикладного пользователя и значительных затрат на их освоение;

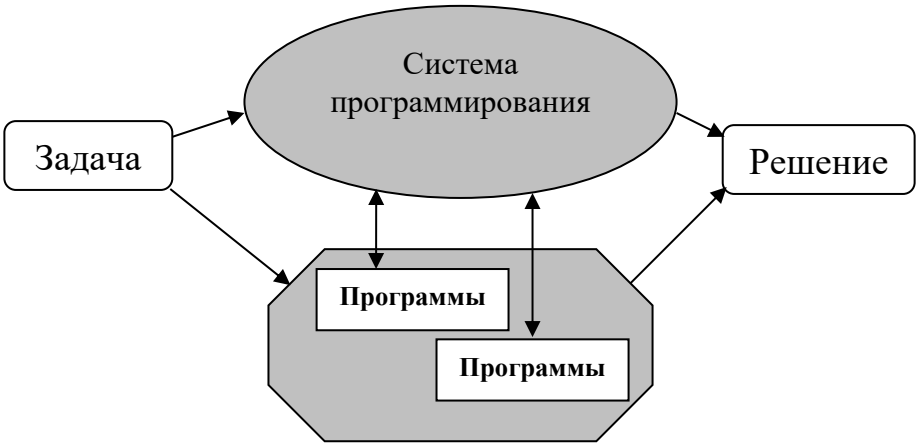


Рис. 15. Третий этап развития прикладных систем.

– функциональные возможности прикладных программ, входящих в состав ППП и их пользовательские интерфейсы позволяли решать разнообразные задачи данной прикладной области;
– возможно совместное использование программных продуктов, входящих в состав ППП при решении конкретных задач.

Примерами наиболее распространенных пакетов прикладных программ могут служить Microsoft Office (Рис. 16), предназначенный для автоматизации офисной деятельности или пакет MathCAD (Рис. 17), предназначенный для решения задач, связанных с математическими и техническими расчетами.

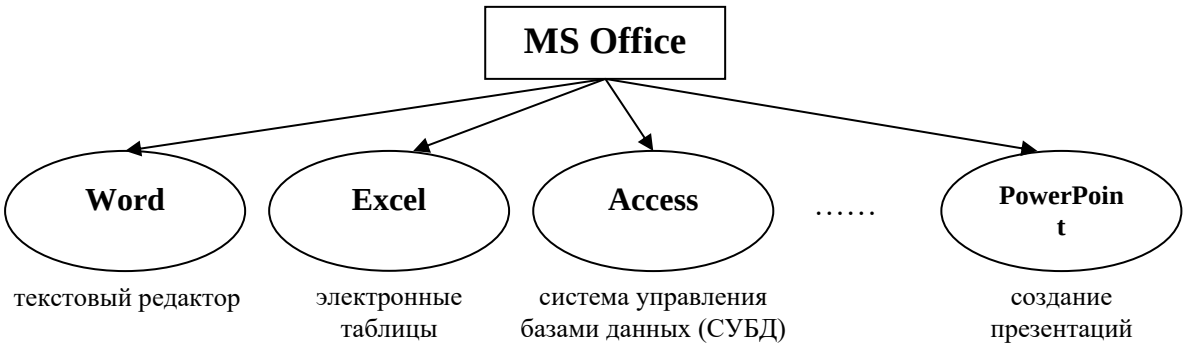


Рис. 16. Пакет программ Microsoft Office.

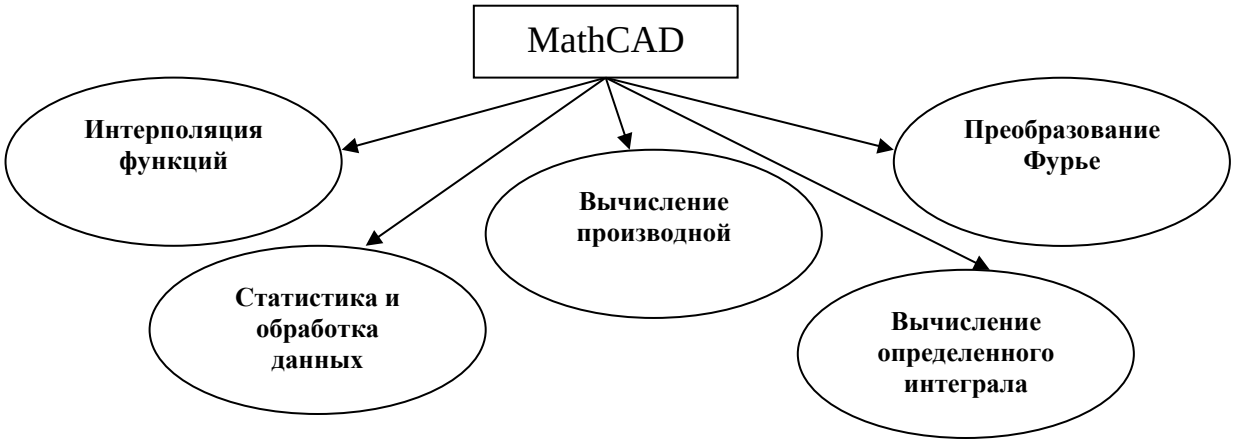


Рис. 17. Пакет MathCAD.

Современный этап — это этап комплексных, адаптируемых к конкретным условиям программных систем автоматизации прикладных процессов, построенных на основе развития концепций пакетов прикладных, интегрированных с современными системами программирования и использующих передовые технологии проектирования и разработки программного обеспечения. Особое развитие получили системы автоматизации бизнес-процессов.

Рассмотрим основные тенденции в развитии современных прикладных систем.

1. **Стандартизация моделей автоматизируемых бизнес-процессов** и построение в соответствии с данными моделями прикладных систем управления. В результате детального анализа и структуризации процессов, происходящих на различных уровнях управления предприятиями, взаимодействия предприятий друг с другом или взаимодействия предприятия с потребителями были стандартизованы разнообразные модели бизнес-процессов и, в свою очередь, появились прикладные системы, ориентированные на их автоматизацию. Примером могут служить следующие разновидности систем:
    a. **B2B**-система (business to business), обеспечивающая поддержку модели межкорпоративной торговли продукцией с использованием Internet (примером может служить электронные биржи);
    b. **B2C**-система (business to customer), обеспечивающая поддержку в Internet модели торговых отношений между предприятием и частным лицом — потребителем (примером может служить Интернет-магазин);
    c. **ERP** (Enterprise Resource Planning) — планирование ресурсов в масштабе предприятия, автоматизированная система управлением предприятием;
    d. **CRM** (Customer Relationship Management) — система управления взаимоотношениями с клиентами.
2. **Открытость системы**: потребителю системы открыты прикладные интерфейсы, обеспечивающие основную функциональность системы, а также стандарты организации внутренних данных. Прикладные интерфейсы (API — Application Programming Interface) совместно со стандартными средствами систем программирования, системы шаблонов и специализированные средства настройки прикладной системы позволяют адаптировать и развивать функциональные возможности прикладных систем к особенностям конкретного потребителя системы. Примером может служить система BAAN, предназначенная для комплексного решения задач автоматизации бизнес-процессов предприятия (Рис. 18). Система включает в себя модули, обеспечивающие мониторинг текущей деятельности предприятия, финансовый учет и отчетность, планирование производства, поддержку управления проектами, финансовыми средствами, инвестициями, закупкой и сбытом продукции, и т.п. Кроме того, система позволяет пользователю дополнять существующую функциональность собственными разработками: для этого предназначена подсистема «Инструментарий», в которой предоставляются средства разработки новых приложений. Стандартизация организации внутренних данных прикладных систем и их открытость создают возможности для существенного упрощения интеграции данных систем с другими прикладными системами и программами. Примером может служить использование XML (Extensible Markup Language — расширяемый язык разметки) в качестве открытого стандарта для описания бизнес-объектов и протоколов обмена данными в B2B приложениях.
3. Использование современных технологий и моделей организации системы: Internet/Intranet-технологии, средства и методы объектно-ориентированного программирования (ООП), модель клиент/сервер, технологии организации хранилищ

данных и аналитической обработки данных с целью выявления закономерностей и прогнозирования решений, и др.

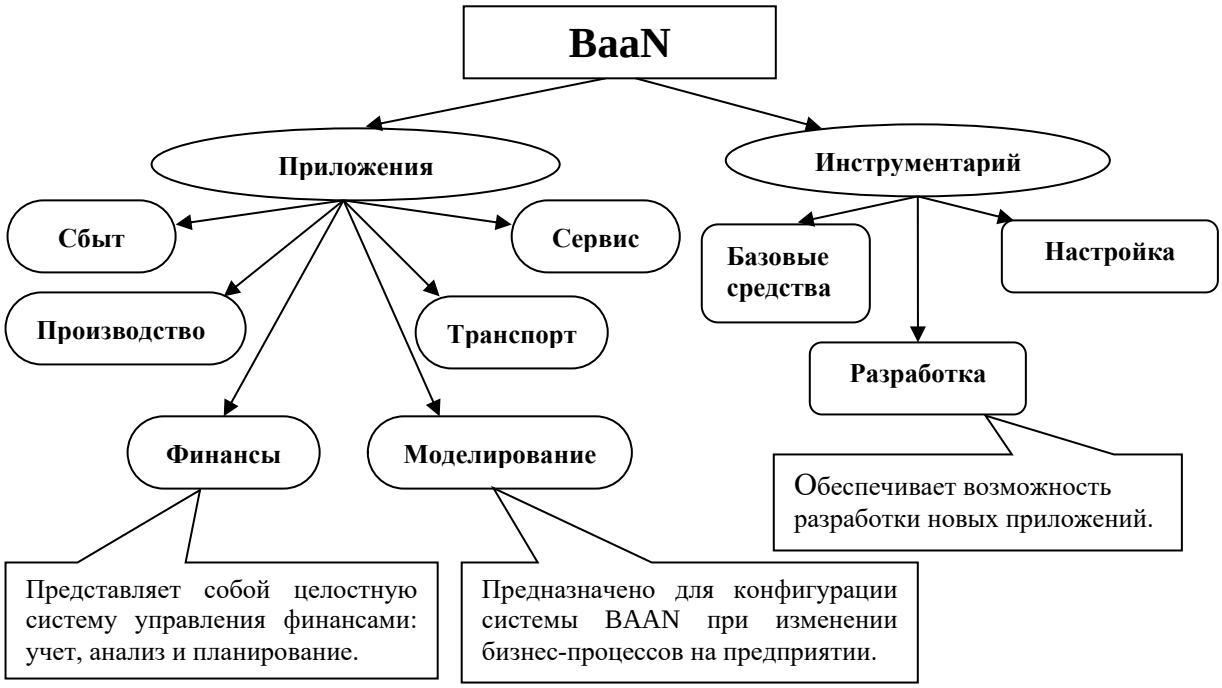


Рис. 18. Система BaaN.

Современная прикладная система предполагает глубокую интеграцию всех компонентов вычислительной системы: аппаратной части, операционной системы, системы программирования. В итоге, возможно разделение пользователей прикладной системы на следующие категории:

- **оператор** или **прикладной пользователь**, оперируя средствами пользовательского интерфейса и функциональными возможностями системы, решает конкретные прикладные задачи. Примером может служить работа инженера по проектированию оборудования с использованием системы AutoCAD или работа менеджера крупной компании, использующей аналитические средства системы управления бизнесом на основе решений BAAN;
- **системный программист** — пользователь компонентов прикладной системы, обеспечивающий возможности интеграции данной системы в конкретной вычислительной системе, возможности настройки в соответствии с конкретными особенностями эксплуатации системы на конкретном предприятии, доработку функциональных возможностей системы, удовлетворяющих потребностям и особенности эксплуатации. Например, применение пакета Microsoft Office с точки зрения системного программиста может варьироваться от автоматизации часто повторяющейся последовательности действий путем написания так называемых «макросов» до создания новых интерактивных приложений, функционирующих в среде MS Office. Основу технологии автоматизации на базе MS Office составляет предоставление офисных приложений в виде унифицированной иерархической объектной модели и использование единого внутреннего механизма программирования приложений на основе Visual Basic for Applications (VBA);
- **системный администратор** обеспечивает выполнение текущих работ по поддержке функционирования программной системы в конкретных условиях: в их состав могут входить регистрация пользователей и распределение полномочий и прав между ними, контроль за обеспечение сохранности и целостности данных, фиксация проблем,

возникающих в процессе эксплуатации, и обоснованное выполнение обновлений системы, поступающих от разработчика.

Каждой категории пользователей прикладной системы предоставлены свои, специализированные средства работы, которые предназначены для решения конкретных задач данного пользователя.

1.1.7 Выводы

Мы рассмотрели основные уровни структурной организации вычислительной системы. Следует отметить, что рассмотренная нами модель организации вычислительной системы не единственная: существуют и другие подходы в определении структуры ВС, но в большинстве случаев отличия не являются принципиальными. Выбранная нами модель служит основой для дальнейшего изложения материала.

Вернемся к вопросу, который в той или иной степени затрагивался при рассмотрении каждого из уровней ВС. Как представляется вычислительная система пользователя ВС на каждом из уровней? Что видит или что доступно пользователю ВС, который находится на одном из уровней структурной организации вычислительной системы? Рассмотрим еще раз уровни структурной организации ВС с позиций обозначенных вопросов (Рис. 19).

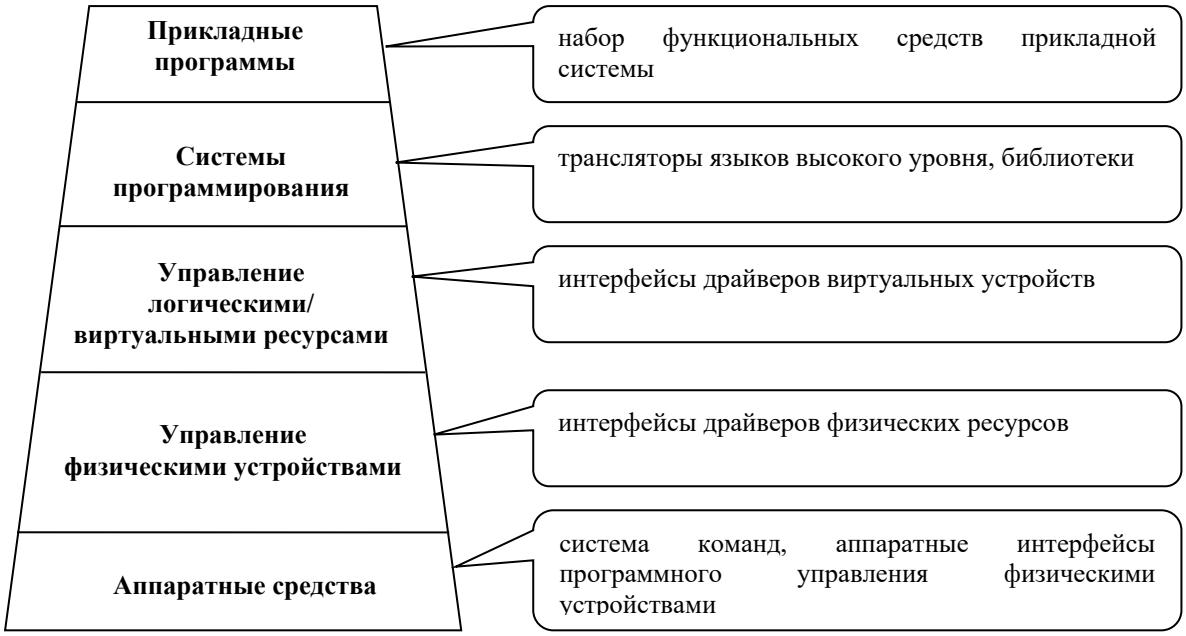


Рис. 19. Структура организации вычислительной системы.

Аппаратный уровень. Пользователь вычислительной системы — программист. Доступные средства программирования: система команд компьютера, аппаратные интерфейсы программного управления внешними устройствами. Таким образом, пользователь ВС, находясь на уровне аппаратуры, работает с конкретным компьютером.

Уровень управления физическими ресурсами. На данном уровне пользователем системы также является программист. Средства программирования, которые предоставляются пользователю на данном уровне, претерпели изменения, т.к. кроме возможности работы с системой команд компьютера, с аппаратными интерфейсами программного управления внешними устройствами пользователю предоставляются интерфейсы драйверов физических устройств (ресурсов) компьютера. С позиций программиста, он работает с компьютером, имеющим расширенные, по сравнению с предыдущим уровнем, возможности. Кроме стандартных аппаратных средств

программирования компьютера (система команд, аппаратные интерфейсы взаимодействия с физическими внешними устройствами) появились интерфейсы драйверов физических устройств (ресурсов) компьютера.

Уровень управления логическими или виртуальными ресурсами. На данном уровне структурной организации вычислительной системы спектр средств программирования расширяется за счет интерфейсов драйверов виртуальных/логических устройств (или ресурсов). В общем случае, для программиста, работающего с системой на данном уровне, средства программирования компьютера представляются:

- системой команд компьютера;
- аппаратными интерфейсами программного управления физическими устройствами;
- интерфейсами драйверов физических устройств;
- интерфейсами драйверов виртуальных устройств.

Операционная система может ограничить доступ пользователей к аппаратным средствам управления внешними устройствами, к драйверам физических устройств, к некоторым драйверам виртуальных устройств. Однако, "условный" пользователь уровня управления виртуальными устройствами вычислительной системы работает с компьютером, имеющим расширенные возможности. При этом пользователь может не знать о том, какие устройства, используемые в его программе, являются физическими, реально существующими, а какие — виртуальными. А даже если он и знает, что какое-то устройство является, к примеру, физическим, то, скорее всего, он не имеет никакого представления о деталях организации управления этого устройства на уровне аппаратных интерфейсов.

Уровень систем программирования. Для иллюстрации проблемы упростим структуру системы программирования, рассмотрим практически вырожденный случай. Пусть система программирования, с которой работает пользователь ВС, состоит только из транслятора языка высокого уровня и стандартной библиотеки программ, — например, языка Си. В этом случае представление пользователя о компьютере, на котором он работает, может свестись к языковым конструкциям языка Си и возможностям, предоставляемым стандартной библиотекой языка Си. Происходит очередное "расширение" возможностей компьютера за счет конструкций языка Си и его стандартной библиотеки. Более того, пользователь может работать на данном "расширенном" компьютере, не подозревая о реальной архитектуре аппаратного уровня ВС, о физических и виртуальных устройствах, поддерживаемых операционной системой, о системе команд и внутренней организации данных реального компьютера.

Уровень прикладных систем. Тенденция "расширения" возможностей компьютера продолжается и на прикладном уровне. При этом для каждой категории пользователей прикладного уровня вычислительной системы существует свое расширение компьютера. Так, например, для оператора прикладной системы компьютер представляется набором функциональных средств прикладной системы, доступной через пользовательский интерфейс. Рассмотрим работу кассира в современном супермаркете, кассовый аппарат которого может являться специализированным персональным компьютером, работающим в составе системы автоматизации деятельности всего магазина. Для кассира работа с этим компьютером и, соответственно, возможности этого компьютера представляются в виде возможностей прикладной подсистемы, автоматизирующей его рабочее место. Заведомо кассир магазина может не иметь никаких представлений о внутренней организации специализированной вычислительной системы, на которой он работает (тип компьютера, тип операционной системы, состав драйверов ОС и т.п.).

Не будет преувеличением утверждение, что не менее 90% современных пользователей персональных компьютеров не имеют представления о системе команд компьютера, о структуре компьютерных данных, об аппаратных интерфейсах управления физическими устройствами — все это скрывают расширения компьютера, которые образуются за счет соответствующих уровней вычислительной системы. Мы будем

говорить, что каждый пользователь, работая в соответствующем расширении компьютера, работает в **виртуальной машине** или **виртуальном компьютере**. Реальный компьютер используется непосредственно исключительно на аппаратном уровне. Во всех остальных случаях пользователь работает с программным расширением возможностей реального компьютера — с виртуальным компьютером. Причем "виртуальность" этого компьютера (или этих компьютеров) возрастает от уровня управления физическими ресурсами ВС до уровня прикладных систем.

Вернемся к замечаниям, с которых начали данный раздел, касающихся неоднозначности определений многих компонентов вычислительных систем и, в частности, неоднозначности определения термина «операционная система».

В некоторых изданиях ошибочно ассоциируют понятие виртуального компьютера исключительно с операционной системой. Это не так. Только что мы показали, что "виртуальность компьютера", с которым работает пользователь вычислительной системы, начинается с уровня управления физическими устройствами и завершается на уровне прикладных систем.

Также не совсем правильным является утверждение, что операционная система предоставляет пользователю удобства работы с вычислительной системой или простоту ее программирования. На самом деле эти свойства в большей степени принадлежат прикладным системам или системам программирования. Одной из возможных причин подобной неоднозначности является то, что на ранних периодах развития вычислительной техники системы программирования рассматривались в качестве компонента операционных систем. Вычислительная система является продуктом глубокой интеграции ее компонентов, и, безусловно, на удобства работы с ВС и на простоту программирования оказывают влияние и аппаратура компьютера, и операционная система, но эти свойства в существенно большей степени характеризуют системы программирования и прикладные системы.

В настоящем разделе были рассмотрены следующие базовые определения, понятия.

Вычислительная система — совокупность аппаратных и программных средств, функционирующих в единой системе и предназначенных для решения задач определенного класса. Рассмотрена пятиуровневая модель организации вычислительной системы: аппаратный уровень, уровень управления физическими ресурсами ВС, уровень управления логическими/виртуальными ресурсами, уровень систем программирования и уровень прикладных систем. Круг задач, на решение которых ориентирована вычислительная система, определяется наполнением уровня прикладных систем, однако возможность реализации тех или иных прикладных систем определяется всеми остальными уровнями, составляющими структурную организацию ВС.

Физические ресурсы (***устройства***) — компоненты аппаратуры компьютера, используемые на программных уровнях ВС или оказывающие влияние на функционирование всей ВС. Совокупность физических ресурсов составляет аппаратный уровень вычислительной системы.

Драйвер физического устройства — программа, основанная на использовании команд управления конкретного физического устройства и предназначенная для организации работы с данным устройством. Драйвер физического устройства скрывает от пользователя детальные элементы управления конкретным физическим устройством и предоставляет пользователю упрощенный программный интерфейс работы с устройством.

Логические, или ***виртуальные***, ***ресурсы*** (***устройства***) ВС — устройство/ресурс, некоторые эксплутационные характеристики которого (возможно все) реализованы программным образом.

Драйвер логического/виртуального ресурса — это программа, обеспечивающая существование и использование соответствующего ресурса, для этих целей при его реализации возможно использование существующих драйверов физических и виртуальных устройств.

Ресурсы вычислительной системы — это совокупность всех физических и виртуальных ресурсов данной вычислительной системы.

Операционная система — это комплекс программ, обеспечивающий управление ресурсами вычислительной системы. В структурной организации вычислительной системы операционная система представляется уровнями управления физическими и виртуальными ресурсами.

Жизненный цикл программы в вычислительной системе — проектирование, кодирование (программная реализация или реализация), тестирование и отладка, ввод программной системы в эксплуатацию (внедрение) и сопровождение.

Система программирования — комплекс программ, обеспечивающий поддержание этапов жизненного цикла программы в вычислительной системе.

Прикладная система — программная система, ориентированная на решение или автоматизацию решения задач из конкретной предметной области.

1.2 Основы компьютерной архитектуры

Изучение принципов структурной организации и функционирования основных компонентов операционной системы невозможно без рассмотрения основ архитектуры компьютера. Настоящая глава посвящена рассмотрению концепций организации компьютера в контексте его функционирования в составе вычислительной системы (вычислительная система – это совокупность взаимосвязанных аппаратных и программных компонентов, функционирующих в единой системе и предназначенных для решения задач определенного класса). Многие функциональные возможности операционных систем (такие как организация асинхронной работы с внешними устройствами, защита памяти от несанкционированного доступа, организация виртуальной оперативной памяти) невозможно рассматривать вне поддержки этих функций в аппаратуре компьютера. На самом деле верно и обратное: многие возможности аппаратуры компьютера сложно представить вне их использования в рамках операционной системы. В процессе рассмотрения основ архитектуры мы будем использовать обобщенную модель организации и свойств основных компонентов, составляющих компьютер, достаточную для построения представления о существующих взаимосвязях аппаратных и программных компонентов вычислительной системы, а также для понимания принципов построения операционных систем.

1.2.1 Структура, основные компоненты

Середина 40-х годов 20-го века может вправе считаться сроком зарождения современной вычислительной техники. С этой датой связано опубликование американским математиком венгерского происхождения Джоном фон Нейманом (John Von Neumann) технического отчета по результатам проектирования компьютера EDVAC (Electronic Discrete Variable Computer — Электронный Компьютер Дискретных Переменных) под названием «Предварительный доклад о компьютере EDVAC» (A First Draft Report on the EDVAC). В данном отчете декларировались основные концепции организации компьютеров, которые были реализованы в EDVAC. Основными разработчиками этого компьютера были Джон Мочли (John Mauchly) и Джон Преспер Эккерт (John Presper Eckert). Следует отметить, что к тому времени Мочли и Эккерт имели успешный опыт разработки компьютера ENIAC (Electronic Numerical Integrator And Computer). Скандальность данной ситуации состояла в том, что внутрикорпоративный отчет, основанный на предложениях Моучли и Эккерта (возможно и совместно с фон Нейманом), был подготовлен и опубликован за авторством только Джона фон Неймана. Распространение данного отчета в научной среде породило появление основополагающих принципов построения вычислительных машин – так называемых "принципов фон

Неймана", которые как минимум должны были именоваться принципами Мочли, Эккерта, фон Неймана. Мы не вправе и не в силах изменить ход истории и сложившуюся терминологию, поэтому в дальнейшем также будем использовать термин "принципы построения компьютера фон Неймана". Итак, в чем же состояли принципы организации машины фон Неймана?

1. **Принцип двоичного кодирования информации**: вся информация, которая поступает и обрабатывается в компьютере, кодируется в двоичной системе счисления.

2. **Принцип программного управления**. Программа состоит из команд, в которых закодированы операция и операнды, над которыми должна выполниться данная операция. Выполнение компьютером программы — это автоматическое выполнение определенной последовательности команд, составляющих программу. В компьютере имеется устройство, обеспечивающее выполнение команд, — **процессор**. Последовательность выполняемых процессором команд определяется последовательностью команд и данных, составляющих программу. То есть, по сути, второй принцип – это принцип последовательной обработки.

3. **Принцип хранимой программы**. Для хранения команд и данных программы используется единое устройство памяти, которое представляется в виде вектора слов. Все слова имеют последовательную адресацию. Команды и данные представляются единым образом. Интерпретация информации памяти и, соответственно, ее идентификация как команды или как данных происходит неявно при выполнении очередной команды. К примеру, содержимое слова, адрес которого используется в команде перехода в качестве операнда, интерпретируется как команда. Если то же слово используется в качестве операнда команды сложения, то его содержимое интерпретируется как данные. То есть одна и та же область памяти в зависимости от команд в одном случае будет интерпретироваться как команда, в другом случае – как данные. Этот принцип фон Неймана замечателен тем, что он определяет возможность программной генерации команд с последующим их выполнением, то есть возможность компиляции программы, когда одна программа порождает другую программу, которая будет выполняться.

Рассмотрим упрощенную структуру компьютера фон Неймана (Рис. 20):

– **Оперативное запоминающее устройство** (ОЗУ), или **основная память**, — устройство хранения данных, в котором находится исполняемая в данный момент программа. То есть оперативная память – это свойство всё-таки системное, а не технологическое (т.е. на основе чего сделано это устройство – на той же элементной базе можно сделать устройство для хранения информации, но оно будет представляться системе не как оперативная память, а как внешнее устройство; тогда как программа будет исполняться из оперативной памяти).

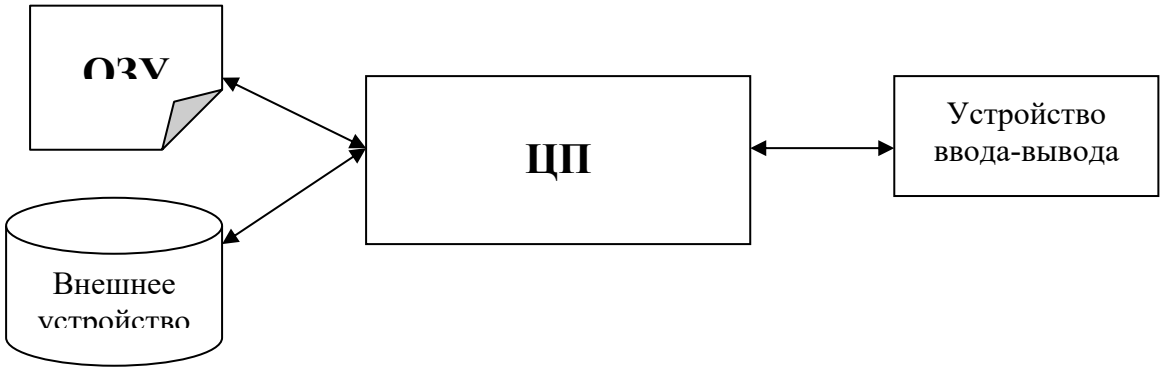


Рис. 20. Структура компьютера фон Неймана.

– **Внешние устройства —** программно управляемые устройства, входящие в состав компьютера, т.е. устройства, с которыми выполняемая программа может осуществлять обмен данными.

– **Процессор**, или **центральный процессор** (**ЦП**), — основной компонент компьютера, обеспечивающий выполнение программ, процессор координирует работу внешних устройств и оперативной памяти. Процессор состоит из **арифметико-логического устройства** (**АЛУ**) и **устройства управления** (**УУ**). Устройство управления обеспечивает последовательную выборку команд, составляющих программу, из памяти, выделение и анализ кода операции, получение значений операндов. В зависимости от кода операции команда выполняется либо в устройстве управления (обычно это могут быть команды передачи управления), либо код операции и операнды передаются для выполнения в АЛУ. После чего выбирается из памяти следующая команда программы и т.д. В системе команд процессора предусмотрены средства для взаимодействия с внешними устройствами.

Современные компьютеры по многим показателям не соответствуют принципам фон Неймана. Во-первых, принцип двоичного кодирования информации нарушается, так как в мире есть, по крайней мере, один пример использования троичной системы счисления. Это машина Сетунь, её автор – Николай Петрович Брусенцов, который работает у нас на факультете. Во-вторых, принцип программного управления в современных компьютерах тоже нарушен, так как он декларирует последовательную обработку и выбор информации, а подавляющее большинство компьютеров начинает обрабатывать команды «с забеганием» вперёд, то есть во время выполнения текущей команды последующие команды уже начинают выбираться (иногда эта работа может пойти насмарку, например, в случае ветвления по условию). В-третьих, принцип хранимой программы также на сегодняшний день нарушен – в подавляющем большинстве компьютеров ОЗУ хранит команды и данные «немного по-разному».

Ниже мы рассмотрим базовые структурные и функциональные особенности современных компьютеров, уделив особое внимание рассмотрению структурной организации компьютера как системы, объединяющей разнородные (по назначению и производительности) аппаратные компоненты и работающей под управлением операционной системы. Рассмотрим простейшую систему (Рис. 21). В оперативной памяти находится исполняемая программа (т.е. все команды и данные последовательно выбираются из ОП). Соответственно, ЦП обеспечивает выборку, анализ и исполнение команд. Скорость обработки информации в процессоре, скорость доступа к данным, размещенным в оперативной памяти, и скорость обмена данными с внешними устройствами могут отличаться друг от друга на порядки. И если в системе не будут предусмотрены средства, компенсирующие этот дисбаланс, то итоговая производительность будет определяться наименее производительным элементом, активно используемым в работе системы. Мы будем смотреть на архитектуру именно с позиций системы – где эти проблемы возникают и как эти проблемы в системе разрешаются. Итоговая производительность вычислительной системы во многом определяется решениями на уровнях аппаратуры и операционной системы, которые позволяют минимизировать последствия дисбаланса в производительности, как аппаратных, так и программных компонентов.

Рассмотрим теперь подробнее характеристики каждого из компонентов компьютера.

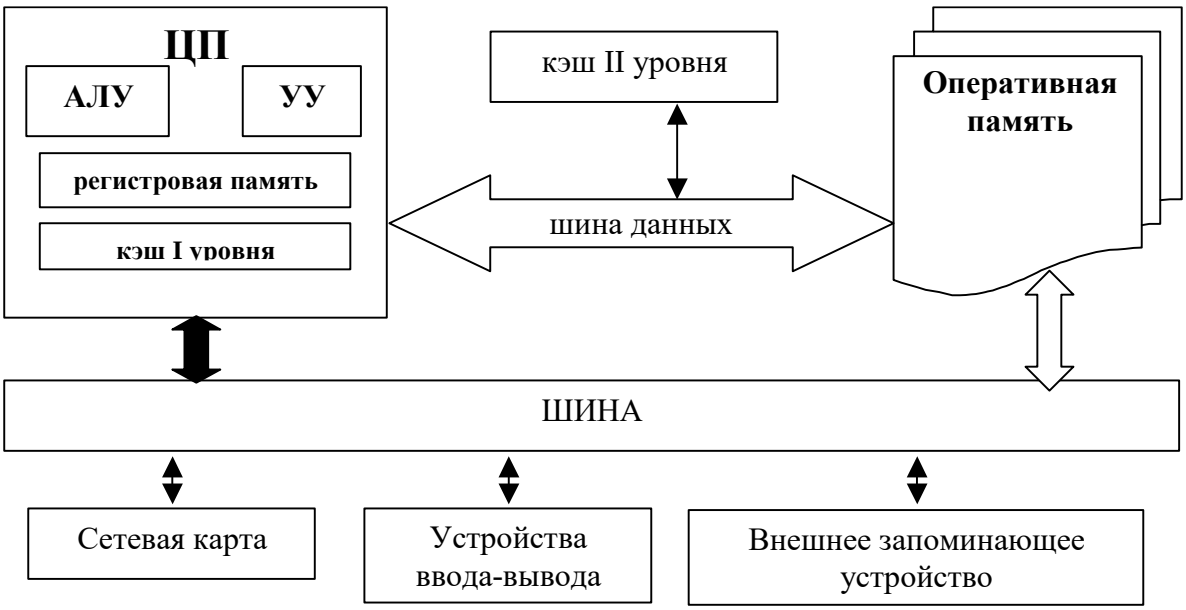


Рис. 21. Базовая архитектура современных компьютеров.

1.2.2 Оперативное запоминающее устройство

Оперативное запоминающее устройство (RAM — Random-Access Memory, память с произвольным доступом) – это устройство для хранения данных, в котором находится исполняемая программа. ОЗУ еще называют основной памятью, или оперативной памятью. Команды программы, исполняемые компьютером, поступают в процессор исключительно из ОЗУ. Таким образом, основным назначением оперативной памяти является хранение программы, которая выполняется в настоящее время компьютером. Оперативная память состоит из **ячеек памяти**. Ячейка памяти — это устройство, в котором размещается информация. Ячейка памяти может состоять из двух полей (Рис. 22). Первое поле — поле машинного слова, второе — поле служебной информации (или ТЕГ). Рассмотрим назначение каждого из этих полей.

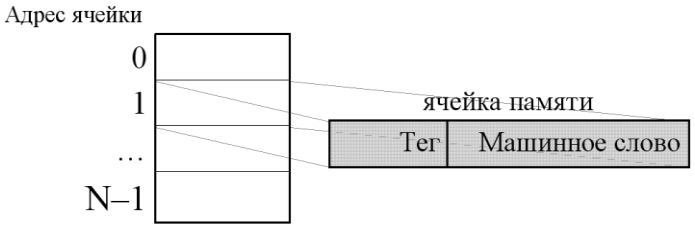


Рис. 22. Ячейка памяти.

Машинное слово — поле программно изменяемой информации. В машинном слове могут располагаться машинные команды (или части машинных команд) или данные, с которыми может оперировать программа. Машинное слово имеет фиксированный для данной ЭВМ размер. Обычно под размером машинного слова понимается количество двоичных разрядов, размещаемых в машинном слове. Когда используются термины «16-ти разрядный компьютер», или «32-х разрядный компьютер», или «64-х разрядный компьютер», это означает, что речь идет о компьютерах, оперативная память которых имеет *машинные слова* размером 16, 32 или 64 разряда соответственно.

Поле служебной информации — ***ТЕГ*** (tag — ярлык, бирка) — поле ячейки памяти, в котором схемами контроля процессора и ОЗУ автоматически размещается информация, необходимая для осуществления контроля за целостностью и корректностью

использования данных, размещаемых в соответствующем машинном слове. В принципе ТЭГ может отсутствовать.

Использование поля служебной информации (ТЭГа) может осуществляться в следующих целях.

– **Контроль за целостностью данных**. Простейшая модель – это одноразрядный ТЭГ, который использовался для контроля чётности. Содержимое поля используется для контрольного суммирования кода, размещенного в машинном слове. При каждой записи информации в машинное слово автоматически происходит контрольное суммирование (количества единичек в записываемом коде) и формирование содержимого поля служебной информации (бит чётности или нечётности). При чтении данных из машинного слова также автоматически происходит контрольное суммирование кода, находящегося в машинном слове, а затем полученный код контрольной суммы сравнивается с кодом, размещенным в поле служебной информации. Совпадение кодов говорит о том, что данные, записанные в машинном слове, не потеряны. Несовпадение говорит о том, что произошел сбой в ОЗУ и информация, находящаяся в машинном слове, потеряна, в этом случае в процессоре происходит прерывание (прерывания будут рассматриваться несколько позднее). На Рис. 23 изображена ячейка памяти с 16-ти разрядным машинным словом и одноразрядным полем ТЕГа. Контрольный разряд дополняет код машинного слова. Вариант А: содержимое машинного слова корректное, вариант Б — ошибка. Здесь следует отметить, что одноразрядное контрольное суммирование может "пропускать" потери пар единиц в коде машинного слова, т.е. эта схема контроля не может отлавливать чётное количество ошибок, — вариант В.

– **Контроль доступа к командам/данным**. Рассмотрим проблемы, возникающие в машинах фон Неймана. Первая — ситуация "потери" управления в программе, т.е. ситуация, при которой из-за ошибок в программе в качестве исполняемых команд начинают выбираться процессором и исполняться данные. Вторая проявляется тогда, когда программа из-за ошибки сама затирает свою кодовую часть: на место команд записываются данные. Отладка подобных ошибок достаточно трудоемка, т.к. возникновение ошибки в программе и ее проявление могут быть существенно разнесены по коду программы и по времени проявления. Контроль доступа к командам/данным обеспечивает защиту от возникновения подобных проблем. Суть этого решения заключается в следующем. Когда мы размещаем программу в памяти и генерируем код, то мы сразу же «раскрашиваем» всю информацию на два цвета – машинные слова, в которых находятся команды, и машинные слова, в которых находятся данные. При включении специального режима работы процессора запись машинных команд в оперативную память сопровождается установкой в ТЕГе специального кода, указывающего, что в данном машинном слове размешена команда. Также соответствующий признак устанавливается при записи данных. При выборке очередной команды из памяти автоматически проверяется содержимое соответствующих разрядов ТЕГа: если в машинном слове размещена команда, то будет продолжена ее обработка и выполнение. Если возникает попытка выполнения в качестве команды кода, записанного как данные, то происходит прерывание, т.е. фиксируется возникновение ошибки. Здесь мы видим первый случай отхода от одного из принципов организации компьютеров фон Неймана — введение контроля за семантикой информации, размещенной в машинном слове.

Это нужно для выполнения одного из основных требований, связанных с надёжностью программирования – система должна всеми силами минимизировать возможные ошибки в программе пользователя. Например, ошибка, связанная с передачей управления на область данных, может проявиться только позже, и такие ошибки ловить очень трудно.

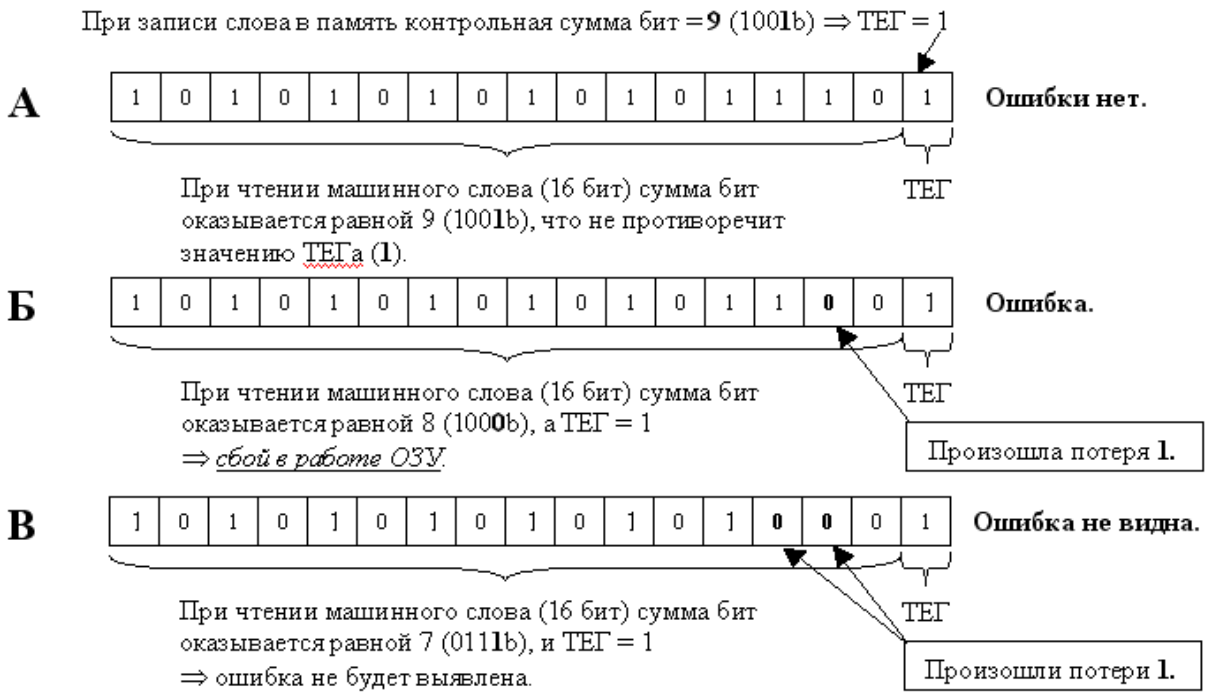


Рис. 23. Контроль четности.

– **Контроль доступа к машинным типам данных**. Развитием контроля за семантикой информации («раскраски информации в два цвета»), размещенной в оперативной памяти, является модель доступа к машинным типам данных. Как известно, каждый компьютер имеет так называемые машинные типы данных. Это означает, что существуют группы машинных команд, которые оперируют с данными одного типа (целые, вещественные с фиксированной точкой, вещественные с плавающей точкой, символьные, логические). Т.е. при выполнении команды используемые операнды интерпретируются согласно машинному типу данных в соответствии с типом команды. Согласно одному из принципов фон Неймана способ интерпретации информации в оперативной памяти зависит исключительно от характера использования этой информации. Т.е. любой код, записанный в машинное слово, может быть использован в качестве кода машинной команды, если устройство управления обратилось за очередной командой к этому машинному слову, и этот же код может быть проинтерпретирован как код любого машинного типа данных, если он используется в качестве операнда команды соответствующего типа. Контроль доступа к машинным типам данных осуществляется за счет фиксации в поле ТЕГа кода типа данных при их записи в машинное слово, а при использовании этих данных в качестве операндов команд осуществляется автоматическая проверка совпадения типа операнда и типа команды. Если они совпадают, то команда продолжает свое выполнение, если нет, то происходит прерывание. Как видим, контроль за использованием машинных типов данных является еще одним проявлением отхода архитектуры компьютеров от принципов фон Неймана.

Наличие или отсутствие поля служебной информации в ячейке памяти, характер его использования зависят от конкретного типа компьютеров. В каких-то компьютерах это поле ячейки памяти может отсутствовать, и в этом случае размер ячейки памяти совпадает с машинным словом. В каких-то — поле со служебной информацией ячейки памяти есть и используется для организации контроля за целостностью данных и корректностью их использования.

В ОЗУ все ячейки памяти имеют уникальные имена, имя — **адрес ячейки памяти**. Обычно адрес — это порядковый номер ячейки памяти (нумерация ячеек памяти возможна как подряд идущими номерами, так и номерами, кратными некоторому значению). Доступ к содержимому машинного слова осуществляется при непосредственном (например, считать содержимое слова с адресом **А**) или косвенном использовании адреса (например,

считать значение слова, адрес которого находится в машинном слове с адресом **B**). Одной из характеристик оперативной памяти является ее производительность, которая определяет скорость доступа процессора к данным, размещенным в ОЗУ. Обычно производительность ОЗУ определяется по значениям двух параметров.

- Первый — **время доступа (access time — t_{access})** — это время между запросом на чтение слова из оперативной памяти и получением содержимого этого слова.
- Второй параметр — **длительность цикла памяти (cycle time — t_{cycle})** — это минимальное время между началом текущего и последующего обращения к памяти.

Обычно, длительность цикла памяти существенно превосходит время доступа (t_{cycle}>t_{access}). Это связано с тем, что устройства памяти устроены таким образом, что после операции чтения многие из устройств памяти требуют регенерации (т.е. при чтении информация из ячейки разрушается – для того, чтобы сохранить информацию, надо её записать). Реальные соотношения между длительностью цикла и временем доступа зависят от конкретных технологий, применяемых для организации ОЗУ (в некоторых ОЗУ /t_{access}>2). Таким образм, время необходимое для чтения N псоледовательных ячеек памяти может быть не меньше **N**×t_{cycle}.

Вернемся к обозначенной в конце предыдущего пункта проблеме дисбаланса производительности аппаратных компонентов компьютера. Скорость обработки данных в процессоре в несколько раз превышает скорость доступа к информации, размещенной в оперативной памяти. Необходимо, чтобы итоговая скорость выполнения команды процессором как можно меньше зависела от скорости доступа к коду команды и к используемым в ней операндам из памяти. Это составляет проблему, которая системным образом решается на уровне архитектуры ЭВМ. В аппаратуре компьютера применяется целый ряд решений, призванных сгладить эту разницу. Одно из таких решений — **расслоение памяти**.

Расслоение ОЗУ — один из аппаратных путей решения проблемы дисбаланса в скорости доступа к данным, размещенным в оперативной памяти, и производительностью процессора. Суть расслоения состоит в следующем (Рис. 24, Рис. 25).

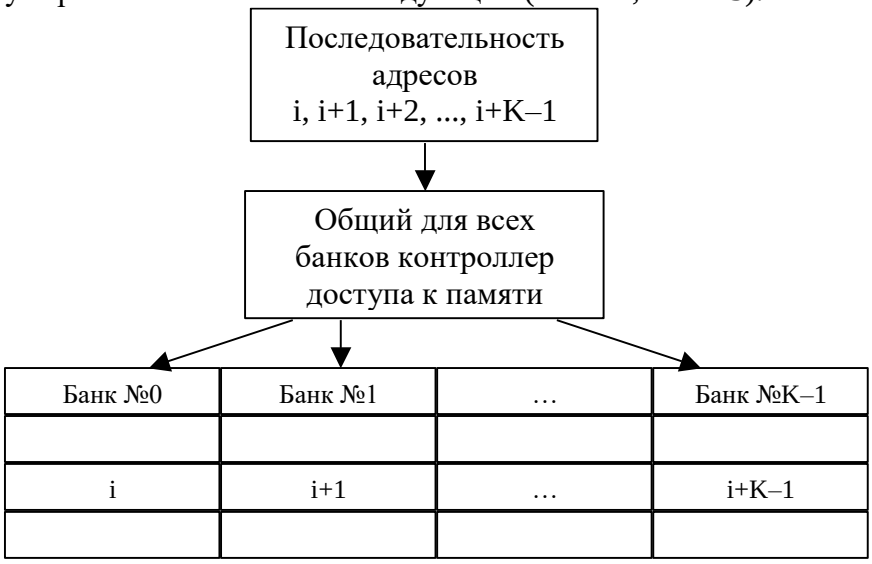


Рис. 24. Централизованный контроллер доступа к ОЗУ— один контроллер на все банки.

Всё пространство ОЗУ делится на **K** (**K** = 2^L). *независимых* подустройств, которые называются *банками памяти*. Адресация в системе организована таким образом, что младшие L разрядов адреса содержат номер банка (Рис. 25). Подряд идущие ячейки памяти распределены между банками таким образом, что у любой ячейки ее соседи размещаются в соседних банках (т.е. они находятся в разных банках). Что дает подобная организация памяти? Расслоение памяти позволяет во многом сократить задержки, возникающие из-за

несоответствия времени доступа и цикла памяти при выполнении последовательного доступа к ячейкам памяти, т.к. при расслоении ОЗУ задержки, связанные с циклом памяти, будут возникать только в тех случаях, когда подряд идущие обращения попадают в один и тот же банк памяти. Таким образом, расслоение памяти позволяет добиться увеличения скорости чтения из памяти при последовательном доступе. Используя организацию параллельной работы банков, в идеальном случае, можно повысить производительность работы ОЗУ в **К** раз. Для этих целей необходимо использовать более сложную архитектуру системы управления памятью.

Возможны две модели организации доступа к памяти с расслоением – *с централизованным контроллером доступа к памяти* (Рис. 24 – один контроллер управляет всеми банками) в этом случае нет проблемы цикла памяти, т.к. соседние ячейки памяти находятся в разных независимых банках и обращение к памяти за следующей ячейкой, происходит сразу после получения предыдущей через $\mathbf{t_{access}}$ - время доступа. В этом сдучае, если идет запрос единовременного чтения К подряд идущих ячеек памяти, то эта операция для данной модели пройдет за время $\mathbf{K \times t_{access}}$. Вторая модель - с *независимыми контроллерами для каждого из банков* (Рис. 25) – в этом случае мы можем организовывать параллельный доступ к памяти, т.е. одновременно мы можем считать порцию данных до **К** ячеек за время $\mathbf{t_{access}}$. В случае, если расслоения памяти нет, то на чтение **К** подряд идущих ячеек памяти потребуется $\mathbf{K \times t_{cycle}}$ (для случая группового запроса к памяти).

Вопрос для самостоятельного изучения. Каковы будут характеристики чтения в случае, если поступил единовременны запрос на чтение К+1 последовательной ячейки памяти для различных моделей организации: без расслоения; с расслоением и *с централизованным контроллером доступа к памяти; с расслоением и независимыми контроллерами для каждого из банков.*

.

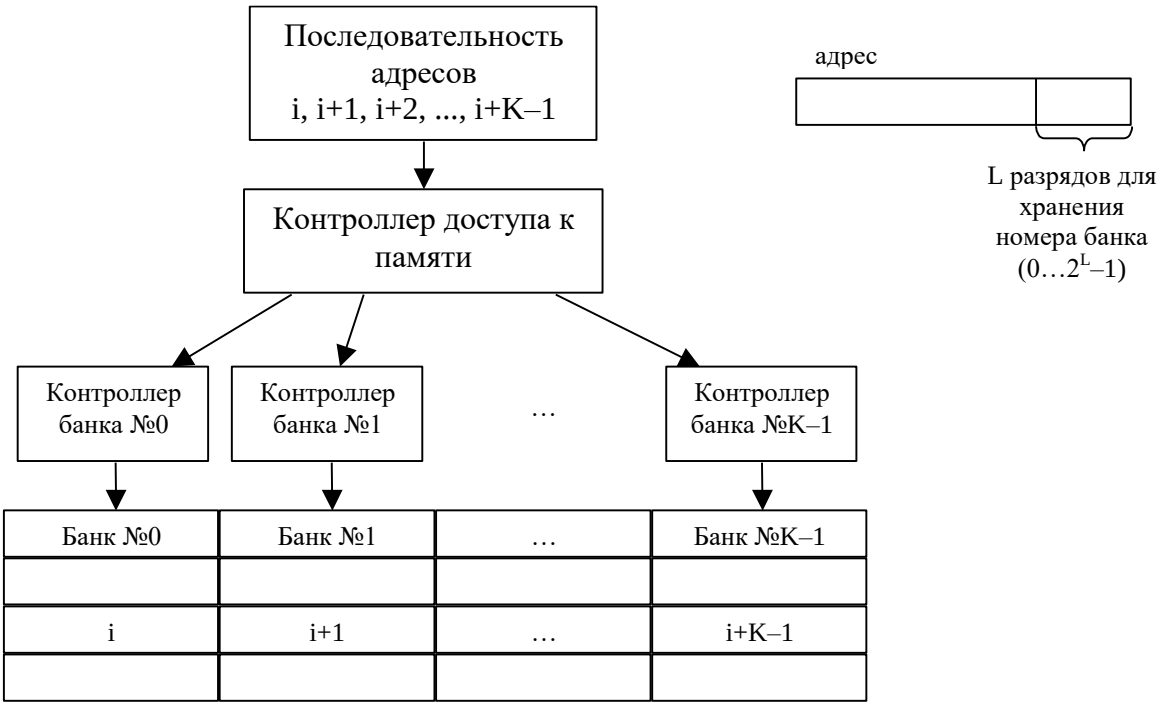


Рис. 25. ОЗУ с расслоением памяти — каждый банк обслуживает отдельный контроллер.

Другие свойства и характеристики оперативного запоминающего устройства мы будем рассматривать позднее по мере знакомства с основами архитектуры компьютеров и с организацией и функционированием компонентов операционных систем.

1.2.3 Центральный процессор

Процессор, или **центральный процессор** (**ЦП**), компьютера обеспечивает выполнение машинных команд, составляющих программу, размещенную в оперативной памяти. Термин «центральный процессор» соответствует ситуации сегодняшнего дня, когда современный компьютер – это многопроцессорная система (при этом компьютер будет называться однопроцессорным, поскольку в выполнении программы принимает участие только один процессор). Практически любой современный компьютер имеет в своем составе значительное количество специализированных управляющих компьютеров. Подобные компьютеры могут осуществлять управление контроллерами устройств, могут быть встроены в сами устройства, выполнять специализированные операции над данными программы.

Рассмотрим основные компоненты обобщенной структурной организации центрального процессора (Рис. 26).

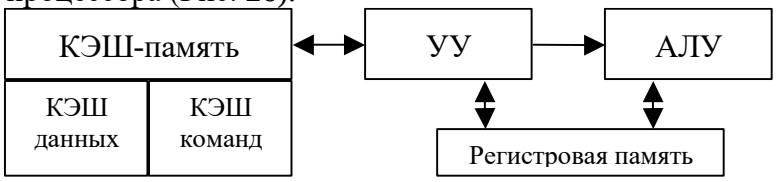


Рис. 26. Структура организации центрального процессора.

1.2.3.1 Регистровая память

Регистровая память, или ***регистровый файл*** (register file), — совокупность устройств памяти процессора (так называемых *регистров*), предназначенных для временного хранения управляющей информации, операндов и/или результатов выполняемых команд. Регистровая память обычно включает в себя **регистры общего назначения** (general-purpose register) и **специальные регистры** (special-purpose register).

Регистры общего назначения (РОН) состоят из доступных для программ пользователей регистров, предназначенных для хранения операндов, адресов операндов, результатов выполнения команд. РОН могут иметь машинную типизацию (например, регистры для хранения данных с плавающей точкой, с фиксированной точкой и т.д.). РОН могут быть скалярными (когда с одним регистром ассоциируется только одна единица памяти) и векторными (например, с одним регистром может ассоциироваться вектор регистров из 64 элементов; примером классических векторных компьютеров являются компьютеры фирмы CRAY). Для чего нужны РОН? Регистровая память работает в темпе процессора, т.е. скорость доступа к содержимому регистров сравнима со скоростью обработки информации процессором, поэтому одной из основных причин появления регистров общего назначения было сглаживание дисбаланса в производительности процессора и скорости доступа к оперативной памяти. РОН были первым аппаратным средством, которое предоставлялось пользователю для оптимизации своей программы. Наиболее часто используемые в программе операнды размещались на регистрах общего назначения, тем самым происходило сокращение количества реальных обращений в оперативную память, что, в итоге, повышало суммарную производительность компьютера. Состав регистров общего назначения существенно зависит от архитектуры конкретного компьютера.

Специальные регистры предназначены для координации информационного взаимодействия основных компонентов процессора. В их состав могут входить специальные регистры, обеспечивающие управление устройствами компьютера, регистры, содержимое которых используется для представления информации об актуальном состоянии выполняемой процессором программы и т.д. Так же, как и в случае регистров

общего назначения, состав специальных регистров определяется архитектурой конкретного процессора. К наиболее распространенным специальным регистрам относятся: ***счетчик команд*** (program counter), ***указатель стека*** (stack pointer), ***слово состояния процессора*** (processor status word). ***Счетчик команд*** — специальный регистр, в котором размещается адрес очередной выполняемой команды программы. Счетчик команд изменяется в устройстве управления согласно алгоритму, заложенному в программу. Более подробно использование счетчика команд проиллюстрируем несколько позднее при рассмотрении рабочего цикла процессора. ***Указатель стека*** — регистр, содержимое которого в каждый момент времени указывает на адрес слова в области памяти, являющегося вершиной стека. Обычно данный регистр присутствует в процессорах, система команд которых поддерживает работу со стеком (операции чтения и записи данных из/в стек с автоматической коррекцией значения указателя стека). ***Слово состояния процессора*** — регистр, содержимое которого определяет режимы работы процессора, значения кодов результата операций и т.п.

1.2.3.2 Устройство управления. Арифметико-логическое устройство

Устройство управления (control unit) — устройство, которое координирует выполнение команд программы процессором. **Арифметико-логическое устройство** (arithmetic/logic unit) обеспечивает выполнение команд, предусматривающих арифметическую или логическую обработку операндов. Эти устройства являются своего рода «мозгом» процессора, т.к. именно функционирование устройства управления и арифметико-логического устройства обеспечивают выполнение программы. Рассмотрим упрощенную (без анализа нештатных ситуаций) схему выполнения процессором программы (Рис. 27) в модельном компьютере.

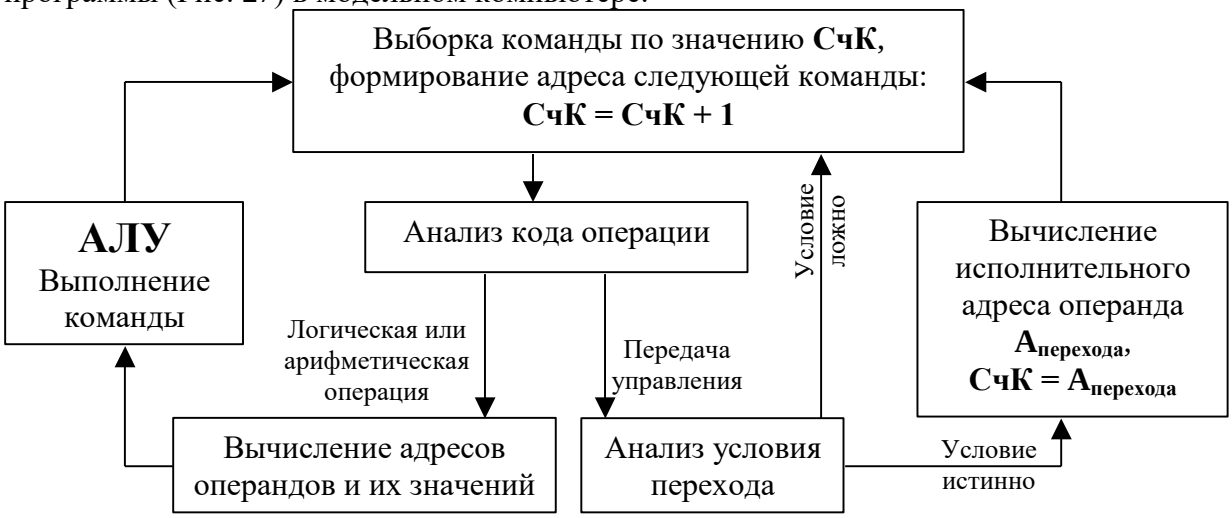


Рис. 27. Алгоритм выполнения процессором программы (рабочий цикл ЦП).

Пусть в начальный момент времени в счетчике команд **СчК** находится адрес первой команды программы. Для упрощения изложения будем считать, что система команд компьютера и система адресации оперативной памяти таковы, что любая команда размещается в одном машинном слове и адреса соседних машинных слов отличаются на единицу. Итак, рассмотрим последовательность действий в устройстве управления процессора при выполнении программы.

1. По содержимому счетчика команд **СчК** из ОП выбирается команда для выполнения. Формируется адрес следующей команды: **СчК = СчК + 1**.
2. Осуществляется анализ кода операции:

– Если это код арифметической или логической операции, то вычисляются исполнительные адреса операндов, выбираются значения операндов, команда передается для исполнения в арифметико-логическое устройство (передается код операции и значения операндов). В арифметико-логическом устройстве происходит выполнение команды, а также происходит формирование кода признака результата в регистре слова состояния процессора или в специальном регистре результата. Переход на п.1.

– Если это команда передачи управления, то происходит анализ условий перехода (анализируется содержимое кода признака результата предыдущей арифметико-логической команды с условиями перехода, соответствующими команде). Если условие перехода не выполняется, то переход на п.1. Иначе, вычисляется исполнительный адрес операнда $\mathbf{A_{перехода}}$, затем: $\mathbf{СчК = A_{перехода}}$ и осуществляется переход на п.1.

– Если команда загрузки данных из памяти в регистры общего назначения, то вычисляются исполнительные адреса операндов, выбираются значения операндов из памяти, значения записываются в соответствующие регистры. Переход на п.1.

Последовательность действий, происходящая в процессоре при выполнении программы, называется ***рабочим циклом процессора***. По ходу рассмотрения материала мы будем уточнять рабочий цикл нашего обобщенного модельного компьютера.

1.2.3.3 КЭШ-память

Ключевой проблемой функционирования компьютеров является проблема несоответствия производительности центрального процессора и скорости доступа к информации, размещенной в оперативной памяти. Мы рассмотрели аппаратные и программно-аппаратные средства, применение которых позволяет частично сократить этот дисбаланс. Однако, ни организация расслоения памяти, ни использование регистров общего назначения для размещения наиболее часто используемых операндов не предоставили кардинального решения проблемы. Решение, которое на сегодняшний день является наиболее эффективным, основывается на аппаратных средствах, позволяющих при выполнении программы автоматически минимизировать количество реальных обращений в оперативную память за операндами и командами программы за счет **кэширования** памяти — размещения части данных в более высокоскоростном запоминающем устройстве. Таким средством является **КЭШ-память** (cache memory) — высокоскоростное устройство хранения данных, используемое для буферизации работы процессора с оперативной памятью. В общем случае, кэш представляет собою аппаратную «емкость», в которой аккумулируются наиболее часто используемые данные из оперативной памяти. Скорость доступа к информации, размещённой в КЭШе, соизмерима со скоростью обработки информации в ЦП. Обмен данными при выполнении программы (чтение команд, чтение значений операндов, запись результатов) происходит не с ячейками оперативной памяти, а с содержимым КЭШа. При необходимости из КЭШа «выталкивается» часть данных в ОЗУ или загружаются из ОЗУ новые данные. Варьируя размеры КЭШа, можно существенно минимизировать частоту реальных обращений к оперативной памяти. Размещение и команд, и данных в одном КЭШе может приводить к тому, что команды и данные начинают вытеснять друг друга, увеличивая при этом обращения к оперативной памяти. Для исключения недетерминированной конкуренции в КЭШе между командами программы и обрабатываемыми данными современные компьютеры имеют два независимых КЭШа: **КЭШ данных** и **КЭШ команд**, каждый из которых работает со своим потоком информации — потоком команд и потоком операндов.

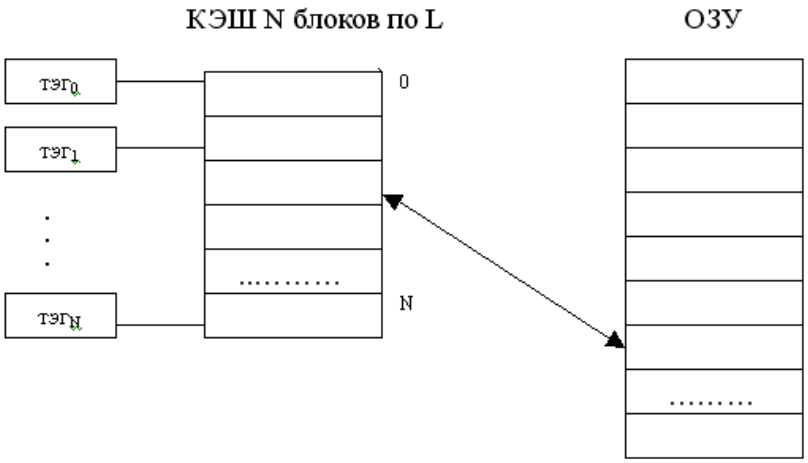


Рис. 28. Общая схема работы КЭШа.

Общая схема работы КЭШа следующая (Рис. 28).

1. Условно, вся память разделяется на блоки одинакового размера. Обмен данными между КЭШем и оперативной памятью осуществляется ***блоками*** фиксированного объёма. Здесь мы можем видеть возможное проявление преимущества использования памяти с расслоением, так как загрузка блока из оперативной памяти в КЭШ осуществляется с использованием параллелизма работы «расслоенной» оперативной памяти.

2. Каждый блок имеет спецификатор доступа – ***тэг***, в котором находится служебная информация, характеризующая данный блок. В тэге может содержаться информация о том, какой области ОП соответствует содержимое данного блока, занят или свободен блок, производились изменения в данном блоке или нет. Когда процессору нужно обратиться за командой или за данными в ОП, сначала происходит обращение к КЭШу. По содержимому адресного тэга можно однозначно адресовать содержимое блока. Анализ тэгов блоков КЭШа производится аппаратно. Таким образом, после вычисления исполнительного адреса операнда или команды устройство управления может определить, находится ли соответствующая информация в одном из блоков КЭШ-памяти или нет. Факт нахождения искомых данных в КЭШе называется **попаданием** (hit) – в этом случае данные берутся из КЭШа, и обращение в ОП не осуществляется. Если искомых данных нет в КЭШе, то фиксируется **промах** (cache miss).

3. При возникновении промаха происходит ***вытеснение*** – обновление содержимого КЭШа. Для этого выбирается блок-претендент на вытеснение, т.е. блок, содержимое которого будет заменено. Стратегия этого выбора зависит от конкретной организации процессора. Существуют КЭШи, вытеснение блоков которых осуществляется *случайным образом*, т.е. номер блока, который должен быть вытеснен, определяется с использованием встроенного генератора случайных чисел. Альтернативой случайного вытеснения является *вытеснение наименее «популярного» блока КЭШа*, т.е. блока, к содержимому которого происходило наименьшее число обращений (LRU — Least-Recently Used).

4. Отдельно следует обратить внимание на организацию вытеснения блока в КЭШе данных, т.к. содержимое блоков КЭШа может не соответствовать содержимому памяти (это возникает при обработке команд записи данных в память). В этом случае также возможно использование нескольких стратегий вытеснения. Первая — **сквозное кэширование** (write-through caching): при выполнении команды записи данных обновление происходит как в КЭШе, так и в оперативной памяти. Таким образом, при вытеснении блока из КЭШа происходит только загрузка содержимого нового блока. Данная стратегия оправдана, т.к. статистические исследования показывают, что частота чтения данных превосходит частоту их записи на порядок. Другой стратегией является **кэширование с**

обратной связью (write-back caching), суть которой заключается в использовании специального *тега модификации* (dirty bit). При выполнении команды записи по адресу, содержимое которого кэшируется в одном из блоков, происходит обновление соответствующей этому адресу информации только в блоке КЭШа, а также установка в блоке тега модификации. Соответственно, при вытеснении блока осуществляется контроль за содержимым тега. Если тег модификации установлен, то содержимое блока перед вытеснением «сбрасывается» в память. Тем самым минимизируется частота выполнения операции записи в память.

Использование КЭШ-памяти позволяет получить следующие *преимущества*. Во-первых, сокращается количество обращений к ОЗУ – обращений, как по выборке команд, так и по выборке операндов. Во-вторых, существенно увеличивается скорость доступа к памяти в случае использования ОЗУ с «расслоением», так как обмены блоков с памятью будут проходить практически параллельно (когда мы работаем с группой подряд идущих слов).

Естественно, при использовании КЭШ-памяти возникают и некоторые *проблемы*. Во-первых, усложнение логики процессора. Организация и использование КЭШ-памяти в процессоре развивает рабочий цикл процессора модельного компьютера, рассмотренный выше: при выборке очередных команд, получении операндов команд и записи результатов выполнения команд в ОЗУ добавляются схемы организации использования КЭШ-памяти. Во-вторых, если КЭШ один (т.е. потоки команд и данных приходятся на один КЭШ), то один из потоков может начать «довлеть» над другим, так как характеристики потоков команд и данных разные (поток команд обладает свойством локализации, поток данных этим свойством не обладает). Поэтому получили распространение архитектуры, в которых КЭШ разделяется на КЭШ команд и КЭШ данных. Это также позволяет повысить производительность системы.

Кэширование памяти в современных вычислительных системах применяется не только для оптимизации взаимодействия центрального процессора и оперативной памяти. В настоящем пункте мы рассмотрели модельный аппарат КЭШ как компонент процессора — это т.н. **КЭШ первого уровня**. Современные компьютеры могут включать в свой состав иерархию устройств, кэширующих более медленные устройства хранения данных. Рассмотрению этого вопроса будет посвящен отдельный раздел.

1.2.3.4 Аппарат прерываний

Если мы обратим внимание на представленный выше рабочий цикл процессора, то увидим, что такая схема не предусматривает возможности обработки ошибочных ситуаций, которые могут возникнуть в системе в ходе выполнения программы. Что будет с компьютером, если в программе, которую он выполняет, встретится команда с кодом операции, обработка которого не предусмотрена аппаратурой? Что будет, если выполняется корректная команда, но значения операндов приводят к невозможности выполнения соответствующей команде операции, например, деление на ноль? Что будет, если при программном обращении к внешнему устройству оно сломалось? В первых компьютерах происходила остановка работы всего компьютера и обработка ситуации, вызвавшей аварийную остановку (АВОСТ). Современные вычислительные системы не могут позволить себе остановку работы всей системы из-за возникновения тех или иных проблем в программе или в компонентах компьютера. Для решения проблем автоматизации обработки предопределённых событий, возникающих в вычислительной системе, в современных компьютерах предусмотрены соответствующие аппаратно-программные средства – т.н. аппарат прерываний.

Прерыванием называется событие в компьютере, при возникновении которого в системе предусмотрена предопределенная последовательность действий, включающая стандартную реакцию процессора на прерывание и этап программной обработки прерываний (функция ОС). Состав прерываний — множество разновидностей событий, на

возникновение которых предусмотрена стандартная реакция центрального процессора, — фиксирован и определяется конструктивно при разработке компьютера. **Аппарат прерываний** компьютера позволяет организовывать стандартную обработку прерываний, возникающих при функционировании вычислительной системы. Традиционно прерывания разделяются на два типа: **внутренние прерывания** и **внешние прерывания**.

Внутренние прерывания инициируются схемами контроля работы процессора. К примеру, внутреннее прерывание может возникнуть в процессоре при попытке выполнения команды деления, операнд-делитель которой равен нулю. Также внутреннее прерывание возникнет в ситуации, когда при обработке очередной команды адрес одного из операндов выходит за пределы адресного пространства оперативной памяти.

Внешние прерывания — события, возникающие в компьютере в результате взаимодействия центрального процессора с внешними устройствами. Примером внешнего прерывания может служить событие, связанное с вводом символа с клавиатуры персонального компьютера.

Обработка прерывания предполагает две стадии: **аппаратную**, которая включает предопределённую реакцию процессора на возникновение прерывания, и **программную**, которая предполагает выполнение специальной программы обработки прерывания, являющейся частью операционной системы.

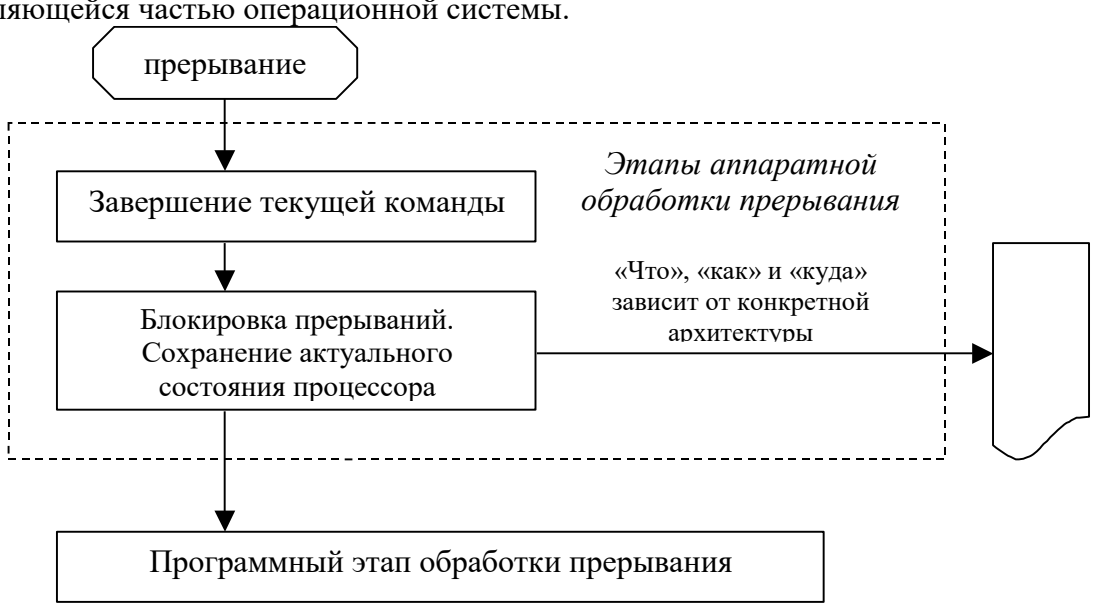


Рис. 29. Схема обработки прерывания.

Рассмотрим обобщенную модель последовательности действий, происходящих в ВС при возникновении прерывания (Рис. 29). Сначала рассмотрим **этап аппаратной обработки прерывания**.

1. Завершается выполнение текущей команды (за исключением случаев, когда прерывание возникает по причине некорректного выполнения команды).

2. Обработка прерывания предполагает остановку выполнения текущей программы, запуск специальной программы обработки прерывания, а затем, возможно, продолжение выполнения прерванной программы (с прерванного места). Для обеспечения этой возможности необходимо зафиксировать актуальное состояние компьютера в момент прихода прерывания (т.к. для обработки прерывания будет работать другая программа – ОС, а, следовательно, актуальное состояние системы изменится). Поэтому аппаратный этап обработки прерываний регламентирует перечень регистров, которые автоматически будут сохранены процессором. Это специальные регистры, содержимое которых описывает состояние процессора в точке прерывания выполнения программы (счетчик команд, регистр результатов, регистры, содержащие режимы работы процессора), а также

несколько регистров общего назначения, которые могут быть использованы программой обработки прерываний в начальный момент времени. Процедура аппаратного сохранения регистров в различных компьютерах может происходить по-разному. Простейшая модель следующая. Важно отметить, что буфер для сохранения актуального состояния в системе один, поэтому, если пришло прерывание и система что-то положила в буфер, нужно временно запретить запись информации относительно нового прерывания в этот буфер. Поэтому **включается режим блокировки прерываний**. При этом режиме в системе запрещается инициализация новых прерываний: возникающие в это время прерывания могут либо игнорироваться, либо откладываться (зависит от конкретной аппаратуры компьютера и типа прерывания).

3. Аппаратное копирование содержимого сохраняемых регистров («малое упрятывание»). Включенный режим блокировки прерывания гарантирует сохранность этих данных до момента завершения предварительной обработки прерывания и выключения блокировки прерываний.

4. Переход на программный этап обработки прерываний. Аппаратно передаём управление на некоторую фиксированную точку в ОЗУ, в которой предполагается наличие программы обработки прерываний операционной системы. Для перехода на программный этап обработки прерываний необходимо решить вопрос, как аппаратура передаст операционной системе информацию о том, прерывание какого типа произошло. Существует несколько моделей аппаратного решения этого вопроса.

– Первая модель — использование **специального регистра прерываний**, каждый разряд которого соответствует конкретному типу прерывания, т.е. если, к примеру, в разряде, соответствующем прерыванию от клавиатуры появляется единица, это означает, что произошло соответствующее прерывание. Для расширения числа обрабатываемых прерываний возможно использование иерархической модели регистров прерывания (Рис. 30). Она предполагает, что имеется **главный регистр прерывания** и **периферийные**. В главном регистре прерывания выделяются разряды, отвечающие не только за появление конкретных прерываний, но и разряды, отвечающие за появление прерываний в периферийных регистрах. В данной модели управление передается в операционную систему на адрес входа в программу.

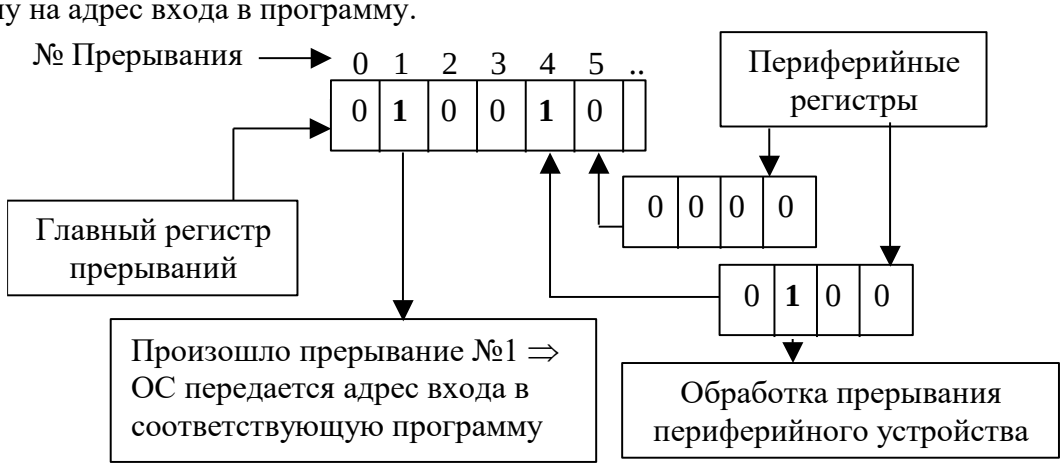


Рис. 30. Иерархическая модель регистров прерываний.

– Вторая модель — использование **вектора прерываний**. Предполагается, что по количеству возможных прерываний в ОЗУ выделена группа машинных слов — *вектор прерываний*. Каждое слово вектора прерываний содержит адрес программы, обрабатывающей данное прерывание (Рис. 31). При возникновении прерывания после сохранения регистров осуществляется передача прерывания по адресу, соответствующему номеру прерывания.

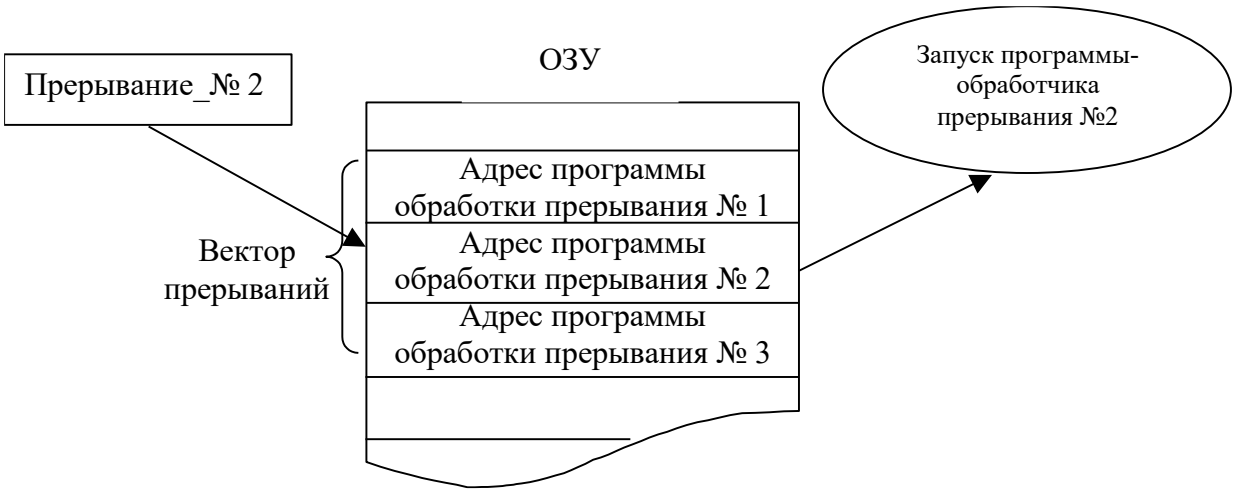


Рис. 31. Модель организации прерываний с использованием «вектора прерываний».

– Третья модель — использование **регистра слова состояния процессора** (Рис. 32). В этом случае в данном регистре резервируется часть разрядов – поле, в которое передаётся номер возникшего прерывания. В этой модели управление передается на фиксированный адрес входа в программу обработки прерываний.

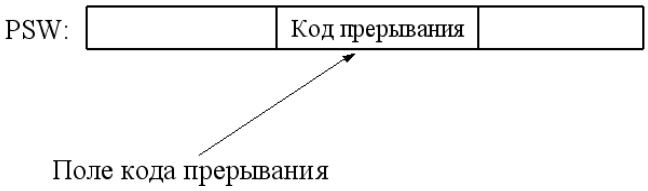


Рис. 32. Модель организации прерываний с использованием регистра слова состояния процессора.

Теперь рассмотрим **этап программной обработки прерывания**. Управление передано на адрес программы ОС, занимающейся обработкой прерывания. При входе в эту точку часть ресурсов ЦП, используемых программами, освобождена (в результате аппаратного упрятывания регистров). Поэтому будет запущена программа ОС, которая может использовать только освобожденные ресурсы ЦП (перечень доступных в этот момент регистров — характеристика аппаратуры). Выполняется следующая последовательность действий (Рис. 33).

Рис. 33. Программный этап обработки прерываний.

1. Анализ и предварительная обработка прерывания. Происходит идентификация типа прерывания, определяются причины.

– Если прерывание «короткое», т.е. обработка не требует дополнительных ресурсов ЦП и времени, то прерывание обрабатывается, выключается режим блокировки прерываний, восстанавливается состояние процессора, соответствующее точке прерывания исходной программы, и передается управление на прерванную точку. Примером подобного «короткого» прерывания может служить прерывание от таймера для коррекции времени в системе. Если прерывание требует использования всех ресурсов ЦП, то переходим к следующему шагу.

– Если прерывание является «фатальным» для программы, т.е. после этого прерывания продолжить выполнение программы невозможно (например, в программе произошло деление на ноль или обращение к несуществующему в ОЗУ адресу), то выключается режим блокировки прерываний, и управление передается в ту часть ОС, которая прекратит выполнение прерванной программы.

2. «Полное упрятывание». Если прерывание не короткое и не фатальное (например, обращение к области памяти, которая закрыта для обращения – например, чтение информации с внешнего носителя), то для обработки такого прерывания потребуются ресурсы. Поэтому осуществляется полное сохранение контекста (т.е. всех регистров ЦП, использовавшихся прерванной программой) в специальную программную таблицу. В данную таблицу копируется содержимое регистровой или КЭШ-памяти, содержащей сохраненные значения ресурсов ЦП, а также копируются все оставшиеся регистры ЦП, используемые программно, но не сохраненные аппаратно. После данного шага программе обработки прерываний становятся доступны все ресурсы ЦП, а прерванная программа получает статус ожидания завершения обработки прерывания. В общем случае, программ, ожидающих завершения обработки прерывания, может быть произвольное количество.

3. До данного момента времени все действия происходили в режиме блокировки прерываний. Почему? Потому что режим блокировки прерываний — единственная гарантия того, что не придет новое прерывание, и при его обработке не потеряются данные, необходимые для продолжения прерванной программы (регистры, режимы, таблицы ЦП). После полного сохранения регистров происходит снятие режима блокировки прерываний,

то есть включается стандартный режим работы процессора, при котором возможно появление прерываний.

4. Операционная система завершает обработку прерывания.

Мы рассмотрели модельную, упрощенную схему обработки прерывания: в реальных системах она может иметь отличия и быть существенно сложнее. Но основные идеи обычно остаются неизменными. Аппарат прерываний позволяет системе фиксировать и корректно обрабатывать различные события, возникающие как внутри компьютера, так и вне него.

1.2.4 Внешние устройства

Внешние устройства во многом определяют эксплуатационные характеристики, как компьютера, так и вычислительной системы в целом. Размер экрана монитора, объем и производительность магнитных дисков, наличие печатающих устройств, модемов, и т.д. — характеристики компьютера, на которые зачастую в первую очередь обращает внимание массовый пользователь. Значимость внешних устройств компьютера в вычислительной системе возрастала по мере развития сфер применения вычислительной техники. Основным применением первых компьютеров было численное решение задач моделирования физических процессов, и для этих целей было достаточным иметь в компьютере высокопроизводительный (по меркам того времени) процессор, достаточный для решения задач данного класса объем оперативной памяти, простейшие устройства печати и ввода данных, внешнее запоминающее устройство для хранения исходных и промежуточных данных. Спектр же внешних устройств современных компьютеров несоизмеримо шире, что соответствует разнообразию задач, решаемых средствами современных вычислительных систем (Рис. 34).

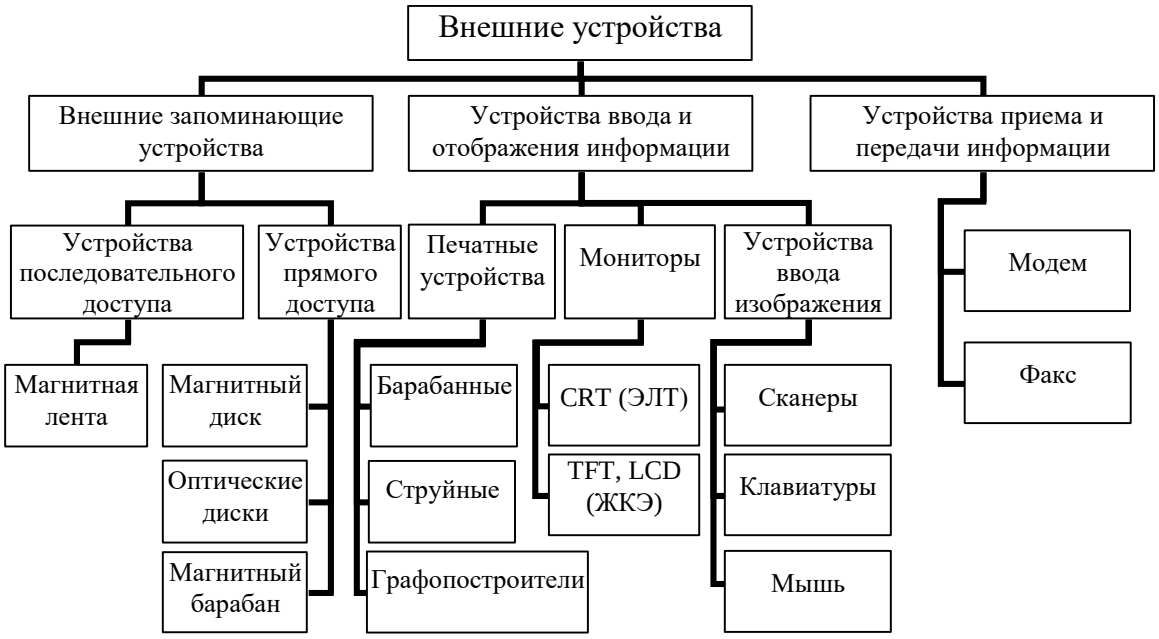


Рис. 34. Иерархия внешних устройств.

Мы более подробно остановимся на характеристиках и особенностях использования внешних запоминающих устройств, как наиболее интенсивно используемых программами внешних устройствах вычислительных систем.

1.2.4.1 Внешние запоминающие устройства

Внешние запоминающие устройства (ВЗУ) предназначены для организации хранения данных и программ. Обычно обмен с ВЗУ происходят некоторыми порциями данных, которые называются ***записями***. Данные, размещенные на ВЗУ, представляются в виде последовательности записей. Существует категория ВЗУ, называемые **блочными устройствами**, которые допускают выполнение обменов исключительно записями фиксированного размера — ***блоками***. Примером блочных устройств могут служить различные типы магнитных дисков. Обычно размер блоков (**физических блоков**), обмен которыми может осуществляться с блочными устройствами, определяется аппаратно и может зависеть от конкретной модели и типа устройства. Альтернативой блочным ВЗУ являются устройства, аппаратно допускающие обмен записями произвольного размера. Примером таких устройств являются устройства хранения информации на магнитных лентах.

ВЗУ могут разделяться на две группы по *возможностям доступа к хранящимся данным*. Первая группа — устройства, аппаратно допускающие как операции чтения, так и операции записи. Примером устройств данной группы может служить жесткий диск. Вторая группа — устройства, позволяющие выполнять только операции чтения данных, например, в эту группу входят устройства CD-ROM (compact disk read-only memory), DVD-ROM (digital video/versatile disc read-only memory).

Внешние запоминающие устройства могут, также подразделяться на **устройства прямого доступа** и **устройства последовательного доступа**. Рассмотрим принципы организации и общие характеристики устройств, принадлежащих каждой из этих групп.

Устройства последовательного доступа — это устройства, при доступе к содержимому произвольной записи которых «просматриваются» все записи, предшествующие искомой. Рассмотрим в качестве примера ВЗУ последовательного доступа устройство хранения данных на магнитной ленте. На магнитной ленте каждая запись имеет специальные маркеры начала и конца. Также, на каждой ленте размещаются маркеры начала и конца ленты (Рис. 35).

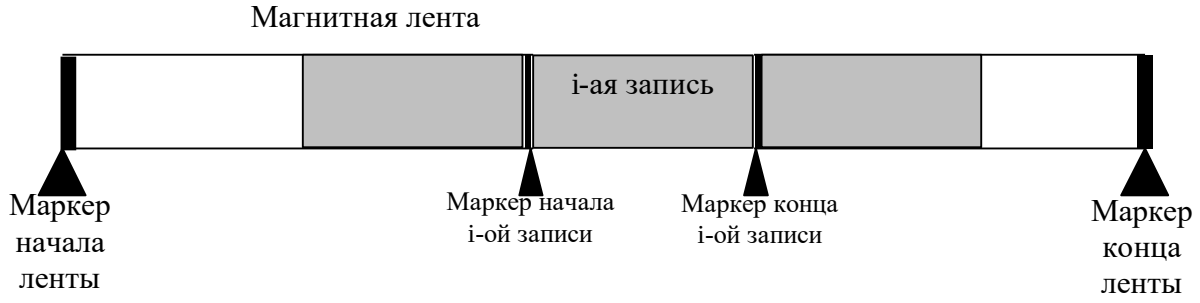


Рис. 35. Магнитная лента.

Каждая запись на ленте имеет свой логический номер. При возникновении запроса на чтение записи с номером *i* выполняется следующая последовательность действий:

– устройство перематывает ленту до маркера начала ленты;

– осуществляется последовательный поиск маркеров начала записей, после нахождения *i*-го маркера считается, что устройство «вышло» на начало искомой записи;

– происходит чтение *i*-ой записи.

Устройство прямого доступа обеспечивает выполнение операций чтения/записи без считывания дополнительной (предыдущей) информации. Примером устройств прямого доступа могут служить **магнитные диски**, или **дисковые устройства**.

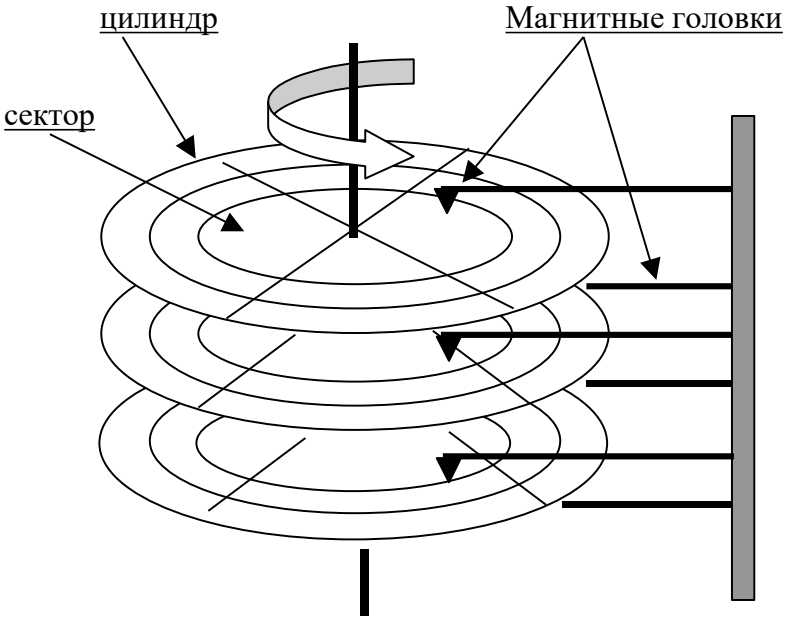


Рис. 36. Принцип устройства магнитного диска.

Магнитные диски являются самыми распространенными устройствами внешней памяти современных компьютеров. Рассмотрим принципиальную схему организации магнитного диска (Рис. 36). Устройство представляет собою вал, вращающийся с достаточно высокой постоянной скоростью. На валу закреплены диски, поверхности которых покрыты материалом, способным на основе магнитоэлектрических эффектов сохранять информацию. Количество дисков варьируется в зависимости от типа дискового устройства. Также в дисковом устройстве присутствует система головок чтения/записи. Количество головок соответствует количеству поверхностей дисков, и каждая головка может работать со своей фиксированной поверхностью. Все головки устройства составляют блок головок магнитного диска. Блок головок может перемещаться от края поверхностей к центру. Перемещение блока головок осуществляется дискретно, каждая позиция остановки блока головок над поверхностями (с учетом вращения дисков) образует цилиндр. Таким образом, каждое дисковое устройство характеризуется фиксированным количеством цилиндров, которые соответствуют позициям, на которых может размещаться блок головок.

Все цилиндры пронумерованы ($0,1,....N_{цилинд}$). Условные линии пересечения цилиндров с поверхностями образуют дорожки. Дорожки, относящиеся к одному цилиндру пронумерованы ($0,1,....N_{дорожки}$). Дорожки, принадлежащие одной поверхности, формируют концентрические круги. Все дорожки разделены на фиксированное (для данного устройства) число равных частей — *секторов*. Секторы каждой дорожки пронумерованы ($0,1,....N_{сектор}$). Начала всех одноименных секторов лежат в одной плоскости, проходящей через вал. При работе магнитного диска предусмотрена возможность индикации факта прохода блока головок через каждую точку начала сектора (это решается с использованием механических или оптических датчиков секторов), таким образом, блок головок всегда может «знать», над каким сектором он находится. В каждый момент времени в блоке головок может проходить обмен с одним из секторов. Рассмотрим пример выполнения операции обмена данными, размещенными в одном из секторов. Для задания координат конкретного сектора в устройство управления магнитным диском должны быть переданы:

- номер цилиндра, в котором расположен данный сектор, — N_c;
- номер дорожки, на которой размещается сектор, — N_t;
- номер сектора — N_s.

После получения координат сектора (N_c, N_t, N_s) выполняется следующая последовательность действий:

– шаговый двигатель перемещает блок головок в цилиндр N_c;

– включается головка чтения/записи, соответствующая номеру дорожки N_t;

– как только головка чтения/записи позиционируется над началом искомого сектора N_s, запускается выполнение операции чтения (или записи).

Таким образом, мы видим, что для выполнения операций обмена с магнитным диском не производится чтение какой-либо дополнительной информации с диска, т.е. обеспечивается «прямой доступ» к информации.

Производительность внешнего запоминающего устройства — время доступа к хранящейся информации — во многом определяется наличием и продолжительностью механических операций, которые необходимо провести при обмене. Так, время обмена с магнитным диском будет определяться, в основном, временем выдвижения блока головок в соответствующий цилиндр (это время перемещения блока головок из начального положения к цилиндру с максимальным номером), а также временем позиционирования головки в начало сектора, с которым будет осуществляться обмен (это время не больше времени полного оборота вала). При работе с магнитной лентой механическая составляющая обмена существенно больше, поэтому магнитные диски являются более высокопроизводительными устройствами и применяются для оперативного хранения обрабатываемых данных. Магнитные ленты используются для организации архивирования и долговременного хранения данных.

Следующее устройство, которое мы рассмотрим, — это **магнитный барабан** (Рис. 37). В данном приборе также имеется электродвигатель, к оси которого прикреплен массивный барабан, поверхность которого покрыта электромагнитным слоем. Двигатель раскручивает барабан до достаточно высокой постоянной скорости. Помимо этого имеется фиксированная штанга, на которой расположены головки чтения-записи. Под каждой головкой логически можно выделить дорожку, которая называется ***треком***. Так же, как и в диске, все дорожки разделены на сектора. Для адресации блока данных в этом случае используется только номер дорожки ($N_{трека}$) и номер сектора ($N_{сектора}$). Для того чтобы произвести операцию чтения или записи, устройство управления должно включить головку, соответствующую указанному номеру дорожки; а после этого происходит ожидание механического поворота цилиндра до выхода головки на начало искомого сектора. Таким образом, по сравнению с жесткими дисками, в этом устройстве отсутствует механическая составляющая выхода головки на нужный трек, поэтому данный тип устройств считается более высокоскоростным.

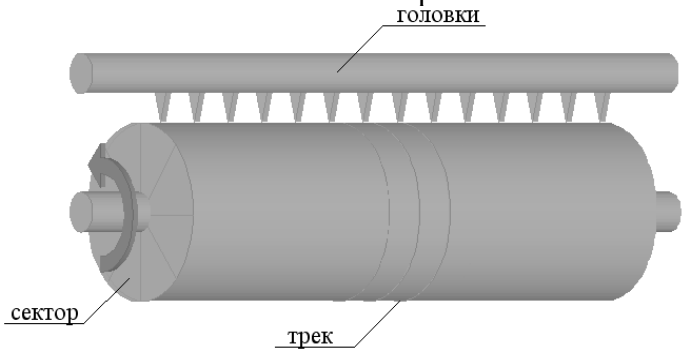


Рис. 37. Принцип устройства магнитного барабана.

Напоследок отметим, что магнитные барабаны на сегодняшний день являются в некотором роде экзотическими устройствами: они используются в основном лишь в больших специализированных высокопроизводительных компьютерах обычно для временного хранения данных из оперативной памяти.

И, наконец, отметим т.н. **память на магнитных носителях (доменах)** – магнитно-электронное ВЗУ прямого доступа. Под ***доменом*** понимается некоторая элементарная единица, способная сохранять свою намагниченность в течение длительного промежутка

времени. Домен может быть намагничен одним из двух способов (отмеченные на Рис. 38 либо как «плюс-минус», либо как «минус-плюс»).

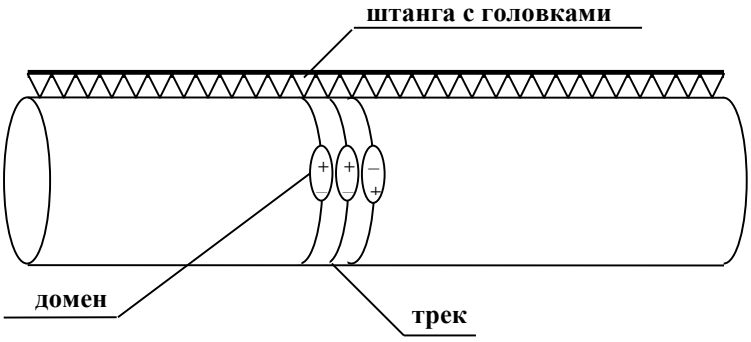


Рис. 38. Принцип устройства памяти на магнитных доменах.

Принцип работы устройства памяти на магнитных доменах основан на том, что под воздействием магнитно-электронных эффектов магнитные домены разгоняются вдоль своего трека до некоторой постоянной скорости. В остальном же принцип работы данного класса устройств ничем не отличается от работы магнитных барабанов. Соответственно, из-за того, что в данном устройстве нет механической составляющей, оно является еще более высокоскоростным по сравнению с предыдущими устройствами.

Для считывания или записи информации на данный носитель устройство управления включает необходимую головку, которая по таймеру синхронизируется с «приходом» начала искомого сектора, после чего происходит обмен с найденным сектором.

1.2.4.2 Модели синхронизации при обмене с внешними устройствами

Важной характеристикой, во многом определяющей эффективность функционирования вычислительной системы, является модель синхронизации, поддерживаемая аппаратурой компьютера при взаимодействии центрального процессора с внешними устройствами.

Для иллюстрации рассмотрим **пример**. Пусть выполняемой в компьютере программе необходимо записать блок данных на магнитный диск. Что будет происходить в системе при обработке заказа на данный обмен? Возможны две модели реализации обмена, рассмотрим их.

Синхронная работа с ВУ. При синхронной организации обмена в момент обращения к внешнему устройству программа будет приостановлена до момента завершения обмена (Рис. 39). Тем самым при использовании такой модели в системе возникали задержки, которые снижали эффективность функционирования ВС.

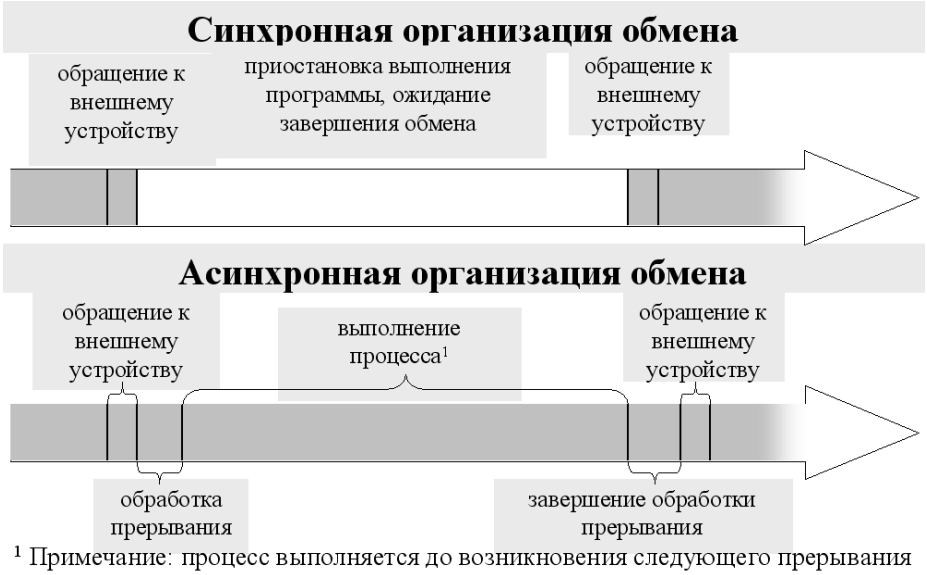


Рис. 39. Синхронная и асинхронная работа с ВУ.

Асинхронная работа с ВУ. При асинхронной организации работы внешних устройств последовательность событий, происходящих в системе, следующая:

1. Для простоты изложения будем считать, что в системе прерываний компьютера имеется специальное внутреннее прерывание «обращение к системе», которое инициируется выполнением программой специальной команды. Программа инициирует прерывание «обращение к системе» и передает заказ на выполнение обмена, параметры заказа могут быть переданы через специальные регистры, стек и т.п. В операционной системе происходит обработка прерывания, при этом конкретному драйверу устройства передается заказ на выполнение обмена.
2. После завершения обработки «обращения к системе» программа может продолжить свое выполнение, или может быть запущено выполнение другой программы.
3. По завершении выполнения обмена происходит прерывание, после обработки которого программа, выполнявшая обмен, может продолжить свое выполнение.

Асинхронная схема обработки обращений к ВУ позволяет сглаживать дисбаланс в скорости выполнения машинных команд и скоростью доступа к ВУ.

В заключении отметим следующее. Представленная выше схема организации обмена является достаточно упрощенной. Она не затрагивает случаев синхронизации доступа к областям памяти, участвующим в обмене. Проблема состоит в том, что, например, записывая область данных на ВЗУ после обработки заказа на обмен, но до завершения обмена, программа может попытаться обновить содержимое области, что является некорректным. Поэтому в реальных системах для синхронизации работы с областями памяти, находящимися в обмене, используется возможность ее аппаратного закрытия на чтение и/или запись. То есть при попытке обмена с закрытой областью памяти произойдет прерывание. Это позволяет остановить выполнение программы до завершения обмена, если программа попытается выполнить некорректные операции с областью памяти, находящейся в обмене (попытка чтения при незавершенной операции чтения с ВУ или записи при незавершенной операции записи данной области на ВУ).

1.2.4.3 Потоки данных. Организация управления внешними устройствами

При рассмотрении работы любого компьютера имеют место два потока информации. Первый поток — это поток управляющей информации, второй поток — это поток данных, над которыми осуществляется обработка в программе. Если рассматривать эти потоки информации в контексте организации работы ВЗУ, то можно выделить также

поток управляющей информации, включающий в себя команды, обеспечивающие управление внешним устройством, а также поток данных, перемещающихся между ВЗУ и оперативной памятью. Рассмотрим теперь различные модели организации управления ВЗУ.

Простейшей моделью является **непосредственное управление процессором** внешними устройствами (Рис. 40). Это означает, что центральный процессор фактически «интегрирован» со схемами управления внешними устройствами, имеет специальные команды управления ими, а также путем интерпретации последовательности команд управления осуществляет управление обменом. Т.е. процессор подает команды устройству на перемещение головок обмена, на включение той или иной головки, на ожидание и синхронизации прихода содержательной информации и пр. Помимо указанного потока команд, центральный процессор обрабатывает и поток данных: он считывает информацию, участвующую в обмене, со специальных регистров и переносит ее в оперативную память (либо же производит обратные манипуляции). Таким образом, и поток управления, и поток данных проходит через центральный процессор, что само по себе является трудоемкой задачей, к тому же эта модель подразумевает лишь синхронную реализацию.

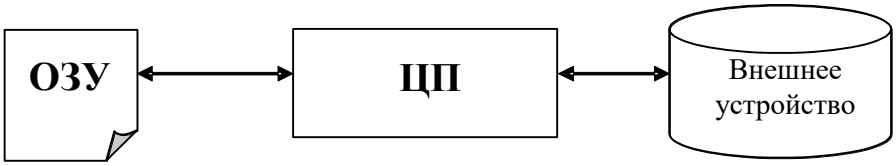


Рис. 40. Непосредственное управление внешними устройствами центральным процессором.

Следующая модель предлагает **синхронное управление** внешними устройствами с использованием контроллеров внешних устройств (Рис. 41). Данная модель появилась вслед за появлением внешних устройств, для которых имелись электронные схемы управления этими устройствами — ***контроллеры***, — взявшие на себя часть работ центрального процессора по управлению обменами. В этом случае контроллер взаимодействует с центральным процессором блоками больших размеров, при этом контроллер может самостоятельно выполнять некоторые работы по непосредственному управлению ВЗУ (например, пытаться локализовать и исправить возможные ошибки, которые могут случиться при чтении или записи данных). Но исторически такой тип управления ВЗУ изначально был синхронным: процессор посылает устройству команды на обмен и ожидает, когда этот обмен завершится. Что касается потока данных, то ничего нового в данной модели не представлено: процессор по-прежнему считывает их со специальных регистров внешнего устройства и помещает их в оперативную память.

Рис. 41. Синхронное/асинхронное управление внешними устройствами с использованием контроллеров внешних устройств.

Вслед за предыдущим типом устройств появились устройства, позволяющие осуществлять **асинхронное управление** с использованием контроллеров ВЗУ (Рис. 41). В этом случае центральный процессор подает команду на обмен и не дожидается, когда эту команду отработают контроллер и устройство, т.е. процессор может продолжить обработку каких-то задач. Но для осуществления указанной модели необходимо, чтобы в системе был реализован аппарат прерываний.

Затем исторически появились т.н. **контроллеры прямого доступа к памяти** (DMA — Direct Memory Access, Рис. 42). Контроллеры данного типа исключили центральный процессор из обработки потока данных, взяв эту функцию на себя. В данной модели предполагается, что центральный процессор занимается лишь обработкой потоком управляющей информации, а данные перемещаются между ВЗУ и ОЗУ уже без его участия.

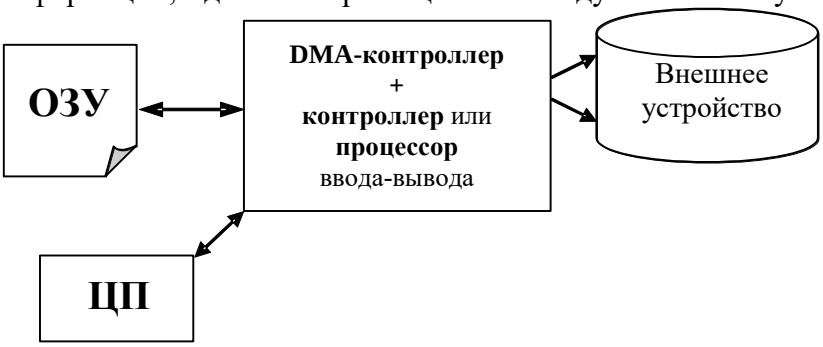


Рис. 42. Использование контроллера прямого доступа к памяти (DMA) или процессора (канала) ввода-вывода при обмене.

И, наконец, последняя модель основана на использовании **процессора** или **канала ввода-вывода** (Рис. 42). В этом случае предполагается наличие специализированного компьютера, который имеет свой процессорный элемент, свою оперативную память, который функционирует под управлением своей ОС, и этот компьютер располагается логически между центральным процессором и внешними устройствами. В функции подобных процессоров или каналов входит высокоуровневое управление внешних устройств. В этом случае центральный процессор оперирует с внешними устройствами в форме высокоуровневых заказов на обмен. Соответственно, реализация непосредственного управления конкретным ВЗУ осуществляется в процессоре ввода-вывода (в частности, в нем может происходить многоуровневая фиксация ошибок, он может осуществлять аппаратное кэширование обменов к конкретному устройству и пр.).

1.2.5 Иерархия памяти

Рассматривая вычислительную систему (или компьютер) можно выстроить некоторую последовательность устройств, предназначенных для хранения информации в некотором ранжированном порядке – иерархии. Этот порядок можно определять на основе различных критериев: например, по стоимости хранения единицы информации или по скорости доступа к данным, но так или иначе устройства будут располагаться примерно в одном порядке (Рис. 43).

Рис. 43. Иерархия устройств хранения информации.

Самой дорогостоящей и наиболее высокопроизводительной памятью является память, которая размещается в центральном процессоре (это **регистровая память** и **КЭШ первого уровня (L1)**).

Следующим звеном в этой иерархии может являться **КЭШ второго уровня (L2)**. Это устройство логически располагается между процессором и оперативной памятью, оно является более дешевым и менее скоростным, чем КЭШ первого уровня, но более дорогое и более скоростное, чем **ОЗУ**, которое располагается на следующей ступени иерархии. Одним из основных свойств оперативной памяти является то, что в ней располагается исполняемая центральным процессором в данный момент программа, т.е. процессор «берет» очередные операнды и команды для исполнения именно из оперативной памяти.

Ниже ОЗУ в приведенной иерархии следуют устройства, предназначенные для оперативного хранения программной информации пользователей и ОС. Сначала естественным образом следуют **ВЗУ прямого доступа с внутренней КЭШ-буферизацией**. Это дорогостоящие устройства, они предназначены для наиболее оперативного обмена. Так, на этих устройствах операционная система может размещать свои всякого рода информационные таблицы.

Следом за предыдущим типом устройств следуют **ВЗУ прямого доступа без КЭШ-буферизации**, которые также обеспечивают оперативных доступ, но уже на более низких скоростях. На подобных устройствах может находиться файловая система пользователей, код ОС (поскольку для системного устройства, с которого происходит загрузка ОС, скорость не особенно актуальна в отличие от устройства, хранящего данные работающей ОС).

И в самом низу иерархии располагаются **ВЗУ долговременного хранения данных**. Это системы резервирования, системы архивирования и т.д. Назначения данного класса устройств могут быть самыми разными, но все они характеризуются низкой скоростью доступа к данным и достаточно низкой стоимостью хранения единицы информации.

1.2.6 Аппаратная поддержка операционной системы и систем программирования

Если мы обратим свое внимание на рассмотрение компьютеров первого поколения, то это были *компьютеры* (*computer — вычислитель*) в прямом смысле слова, т.е. производители первых компьютеров ставили перед собой цель создание автоматических вычислений (причем достаточно в большом количестве). Но со временем круг

пользователей расширялся, что привело к возникновению необходимости в присутствии в аппаратуре компьютера компонентов, предназначенных не столько для организации автоматизации вычислений, сколько для организации управления этими вычислениями. Этот раздел посвящен таким компонентам компьютера, которые изначально предназначались для аппаратной поддержки функционирования программного обеспечения (ПО), в частности, операционной системы и систем программирования.

Мы можем проследить историческую цепочку событий, связанных с развитием аппаратной поддержки ПО. Первый шаг – это появление программируемого компьютера, т.е. устройства, которое умеет исполнять программу. Второй шаг – это появление регистров общего назначения, которые позволили на программном уровне сгладить разницу производительности центрального процессора и оперативной памяти. Третий шаг – это появление аппарата прерываний, который позволил организовать асинхронную работу с внешними устройствами и т.д. Вообще, концепция аппаратной поддержки ПО всё более и более распространяется в современной архитектуре. Вспомним, например, теги, которые присутствуют в ячейках памяти. Теги используются как для контроля содержимого, целостности данных, так и для возможности «раскрашивания» данных по типам. Давайте продолжим эту цепочку в связи со средствами поддержки мультипрограммного режима работы вычислительной системы.

1.2.6.1 Требования к аппаратуре для поддержки мультипрограммного режима

Выше уже шла речь о **мультипрограммном режиме** – режиме работы ВС, при котором в каждый момент времени в системе могут обрабатываться две и более программ пользователей (эти программы иногда называют *смесью программ*). Каждая из этих программ может находиться в одном из *трех состояний*: во-первых, программа может выполняться на процессоре (т.е. ее команды исполняются центральным процессором), во-вторых, программа может ожидать завершения запрошенного ею обмена (для продолжения ее выполнения необходимо окончания обмена), и, наконец, в-третьих, программы могут находиться в ожидании освобождения центрального процессора (эти программы готовы к выполнению на процессоре, но процессор в данный момент занят иной программой). Мультипрограммный режим — это режим наиболее эффективной загрузки центрального процессора. На сегодняшний день мультипрограммный режим позволяет обрабатываться на компьютере большому числу процессов (задач), предоставляющих пользователю широкий круг различных услуг.

Рассмотрим *схему организации мультипрограммного режима* (Рис. 44). Пусть в начальный момент времени на процессоре обрабатывается *Программа 1*, которая в некоторый момент времени t_1 выдает запрос на обмен, при этом дальнейшая обработка на процессоре невозможна до завершения этого обмена. В случае синхронной организации *Программа 1* будет приостановлена, и процессор будет простаивать до завершения обмена *Программы 1*. Соответственно, со временем последовало естественное предложение запускать на обработку центральным процессором другие программы, пока *Программа 1* ожидает завершения своего обмена. На Рис. 44 проиллюстрирована ситуация, когда при запуске обмена для *Программы 1* на счет ставится *Программа 2*, которая выполняется до некоторого момента времени t_2, после чего она приостанавливается по тем или иным причинам, и запускается *Программа 3*. После завершения обмена на обработку вновь ставится *Программа 1*, сменяя *Программу 3* в момент времени t_3.

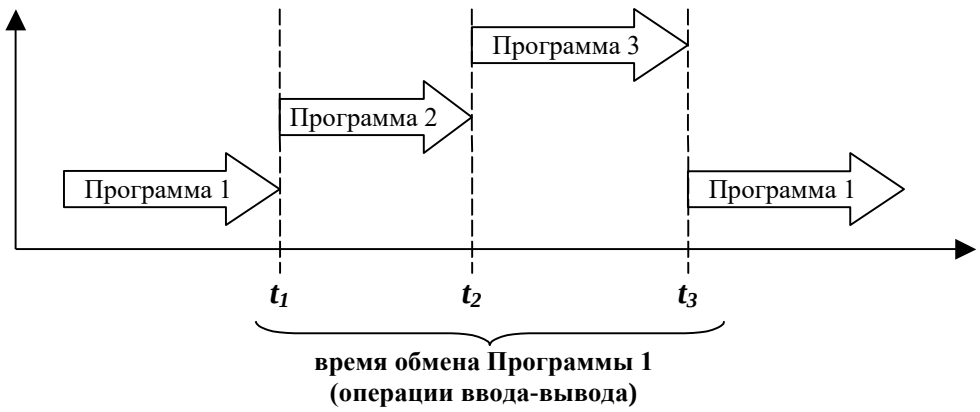


Рис. 44. Мультипрограммный режим.

Естественно, для предложенного подхода возникает вопрос, какие аппаратные средства необходимы для корректного функционирования указанной системы. Под **корректным** функционированием мы будем понимать, что в независимости от степени мультипрограммирования (от количества обрабатываемых в системе программ) результат работы конкретной программы не зависит от наличия и деятельности других программ. Чтобы понять, какие требования предъявляются подобным системам, разберем сначала, какие трудности и проблемы могут возникнуть при мультипрограммном режиме. Это проблемы, связанные, в первую очередь, с «общим ведением хозяйства», т.е. когда обрабатываемые программы претендуют на доступ к одним и тем же ресурсам ВС. Возникает ситуация конкуренции – поэтому для корректной работы необходимо обеспечивать ограничение доступа.

Первая проблема, которая может возникнуть, — это влияние программ друг на друга. Очень нежелательна ситуация, когда одна программа может обратиться в адресное пространство другой программы и считать оттуда данные (поскольку все-таки необходимо обеспечивать конфиденциальность информации), и уж совсем плохо, когда другая программа может что-то записать в чужое адресное пространство. Соответственно, для корректного мультипрограммирования система должна обеспечивать программам эксклюзивное владение выделенными им участками памяти. Если возникает задача обеспечения множественного доступа к памяти, то это должно осуществляться с согласия владельца этой памятью. Итак, первое требование к системе — это наличие т.н. ***аппарата защиты памяти*** обрабатываемых программ. Это аппаратно-программное средство компьютера. Это означает, что операционная система, когда загружает программу в оперативную память, настраивает работу процессора, т.е. устанавливает в специальные регистры процессора режимы, которые ограничивают использование данной программой оперативной памяти. Сразу отметим, что режим защиты памяти *нельзя делать чисто программным способом*, поскольку если данный режим будет обеспечивать операционная система (т.е. каждый раз сравнивать получаемый исполнительный адрес – не вышел ли он за границы дозволенного программе диапазона адресов), то производительность вычислительной системы в целом будет крайне низкой.

Реализация аппарата защиты памяти может быть достаточно простой: в процессоре могут быть специальные регистры (***регистры границ***), в которых устанавливаются границы диапазона доступных для исполняемой задачи адресов оперативной памяти. Соответственно, когда устройство управления в центральном процессоре вычисляет очередной исполнительный адрес (это может быть адрес следующей команды или же адрес необходимого операнда), **автоматически** проверяется, принадлежит ли полученный адрес заданному диапазону. Если адрес принадлежит диапазону, то продолжается обработка

задачи, иначе же в системе возникает прерывание (т.н. ***прерывание по защите памяти***). Отметим, что предложенная модель в реальной аппаратуре может быть реализована множеством способов, но главное, что при постановке программы на обработку операционная система (программным способом) задает значения указанных регистров границ, а дальнейшая проверка адресов осуществляется аппаратным способом.

Рассмотрим следующий круг возникающих при мультипрограммном режиме проблем. Предположим, в нашей мультипрограммной системе (например, терминальном классе) имеется единственное печатающее устройство, и есть несколько программ, которые выводят свои данные на печать данному устройству. Соответственно, если каждая программа будет иметь доступ к командам управления конечными физическими устройствами, то при совместной работе в режиме мультипрограммирования эти программы будут вперемешку обращаться к печатающему устройству и печатать на нем порции своих данных, что в итоге приведет к невозможности интерпретации напечатанной информации.

Другим примером может служить только что обсуждавшийся аппарат защиты памяти. Значения указанных регистров границ устанавливаются посредством специальных машинных команд (ОС настраивает границы, а аппаратура автоматически проверяет). Представьте ситуацию, когда к указанным командам смогут обращаться произвольные программы: тогда смысла в аппарате защиты памяти просто не будет — любая программа сможет обойти этот режим подменой своих регистров границ.

Рассмотрение представленных примеров должно наводить на мысль, что система должна каким-то способом ранжировать команды и в соответствии с этим ранжированием ограничивать доступ пользователей различных категорий к машинным командам. Решением стала **аппаратная** возможность работы центрального процессора в двух режимах: в ***режиме работы операционной системы*** (или ***привилегированном режиме***, или ***режиме супервизора***) и в ***пользовательском режиме*** (или ***непривилегированном режиме***, еще раньше использовался термин ***математического режима***). В режиме работы ОС процессор может исполнить любую из своих машинных команд. Если же программа исполняется в пользовательском режиме, то ей доступно для исполнения лишь некоторое подмножество машинных команд (если же при обработке такой программы встретится недопустимая команда, то в системе возникнет прерывание по запрещенной команде). В частности, в состав запрещённых команд включаются все команды, обеспечивающие управление режимами центрального процессора, управление физическими устройствами и т.д.

Тогда возникает вопрос, что же должна делать программа, обрабатываемая в пользовательском режиме, для печати своих данных, например. Решений здесь может быть достаточно много, одним из которых может быть наличие в системе специальных команд, интерпретируемых как обращения к операционной системе (которые в некоторых системах рассматриваются как прерывания, в других системах — не как прерывания; мы будем рассматривать их как *прерывания по обращению к операционной системе*). Тогда программа, работающая в непривилегированном режиме, может вызывать команды обращения к операционной системе, а через параметры передать необходимые данные, которые могут свидетельствовать о желании данной программы распечатать какую-либо информацию на устройстве печати. Тогда схема организации печати данных на устройстве печати может выглядеть следующим образом. Операционная система получает от пользователей (т.е. от пользовательских программ) заказы на печать, и для каждой из программы она формирует некоторую таблицу или область памяти, в которой будет аккумулироваться информация, которую необходимо вывести на принтер. Тогда каждый запрос программ на печать порции данных не является реальным обращением к устройству печати, но свидетельствует лишь о том, что передаваемая порция данных должна быть распечатана, а ОС их аккумулирует. Реальная печать будет осуществляться при возникновении одного из трех событий. Во-первых, программа, посылающая данные на

печать, успешно завершилась. Это означает, что гарантированно она не будет более посылать данные на печать. Во-вторых, в программе обнаружилась фатальная ошибка, что ведет к безусловному завершению этой программы, что опять-таки гарантирует отсутствие будущих запросов данной программы на печать. И, в-третьих, операционная система может получить (от некоторого виртуального оператора – т.н. *планировщика*) команду разгрузить буфер печати данной конкретной программы.

И, наконец, ещё одна серьезная проблема, которая может возникнуть при организации мультипрограммного режима, связана с появлением в выполняемой в текущий момент программе семантической ошибки — *программа зациклилась*. Соответственно, если в этом цикле не встречаются команды, которые могут привести к тем или иным прерываниям, то в этом случае вся вычислительная система «зависает»: никакие новые задачи не ставятся на счет и пр. Решение данной проблемы может быть довольно простым: необходима функция управления временем. Это означает, что операционная система должна контролировать время использования центрального процессора программами пользователей. Для этих целей компьютеру требуется **прерывание по таймеру**. Резюмируя, можно сказать, что для реализации мультипрограммного режима необходимо наличие аппарата прерываний, и этот аппарат, как минимум, должен включать в себя аппарат прерывания по таймеру. В этом случае зацикленная программа автоматически будет периодически прерываться, управление периодически будет передаваться операционной системе (ОС может, например, считать время, которое текущий процесс использовал за последний сеанс работы с ЦП). Это даст возможность операционной системе принять решение – поставить на счет другую программу либо снять со счета (например, по команде пользователя) эту зависшую программу. Существуют две стратегии работы со временем. Первая стратегия (которая используется в Windows, в «ширпотребовской» Unix) состоит в том, что любой процесс берёт время ЦП до тех пор, пока его не убьют. Тем не менее, при этой стратегии зацикленная программа не будет никому мешать, т.к. ОС будет периодически получать управление и переключать процессы. Другая стратегия (которая обычно применяется для высокопроизводительных машин) основывается на том, что процесс предварительно оговаривает, сколько времени ему может потребоваться. Если процесс пытается превысить это время, то ОС считает, что процесс зациклился.

Задание для размышления – попробовать смоделировать корректную мультипрограммную систему, которая будет работать, имея аппарат защиты памяти, специальный режим и только прерывание по таймеру. Например, вместо того, чтобы иметь прерывание по завершению обмена, можно периодически (по таймеру) просто считывать какие-то регистры и их анализировать.

Итак, требуются *три аппаратных средства компьютера*, необходимых для поддержки мультипрограммного режима: аппарат защиты памяти, специальные режимы исполнения команд и аппарат прерываний, состоящий, как минимум, из аппарата прерывания по таймеру. Отметим, что специальных режимов может быть больше двух: т.е. часть команд доступна всем программам, часть команд могут выполняться лишь в защищенном режиме, еще часть — в более защищенном режиме, и т.д.

Может возникнуть резонный вопрос, как происходит включение режима супервизора. Ответ здесь будет зависеть от архитектуры конкретной системы. Например, в некоторых архитектурах считается, что операционная система занимает некоторое предопределенное адресное пространство физической памяти. И если управление попадает на эту область, то включается режим операционной системы. А вот выключение режима операционной системы может происходить программно: например, операционная система, запуская процесс, может предварительно программным способом установить его в непривилегированный режим.

1.2.6.2 Проблемы, возникающие при исполнении программ

Рассмотрим круг проблем, которые, так или иначе, возникают при исполнении программ. Эти проблемы приводят к снижению эффективности системы.

Вложенные обращения к подпрограммам (Рис. 45). Несколько лет назад проводились исследования, которые анализировали распределение времени исполнения программы на разных компонентах программного кода, и выяснилось, что в системах, рассчитанных не только (и не столько) на выполнение математических вычислений (например, в системах обработки текстовой информации), порядка 70% времени тратится на обработку входов и выходов из подпрограмм. Это объясняется тем, что при обращении к подпрограмме необходимо зафиксировать адрес возврата, сформировать параметры, передаваемые вызываемой подпрограмме, как-то сохранить регистровый контекст (т.е. сохранить содержимое тех регистров, которые использовались в программе на данном текущем уровне).

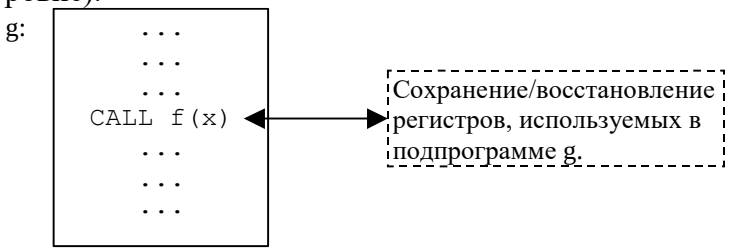


Рис. 45. Вложенные обращения к подпрограммам.

Накладные расходы при смене обрабатываемой программы. Это аналогичная проблема, связанная со сменой обрабатываемых программ (или процессов): *операционная система* должна сохранить контексты процессов. К этому необходимо добавить, что в современных компьютерах количество одновременно обрабатываемых процессов очень велико, что лишь увеличивает объем возникающих накладных расходов.

Перемещаемость программы по ОЗУ. Рассмотрим процесс получения исполняемого кода программы (Рис. 46). После того, как исходный текст программы попадает на вход компилятору, образуется объектный модуль. А уже из пользовательских модулей и библиотечных формируется исполняемый код, т.е. тот модуль, который можно загрузить в оперативную память и начать его исполнять, причем момент создания исполняемого модуля и момент запуска его на исполнение разнесены во времени. Возникает проблема привязки исполняемого модуля к тому адресному пространству, в котором он будет исполняться (Рис. 47). Исторически первые исполняемые модули настраивались на те адреса оперативной памяти, в рамках которых эти исполняемые модули должны были исполняться. Это означает, что если память в данный момент занята другой программой, то эту программу поставить на счет не удастся (пока память не освободится). И, соответственно, возникает проблема перемещаемости программы по ОЗУ: ресурс свободной памяти в ОЗУ может быть достаточно большим, чтобы в ней разместилась вновь запускаемая программа, но в силу привязки каждой программы к конкретным адресам ОЗУ эту программу запустить не удается.

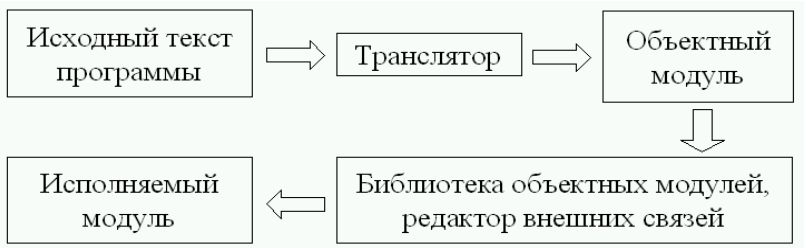


Рис. 46. Процесс получения исполняемого модуля программы.

Рис. 47. Перемещаемость программы по ОЗУ.

Фрагментация памяти. Положим, что предыдущая проблема, связанная с перемещаемостью программы, решена в нашей системе: любой исполняемый модуль может быть загружен в произвольное место ОЗУ для дальнейшего выполнения, но программа загружается в непрерывный фрагмент памяти. В этом случае возникает иная проблема.

Пусть наша система функционирует в мультипрограммном режиме. И в начале работы были загружены в ОП и запущены на исполнение *Программа 1*, *Программа 2* и т.д., вплоть до некоторого номера *К*. Со временем некоторые задачи завершаются, а, соответственно, место, занимаемое ими в ОЗУ, высвобождается. Операционная система способна оценивать свободное пространство оперативной памяти и из буфера программ, готовых к исполнению, выбрать ту программу, которая может поместиться в свободный фрагмент памяти. Но зачастую размер загружаемой программы несколько меньше того фрагмента, который был свободен. И постепенно проявляется т.н. проблема ***фрагментации оперативной памяти*** (Рис. 48). В некоторый момент может оказаться, что в ОЗУ находится несколько процессов, между которыми имеются фрагменты свободной памяти, каждый из которых (по отдельности) не достаточен для того, чтобы загрузить какую-либо готовую к исполнению программу. При этом количество подобных фрагментов может быть настолько большим, что суммарно свободное пространство ОЗУ позволило бы разместить в нем хотя бы один готовый к исполнению процесс. Таким образом, система начинает деградировать: имея ресурс свободной памяти, мы не можем его использовать, а это означает, что система используется в усеченном качестве.

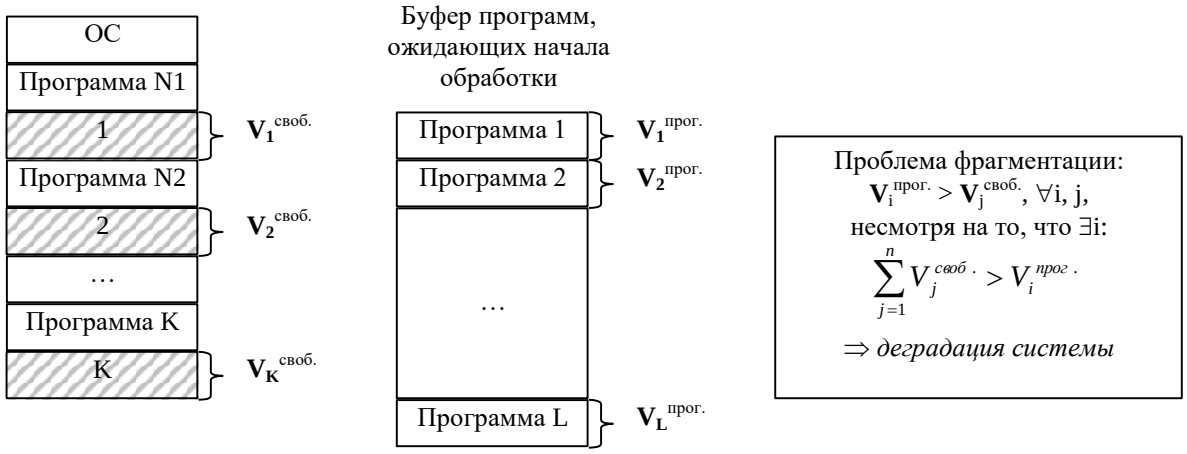


Рис. 48. Фрагментация памяти.

После того, как мы указали основные проблемы, возникающие при исполнении программ, рассмотрим, какие аппаратные средства применяются для разрешения этих проблем.

1.2.6.3 Регистровые окна

Одно из более или менее новых решений, предназначенное для минимизации накладных расходов, связанных с обращениями к подпрограммам, основано на использовании в современных процессорах т.н. ***регистровых окон*** (***register windows***). Это решение нацелено на решение проблем сохранения/ восстановления регистров. Суть этого решения заключается в следующем (Рис. 49). В процессоре имеется некоторое количество *K* физических регистров общего назначения, предназначенных для использования в пользовательских программах. Эти регистры пронумерованы от 0 до *K–1*. В каждый момент времени программе доступно т.н. *регистровое окно*, состоящее из *L* регистров (*L*<*K*). Соответственно, все *K* физических регистров разделяются на регистровые окна некоторым способом. Один из способов предполагает, что с нулевого физического регистра начинается нулевое физическое окно, причем в этом нулевом физическом окне программе пользователя доступны физические регистры с номерами от 0 до *L–1*. Первое физическое окно представляет собою очередные *L* регистров, которые внутри окна также имеют нумерацию от 0 до *L–1*, но в реальности им соответствуют физические регистры с номерами, начинающимися с *L–1*. Т.е. окна организованы таким способом, что последний регистр предыдущего окна отображается на тот же физический регистр, что и нулевой регистр следующего окна.

При обращении из текущей программы в другую программу автоматически происходит смена окна, т.е. текущей программе становится доступно другое окно, состоящее тоже из *L* регистров. Структура регистрового окна представима в виде следующей последовательности регистров (Рис. 50). Первая группа регистров – регистры формальных параметров (включая адреса возврата), вторая – регистры локальных параметров, третья – регистры фактических параметров. Каждый из регистров окна отображается на один из регистров базового регистрового файла. Далее, следующее регистровое окно, которое получит программа при обращении к подпрограмме, имеет пересечение с текущим окном, через начало и конец регистрового окна. Это означает, что если в текущей программе мы загрузили значения на регистры, через которые будем передавать фактические параметры, то при обращении к подпрограмме она в своём новом

окне получит фактические параметры через соответствующие формальные параметры. При этом локальные регистры текущей подпрограммы сохранять не нужно.

Итак, весь имеющийся регистровый файл, состоящий из *K* физических регистров, разбит на *N* окон, в каждом из которых регистры имеют номера от *0* до *L–1*. Соответственно, в системе организована логика таким способом, что все окна расположены в циклическом списке: нулевое окно пересекается с первым, первое — со вторым, и так далее, вплоть до N–1-ого окна, которое пересекается снова с нулевым. Также в системе имеется команда смены окна. Соответственно, при обращении к подпрограмме через пересекающиеся точки передаются адреса возвратов, а внутри окна можно работать с регистрами, причем при обращении к подпрограмме не встает необходимость их сохранения.

Имеются два управляющих регистра: указатель текущего окна (**CWP)** и указатель сохранённого окна **(SWP)**. Суть заключается в том, что количество регистровых окон ограничено. И если глубина вложенности больше, чем количество регистровых окон, то возникает проблема. В этом случае какое-то окно необходимо сохранить в ОП или стеке, чтобы потом его использовать. При этом эффективность, естественно, начнёт падать. Считается, что наиболее оптимальный эффект оптимизации достигается при четырех окнах – это означает, что средний уровень вложенности подпрограмм не более четырех. Недостатком такого решения является фиксированный размер каждого окна, что на практике часто оказывается неоптимальным (т.к. иногда требуется больше регистров, иногда — меньше).

Ниже на Рис. 51 приведены простейшая последовательность действий при входе и выходе из подпрограмм. Простой пример работы с двумя регистровыми окнами представлен на Рис. 52.

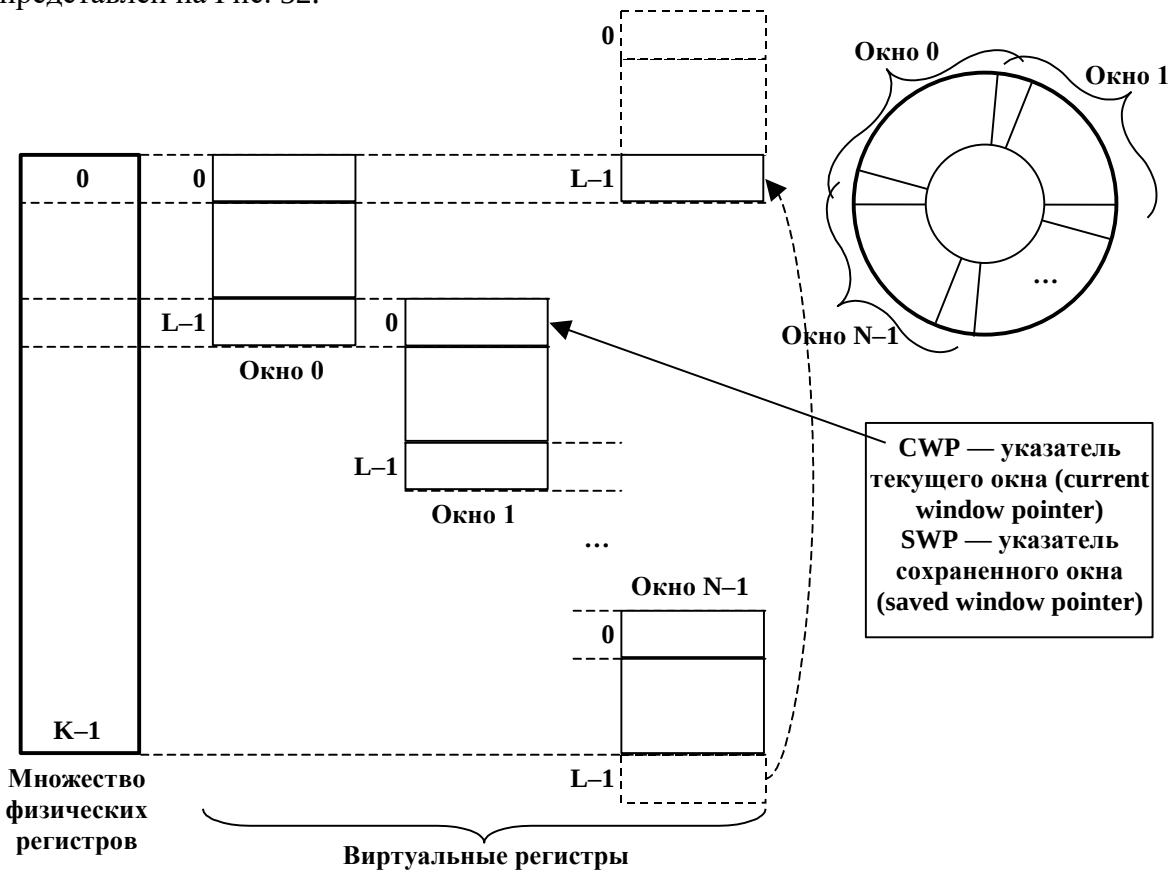


Рис. 49. Регистровые окна.

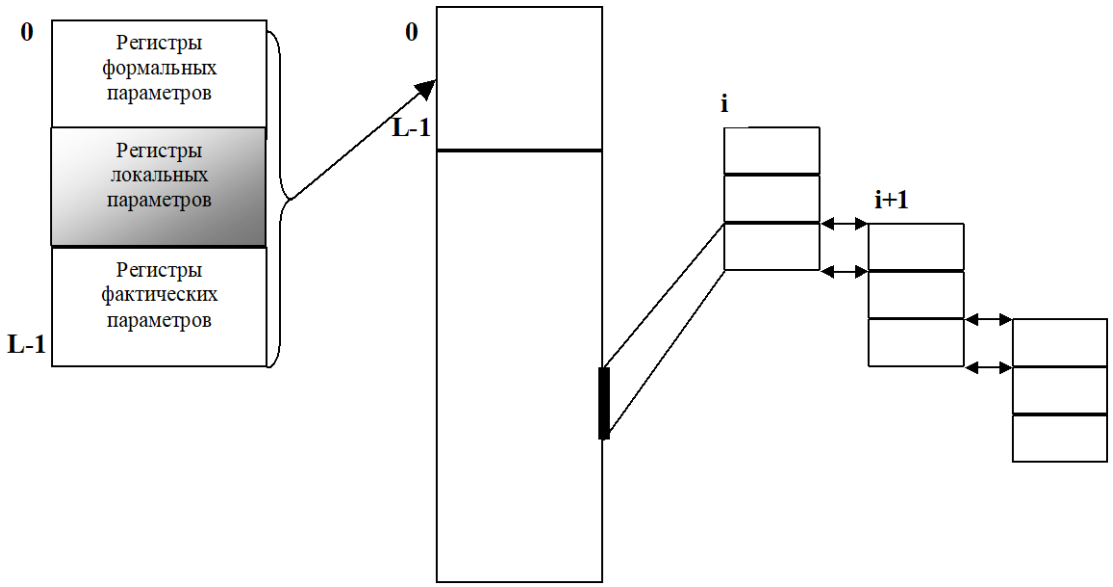


Рис. 50. Структура регистрового окна.

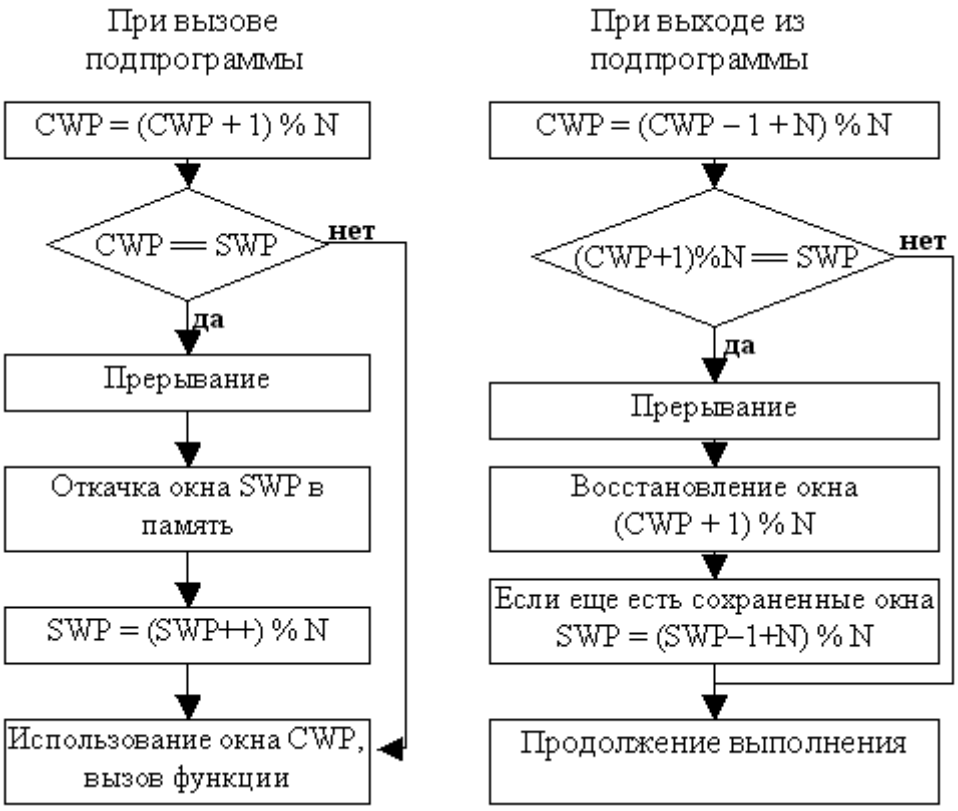


Рис. 51. Регистровые окна. Вход и выход из подпрограммы.

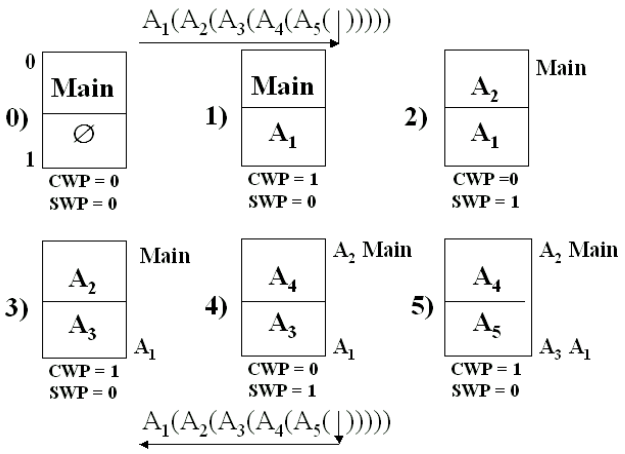


Рис. 52. Пример работы с регистровыми окнами.

Модель организации регистровой памяти в Intel Itanium (Рис. 53). Рассмотрим более эффективную модель работы с регистровыми окнами. В современных компьютерах имеется возможность варьирования размера регистрового окна. В частности, в 64-разрядных процессорах Itanium компании Intel размер окна *динамический*. В данном процессоре в регистровом файле, состоящем из 128 регистров, первые 32 регистра (с номерами от 0 до 31) являются общими, а на регистрах с номерами от 32 по 127 организуются регистровые окна, причем окно может быть произвольного размера (например, от регистра GR до регистра с номером 32+N, где N=0..95). Такая организация позволяет оптимизировать работу с точки зрения входов-выходов из функций и замены функциональных контекстов.

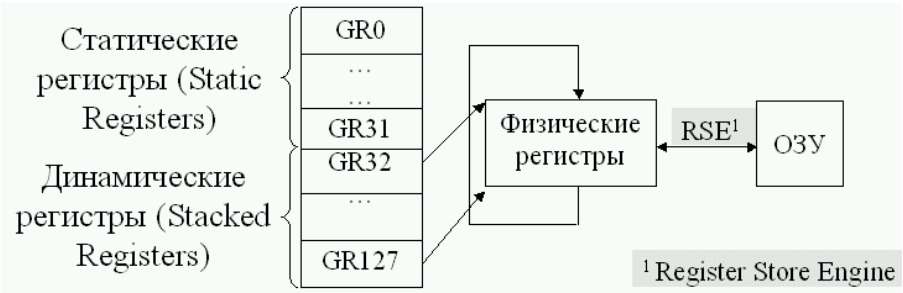


Рис. 53. Модель организации регистровой памяти в Intel Itanium.

1.2.6.4 Системный стек

Будем рассматривать системы, в которых имеется аппаратная поддержка стека. Это означает, что имеется регистр, который ссылается на вершину стека, и есть некоторый механизм, который поддерживает работу со стеком. Использование системного стека может частично решать проблему минимизации накладных расходов при смене обрабатываемой программы. В частности, этот механизм может использоваться при обработке прерывания: если в системе возникает прерывание, процессор просто сохраняет в стеке содержимое необходимых регистров («малое упрятывание»). Если же возникнет второе прерывание, то процессор поверх предыдущих данных скинет в стек новое содержимое регистров, чтобы обработать вновь пришедшее прерывание.

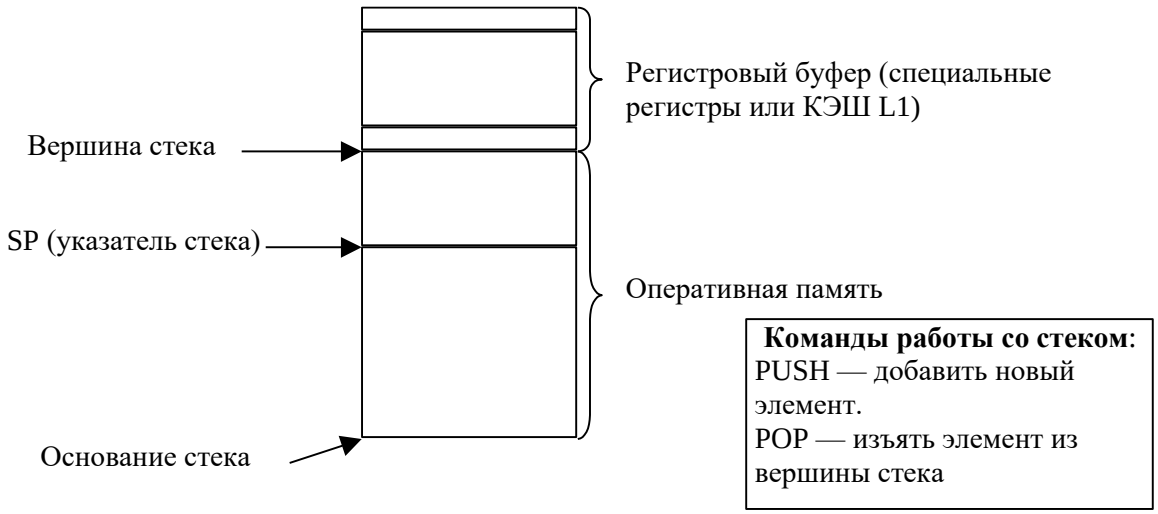


Рис. 54. Системный стек.

Но у данного подхода есть и недостаток. Поскольку стек располагается в оперативной памяти, то при каждой обработке прерывания процессору придется обращаться к оперативной памяти, что сильно снижает производительность системы при частых возникновениях прерываний. Решений может быть несколько (Рис. 54). Во-первых, в процессоре могут использоваться специальные регистры, исполняющие роль буфера, аккумулирующего вершину стека непосредственно в процессоре. Во-вторых, работу со стеком можно организовать посредством буферизации в КЭШе первого уровня (L1), но при кэшировании стека мы добавляем ещё один поток информации.

1.2.6.5 Виртуальная память

Перейдём теперь к проблеме организации, использования и управления оперативной памятью. Рассмотренные выше проблемы перемещаемости программы по ОЗУ и фрагментации памяти связаны с необходимостью наличия т.н. аппарата ***виртуальной памяти***, т.е. аппаратного средства процессора, которое обеспечивает преобразование (установление соответствия) логических адресов, используемых внутри программы, в те адреса физической оперативной памяти, в которой размещается программа во время выполнения.

Что понимается под виртуальной памятью и виртуальным адресным пространством? Неформально *виртуальное адресное пространство* можно определить как то адресное пространство, которое используется внутри программ (написанных, например, на языках программирования высокого уровня). Ведь когда программист пишет программу, оперируя теми или иными адресами, он зачастую не задумывается, к каким физическим адресам эти адреса будут привязаны. В исполняемом модуле используется т.н. программная (логическая, виртуальная) адресация. Виртуальные адреса существуют «вне машины». Соответственно, стоит проблема установления соответствия между программной адресацией и физической памятью. И эта проблема решается за счет аппарата виртуальной памяти.

Реализацией одной из моделей аппарата виртуальной памяти является аппарат ***базирования адресов***. Механизм базирования адресов основан на двоякой интерпретации (Рис. 55) получаемых в ходе выполнения программы исполнительных адресов ($A_{исп.}^{прог.}$). С одной стороны, его можно интерпретировать как ***абсолютный исполнительный адрес***, когда физический адрес в некотором смысле соответствует исполнительному адресу программы ($A_{исп.}^{физ.} = A_{исп.}^{прог.}$). Например, требуется «прочитать ячейку с адресом (абсолютным адресом) 0», или «передать управление по адресу входа в обработчик

прерывания». С другой стороны, исполнительный адрес программы можно проинтерпретировать как ***относительный адрес,*** т.е. адрес, зависящий от места дислокации программы в ОЗУ (адрес относительно точки загрузки программы). Иными словами, имеется оперативная память с ячейками с номерами от 0 до некоторого L–1, и, начиная с некоторого адреса *K*, расположена программа. Тогда адрес $A_{исп.}^{прог.}$ внутри программы можно трактовать, как отступ от физической ячейки с адресом K на величину $A_{исп.}^{прог.}$. Для реализации модели базирования используется специальный ***регистр базы***, в который в момент загрузки процесса в оперативную память операционная система записывает начальный адрес загрузки (т.е. *K*). Тогда реальный физический адрес получается, исходя из формулы $A_{исп.}^{физ.} = A_{исп.}^{прог.} + \langle R_{базы} \rangle$.

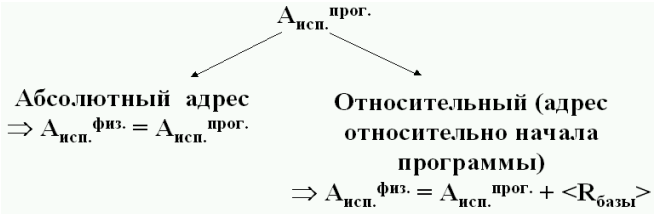


Рис. 55. Базирование адресов – решение проблемы перемещаемости программы по ОЗУ.

Таким образом, базирование адресов – это средство отображения виртуального адресного пространства программы в физическую память «один в один». Аппарат базирования позволяет разрешить проблему перемещаемости программ по ОЗУ, поскольку процесс можно загрузить в любую область памяти. Но при этом необходимо помнить, что программа представляется в виде непрерывной области виртуальной памяти, которая загружается в непрерывный фрагмент физической памяти. Поэтому для решения проблемы фрагментации аппарата базирования недостаточно, и используются более развитые механизмы организации ОЗУ и виртуальной памяти.

Развитием аппарата виртуальной памяти является аппарат ***страничной организации памяти***. Ниже мы рассмотрим ***модельный*** сильно упрощенный ***пример*** страничной памяти (наша цель – рассмотрение основных концепций). Данная модель представляет все адресное пространство оперативной памяти в виде последовательности блоков фиксированного размера, называемых страницами. ***Страница*** — это область адресного пространства фиксированного размера: обычно размер страницы кратен степени двойки – будем считать, что размер страницы 2^k. Тогда структура адреса представима в виде двух полей (Рис. 56): правые *k* разрядов представляют адрес внутри страницы, а оставшиеся разряды отвечают за номер страницы. Тогда количество виртуальных страниц в системе ограничено разрядностью поля «Номер виртуальной страницы» в адресе.

Итак, ***виртуальное адресное пространство*** — это множество виртуальных страниц, доступных для использования в программе. ***Физическое адресное пространство*** — это оперативная память, подключенная к данному компьютеру. Физическая память может иметь произвольный размер по отношению к размеру виртуальной памяти (число физических страниц может быть меньше, больше или равно числу виртуальных страниц).

Рис. 56. Страничная организация памяти.

В центральном процессоре имеется аппаратная (регистровая) таблица, называемая ***таблицей страниц***, предназначенная для организации отображения между виртуальными и физическими адресами при страничной организации памяти. Количество строк в этой таблице определяется максимальным числом виртуальных страниц, которое зависит от схем работы процессора и максимальной адресной разрядности процессора. Каждой виртуальной странице исполняемой программы ставится в соответствие строка таблицы страниц с тем же номером (нулевой странице соответствует нулевая строка и т.п.). Внутри каждой записи таблицы страниц находится номер физической страницы, в которой размещается соответствующая виртуальная страница программы. Соответственно, аппарат виртуальной страничной памяти позволяет **автоматически** (т.е. **аппаратно)** преобразовывать номер виртуальной страницы в номер физической страницы посредством обращения к таблице страниц (Рис. 57). Программных действий при таком подходе требуется минимально: при выборе операционной системой очередного процесса, который ставится на обработку на центральный процессор, она должна лишь корректно заполнить аппаратную таблицу страниц процессора для данного процесса.

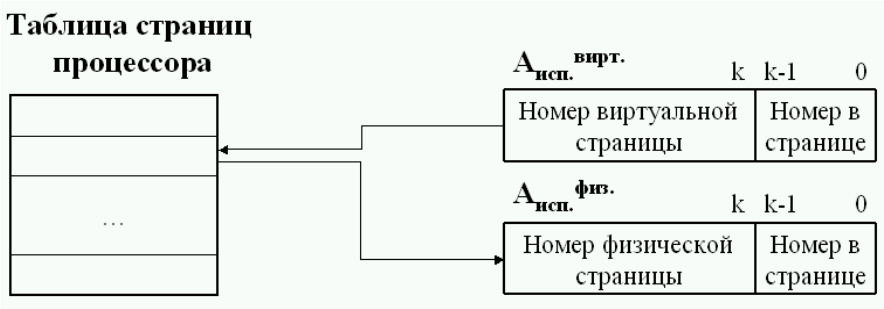


Рис. 57. Страничная организация памяти. Преобразование виртуального адреса в физический.

Типовая *схема преобразования адресов* достаточно проста (Рис. 58). Пусть в таблице страниц имеется N строк. Это означает, что в компьютере дозволено использовать N страниц. Содержимое каждой i-ой строки таблицы — α_i – определяется операционной системой в момент запуска процесса. Пусть в нашем модельном примере если $\alpha_i > 0$, то это номер физической страницы, которая соответствует i-ой виртуальной странице. Заметим, что нулевая физическая страница обычно принадлежит ОС и не может использоваться в памяти пользовательских процессов. Если $\alpha_i = 0$, то это означает, что данной страницы у программы нет, и если в ходе обработки процесса процессор обращается к строке таблицы страниц с отрицательным содержимым, происходит прерывание по защите памяти, и управление получает ОС. *Причин возникновения прерывания в данном случае может две*. Во-первых, может оказаться, что действительно i-ой виртуальной страницы у программы нет, что свидетельствует об ошибке в программе. Во-вторых, может оказаться, что соответствующей страницы нет в оперативной памяти – она расположена на внешнем

запоминающем устройстве (ВЗУ), т.е. данная i-ая виртуальная страница легальна, но в данный момент ее нет в ОЗУ. Так или иначе, операционная система анализирует причину возникновения прерывания и для последнего случая осуществляет подкачку из ВЗУ в ОЗУ требуемой страницы. Таким образом, замечательным свойством страничной организации памяти является то, что при такой организации не обязательно загружать всю программу в оперативную память. При смене задачи система должна сначала скопировать содержимое таблицы страниц в свою программную таблицу, а после этого восстановить содержимое таблицы страниц той программы, которая будет исполняться следующей.

Отметим, что страничная организация памяти решает все вышеперечисленные проблемы, связанные с выполнением программ. Здесь имеется *механизм защиты памяти* (в этой схеме процесс никогда не сможет обратиться к «чужой» странице), но также имеется возможность разделять некоторые страницы между несколькими процессами (в этом случае операционная система каждому из процессов допишет в таблицу страниц номер общей страницы). Данная схема обладает достаточной *производительностью*, поскольку ее функционирование основано на использовании регистров. Основным достоинством данного подхода является решение *проблемы фрагментации*, поскольку все программы оперируют в терминах страниц (каждая из которых имеет фиксированный размер), и мы можем размещать программу по страницам в произвольном порядке. Помимо этого решается еще и *проблема перемещаемости программ по ОЗУ*, причем даже в рамках одной программы соответствие между виртуальными и физическими страницами может оказаться произвольным: нулевая виртуальная страница программы может располагаться в одной физической странице, первая виртуальная — в другой (совершенно не связанной с первой) физической странице, и т.д. Еще одним важным достоинством страничной организации памяти заключается в том, что *нет необходимости держать в оперативной памяти весь исполняемый процесс*. Это свойство позволяет существенно повысить эффективность использования ОП. Реально в ОЗУ может находиться лишь незначительное число страниц, в которых расположены команды и требуемые для текущих вычислений операнды, а все оставшиеся страницы могут находиться на внешней памяти — в областях подкачки. Следствием только что сказанного является то, что размеры физической и виртуальной памяти могут быть произвольными. Может оказаться, что физической памяти в компьютере больше, чем размеры адресного пространства виртуальной памяти, а может оказаться и наоборот: физической памяти существенно меньше виртуальной. Но во всех этих случаях система окажется работоспособной.

Рис. 58. Модельный пример организации страничной виртуальной памяти. Схема преобразования адресов.

Но данный подход имеет и свои *недостатки*. Во-первых, это страничная фрагментация, или ***внутренняя*** (скрытая) ***фрагментация***: если в странице используется хотя бы один байт, то вся страница отводится процессу и считается занятой (т.е., решив вопрос с т.н. ***внешней фрагментацией***, мы в данном случае не используем память, размером со страницу минус один байт). Во-вторых, описанная выше модель является *вырожденной*: если таблица страниц целиком располагается на регистровой памяти, то в силу дороговизны последней размеры подобной таблицы должны быть слишком малы (а следовательно, будет невелико количество физических страниц). Реальные современные системы имеют более сложную логическую организацию, и речь о ней пойдет ниже. Кроме того, при смене процессов таблицу страниц сначала обязательно надо сохранить, а потом обновить – дополнительные накладные расходы.

Напоследок заметим, что данное нами определение аппарата виртуальной памяти расходится с определениями некоторых других источников. Повторим, что мы рассматриваем механизм виртуальной памяти как механизм преобразования виртуального адресного пространства в физическое. Во многих изданиях, посвященных рассмотрению операционных систем, виртуальной памятью считается то, что позволяет часть программы размещать на внешних устройствах, т.е. считают механизм виртуальной памяти как средство увеличения объема физической памяти. Мы считаем такое определение некорректным. Если рассматривать, например, виртуальную память как механизм увеличения объема памяти, то возникает вопрос: в случае большого объема физической памяти разве виртуальная память отсутствует? Соответственно, возникают проблемы с подобным определением.

И еще одно важное замечание. В компьютере имеется физическое адресное пространство и виртуальное. Физическое пространство — это та оперативная память, которая физически может быть подключена к компьютеру, а виртуальное адресное пространство — это то пространство, которое доступно программе. И возникает вопрос, что и каким способом задает максимальные размеры этих адресных пространств. На размер виртуального адресного пространства влияет разрядность исполнительных адресов,

получаемых в ходе обработки программы на центральном процессоре. Размер физического пространства определяется характеристикой компьютера: зависит от того, сколько физически можно подключить памяти к машине, и какова разрядность внутренней аппаратной шины. Но и то, и другое являются аппаратными характеристиками компьютера.

1.2.7 Многомашинные, многопроцессорные ассоциации

В настоящее время одиночный компьютер можно сравнить с телефонным аппаратом без телефонной сети. Т.е., говоря об ЭВМ, мы подразумеваем машину в некотором окружении и взаимодействии с другими машинами. В зависимости от степени интегрированности машин в рамках одного комплекса различают многопроцессорные ассоциации, где степень связанности машин довольно велика, и многомашинные ассоциации, в которых наблюдаются слабые связи между машинами (в некоторых случаях говорят о сетях ЭВМ).

Начиная данную тему, мы, следуя традиционному научному подходу, сначала рассмотрим классификацию — это позволит выявить среди большого разнообразия машинных ассоциаций группы с идентичными свойствами, которые помогут нам познакомиться с наиболее общими подходами, абстрагируясь от деталей реализации.

Для классификации существуют множество методов, проводящих деление по различным характеристикам (например, по производительности). Одна из наиболее простых классических классификаций многопроцессорных систем — это **классификация по Флинну** (M.Flynn), основанная на анализе некоторых характеристик потоков информации в машине. Основная концепция этой классификации – перебор всевозможных характеристик потока команд (инструкций) и потока данных. Обработка каждого из этих потоков может быть одиночная либо множественная.

В контексте машины можно выделить два потока информации: **поток управления** (для передачи управляющих воздействий на конкретное устройство) и **поток данных** (циркулирующий между оперативной памятью и внешними устройствами). Возможны некоторые оптимизации данных потоков. В потоке команд — это переход от команд низкого уровня к высокоуровневым (когда ЦП вместо работы с микрокомандами начинает вырабатывать высокоуровневые команды, которые передаются «умному» устройству управления, непосредственно реализующему данные команды); в потоке данных — это исключение участия ЦП в обменах между внешними устройствами и оперативной памятью.

Флинн предлагает рассматривать компьютер с позиции 2 потоков:

- поток команд: выбор одной или группы команд
- поток данных, операндов: с выполнением каждой команды выбирается либо единичная, либо множественная порция данных. В результате получаем *четыре класса архитектур*:

– **ОКОД** (одиночный поток команд, одиночный поток данных, или **SISD** — single instruction, single data stream) — это традиционные компьютеры (близкие машине фон Неймана) с единственным ЦП. Они имеют одно устройство управления, которое последовательно выбирает команды, и каждая команда обрабатывает единичную порцию данных.

– **ОКМД** (одиночный поток команд, множественный поток данных, или **SIMD** — single instruction, multiple data stream) — например, векторные компьютеры, способные оперировать векторами данных, матричная обработка данных. Обычно для этих целей в данных машинах существуют векторные регистры, а также обычно имеются векторные операции, предполагающие векторную обработку. В этой архитектуре имеется одно УУ, которое последовательно выбирает команды, а обработка данных ведётся агрегировано. Заметим, что принадлежность конкретной системы к конкретному классу относительно условна.

– **МКОД** (множественный поток команд, одиночный поток данных, или **MISD** — multiple instruction, single data stream) — имеется смесь команд, которая оперирует над одними и теми же данными. Этот класс архитектур является спорным. Существуют различные точки зрения о существовании каких-либо систем данного класса, и если таковые имеются, то какие именно. В некотором смысле сюда можно отнести специализированные системы обработки видео- и аудиоинформации (DSP-процессоры), а также конвейерные системы.
– **МКМД** (множественный поток команд, множественный поток данных, или **MIMD** — multiple instruction, multiple data stream) — это системы, которые содержат не менее двух устройств управления (это может быть один сложный процессор с множеством устройств управления). Множество процессоров одновременно выполняют различные последовательности команд над своими данными. Это наиболее распространённая категория архитектур. На сегодняшний день данная категория во многом определяет свойства и характеристики многопроцессорных и параллельных вычислительных систем.

Среди систем МКМД можно выделить два подкласса: **системы с общей оперативной памятью** и **системы с распределенной памятью** (Рис. 59). Для систем первого типа характерно то, что любой процессор имеет **непосредственный** доступ к любой ячейке этой общей оперативной памяти. Слово «непосредственно» означает, что любой адрес может появляться в произвольной команде в любом из устройств управления. Системы с распределенной памятью представляют собою обычно объединение компьютерных узлов. Под узлом понимается самостоятельный процессор со своей локальной оперативной памятью. В данных системах любой процессор **не может** произвольно обращаться к памяти другого процессора. Указанные системы иллюстрируют противоположные подходы – на практике обычно встречаются промежуточные решения.

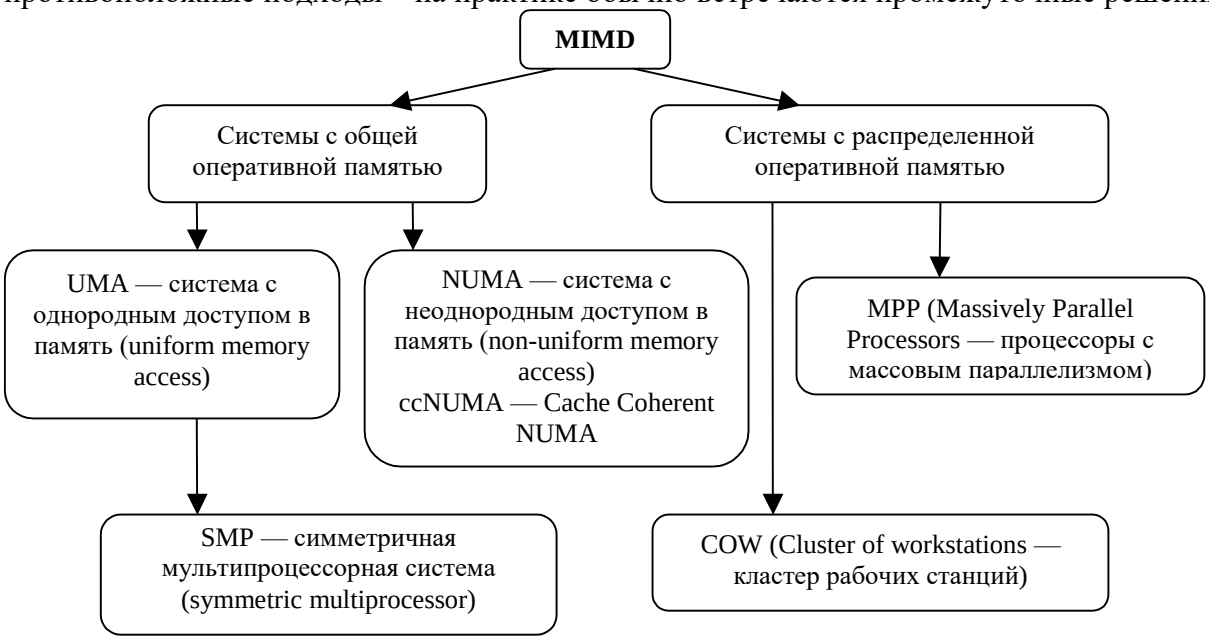


Рис. 59. Классификация МКМД.

Рассмотрение ***систем с общей оперативной памятью*** начнем с UMA. **UMA** (uniform memory access) — система с однородным доступом к памяти. В данной модели произвольный процессорный элемент имеет доступ к произвольной точке оперативной памяти (доступ с одинаковым временем). То есть характеристики доступа любого процессорного элемента в любую точку ОЗУ не зависят от конкретного элемента и адреса

(Все процессоры равноценны относительно доступа к памяти). Подвидом UMA-систем является модель **SMP** (symmetric multiprocessor — симметричная мультипроцессорная система). В этой модели (Рис. 60) к общей системной шине, или магистрали, подсоединяются несколько процессоров и блок общей оперативной памяти. У данного решения можно отметить следующие *недостатки*. Во-первых, это централизованная система, и общая шина в ней является «узким горлом», поэтому данная модель накладывает существенные ограничения на количество подключаемых процессорных элементов (обычно 2, 4, 8, вплоть до 32). Во-вторых, возникают дополнительные проблемы синхронизации КЭШей первого уровня каждого процессора. Решений тут как минимум два: либо не использовать КЭШ, либо реализовать КЭШ-память со слежением. В последнем случае каждый КЭШ слушает шину и реагирует на ситуацию в системе. Различные ситуации поведения кэш-памяти с отслеживанием при чтении/записи приведены в следующей таблице:

Операции	Локальный кэш	Кэш других процессоров
Промах при чтении	Запись из памяти в кэш	Ничего
Попадание при чтении	Использование кэша	Ничего (операция «не видна»)
Промах при записи	Запись в память	Соответствующая запись в кэше удаляется
Попадание при записи	Запись в память и кэш	Соответствующая запись в кэше удаляется

При промахе при чтении в локальный КЭШ происходит запись соответствующего блока; эта операция проходит через общую магистраль, которую слушают другие КЭШи; никаких действий КЭШи других процессоров не производят. При попадании при чтении мы используем КЭШ, и эта операция не даёт никакой нагрузки на общую шину. При промахе при записи осуществляется запись в память; все другие КЭШи информацию о записи слушают и проверяют наличие соответствующего блока у себя – если блок есть, то они удаляют его из своих КЭШей. То есть при промахе по записи производится только обновление памяти, т.к. реализуется стратегия, ориентированная на преимущественное чтение. При попадании при записи ситуация аналогичная, только запись осуществляется ещё и в локальный КЭШ.

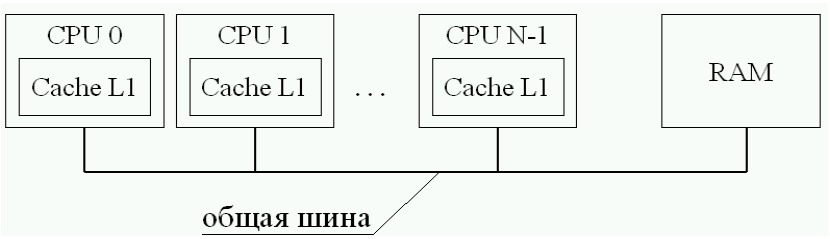


Рис. 60. Схема SMP-системы.

Иной подход к реализации систем с общей оперативной памятью предлагает архитектура **NUMA** (non-uniform memory access — система с неоднородным доступом к памяти). Степень параллелизма в NUMA-системах выше, чем в SMP. Для NUMA-систем (Рис. 61) характерны следующие свойства:

- процессорные элементы работают на общем адресном пространстве;
- характеристики доступа процессора к области оперативной памяти зависят от того, к каким областям идет обращение (к локальной или нелокальной памяти).

Контроллер памяти позволяет осуществлять взаимодействие между вычислительными узлами. Доступ процессора к своей ОП осуществляется через свой

контроллер, к чужой – через два контроллера. При обращении не к своей памяти контроллер выбрасывает запрос на общую шину, целевой контроллер его принимает и возвращает результат.

В NUMA-системах остаётся проблема синхронизации КЭШа. Существует несколько способов её решения:

- использовать процессоры без КЭШа (использовать только Cache L2);
- использовать модель ccNUMA (Cache coherent NUMA) – это NUMA-система с когерентными КЭШами.

CcNUMA-системы позволяют отслеживать соответствие локальных КЭШей друг другу и состояние всей системы в целом. CcNUMA-системы подключают несколько сотен процессоров, но остаются ограничения, связанные с централизацией – использованием системной шины, а также возникают ограничения, связанные с cc-архитектурой: появляются системные потоки служебной информации, что ведет к дополнительным накладным расходам – загрузке общей шины служебной информацией.

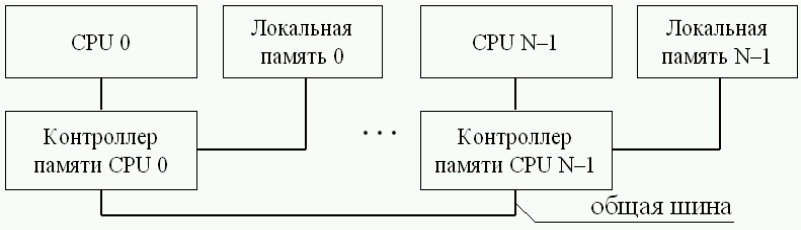


Рис. 61. Схема NUMA-системы

Теперь рассмотрим ***системы с распределенной оперативной памятью***. Эти системы представляются как объединение вычислительных узлов, каждый из которых состоит из процессора и ОЗУ, непосредственный доступ к которой имеет только «свой» процессорный элемент. Данный класс систем является наиболее перспективным с точки зрения их массового распространения и использования. Среди них можно выделить два основных класса: **MPP** (Massively Parallel Processors — процессоры с массовым параллелизмом) и **COW** (Cluster of Workstations — кластеры рабочих станций).

Преимуществом MPP-систем является высокая эффективность при решении определённого класса задач. Однако MPP обычно являются дорогостоящими узкоспециализированными многопроцессорными системами, поэтому не находят массового применения. Системы данного класса имеют разнообразные топологии архитектур (Рис. 62): это могут быть макроконвейерные архитектуры, кубы и гиперкубы.

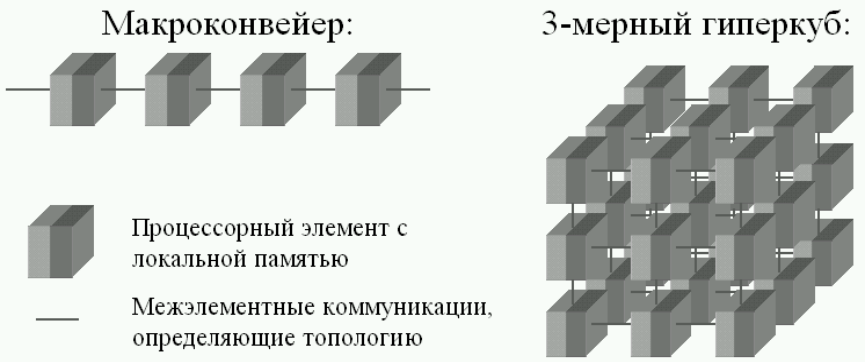


Рис. 62. Примеры топологий MPP-систем.

Что касается COW, то это многомашинные системы, состоящие из множества узлов, каждый из которых может быть обыкновенным компьютером. В качестве минимального узла может выступать процессор со своей локальной оперативной памятью и аппаратурой сопряжения с другими вычислительными узлами. Для сопряжения вычислительных узлов в кластере используются специализированные компьютерные сети.

Кластеры могут использоваться для достижения следующих основных ***целей***:

– построение кластера, обеспечивающего ***надежность***. Создаются программно-аппаратные комплексы для обеспечения устойчивого функционирования системы. Данный тип кластеров строится для решения конкретной прикладной задачи (например, сервер базы данных авиабилетов), при этом выход из строя некоторых узлов не означает отказ всей системы: система продолжает функционировать, пусть и со сниженной производительностью;
– построение кластера как ***высокопроизводительной вычислительной системы***, т.е. вычислительного кластера (критерием эффективности выступает скорость обработки информации). Это многопроцессорная система, состоящая из вычислительных узлов. Каждый вычислительный узел – это компьютер традиционной архитектуры. Все узлы связаны друг с другом через специальные высокоскоростные сети, и вся система в целом используется для достижения максимальной производительности.

Для построения вычислительных кластеров зачастую используют Unix-системы, а для кластеров надежности — Windows-системы. На сегодняшний день ***кластеры*** — это специализированные системы с соответствующей архитектурой (например, alpha-системы), при этом речь идет о супервычислительных кластерах, включающих в себя сотни – тысячи узлов. *Преимуществом* кластерных систем является высокая эффективность при решении широкого круга задач (за приемлемую цену); кластерные системы используют унифицированные средства программирования (типа MPI). Основными *проблемами* кластерных систем являются отвод тепла и коммуникация (если будет использоваться единственная магистраль, то она «захлебнется» от потоков передаваемой информации, а большинство узлов будут простаивать).

Напоследок хочется отметить, что в рейтингах наиболее высокоскоростных вычислительных систем (Top100, Top500 и пр.) верхние строчки занимают именно кластерные системы.

Теперь рассмотрим проблемы сетевого взаимодействия.

1.2.8 Терминальные комплексы (ТК)

Современные многопроцессорные системы строятся как специализированные компьютерные сети. Основой современных компьютерных сетей было появление терминальных комплексов. ***Терминальный комплекс*** — это многомашинная ассоциация, предназначенная для организации массового доступа удаленных и локальных пользователей к ресурсам некоторой вычислительной системы.

Суть ТК заключается в следующем. Пусть имеется некая вычислительная система. Ставится задача организовать массовый доступ к ее ресурсам. Под ресурсами в данном случае могут пониматься, например, высокая производительность рассматриваемой системы или же информационный ресурс (информационное наполнение, интересное для массового пользователя, такое как электронная библиотека).

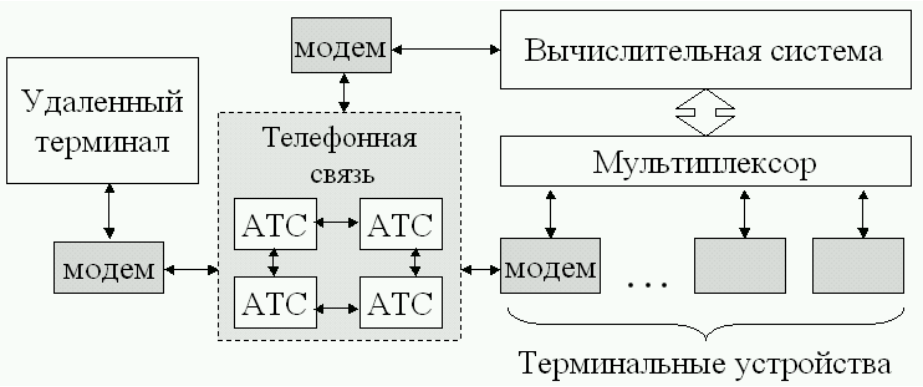


Рис. 63. Структура терминального комплекса.

Терминальные комплексы предполагают наличие в своем составе следующих компонентов (Рис. 63):

- основная вычислительная система (к которой обеспечивается доступ);
- локальные мультиплексоры;
- локальные терминалы;
- модемы;
- удаленные терминалы;
- удаленные мультиплексоры.

Считается, что пользователь может использовать терминальное устройство (алфавитно-цифровой терминал, т.е. дисплей с клавиатурой), печатающее устройство, внешнее запоминающее устройство (например, магнитная лента). Изначально все терминальные устройства располагались на незначительном расстоянии от вычислительного комплекса.

Терминальные устройства можно разделить на **локальные** и **удаленные**. Локальные терминальные устройства имеют либо непосредственное подключение к ВС по локальному каналу связи, либо подключение через мультиплексор (устройство, позволяющее через один высокоскоростной канал связи подключить *N* менее скоростных каналов). **Удаленные** терминальные устройства подключаются к ВС через коммуникационную среду. Изначально этой средой являлись обычные телефонные сети, которые исторически основывались на аналоговом способе передачи информации; компьютерные сети основаны на цифровом (дискретном) способе передачи данных. Для передачи цифровой информации через аналоговые сети необходимы аналогово-цифровой и цифро-аналоговый преобразователи (модем — **мо**дулятор-**дем**одулятор). Таким образом, для удаленного соединения необходимы, как минимум, два модема. Удалённые терминалы могут также мультиплексироваться, и это мультиплексирование может быть многоуровневым.

Линии связи/каналы. Существуют три критерия, по которым можно классифицировать каналы.

Во-первых, с точки зрения организации канала все каналы можно поделить на **коммутируемые** и **выделенные**.

Коммутируемый канал — это линия, выделяемая на весь сеанс работы терминального устройства. Физически коммутируемая линия может иметь различные топологии. Примером коммутируемого канала может служить телефонный разговор.

Выделенный канал обеспечивает связь терминального устройства с ВС на постоянной основе. И здесь существуют два подхода: с одной стороны, можно протянуть между терминальным устройством и ВС физический провод (это дорого, но это наилучший способ обеспечить выделенный канал), а с другой стороны, можно взять один из коммутируемых маршрутов телефонной сети (этот подход несколько дешевле предыдущего, но он особенно плох тем, что уменьшает количество коммутируемых маршрутов в телефонной сети, что, в конечном счете, негативно сказывается на качестве ее работы «по прямому назначению» — на обеспечении телефонной (голосовой) связи). Преимущества выделенного канала: постоянная готовность (отсутствие отказа на соединение), детерминированное качество соединения.

Во-вторых, все каналы можно поделить *по количеству участников общения*:

- ***канал точка-точка*** —одно устройство общается с одним устройством (подключение к удалённому терминалу без мультиплексирования);
- ***многоточечный канал*** — общение многих устройств (например, при мультиплексировании): подключение терминала осуществляется через локальный мультиплексор.

И, наконец, каналы можно делить с точки зрения *направления движения информации* в канале:

– ***симплексный*** — канал позволяет осуществлять передачу информации только в одном направлении (например, репродуктор громкой связи на вокзале или в организациях; идеальная лекция);
– ***дуплексный*** (двунаправленный) — канал передаёт информацию в двух направлениях (например, телефон);
– ***полудуплексный*** — канал позволяет передавать информацию в двух направлениях, но в каждый момент времени возможна передача информации лишь в одном направлении (например, рация).

1.2.9 Компьютерные сети

Развитие терминальных комплексов положило основу развития компьютерных сетей. И первым шагом к сетям стала замена терминальных устройств компьютерами.

Компьютерная сеть — это объединение компьютеров (или вычислительных систем), взаимодействующих через коммуникационную среду (Рис. 64).

Коммуникационная среда — каналы и средства передачи данных.

Рис. 64. Компьютерная сеть.

Можно выделить следующие *свойства компьютерных сетей*. Во-первых, логически *сеть может состоять из значительного числа связанных между собой автономных компьютеров*, каждый из которых обеспечивает решение определённых задач. Два компьютера называются связанными между собой, если они могут обмениваться информацией. Требование автономности используется, чтобы исключить из рассмотрения компьютерные системы, в которых один компьютер может принудительно запустить, остановить другой компьютер или управлять его работой.

Во-вторых, компьютерная сеть предполагает *возможность распределенной обработки информации*, когда пользователь компьютерной сети может получать в качестве результата данные, обработанные за счет интеграции этапов обработки данных на разных компьютерах сети.

Следующее свойство — *расширяемость сети*. То есть компьютерная сеть должна обеспечивать возможность развития по протяженности, по расширению пропускной способности каналов, по составу и производительности компонентов сети.

И, наконец, *возможность применения симметричных интерфейсов обмена информацией между компьютерами сети*. Эти интерфейсы позволяют произвольным способом распределять функции сети между компьютерами.

Будем говорить, что логически компьютеры, составляющие сеть, можно подразделить на **абонентские машины** (или основные компьютеры — хосты) и **коммуникационные** (или вспомогательные) компьютеры (шлюзы, маршрутизаторы и пр.) (Рис. 65). Последние выполняют фиксированные функции по обеспечению функционирования сети (выбор маршрута, передача информации и т.д.). Это деление логическое – на практике может оказаться, что одна машина исполняет роль, как абонентской машины, так и коммуникационной.

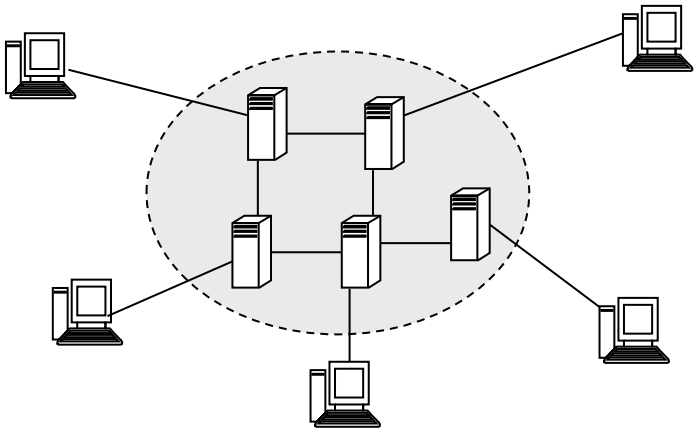


Рис. 65. Составляющие компьютерной сети.

С точки зрения взаимодействия компьютеров в сети, традиционно для функционирования компьютерных сетей используются 3 модели организации каналов, или 3 модели сетей, — это **сети коммутации каналов**, **сети коммутации сообщений** и **сети коммутации пакетов**. Сразу отметим, что большинство современных сетей являются комбинациями этих основных моделей сетей.

Будем говорить, что два абонента сети осуществляют взаимодействие сеансами связи (каждый *сеанс связи* является последовательностью обменов сообщениями между абонентами сети). Под ***сообщением*** будем понимать логически целостную порцию данных, имеющую произвольный размер. В **сети коммутации каналов** коммутация каналов происходит *на весь сеанс связи*.

К *достоинствам* данных сетей можно отнести следующие свойства:

- канал всегда находится в состоянии готовности;
- требования к коммуникационному оборудованию минимальны: по сути, нет требований к обеспечению буферизации;
- обмены происходят сообщениями, что ведет к уменьшению накладных расходов по передаче информации;
- детерминированная пропускная способность сети.

Среди *недостатков* данной модели можно отметить следующие:

- дороговизна: любой выделенный канал требует больших материальных затрат;
- наличие высокой избыточности в сети: должно быть либо много каналов, чтобы не было коллизий, либо в сети будут коллизии – при этом период ожидания свободного канала недетерминирован;
- неэффективность использования выделенного коммутационного канала: в отдельно взятом сеансе может быть низкая интенсивность обмена сообщениями;
- при сбоях или отказах повторение переданной информации является сложной задачей.

Сети коммутации сообщений — это сети, которые оперируют термином «передача сообщения», а не «сеанс связи». Абонент-отправитель передаёт сообщение в сеть по первому свободному каналу, который у этого абонента есть. Выделение канала для передачи каждого сообщения происходит поэтапно (по мере продвижения сообщения) от одного узла к другому. На каждом узле на пути следования принимается решение, свободен ли канал к следующему узлу. Если свободен, то сообщение передается далее, иначе происходит ожидание освобождения канала. К *достоинствам* и *недостаткам* данной модели можно отнести:

- отсутствие выделенного канала и, соответственно, занятости ресурса коммутируемого канала на недетерминированный промежуток времени, т.е. устранена деградация системы, возникающая при организации сетей коммутации каналов;

– в связи с тем, что сообщения могут быть произвольного размера, возникает необходимость наличия в коммутационных узлах средств буферизации (в общем случае неизвестно, какой мощности, поскольку сообщение имеет произвольную длину) – таким образом, данная сеть имеет недетерминированные характеристики (скорость передачи по сети и т.д.);
– обеспечение буферизации требует дорогостоящего коммутационного оборудования и ПО;
– необходимость повторения сообщения в случае сбоя при передаче, что само по себе является сложной задачей, хотя менее трудоемкой, чем для сетей коммутации каналов.

Модель **сети коммутации пакетов** строится на предположении использования в сети ненадежных средств связи. Функционирование данной сети состоит в следующем: каждое сообщение разбивается на блоки фиксированного размера, которые называются ***пакетами***. То есть передача сообщения – это передача цепочки пакетов (движение пакетов по сети – аналогично предыдущей модели, но существенное различие между этими моделями заключается в детерминированности размера пакета). Соответственно, на стороне отправителя происходит разбиение сообщения на пакеты, а на стороне получателя — сборка. Любой пакет помимо непосредственно самого сообщения (или его части) имеет служебную информацию (которая обычно представлена в заголовке пакета), обеспечивающую внутреннюю целостность пакета (контрольная сумма пакета и пр.); адресную составляющую (данные об отправителе и адресате), а также информацию для сборки сообщения из пакетов.

При передаче пакета используется следующая стратегия: любой узел, получив пакет, пытается сразу от него избавиться. Поскольку любая сеть имеет фиксированную топологию, а также фиксированные количество и расположение абонентов, то возможно просчитать ее характеристики и предъявить требования к коммуникационным узлам. Данная модель допускает буферизацию в узлах передачи: пакет, придя на узел, может быть послан несколько позже, если все необходимые выходные каналы заняты. Но период занятости канала при известной стратегии обработки буфера предопределен, поэтому можно оценить *предельный размер буфера*, а также *предельные периоды ожидания* пакетов при передаче их по сети. Таким образом, если известна стратегия передачи, пропускная способность является детерминированной величиной.

Среди положительных свойств данной системы можно отметить ее *детерминированность* (детерминированность перемещений, детерминированность требований к коммуникационному оборудованию), а также то, что при сбое достаточно заново послать потерянные пакеты, а не все сообщение целиком.

К недостаткам модели можно отнести увеличение трафика из-за того, что по сети перемещается накладная информация, которая прибавляется к каждому пакету при разбиении сообщения на пакеты. Как известно, эта служебная информация занимает 80-90% пропускной способности канала. Еще одной проблемой, связанной с разбиением сообщения на пакеты, является их сборка — это аккумуляция пакетов, а также сама сборка (необходимо обеспечить наличие всех переданных пакетов и их правильный порядок).

1.2.10 Организация сетевого взаимодействия. Эталонная модель ISO/OSI

С появлением компьютерных сетей появилась проблема *организации сетевого взаимодействия* через коммуникационную среду двух и более компьютеров. Распределение функций между этими компьютерами могло быть как централизованным (например, архитектура терминального комплекса), так и децентрализованным (сеть, состоящая из равнозначных по ролям и по функциям компьютеров). В рамках проблемы организации сетевого взаимодействия мы должны решать проблемы не только аппаратного взаимодействия устройств, но сопоставления взаимодействующих программ.

При рассмотрении истории развития компьютерных сетей можно использовать аналогию развития всей вычислительной техники. Изначально компьютер строился как уникальное техническое образование, в котором были уникальные, ориентированные именно на этот компьютер компоненты. Архитектура и структура компьютера была монолитной, т.е. не было чёткой структуризации компьютера. Всё это приводило к ряду проблем. Во-первых, было крайне сложно подключать к компьютеру новые устройства (должна была проводиться очень тяжёлая аппаратная проработка). Во-вторых, появление новых компьютеров практически зачёркивало всё то, что было для старых – отсутствовала программная преемственность (переносимость программ). Все эти проблемы привели к необходимости стандартизации (компьютерных комплектующих, программных интерфейсов). Стандартизация интерфейсов дает возможность взаимозаменяемости компонентов и их работоспособности. Стандартизация позволяет нам, во-первых, децентрализовать производство компьютера (появились независимые производители центральной части, периферийного оборудования, программного обеспечения и т.д.), во-вторых, появилась возможность достаточно простой стыковки адаптации, модернизации техники. Та же самая проблема стандартизации прослеживалась и в программном обеспечении. В первую очередь подверглись стандартизации языки программирования. Описание любого языка нестрогое – 95% в нём совершенно чёткой и однозначной декларации синтаксиса и семантики языка, а 5% чётко не продекларированы, что могло приводить ко многим проблемам. Поэтому необходима стандартизация. Следующий этап стандартизации связан с появлением Unix, что дало стимул к многоуровневой стандартизации интерфейсов ОС (от стандартизации интерфейсов системных вызовов до стандартизации интерфейсов систем обработки команд). Проявился стандарт POSIX (Portable Operating System Interface), который сейчас используется при разработке операционных систем.

Далее речь пойдет об одном аспекте сетевого взаимодействия, который во многом является ключевым, причем важность этого аспекта прослеживается очень давно — с момента появления компьютерных сетей и их массового распространения. Этот аспект связан со стандартизацией, применяемой в вычислительной технике.

На сегодняшний день почти все производственные или технологические процессы, которыми пользуется человек, строятся на достаточно глубокой эшелонированной стандартизации. Стандартизация позволяет современным производствам и организациям производственных процессов быть развиваемыми, ремонтоспособными, обслуживаемыми.

Изначально компьютерные сети развивались на основе терминальных комплексов, которые строились по внутрикорпоративным правилам. Такие сети использовали «свои» корпоративные стандарты на подключение оборудования, на передачу данных, на правила взаимодействия компьютеров и программ в сети.

Развитие сетей определило массовость их использования. Возникла необходимость создания сетей, которые могли бы достаточно прочно расширяться без привлечения существенных переделок, взаимодействовать друг с другом, модернизироваться, в которых могло бы меняться ПО, добавляться новые службы. Всё это приводит к необходимости стандартизации в компьютерных сетях. В связи с этим появился целый спектр моделей организации сетей (т.н. «открытых» сетей), в основе которых используется модель системы открытых интерфейсов (OSI — Open Systems Interconnection), предложенная **Международной организацией по стандартизации** (ISO — International Organization for Standardization). Работа над созданием этой модели велась с середины 70-х гг. по середину 80-х гг. прошлого века. Целью разработки этой модели было построение стандарта, на основе которого можно было создавать компьютерные сети, открытые к расширению и модификации. Стоит отметить, что модель ISO/OSI является в некотором смысле не стандартом, а рекомендацией, которая в полном объёме никогда нигде не была реализована. Однако, подобно принципам фон Неймана, важность этой модели трудно переоценить. Модель ISO/OSI рассматривает сеть и взаимодействие компьютеров в сети в виде семи функциональных уровней (Рис. 66). Данная модель основана на анализе исторического развития реальных компьютерных сетей.

Рис. 66. Модель организации взаимодействия в сети ISO/OSI.

Уровни этой модели располагаются от физической среды передачи данных до прикладного уровня. При этом предполагается, что взаимодействие в сети может осуществляться между одноимёнными (одноранговыми) уровнями. Для осуществления этого взаимодействия используются протоколы.

Протокол — это формальное описание сообщений и правил, по которым сетевые устройства (вычислительные системы) осуществляют обмен информацией. Протокол обеспечивает взаимодействие в сети между различными сетевыми устройствами на одноимённых уровнях. Любой из уровней может содержать произвольное число протоколов, но общаться могут лишь протоколы одного уровня. Под протоколом будут пониматься правила взаимодействия одноименных, или одноранговых, уровней.

Интерфейс — правила взаимодействия вышестоящего уровня с нижестоящим.

Служба или сервис — набор операций, предоставляемых нижестоящим уровнем вышестоящему.

Стек протоколов — перечень разноуровневых (от первого до максимального реализованного) протоколов, реализованных в системе. Стек может быть произвольной

глубины, т.е. в нем, возможно, не будут представлены протоколы некоторых уровней модели ISO/OSI.

При осуществлении взаимодействия информация должна быть сначала передана с текущего на первый уровня на данном сетевом устройстве, затем передана по коммуникационной среде, принята на другом сетевом устройстве, и, наконец, поднята до соответствующего уровня на другом сетевом устройстве. Схема логического взаимодействия сетевых устройств по *i*-ому протоколу приведена на Рис. 67.

Теперь более детально рассмотрим назначение каждого уровня.

Физический уровень. На этом уровне обеспечивается непосредственно передача неструктурированного потока двоичной информации. Для передачи используется конкретная физическая среда (кабель, радиоволны и т.п.). На данном уровне декларируется стандартизация сигналов и соединений.

Канальный уровень (или уровень **передачи данных**). На этом уровне решаются задачи обеспечения передачи данных по физической линии, обеспечения доступности физической линии, обеспечения синхронизации (например, передающего и принимающего узлов), а также задачи по борьбе с ошибками. Канальный уровень манипулирует порциями данных, которые называются ***кадрами***. В кадрах присутствует избыточная информация для фиксации и устранения ошибок. Таким образом, основная задача канального уровня – обеспечение надёжной линии связи. На канальном уровне также может решаться задача внутренней адресации устройств в локальной сети.

Сетевой уровень. На этом уровне решаются задачи взаимодействия сетей: обеспечивается управление операциями сети (в т.ч. адресация абонентов, маршрутизация), а также обеспечивается связь между взаимодействующими сетевыми устройствами. Также на этом уровне происходит управление движением пакетов, и при необходимости поддерживается их буферизация.

Транспортный уровень. На данном уровне обеспечивается корректная транспортировка данных и взаимодействие между программой-отправителем и программой-получателем данных, т.е. обеспечивается программное взаимодействие (а не взаимодействие устройств). На этом же уровне принимается решение о выборе типа транспортных услуг (транспортировка данных с установлением виртуального канала или же без оного). В случае установления виртуального канала осуществляется контроль за фактом доставки и обработка ошибок (при этом взаимодействие программы-отправителя и программы-получателя обеспечивается в терминах сообщений). Если же виртуальный канал не устанавливается, то уровень не несет ответственности за доставку пакетов. На транспортном уровне *может* обеспечиваться выявление и исправление ошибок при передаче.

Сеансовый уровень. Этот уровень обеспечивает управление сеансами связи. На этом уровне решаются задачи определения активной стороны, подтверждения полномочий и паролей, а также решаются задачи организации меток, или контрольных точек по сеансу, которые отражают состояние сеанса связи и позволяют в случае возникновения сбоя восстанавливать сеанс с последней контрольной точки (т.е. повторять передачу не с начала, а с последней установленной контрольной точки).

Уровень представления данных обеспечивает унификацию используемых в сети кодировок и форматов передаваемых данных.

Прикладной уровень (уровень прикладных программ). На этом уровне формализуются правила взаимодействия с прикладными системами (например, с веб-браузером). Ради этого уровня выстраивается вся структура организации сетевого взаимодействия.

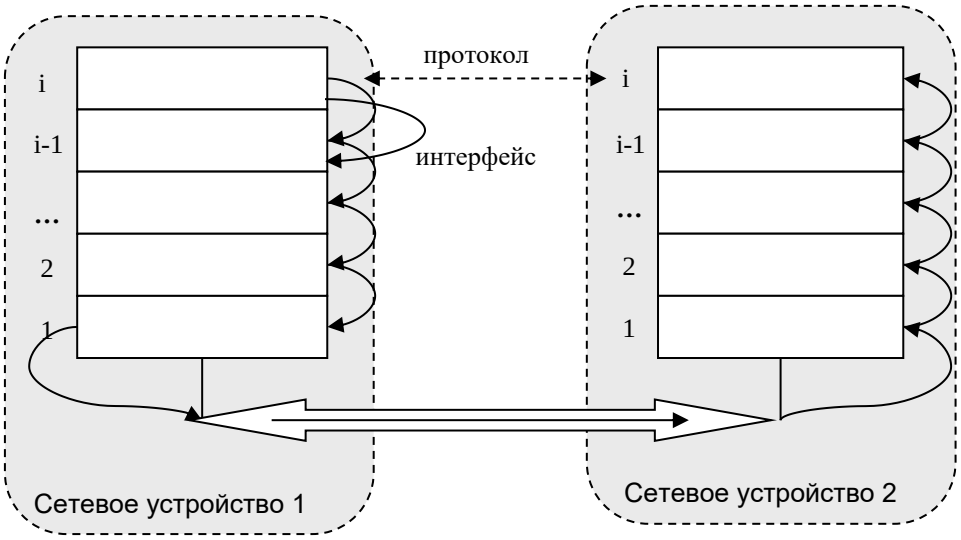


Рис. 67. Логическое взаимодействие сетевых устройств по i-ому протоколу.

1.2.11 Семейство протоколов TCP/IP. Соответствие модели ISO/OSI

Рассмотрим еще одну модель организации сетевого взаимодействия — семейство протоколов TCP/IP (Рис. 68). Это классическая четырехуровневая модель организации сетевого взаимодействия. Протоколы семейства TCP/IP основаны на сети коммутации пакетов. Изначально данные протоколы были разработаны как стандарт военных протоколов в агентстве перспективных разработок DARPA министерства обороны США. Цель этой разработки – создание устойчивой децентрализованной сети, которая могла бы функционировать в коммуникационной среде, имеющей недетерминированную надёжность и производительность. В итоге агентство DARPA разработало сеть ARPANET, которая в своем развитии легла в основу современной сети Internet (поскольку это семейство протоколов было интегрировано в ОС BSD Unix).

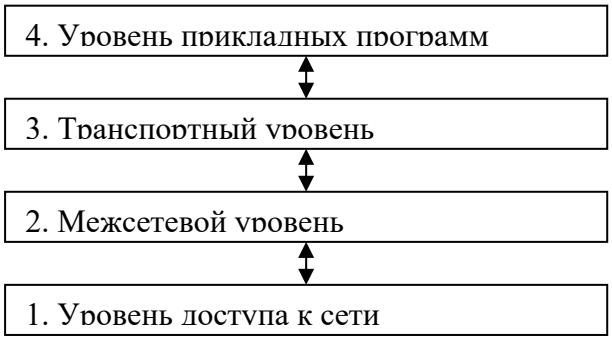


Рис. 68. Семейство протоколов TCP/IP.

Проведём сопоставление моделей TCP/IP и ISO/OSI.

Уровень доступа к сети. Этот уровень соответствует физическому и канальному уровням модели ISO/OSI. Уровень доступа к сети специфицирует доступ к физической сети. На этом уровне решаются проблемы сетевого адаптера, драйвера сетевого адаптера и проблемы среды передачи данных.

Межсетевой уровень (или **internet-уровень**). В некотором смысле ему соответствует сетевой уровень модели ISO/OSI. Т.е. на этом уровне решаются проблемы адресации и маршрутизации по сети. В отличие от сетевого уровня модели OSI, межсетевой уровень модели TCP/IP не устанавливает соединений с другими машинами.

Транспортный уровень. Этому уровню соответствуют сеансовый и транспортный уровни модели ISO/OSI. Транспортный уровень модели TCP/IP обеспечивает доставку данных от компьютера к компьютеру, обеспечивает средства для поддержки логических соединений между прикладными программами. В отличие от транспортного уровня модели OSI, в функции транспортного уровня TCP/IP не всегда входят контроль за ошибками и их коррекция. На этом уровне имеется возможность использования протоколов передачи, которые устанавливают виртуальное соединение или не устанавливают его.

Уровень прикладных программ. Этот уровень разрешает проблемы уровня представления и уровня прикладных программ модели ISO/OSI. Уровень прикладных программ модели TCP/IP состоит из прикладных программ и процессов, использующих сеть и доступных пользователю. В отличие от модели OSI, прикладные программы сами стандартизуют представление данных.

Эти уровни модели TCP/IP являются пакетными: на каждом уровне система оперирует порциями данных, обладающими характеристиками соответствующего уровня (Рис. 69). Двигаясь от верхнего уровня модели к нижнему, содержательная информация при необходимости дробится на пакеты фиксированного размера, и к каждому из них добавляется заголовочная информация. При этом пока содержательная информация доходит до уровня доступа к сети, объём соответствующей ей служебной информации становится достаточно большим.

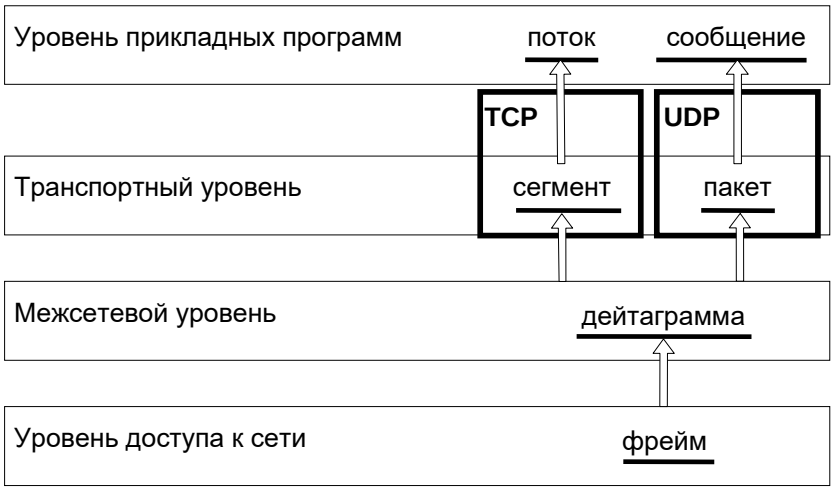


Рис. 69. Взаимодействие между уровнями протоколов TCP/IP.

Итак, основные свойства протоколов семейства TCP/IP следующие:

- Устойчивая работа в недетерминированных условиях линий связи
- Открытость (доступность для использования) стандартов протоколов
- Независимость от аппаратного обеспечения сети передачи данных (за счёт наличия уровня доступа к сети)
- Унифицированная модель именования сетевых устройств
- Стандартизованные протоколы прикладных программ

Теперь остановимся на каждом из уровней модели TCP/IP более подробно.

Протоколы **уровня доступа к сети** обеспечивают систему средств для передачи данных другим устройствам в сети. В качестве примера можно привести семейство протоколов **Ethernet**, являющееся разработкой исследовательского центра компании Xerox (1976 г.). Протоколы Ethernet предполагают наличие широковещательной сети, использующей единую магистраль (общую шину). Для сетевых устройств, подключаемых к этой магистрали, обеспечивается режим, называемый *множественный доступ с контролем несущей и обнаружением конфликтов* (Carrier Sense Multiple Access with Collision Detection — CSMA/CD). Эти сетевые устройства работают в следующем режиме.

Так как сеть ***широковещательная***, все сообщения, которые перемещаются по общей шине, видны всем адаптерам сетевых устройств, каждое из которых имеет уникальный физический адрес (т.н. mac-адрес). Каждый адаптер слушает шину и забирает те сообщения, которые адресованы соответствующему сетевому устройству. Термин ***множественный децентрализованный доступ*** означает, что в любой момент любое из устройств может попытаться выдать сообщение на общую шину. При этом возможно возникновение конфликтных ситуаций. Термин ***контроль несущей*** означает, что каждый абонент, «слушая» сеть, распознает, свободна она или занята. Как только сеть становится свободной, устройство может «закидывать» очередную порцию данных. При этом устройство «слушает» как свою передачу, так и передачи других абонентов. «Бросая» сообщение в сеть, устройство способно распознать искажения, которые означают, что какое-то еще устройство также пытается послать данные в сеть. В этом случае обычно реализуется следующая стратегия: оба абонента прекращают вещание и берут тайм-аут на некоторый случайный промежуток времени (чтобы минимизировать повторные коллизии), а затем повторяют свои попытки. Данная сеть обладает типичными недостатками широковещательной сети: при интенсивной работе часто возникает ситуация, когда общая шина занята. Также при интенсивной работе возрастает частота конфликтов, что ведет к снижению производительности системы.

В качестве физической среды передачи данных используются самые разные среды: это может быть «толстый» Ethernet (к толстой медной полосе адаптеры подключаются с помощью «клёпочного» механизма), «тонкий» Ethernet (аналог домашнего телевизионного кабеля), витая пара (скрутка служит для погашения электромагнитных помех), оптоволокно, радиосигнал. Все эти среды передачи данных стандартизируются протоколами Ethernet.

Семейство протоколов Ethernet является широко распространённым, но кроме протоколов этого семейства на уровне доступа к сети есть и другие протоколы.

Межсетевой уровень. ***Протокол IP*** — один из основных протоколов, в честь которого получило название всё семейство TCP/IP. Данный протокол реализует следующие функции:

- формирование дейтаграмм;
- поддержание системы адресации;
- обмен данными между транспортным уровнем и уровнем доступа к сети;
- организация маршрутизации дейтаграмм;
- разбиение и обратная сборка дейтаграмм.

Отметим, что протокол IP – без установления логического соединения, и он не обеспечивает обнаружение и исправление ошибок.

Основная функция этого протокола — поддержание системы *межсетевой адресации* (internet-адресации), позволяющей объединять различные (гетерогенные) сети в единое целое, а также организация маршрутизации. ***IP-адрес*** — это 32-х или 64-х число, которое кодирует информацию о номере конкретной сети и номере сетевого устройства внутри этой сети. Интерпретация адреса происходит по префиксным двоичным разрядам. Рассмотрим структуру 32-разрядных IP адресов. Адреса разбиваются на следующие классы (Рис. 70).

Формат **класса A** позволяет задавать адреса до 126 сетей с 16 млн. хостов в каждой (сетью класса A обладают ведущие межнациональные корпорации), **класса B** — до 16382 сетей с 65536 хостами, и, наконец, **класса C** — 2 млн. сетей с 254 хостами в каждой. Среди IP-адресов классов A, B и C имеются предопределённые (например, если номер сети и номер сетевого устройства равны нулю, то считается, что это IP-адрес текущего сетевого устройства). Формат **класса D** предназначен для многоадресной рассылки. Остальные адреса используются для служебных целей. Отметим, что на сегодняшний момент в мире складывается ситуация, когда 32-битных IP-адресов не хватает, и ведутся разработки по использованию более длинной адресации. Система TCP/IP предполагает возможность

экономного расходования IP-адресов, т.е. имеется возможность выделения IP-адресов по двум стратегиям: долговременная (постоянные IP-адреса) и кратковременная (IP-адреса выделяются на время сеанса).

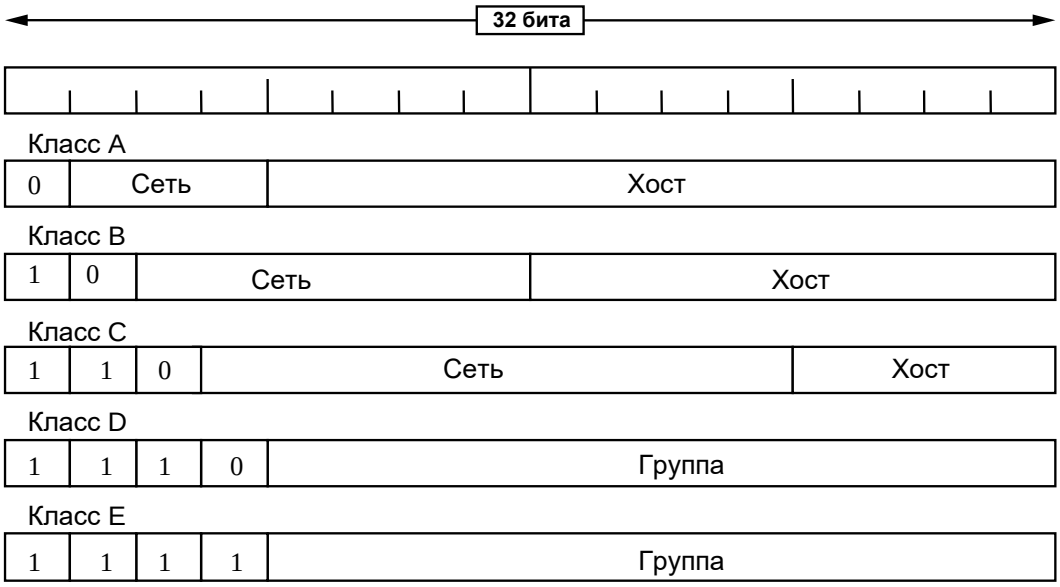


Рис. 70. Система адресации протокола IP.

Как отмечалось выше, каждый из уровней взаимодействует с соседними уровнями (в соответствии с теми или иными протоколами) порциями данных, имеющими специфичные для каждого уровня названия. Так, для межсетевого уровня пакет называется *дейтаграммой*. Одна из основных задач протокола IP – это маршрутизация дейтаграмм.

Протокол IP подразумевает использование некоторых специализированных компьютеров. Это компьютеры, предназначенные для организации физического объединения различных сетей, называются **шлюзами**. То есть шлюз – это устройство, которое единовременно может принадлежать двум и более сетям. В общем случае шлюз имеет два и более сетевых адаптера, на которых функционирует соответствующее число (два или более) стеков протоколов.

Перед межсетевым уровнем стоит задача **маршрутизации** — по имеющему IP-адресу получателя необходимо определить маршрут следования пакета. Эта задача распадается на две подзадачи. Первая подзадача — это проблема организации адресации в локальной сети, в рамках которой происходит взаимодействие. И здесь особых сложностей не возникает, поскольку специфика межсетевого уровня позволяет относительно просто организовать взаимодействие машин в рамках одной локальной сети. Вторая подзадача — это организация адресации между различными сетями. Для решения этой задачи используются шлюзы, которые одновременно принадлежат разным сетям, а также маршрутизаторы, которые решают задачу, через какой шлюз необходимо отправить пакет. Отметим, что стек протоколов TCP/IP позволяет компьютерам совмещать несколько функций: одна и та же машина может быть одновременно и шлюзом, и маршрутизатором, и хостом, причем работающий за ней пользователь может не догадываться об организации локальной сети, в которой он работает.

Рассмотрим **пример** (Рис. 71). Пусть необходимо послать сообщение от машины A1 машине A2. Машина A1 находится в сети A, а машина A2 — в сети C, причем сеть A соединена лишь с сетью B посредством шлюза G1, а сеть C соединена также лишь с сетью B, но посредством шлюза G2. Соответственно, маршрутизатор должен учитывать эти особенности при решении задачи маршрутизации. Обратим ваше внимание, что на компьютерных шлюзах реализовано только два уровня протоколов, поскольку для решения

задачи транспортировки пакетов из одной сети в другую достаточны лишь наличие этих двух уровней.

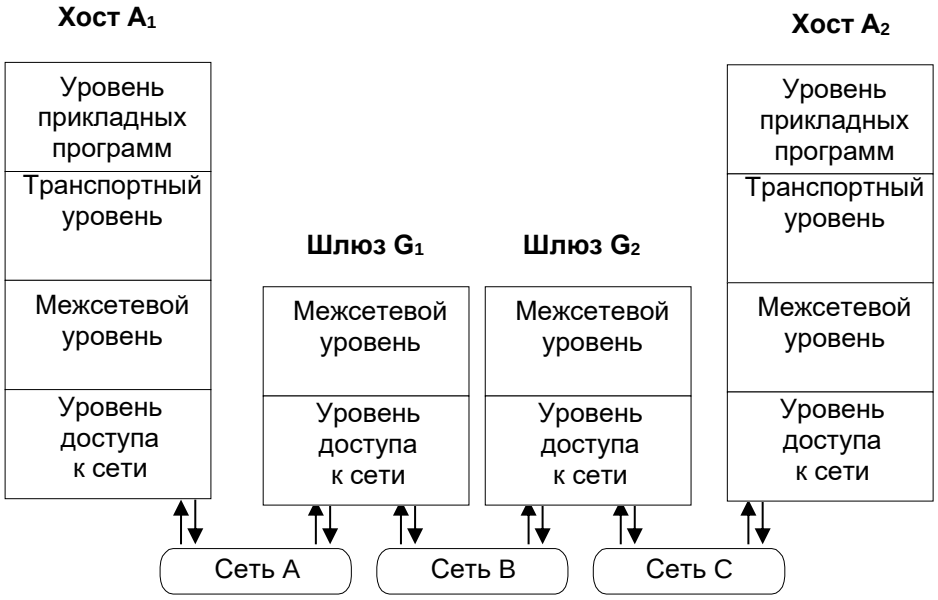


Рис. 71. Использование шлюзов для передачи пакетов.

Транспортный уровень. Два основных протокола транспортного уровня – TCP и UDP. Одним из важнейших протоколов данного уровня является протокол **TCP** (Transmission Control Protocol — протокол управления передачей данных), который, равно как и протокол IP, дал свое название всему семейству протоколов. Этот протокол послужил некоторым «прародителем» этого семейства протоколов, поскольку Министерство Обороны США, когда начинало исследование ARPANET, ставило перед собой задачу разработки сети, устойчивой к недетерминированной физической среде передачи данных. И одним из условий было, чтобы полученная сеть работала корректно как на линиях с устойчивой передачей данных (в которых количество ошибок мало), так и на линиях, в которых возникает большое число ошибок. Это требование и его реализация обусловило распространение семейства протоколов TCP/IP (и, в общем-то, развитие современных сетей, поскольку проблема дисбаланса различных сетей с точки зрения надежности каналов актуальна и по сей день, а разработанные протоколы решали эту проблему).

Протокол TCP — это протокол, обеспечивающий установление виртуального канала, а это означает, что он обеспечивает последовательную передачу пакетов, контролирует доставку пакетов и отрабатывает сбои (пакет либо не доставляется, либо доставляется в целостном состоянии). Для обеспечения заявленных качеств данный протокол подразумевает отправку по сети подтверждающей информации, из-за чего содержательная пропускная способность может очень сильно падать, особенно на линиях связи с плохими техническими характеристиками. Итак, этот протокол подразумевает, что для каждого полученного пакета адресат обязан отправить подтверждение о доставке. К этому необходимо добавить, что в данном протоколе действует поддержка времени: если через некоторое время после отправки пакета подтверждение так и не пришло, то считается, что отправленный пакет пропал, и начинается повторная посылка пропавшего пакета.

Некоторой альтернативой служит протокол **UDP** (User Datagram Protocol — протокол пользовательских дейтаграмм). Протокол UDP не обеспечивает установление виртуального соединения. Данный протокол подразумевает отправку пакетов по сети без гарантии их доставки (он выбрасывает пакет и сразу же «забывает» о нем).

Таким образом, те прикладные протоколы, которые основываются на TCP, оперируют передачей сообщений. Сообщения разбиваются на пакеты, и эти пакеты выбрасываются в сеть. Принимающая сторона на каждый пришедший пакет отправляет

подтверждение и собирает соответствующие пакеты в сообщение. При этом порядок отправки пакетов может не соответствовать порядку их прибытия на принимающую сторону. Прикладные протоколы, основанные на UDP, оперируют передачей пакетов.

Уровень прикладных программ. Этот уровень специфицирует протоколы построения распределённых сетевых приложений. На этом уровне находятся протоколы, часть которых опираются на протокол TCP, а часть — на UDP. Выбор между TCP и UDP основывается на том, насколько критична потеря информации, и на сфере реализации приложений (детерминированное или недетерминированное качество линий связи).

Протоколы, которые основываются на принципах работы протокола *TCP*, обеспечивают доступ и работу с заведомо корректной информацией, причем именно в среде межсетевого взаимодействия (internet), и эти протоколы требуют корректной доставки. В частности, это протокол **TELNET** (Network Terminal Protocol) — прикладной протокол, эмулирующий терминальное устройство (сетевой терминал); протокол межсетевого перемещения файлов **FTP** (File Transfer Protocol); протокол передачи почтовых сообщений **SMTP** (Simple Mail Transfer Protocol).

Есть ряд прикладных протоколов, основанных на использовании протокола *UDP*. Эти протоколы оказываются относительно быстрыми, поскольку максимально снижены накладные расходы на передачу, но они допускают наличие ошибок.

Часть подобных протоколов действуют в рамках локальной сети, где качество линий связи детерминировано. В частности, в большинстве случаев протокол **NFS** (Network File System) сетевой файловой системы функционирует именно в рамках *локальной сети*, и очень редко его запускают в межсетевом режиме.

Другая часть протоколов должна контролироваться, с одной стороны, на прикладном уровне, а с другой стороны, эти протоколы предполагают обмен очень небольшими порциями данных. К таким протоколам относится **DNS** (Domain Name Service), который позволяет мнемоническим способом именовать сетевые устройства. В частности, этот протокол осуществляет преобразования IP-адресов в мнемонические имена и обратно. Мнемонический адрес строится справа налево перечислением доменных имён соответствующих уровней (пример – *jaffar.mlab.cs.msu.su*). Доменные имена первого уровня определяют принадлежность данного имени по двум категориям: национальной (когда доменное имя определяет страну – fi, ru, su, de и др.) и по принадлежности компьютера к организации, занимающейся определённой деятельностью (com, org, gov, net и др.). Существует организация, которая распределяет доменные имена первого уровня. Владелец доменного имени i-го уровня может по своему усмотрению распределять доменные имена (i+1)-го уровня.

1.3 Основы архитектуры операционных систем

Этот раздел мы начнем с определения базовых понятий, среди которых очень важным для нас станет понятие операционной системы. Этот термин имеет различные толкования в разных изданиях – мы остановимся на следующем.

Операционная система — это комплекс программ, в функции которого входит обеспечение контроля за существованием, использованием и распределением ресурсов вычислительной системы. Напомним, что вычислительная система может включать в свой состав как физические, так и виртуальные ресурсы. Чтобы дать более ясную картину того, что же мы будем считать операционной системой, разберем детально её функции.

Начнем с того, что операционная система обеспечивает **контроль за существованием ресурсов**. Под этим понимается обеспечение операционной системой реализации виртуальных ресурсов и предоставление средств доступа к физическим ресурсам. Для любого ресурса степень его доступности зависит от операционной системы. Существуют ресурсы, которые полностью зависят от того, имеется ли их реализация в операционной системе или нет, если есть, то какая именно это реализация. Примером подобного ресурса служит файловая система: этого ресурса может и не быть в операционной системе, может существовать одна модель, или другая модель, или сразу несколько моделей.

Следующий пункт — **использование** ресурсов. Здесь имеется в виду, что операционная система предоставляет все средства, обеспечивающие доступность ресурсов ВС пользователю (точнее программам). При использовании любых ресурсов ВС может возникнуть конкуренция.

И, наконец, **распределение**: под этим будем понимать выбор стратегии распределения и обеспечение всевозможных моделей регламентации доступа.

Любая операционная система опирается на набор базовых сущностей, на основе характеристик которых выстраиваются почти все эксплуатационные свойства конкретной операционной системы. При этом, для различных операционных систем наборы базовых сущностей зачастую различаются: одни основаны на понятии устройства, другие — на понятии файла, третьи — на понятии набора данных. Но в большинстве случаев в состав базовых включается сущность, обозначающая исполняемую программу, задачу, задание или **процесс**. Эта сущность определяет некоторый процесс исполнения последовательности команд, причем здесь может участвовать единственная ветвь вычислений, а может сразу и несколько параллельных ветвей. Из множества трактовок этой сущности мы выберем понимание ее именно как ***процесса***.

Процесс — это совокупность машинных команд и данных, обрабатывающаяся в рамках вычислительной системы и обладающая правами на владение некоторым набором ресурсов ВС.

Разберемся в этом определении. Понятие **совокупности машинных команд и данных** обозначает то, что принято называть исполняемой программой (т.е. это код и операнды, используемые в этом коде). Далее, под термином **обработки в рамках ВС** будем понимать, что эта программа сформирована и находится в системе в режиме обработки (это может быть и ожидание, и исполнение на процессоре, и т.п.). И, третье, понятие **обладания правами на владение некоторым набором ресурсов** обозначает, по сути, возможность доступа. Отметим, что здесь речь не идет об эксклюзивных правах, поскольку в общем случае это было бы некорректно. Итак, иными словами, процесс можно определить как исполняемую программу, которая введена в систему для ее обработки и с которой ассоциированы некоторые ресурсы вычислительной системы.

Ресурсы, выделяемые процессам, могут быть *двух типов*. Первая категория ресурсов состоит из тех ресурсов, которые выделяются процессу ***на эксклюзивных правах***. Это

означает, что этот ресурс, пока процесс им владеет, принадлежит ему и только ему, и никакой иной процесс не имеет право работать с данным ресурсом. Вторая категория — это те ресурсы, которые одновременно могут принадлежать двум и более процессам, – такие ресурсы принято называть ***разделяемыми ресурсами***. Здесь сделаем небольшое пояснение: то, что разделяемый ресурс может *одновременно принадлежать* нескольким процессам, *не означает*, что к нему возможен *одновременный доступ*. Обозначенная проблема решается на другом уровне посредством использования разных схем синхронизации доступа к разделяемому ресурсу, и об этом речь пойдет несколько позже.

С точки зрения выделения ресурса процессу используются *две стратегии организации* этого *выделения*. Первый способ — это *предварительная декларация ресурсов*. В этом случае до ввода программы в систему и формирования для нее процесса декларируется перечень тех ресурсов, которыми процесс будет обладать. Например, это может быть перечень областей оперативной памяти, которые будут доступны данному процессу (если система поддерживает механизм виртуальной памяти, то это будет перечень областей виртуальной памяти, доступных процессу). Или же это может быть предельное время центрального процессора, которое может быть потрачено на исполнение данного процесса. Так или иначе, при вводе программы и формировании процесса операционная система постарается выделить *все* необходимые ресурсы, которые были предварительно декларированы. Если в системе нет заказанного ресурса, то она, скорее всего, не станет запускать процесс, который запросил этот ресурс.

Вторая модель — это *динамическое пополнение списка ресурсов*. Данная модель предполагает выделение процессу ресурса уже во время выполнения этого процесса. Это означает, что в системе происходит запуск процесса с выделением ему минимально необходимой области виртуальной памяти, а затем, когда процесс обращается к системе за выделением дополнительной области, то ОС обрабатывает эти запросы соответствующим образом. Отметим также, что на практике также применяются и комбинированные подходы, но во многих системах, с которыми мы сталкиваемся в нашей повседневной жизни, ориентация сделана на динамическую модель выделения ресурсов.

Многие операционные системы разрабатывались и разрабатываются таким образом, чтобы обладать следующими важными ***свойствами***: надежность, защита, эффективность и предсказуемость.

Надежность означает, что система должна быть надежной как программный комплекс, т.е. число программных ошибок в системе должно быть сведено к минимуму и должно быть соизмеримо с количеством возможных аппаратных сбоев.

Защита информации на сегодняшний день является одним из основных требований, предъявляемых к системе. ОС должна обеспечивать защиту информации и ресурсов от несанкционированного доступа. Эта проблема на сегодняшний день остаётся открытой.

Свойство **эффективности** означает, что функционирование системы должно удовлетворять некоторым требованиям – критериям эффективности, которые, по сути, являются оценкой соответствия.

И, наконец, это **предсказуемость** системы, являющаяся также одним из важных свойств ОС, поскольку большинство систем, которые, так или иначе, являются массово распространенными, при возникновении разного рода форс-мажорных обстоятельств должны вести себя строго определенным способом. Это свойство должно очерчивать круг всевозможных проблем, которые могут возникнуть в той или иной ситуации, предопределять последствия этих проблем, а также подразумевать устойчивость системы к возникновению сбоев (как аппаратных, так и программных).

1.3.1 Структура ОС

Существует множество взглядов, касающихся структуры операционной системы, и в этом разделе речь пойдет о некоторых из них.

Простейшая структурная организация основана на представлении операционной системы в виде композиции следующих компонентов (Рис. 72).

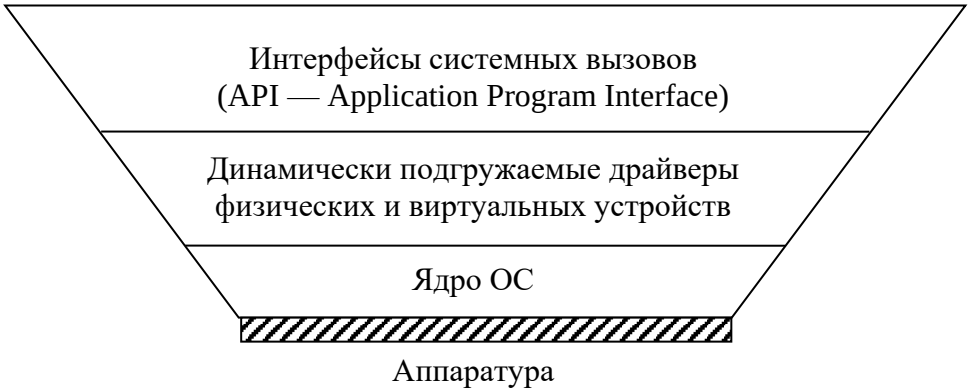


Рис. 72. Структурная организация ОС.

Ядро (***kernel***) ОС — это резидентная (постоянно размещаемая в ОП) часть ОС, реализующая некоторую базовую функциональность ОС и работающая в режиме супервизора (т.е. в привилегированном режиме). Итак, по определению ядро обеспечивает реализацию некоторого набора функций операционной системы. Это может быть очень большой набор функций, а может быть маленький — все зависит от конкретной реализации системы. Ядро может включать в свой состав драйверы основных физических или виртуальных устройств.

Над уровнем ядра может надстраиваться следующий уровень — это ***уровень динамически подгружаемых драйверов физических и виртуальных устройств***. Под **динамически подгружаемыми** понимается то, что в зависимости от ситуации состав этих драйверов *при инсталляции и загрузке* системы может меняться. Соответственно, эти драйверы можно поделить на две категории: ***резидентные драйверы*** и ***нерезидентные***. ***Резидентные драйверы*** подгружаются в систему в процессе ее загрузки и находятся в ней до завершения ее работы. Примером резидентного драйвера может быть драйвер физического диска. ***Нерезидентные драйверы*** вызываются операционной системой на сеанс работы с соответствующими устройствами (например, драйвер флэш-памяти).

Отметим, что большинство современных операционных систем имеют в своем составе набор драйверов широкого спектра конкретных физических устройств и, в частности, физических дисков. Поэтому зачастую при смене устройства драйвер менять не надо: он уже есть в системе. Но при этом системе незачем держать драйвера всех устройств в оперативной памяти. Соответственно, следуя той или иной стратегии, будут загружаться драйверы тех физических устройств, которые реально будут обслуживаться системой. Стратегии могут быть различными, одной из них: может быть явное указание системе списка драйверов, которые необходимо подгрузить (в этом случае, если в списке что-то будет указано неправильно, то соответствующее устройство, возможно, просто не будет работать). Вторая стратегия предполагает, что система при загрузке самостоятельно сканирует подключенное к ней оборудование и выбирает те драйверы, которые должны быть подгружены для обслуживания найденного оборудования.

Итак, примером резидентного драйвера может служить драйвер физического диска. Это объясняется тем, что диск является устройством оперативного доступа, поэтому к моменту полной загрузки системы все должно быть готово для работы. А, например, в системах, где пользователи редко используют сканер, держать соответствующий драйвер резидентно не имеет смысла, поскольку скорость работы самого устройства много медленнее, чем скорость загрузки драйвера из внешней памяти в оперативную. Соответственно, драйвер сканера в этом случае служит одним из примеров нерезидентных драйверов, т.е. тех драйверов, которые могут находиться в ОЗУ, а могут быть и отключенными, но они также динамически подгружаемые.

В общем случае драйверы могут работать как в привилегированном режиме, так и в пользовательском.

И, наконец, некоторой логической вершиной рассматриваемой структуры ОС будут являться ***интерфейсы системных вызовов*** (API — Application Program Interface). Под ***системным вызовом*** будем понимать средство обращения процесса к ядру операционной системы за выполнением той или иной функции (возможности, услуги, сервиса). Примерами системных вызовов являются открытие файла, чтение/запись в него, порождение процесса и т.д. Отличие обращения к системному вызову от обращения к библиотеке программ заключается в том, что библиотечная программа присоединяется к исполняемому коду процесса, поэтому вычисление библиотечных функций будет происходить в рамках процесса. Обращение к системному вызову — это вызов тех команд, которые инициируют обращение к системе. Как уже отмечалось выше, инициацией обращения к операционной системе может служить либо прерывание, либо исполнение специальной команды. Следует понимать различие между системным вызовом и библиотечной функцией. Например, осуществляя работу с файлом, имеется возможность работы с ним посредством обращения к системным вызовам либо посредством использования библиотеки ввода-вывода. В последнем случае в тело процесса включаются дополнительные функции из данной библиотеки, а уже внутри данных функций происходит обращение к необходимым системным вызовам.

Итак, существует несколько подходов к структурной организации операционных систем. Один из них можно назвать классическим: он использовался в первых операционных системах и используется до сих пор — это подход, основанный на использовании ***монолитного ядра***. В этом случае ядро ОС представляет собою единую монолитную программу, в которой отсутствует явная структуризация, хотя, конечно, в ней есть логическая структуризация. Это означает, что монолитное ядро содержит фиксированное число реализованных в нем базовых функций, поэтому модификация функционального набора достаточно затруднительна (необходима практически полная переделка ядра). Устройство монолитного ядра напоминает физическую организацию первых компьютеров: в них также нельзя было выделить отдельные физические функциональные блоки — все было единым, монолитным и интегрированным друг с другом. Аналогичными свойствами обладают одноплатные компьютеры, у которых все необходимые компоненты (ЦПУ, ОЗУ и пр.) расположены на одной плате, и, чтобы что-то изменить в этой конфигурации, требуются соответствующие инженерные знания.

На Рис. 73 проиллюстрирована структурная организация классической системы Unix. В данном случае ядро имеет фиксированный интерфейс системных вызовов. В нем реализовано управление процессами, а также драйвер файловой системы, реализована вся логика системы по организации работы с устройствами (которые можно разделить на байт-ориентированные и блок-ориентированные) и пр.

Рис. 73. Структура ОС с монолитным ядром.

Что касается достоинств данного подхода, то можно утверждать, что для конкретного состава функциональности и логики ядра это будет *наиболее эффективное* решение (т.к. оно имеет минимальное количество интерфейсных сочленений, связанных со структуризацией). Однако при таком подходе отсутствует универсальность, и внутренняя организация ядра рассчитана на конкретную реализацию. Недостаток в этом случае заключается в необходимости перепрограммировать ядро при внесении изменений, и это является прерогативой разработчика. Соответственно, для внесения новой функциональности пользователю системы приходится обращаться к разработчику, что зачастую ведет к материальным затратам.

Альтернативу данному подходу предлагают **многослойные операционные системы**. В этом случае все уровни разделяются на некоторые функциональные слои. Здесь можно провести аналогию с моделью сетевых протоколов. Между слоями имеются фиксированные интерфейсы. Управление происходит посредством взаимодействия соседних слоев. Поскольку любая структуризация снижает эффективность (программа, написанная в виде одной большой функции, работает быстрее, чем аналогичная программа, разбитая на подпрограммы, т.к. любое обращение к подпрограмме ведет к накладным расходам), то подобные системы обладают более низкой эффективностью.

Итак, каждый слой предоставляет определенный сервис вышестоящему слою. Деление на слои является индивидуальным для каждой конкретной операционной системы. Это может быть слой файловой системы, слой управления внешними устройствами и т.д. Тогда модернизация подобных систем сводится к модернизации соответствующих слоев. Вследствие чего проблема несколько упрощается, но при этом остаются ограничения на структурную организацию (например, имея слой файловой системы, можно заменить его другим вариантом этого слоя, но использовать одновременно две различные файловые системы не представляется возможным).

Третий подход предлагает использовать **микроядерную архитектуру** (Рис. 74). Функционирование операционных систем подобного типа основывается на использовании т.н. ***микроядра***. В этом случае выделяется *минимальный* набор функций (например, первичная обработка прерываний и некоторые функции управления процессами), которые

включаются в ядро. Вся остальная функциональность представляются в виде драйверов, которые подключаются к ядру посредством некоторого стандартного интерфейса.

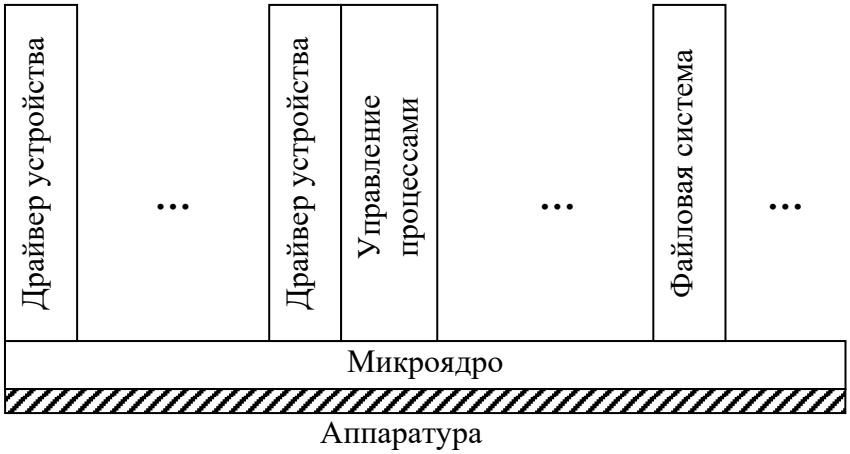


Рис. 74. Структура ОС с микроядерной архитектурой.

Такая архитектура получается хорошо расширяемой, она почти не имеет никаких ограничений по количеству подключаемых драйверов и их функциональному наполнению; требуется только соблюдение драйвером интерфейса для обращения к микроядру. Таким образом, данная архитектура представляется высоко технологичной, хорошо подходит для применения в современных многопроцессорных вычислительных системах (например, в SMP-системах, тогда можно распределять драйверы по различным процессорами и получать соответствующую эффективность).

Микроядерная система может служить основой для надстройки над микроядром разных операционных систем. В частности, такой подход используется в ряде систем, в основе которых используется микроядро системы Mach.

Итак, только что были продемонстрированы достоинства данного подхода. Что касается недостатков, то они непосредственно связаны с достоинствами и проявляются в значительном возрастании *накладных расходов*. Положим, процесс обращается к файловой системе, чтобы произвести обмен с конкретным файлом посредством соответствующего системного вызова. Драйвер файловой системы, получив запрос от процесса, перерабатывает его в последовательность запросов на обмен с диском (пусть, сначала это будут виртуальный диск). После чего файловая система обращается к микроядру, которое, в свою очередь, находит драйвер виртуального диска и передает ему соответствующий запрос. Драйвер виртуального диска определяет, с каким физическим диском будет происходить обмен, и трансформирует поступивший ему запрос в запросы к этому физическому диску, которые ему и передаются по той же схеме. Таким образом, один запрос распадается на множество подзапросов, следующих от драйвера через микроядро к другому драйверу, из-за чего эффективность системы снижается.

Напоследок отметим, что в реальности используются системы, получаемые комбинацией указанных подходов.

1.3.2 Логические функции ОС

Функциональность ОС можно представить в виде объединения некоторого фиксированного количества блоков функций. Состав этого набора варьирует от системы к системе, но в большинстве случаев можно выделить следующие функции: *управление процессами*, *управление оперативной памятью*, *планирование* и, наконец, *управление*

данными, файловой системой и устройствами, а также в последнее время стали добавлять блок функциональности *сетевого взаимодействия*.

Функция ***управления процессами*** охватывает проблемы формирования процессов, поддержания жизненного цикла процесса, проблемы организации взаимодействия процессов (т.е. организацию взаимодействия процесса с системой в целом и с другими процессами в частности) и работы процессов с ресурсами. Функция управления процессами является ключевой для всех операционных систем.

Функция ***управления оперативной памятью*** во многом определяется той аппаратурой, на которой реализуется ОС. С точки зрения программной составляющей, управление ОП включает компоненты, связанные с выбором стратегии организации ОП (в частности, реализуется поддержка аппарата виртуальной памяти), решает задачи защиты ОП от несанкционированного доступа, задачи корректности работы процесса с выделенной ему ОП. Некоторые из задач управления ОП пересекаются с функциями планирования (например, задачи выделения и изъятия памяти у процессов).

Функция ***планирования*** во многом определяет базовые характеристики ОС. Ранее говорилось, что в системе практически всегда возникает ситуация конкуренции за обладание теми или иными ресурсами. При этом сразу же возникает задача регламентации доступа конкурирующих процессов к ресурсам (т.е. задача обработки очередей запросов к ЗЗЗресурсам). Классическая задача планирования – *распределение времени центрального процессора* (т.е. планирование доступа процессов к центральному процессору). Ещё одна задача планирования – *организация и обработка очередей обмена*. В процессе функционирования системы формируется поток запросов на обмен, и очень часто этот поток может превышать пропускную способность устройства, то есть образуется конкуренция по доступу к устройству — выстраивается очередь запросов на обмен. Возникает проблема планирования очередей обмена. При этом необходимо учитывать приоритетность при обработке категорий запросов, которые идут от ОС и которые идут от пользователей. Следующая задача планирования – *обработка прерываний*. При обработке прерываний также есть приоритетность, так как прерывание иногда требует достаточно больших ресурсов системы и может возникать очередь прерываний.

Следующая функция ОС - ***управление устройствами и файловой системой***. Здесь рассматривается управление внешними устройствами, и файловая система рассматривается как виртуальное устройство, которое является достаточно значимым для операционной системы. Эту функцию можно разделить на управление устройствами, которые не требуют оперативного доступа, и управление устройствами, которые требуют оперативный доступ (которые могут содержать важную информацию и которые должны обеспечивать надёжность сохранения этой информации). Организация управления внешними устройствами должна осуществляться надёжно и эффективно. Это достигается посредством многоуровневой буферизации запросов к внешним устройствам, использованием достаточно сложных схем обеспечения надёжности обменов с внешними устройствами (например, программные средства дублирования) и т.д.

Следующая функция – ***обеспечение сетевого взаимодействия***. Практически любая современная ОС должна иметь средства взаимодействия с другими компьютерами в сети, то есть ОС должна обеспечивать функционирование и реализацию сетевых протоколов. При этом заметим, что не обязательно весь реализованный в системе стек протоколов должен являться компонентом ОС.

Следующая очень важная функция – функция ***обеспечения безопасности***. История возникновения проблемы компьютерной безопасности связана с появлением многопользовательских вычислительных систем, а именно, прежде всего, необходимо было обеспечить, чтобы один зарегистрированный в системе не мог добраться до информации другого зарегистрированного в системе пользователя и чтобы незарегистрированный пользователь не мог получить доступ к системе. Развитие сетевого взаимодействия существенно усложнило проблему безопасности, так как из-за концепции открытых

интерфейсов (ISO/OSI, TCP/IP) у компьютера появилось множество логических «входов», через которые программы других компьютеров могут связываться с данным. Появилось понятие компьютерной атаки на сетевое устройство с целью нарушения защиты (либо с целью несанкционированного доступа к данным, либо с целью нарушения функциональности устройства). То есть функция безопасности должна обеспечивать, во-первых, устойчивость системы к возможным атакам, а, во-вторых, анализ функционирования системы и выявление попыток вторжения в систему (это также важно, поскольку очень редко, когда взлом происходит сразу). Это очень сложная проблема, которая до настоящего момента является открытой. Следующий аспект обеспечения безопасности – это минимизация ущерба в случае вторжения.

1.3.3 Типы операционных систем

Операционные системы можно классифицировать с точки зрения критериев эффективности и стратегий использования центрального процессора. Можно выделить три основных класса операционных систем: ***пакетные*** операционные системы, системы ***разделения времени*** и системы ***реального времени***. Остановимся на каждой из них поподробнее.

Пакетная операционная система — это система, критерием эффективности функционирования которой является максимальная загрузка центрального процессора (т.е. минимизация накладных расходов). Иными словами, отношение всего времени работы процессора ко времени исполнения пользовательских программ должно быть близко к единице. Традиционно пакетные системы предназначались для решения расчетных задач, т.е. задач, требующих определенного объема времени работы процессора. Как следует из названия, эти системы оперируют термином ***пакет программ***.

Пакет программ — это некоторая совокупность программ, которые системе необходимо обработать. Особенность пакетных систем прослеживается в стратегии переключения выполнения процессов на процессоре – переключение выполнения процессов происходит только по одной из трех причин.

Первая причина — завершение выполнения процесса (в силу успешного перехода на точку завершения программы или же в силу возникновения ошибки).

Вторая причина — обращение к внешнему устройству с целью осуществления обмена, т.е. возникновение прерывания по вводу-выводу, поскольку операция обмена так или иначе требует какого-то минимального интервала времени.

И, наконец, третья причина — фиксация факта зацикливания процесса. В принципе точно определить факт зацикливания программы сложно, но все-таки возможно. На практике зачастую под фактом зацикливания считают исчерпание процессорного времени (положим, полтора часа).

Очевидно, что переключение процессов в подобных системах происходит лишь по необходимости, а это означает, что происходит редкое обращение к функции ОС смены контекстов обрабатываемых процессов, что ведет к максимальному снижению накладных расходов. В подобных системах степень полезной загрузки процессора составляет от 90% и выше.

Следующая модель — ***система разделения времени***. Данная модель может рассматриваться как развитие модели пакетных систем. В дополнение ко всем свойствам пакетных систем необходимо добавить дополнительную характеристику – для каждого процесса в системе определяется ***квант*** процессорного времени, который может быть единовременно использован процессом. Под ***квантом*** времени центрального процессора понимается некоторый фиксированный операционной системой промежуток времени работы процессора. Соответственно, переключение процессов происходит по тем же причинам, что и в пакетных системах (завершение процесса, возникновение прерывания, фиксация факта зацикливания), но необходимо добавить еще одну причину — исчерпался выделенный квант времени.

Критерием эффективности подобных систем служит вовсе не загрузка процессора, а минимизация времени отклика системы на запрос пользователя (положим, если пользователь набирает текст в текстовом редакторе, то будет важно, чтобы набранные им только что символы отображались на экране достаточно быстро, иначе работать с системой ему будет неудобно). Очевидно, что в подобных системах происходит частая смена контекстов, что связано с большими накладными расходами. В подобных системах эффективность может составлять порядка 30–40%, а, соответственно, 60–70% будут составлять накладные расходы.

Варьируя размерами кванта времени, можно получать системы для решения тех или иных задач. Увеличивая квант времени до некоторого среднего размера (порядка нескольких секунд), можно получить пакетную систему, ориентированную на обработку отладочных программ. А если увеличить размер кванта до бесконечности, получится пакетная система в чистом виде.

При организации планирования времени центрального процессора в системах разделения времени необходимо следующее. Во-первых, разделить все процессы на группы по некоторым критериям (например, интерактивные, отладочные и т.д.). Во-вторых, для каждой из этих групп определить квант времени ЦП, который будет выделяться процессу из конкретной группы. В-третьих, решить вопрос справедливой организации вычислительного процесса, т.е. определить приоритеты для каждой из категорий процессов. При этом в критерии смены обрабатываемого процесса можно добавить ещё один пункт – появление процесса из более приоритетной группы. В-четвёртых, приоритеты категорий процессов должны определяться по расписанию (например, в зависимости от времени суток).

Еще один класс систем представляют ***операционные системы реального времени***. Это специализированные системы, которые предназначены для функционирования в рамках вычислительных систем, обеспечивающих управление и взаимодействие с различными технологическими процессами. При разработке подобных систем все функции планирования ориентированы на обработку некоторого фиксированного набора событий, при возникновении любого из которых гарантируется обработка этого события за некоторый промежуток времени, не превосходящий определенного предельного значения.

Для иллюстрации можно привести следующий пример. Рассмотрим процесс кипячения молока. Если емкость с молоком постоянно нагревать, то через некоторое время оно начинает кипеть, а еще через некоторый достаточно короткий период оно «убегает» (после чего вообще начинает подгорать). Процесс кипячения молока можно автоматизировать, если в сосуд с молоком поместить датчик температуры, который снимает текущее значение температуры молока и передает это значение компьютеру. Соответственно, ставится задача «поймать» момент фиксации температуры кипения молока, причем среагировать необходимо за некоторый фиксированный промежуток времени. Если реакция произойдет, положим, через минуту, то молоко «убежит», и, соответственно, польза от такой системы будет минимальной. Таким образом, имеется фиксированный период времени, в течение которого компьютер должен снять показания датчика, определить, не достигнута ли точка кипения молока, и в случае кипения выключить подогрев сосуда с молоком.

Сфер применения систем реального времени в жизни очень много. Достаточно часто системы реального времени строятся под конкретные задачи. Выделяют различные группы систем реального времени: жесткого времени (например, управление бортовой системой самолёта), мягкого времени и пр.; но основной принцип их функционирования одинаков и подобен тому, который был проиллюстрирован выше.

И, в заключение, кратко остановимся на рассмотрении ***сетевых*** и ***распределенных операционных систем***. Как уже отмечалось выше, одиночные однопроцессорные системы уходят в прошлое, и во многих случаях процессорный элемент или компьютерный элемент

рассматривается как составляющая многопроцессорных или многомашинных ассоциаций. И с этой точки зрения операционные системы можно разделить на две категории.

В первую категорию можно отнести т.н. ***сетевые*** ОС (Рис. 75). Сетевая операционная система — это система, обеспечивающая функционирование и взаимодействие вычислительной системы в пределах сети. Это означает, что сетевая ОС устанавливается на каждом компьютере сети и обеспечивает функционирование распределенных приложений, т.е. тех приложений, реализация функций которых распределена по разным компьютерам сети. Примеров можно привести достаточно много. Так, почтовое приложение может быть распределенным: есть функции перемещения, есть сервер-получатель, есть клиентская часть, обеспечивающая интерфейс работы пользователя с указанным сервером.

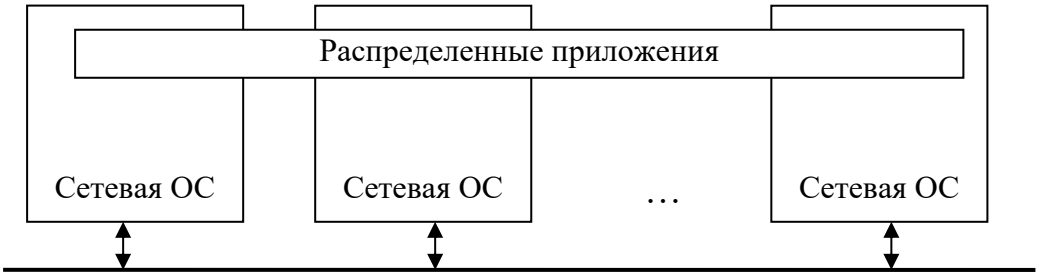


Рис. 75. Структура сетевой ОС.

Вторую категорию составляют ***распределенные*** ОС (Рис. 76). Распределенной операционной системой считается система, функционирующая на многопроцессорном или многомашинном комплексе, в котором на каждом из узлов функционирует отдельное ядро, а сама система обеспечивает реализацию распределенных возможностей ОС (т.н. сервисы или услуги). Примером распределенных функций может служить функция управления заданиями (напомним, что в кластерных системах задание может представлять собою целое множество процессов, и ставится задача распределить эти процессы по имеющимся процессорным узлам). Другим примером может служить распределенная файловая система. Традиционная файловая система ОС Unix просто не справится с потоками информации между узлами многопроцессорных систем, поэтому необходимы принципиально новые решения организации хранения и доступа к файлам.

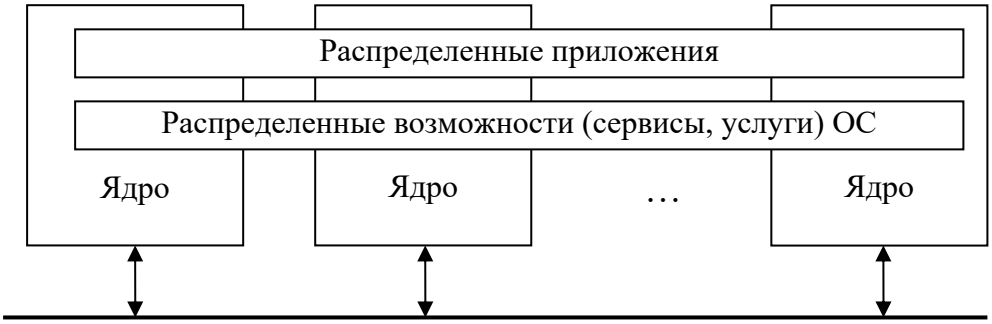


Рис. 76. Структура распределенной ОС.

2 Управление процессами

2.1 Основные концепции

Выше уже встречалось понятие ***процесса*** и некоторые его определения. Под ***процессом*** понимается совокупность машинных команд и данных, обрабатываемая в вычислительной системе и обладающая правами на владение некоторым набором ***ресурсов*** ВС. Также уже говорилось, что ресурсы могут декларироваться посредством различных стратегий, причем ресурс может эксклюзивно принадлежать одному единственному процессу, а может быть ***разделяемым***, когда ресурс может одновременно принадлежать нескольким процессам. С точки зрения выделения ресурса процессу используются две основные стратегии: предварительная декларация ресурсов и динамическое пополнение списка принадлежащих процессу ресурсов; также применяются и комбинированные подходы.

Также ранее отмечалось, что одной из основных задач ОС является поддержание ***жизненного цикла*** процесса. ***Жизненный цикл*** процесса — это те этапы, через которые может проходить процесс с момента его создания, в ходе его обработки и до завершения в рамках вычислительной системы. В реальности жизненный цикл процесса представляет собою характеристику конкретной операционной системы.

Выделим следующие типовые этапы жизненного цикла процесса:

- образование (порождение или формирование) процесса,
- обработка (выполнение) процесса на процессоре,
- ожидание постановки процесса на исполнение — обычно это ожидание какого-либо события: ожидание окончания обмена, ожидание выделения ресурса центрального процессора и пр.,
- завершение процесса — этап, связанный с возвратом процессом принадлежащих ему ресурсов.

2.1.1 Модели операционных систем

Обозначим основные характеристики **модельной операционной системы**, относительно которых будем рассматривать проблемы управления процессами. Будем считать, что ОС обеспечивает существование процессов в двух состояниях. Первое состояние — это размещение процесса, или программы, в ***буфере ввода процессов*** (***БВП***). В этом буфере размещаются процессы с момента их формирования, или ввода в систему, до начала обработки его центральным процессором. Отметим, что буфер ввода процессов является неотъемлемой частью пакетных систем. Второе состояние объединяет состояния процесса, связанные с размещением процесса в ***буфере обрабатываемых процессов*** (***БОП***), т.е. будем считать, что все процессы, которые начали обрабатываться центральным процессором, размещаются в данном буфере. Мы выделили именно два логических состояния, т.к. такая модельная ОС отражает наиболее общую картину. Процесс после его формирования не обязательно сразу попадает на процессор. Заметим, что многие информационные системные структуры образуются только тогда, когда процесс начинает обрабатываться, поэтому, соответственно, можно провести разделение по структурной организации. Размеры буферов в различных системах могут варьироваться.

Теперь рассмотрим некоторые модельные примеры совокупности состояний процессов в зависимости от типа ОС. Начнем рассмотрение с модели **пакетной однопроцессной системы** (Рис. 77). В подобной системе жизненный цикл процесса состоит всего из трех этапов. Первый этап — формирование процесса и ожидание начала обработки, т.е. поступление процесса в очередь на начало обработки процессором и ожидание им начала своей обработки (процесс попадает в БВП). Второй этап — обработка

(переход из БВП в БОП). Последний этап — завершение процесса, освобождение системных ресурсов. Данная система не имеет ожиданий готовых процессов или ожиданий ввода-вывода — это однопроцессная система, которая обрабатывает один процесс, причем все обмены синхронные, и процесс никогда не откладывается.

Рис. 77. Модель пакетной однопроцессной системы. 0 — поступление процесса в очередь на начало обработки ЦП (процесс попадает в БВП). 1 — начало обработки процесса на ЦП (из БВП в БОП). 2 — Завершение выполнения процесса, освобождение системных ресурсов.

Следующая модель — **пакетная мультипроцессная система** (Рис. 78). Данная модель уже имеет более богатый набор состояний процесса. Есть состояние ожидания начала обработки в БВП, после которого процесс попадает в БОП на обработку центральным процессором. Поскольку мы рассматриваем модель пакетной системы, то обрабатываемый процесс может либо завершиться, либо перейти в состояние ожидания ввода-вывода (если процесс обращается к операции обмена). Когда процесс переходит из состояния обработки на процессоре, система может поставить на счет некоторый процесс либо из БВП, либо из очереди готовых на выполнение процессов в зависимости от той или иной реализованной стратегии. Соответственно, после того, как процесс завершил обмен, он меняет свой статус и попадает в очередь на выполнение, из которой позже он попадет снова на выполнение.

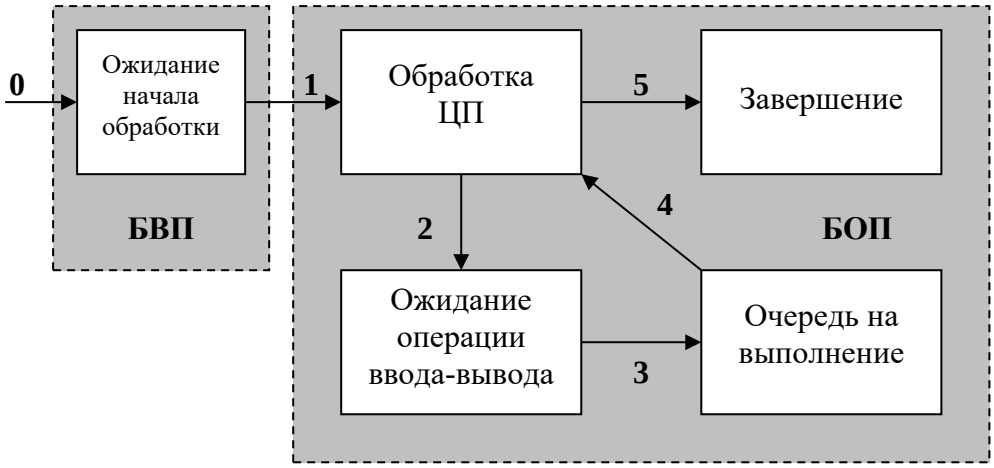


Рис. 78. Модель пакетной мультипроцессной системы. 0 — поступление процесса в очередь на начало обработки ЦП (процесс попадает в БВП). 1 — начало обработки процесса на ЦП (из БВП в БОП). 2 — процесс прекращает обработку ЦП по причине ожидания операции ввода-вывода, поступает в очередь завершения операции обмена (БОП). 3 — операция обмена завершена, и процесс поступает в очередь ожидания продолжения выполнения ЦП (БОП). 4 — выбирается процесс для выполнения на ЦП. 5 — завершение выполнения процесса, освобождение системных ресурсов.

Произведя в рассмотренной модели пакетной мультипроцессной системе небольшие изменения, можно получить модель **операционной системы с разделением времени** (Рис. 79). Структурно для этого достаточно добавить возможность перехода из состояния обработки центральным процессором в очередь готовых на выполнение процессов. Т.е. система имеет возможность прервать выполнение текущего процесса (если исчерпался выделенный квант времени) и поместить процесс в указанную очередь. Но такая модель не предполагает свопинга, или механизма откачки процесса во внешнюю память. В принципе, такую возможность можно также добавить в модель системы (Рис. 80) – тогда появляется еще одно состояние, характеризующее процесс, как откачанный во внешнюю память. Заметим, что в новое состояние могут переходить процессы лишь из очереди готовых на выполнение процессов, а процессы, ожидающие окончания ввода-вывода, свопироваться не могут, иначе в системе будут «зависать» заказы на обмен.

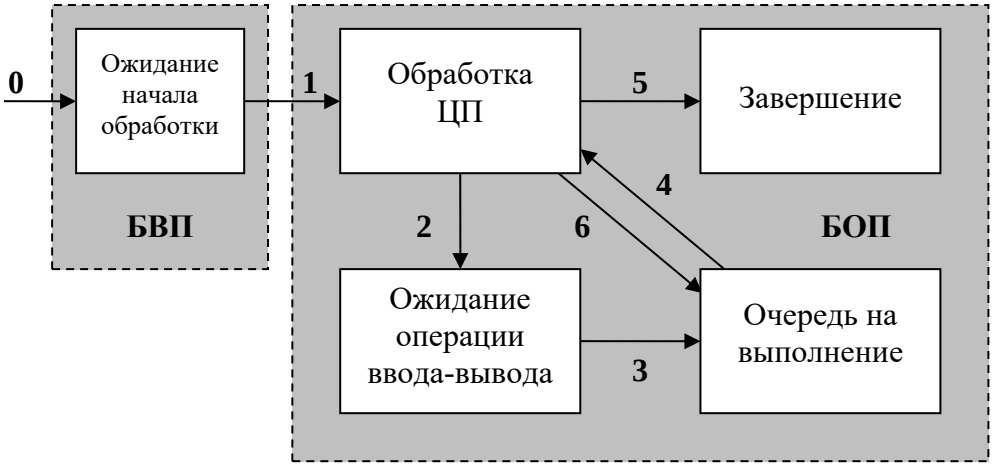


Рис. 79. Модель ОС с разделением времени. 6 — процесс прекращает обработку ЦП, но в любой момент может быть продолжен (истек квант времени ЦП, выделенный процессу). Поступает в очередь процессов, ожидающих продолжения выполнения центральным процессором (БОП).

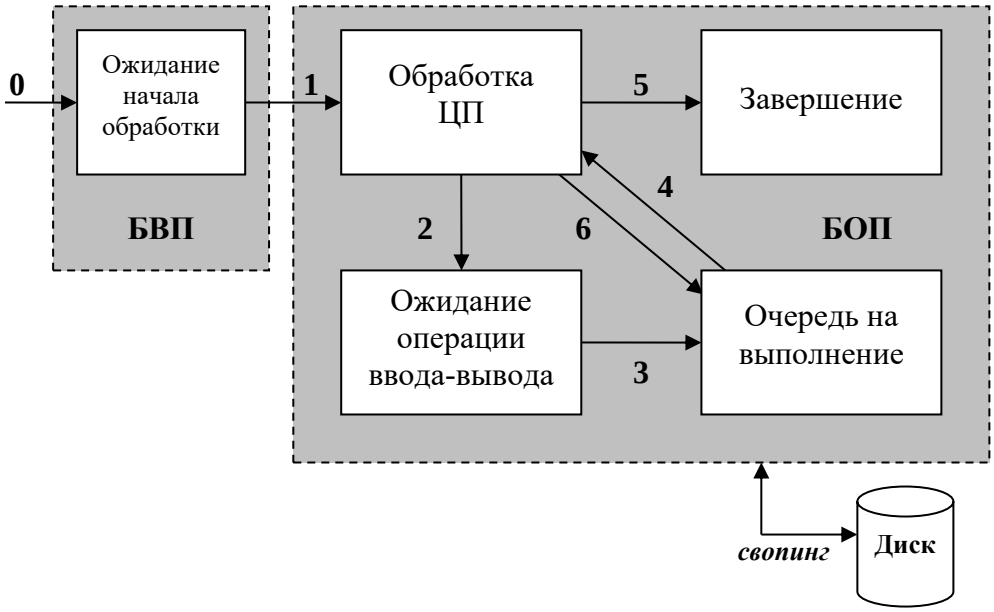
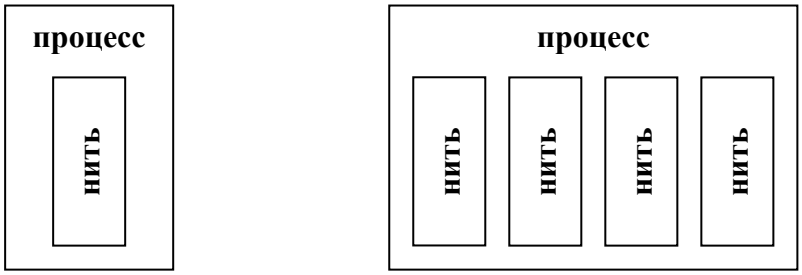


Рис. 80. Модель ОС с разделением времени (модификация). Заблокированный процесс может быть откачан (свопирован) на внешний носитель, а на освободившееся место может быть подкачен процесс с внешнего носителя, который был откачен ранее, либо взят новый.

2.1.2 Типы процессов

До данного момента мы рассматривали процессы как некоторые субъекты ОС, которые владеют ресурсами и являются объектами планирования. На самом деле, внутри процесса может быть не один объект планирования, а два и более (т.е. в одном процессе может быть две и более *нити*, или *потока*). При этом мы как бы вводим ещё один уровень определения процесса (Рис. 81). Процесс (или ***полновесный процесс***) – является объектом планирования и выполняется внутри защищённой области памяти. Альтернативой являются т.н. ***легковесные процессы***, известные также как ***нити*** (или ***потоки***), — это процессы, которые могут активироваться внутри полновесного процесса, могут быть объектами планирования, и при этом они могут функционировать внутри общей (т.е. незащищённой от других нитей) области памяти. Обратим внимание, что мы говорим о легковесных процессах, как о процессах другого иерархического уровня, нежели полновесных (т.е. легковесные процессы реализуются внутри полновесного процесса). Причём нити, которые функционируют внутри *одного* полновесного процесса, работают в *едином* адресном пространстве, и они незащищены друг от друга. При этом планированию подвергается каждая нить, т.е. планировщик может осуществлять переключение с нити на нить.

Основная причина использования многонитевой организации – это минимизация накладных расходов. Мультипроцессирование внутри полновесного процесса получается очень эффективным, благодаря уменьшению количества смен контекстов.



а. б.

Рис. 81. Типы процессов: однонитевая (а) и многонитевая (б) организации процессов.

Также отметим, что многонитевые процессы хорошо ложатся на современные многопроцессорные системы (например, SMP-системы), т.е. в некоторых случаях при такой организации повышается эффективность системы.

Кроме того, механизм нитевой организации позволяет осуществлять взаимодействие нитей в рамках одного процесса, причем адресное пространство, посредством которого они взаимодействуют, остается защищенным от других процессов в системе.

Соответственно, перед операционной системой, помимо управления полновесными процессами, планирования и выделения им ресурсов, возникает задача управления нитями.

Тогда определение процесса можно обобщить – понятие «***процесса***» включает в себя следующее:

- исполняемый код;
- собственное адресное пространство, представляющее собой множество виртуальных адресов, которые может использовать процесс;
- ресурсы системы, которые назначены процессу операционной системой;
- хотя бы одну выполняемую нить.

В заключение отметим, что многие современные операционные системы (как семейства Unix, так и Windows-системы, и др.) обеспечивают работу с нитями.

2.1.3 Контекст процесса

Рассмотрим теперь, какая поддержка операционной системы необходима для функционирования процессов. Здесь в первую очередь следует обратить внимание на понятие ***контекста процесса***. Под ***контекстом процесса*** будем понимать совокупность данных, характеризующих актуальное состояние процесса. Обычно контекст процесса состоит из трёх компонент:

- **пользовательская составляющая** — это текущее состояние программы (т.е. совокупность машинных команд и данных, размещённых в ОЗУ и характеризующих выполнение данного процесса);
- **аппаратная составляющая** – отражает актуальное состояние центрального процессора в момент выполнения данного процесса (т.е. это актуальное состояние регистров, настроек процессора и т.д.);
- **системная составляющая** – это структуры данных операционной системы, содержащие характеристики процесса:
    - содержат информацию идентификационного характера (PID процесса, PID «родителя» и т.д.); информацию о содержимом регистров (РОН, индексные регистры, флаги и т.д.);
    - информация о выделенных процессу ресурсах, настройках и использовании виртуальной и физической оперативной памяти;
    - информацию, необходимую для управления процессом (состояние процесса, приоритет и т.д.). Отметим, что системная составляющая процесса содержит копию аппаратно-системной составляющей, если процесс приостановлен: к примеру, в случае, если используется виртуальная страничная память, то в контексте разиещается «»расщиренная» копия таблицы страниц в которой размещается вся информация о кажой витруальной странице процесса (номер физической страницы, если она загружена в память; расположение страницы на внешнем устройстве, если такая страница есть и не загружена в память; признак отсутствия страницы).

Таким образом, когда процесс находится в состоянии выполнения (прцессором), то актуальна аппаратная составляющая, когда процесс отложен — актуальна системная составляющая.

2.2 Реализация полновесных процессов в ОС Unix

2.2.1 Процесс ОС Unix

Механизм управления и взаимодействия процессов в ОС Unix послужил во многом основой для развития операционных систем в целом, и логического блока управления процессами в частности. Во многом организация управления процессами в ОС Unix является эталонной, поэтому мы рассмотрим ее теперь более детально.

С точки зрения понимания термина ***процесса*** в ОС Unix, данное понятие можно определить двояко. С одной стороны, ***процесс*** можно определить как объект, зарегистрированный в таблице процессов операционной системы. Таблица процессов – одна из специальных системных таблиц, которая является программной таблицей. Второе определение объявляет ***процессом*** объект, порожденный системным вызовом *fork()*. Оба определения являются корректными и равносильными (если учесть, что в системе

существуют два особых процесса с нулевым и первым номерами, и об их особенностях речь пойдет ниже).

Остановимся сначала на первой трактовке процесса. В операционной системе имеется таблица процессов, предназначенная для регистрации всех существующих в данный момент процессов в системе. Размер этой таблицы является параметром настройки ОС, и, соответственно, количество процессов в системе является системным ресурсом. Таблица процессов обеспечивает уникальное именование процессов: она устроена позиционным образом, т.е. именование процесса осуществляется посредством номера записи таблицы, соответствующей данному процессу (нумерация строится от нуля до некоторого фиксированного значения). Этот номер записи называется ***идентификатором процесса*** (PID — Process Identifier). Как только что отмечалось, две первые записи таблицы предопределены и используются для системных нужд. Каждая запись таблицы процессов (Рис. 82) имеет ссылку на контекст процесса, который структурно состоит из пользовательской, системной и аппаратной составляющих.

Пользовательская (программная) составляющая — это тело процесса. Обычно тело процесса состоит из двух частей: сегмент кода и сегмент данных. *Сегмент кода* содержит машинные команды и неизменяемые константы; сегмент кода — это обычно не изменяемая *программным* способом часть тела процесса (отметим, что в принципе изменение может осуществляться посредством системных вызовов). Также в пользовательскую составляющую входит *сегмент данных*, включающий область статических данных процесса (в т.ч. статические переменные), область разделяемой памяти (т.е. область памяти, которая может принадлежать двум и более процессам одновременно), а также область стека. На стеке в системе реализуется передача фактических параметров функциям, реализуются автоматические и регистровые переменные, а также в этой области организуется динамическая память (т.н. куча).

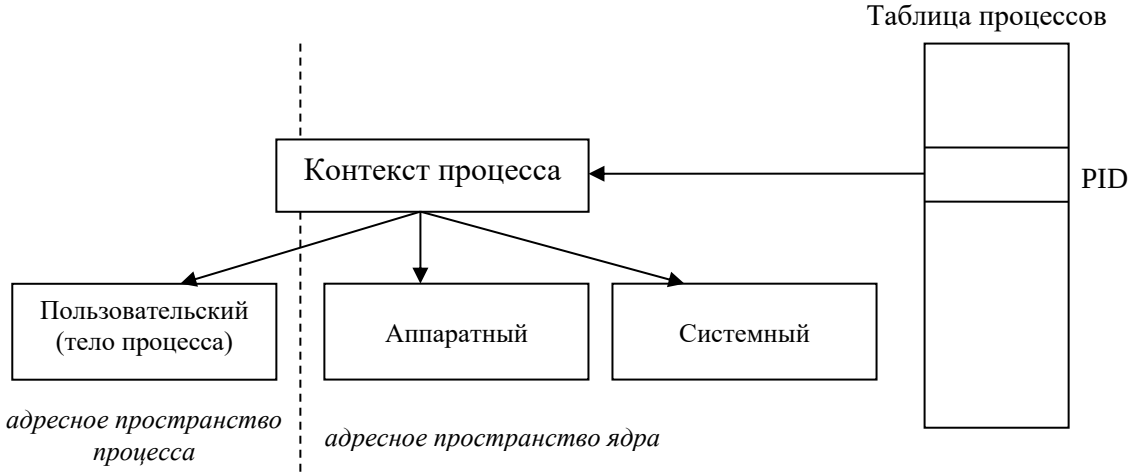


Рис. 82. Таблица процессов в ОС Unix.

В ОС Unix реализована такая возможность, как **разделение сегмента кода** (Рис. 83). Эта возможность предназначена для оптимизации использования ОП. Допустим, в системе работает несколько (пусть 100) пользователей, каждый из которых помимо прочего работает с одним и тем же текстовым редактором. Таким образом, в системе обрабатываются 100 копий текстового редактора. Ставится вопрос о необходимости держать в ОЗУ все сегменты кода для этих 100 процессов. Для оптимизации подобных ситуаций в ОС Unix используется указанный механизм разделения сегмента кода. Тогда каждый обрабатываемый в системе процесс текстового редактора в пользовательской составляющей хранит ссылку на единственную копию сегмента кода редактора, а сегмент данных у каждого из процессов свой. Соответственно, в приведенном примере в памяти

будут находиться один сегмент кода и 100 сегментов данных. Но рассмотренный механизм может иметь место в ОС только в том случае, когда сегмент кода нельзя изменить: он закрыт на запись.

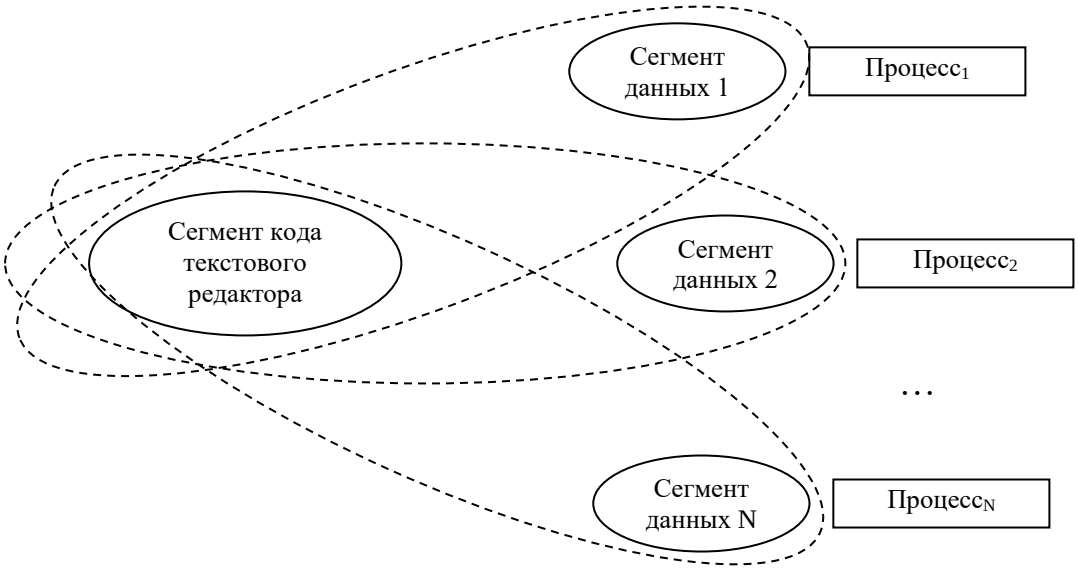


Рис. 83. Разделение сегмента кода.

Аппаратная составляющая включает в себя все регистры, аппаратные таблицы процессора и т.д., характеризующие актуальное состояние процесса в момент его выполнения на процессоре. Конкретная структура аппаратной составляющей зависит от конкретного процессора; обычно она включает счётчик команд, регистр состояния процессора, аппарат виртуальной памяти, регистры общего назначения и т.д.

Системная составляющая содержит системную информацию об идентификации процесса (т.е. идентификация пользователя, сформировавшего процесс), системную информацию об открытых и используемых в процессе файлах и пр., а также сохраненные значения аппаратной составляющей. Итак, в системной составляющей контекста процесса содержатся различные атрибуты процесса, такие как:

•идентификатор родительского процесса;

•текущее состояние процесса;

•приоритет процесса;

•реальный идентификатор пользователя-владельца (идентификатор пользователя, сформировавшего процесс);

•эффективный идентификатор пользователя-владельца (идентификатор пользователя, по которому определяются права процесса);

•реальный идентификатор группы, к которой принадлежит владелец (идентификатор группы, к которой принадлежит пользователь, сформировавший процесс);

•эффективный идентификатор группы, к которой принадлежит владелец (идентификатор группы «эффективного» пользователя, по которому определяются права процесса);

•список областей памяти;

•таблица дескрипторов открытых файлов процесса (именно дескрипторов, т.к. один и тот же файл может быть открыт в системе многократно); файловый дескриптор – это системная структура данных, отражающая актуальное состояние работы процесса с файлом;

•информация о том, какая реакция установлена на тот или иной сигнал (аппарат сигналов позволяет передавать воздействия от ядра системы процессу и от процесса к процессу);

•информация о сигналах, ожидающих доставки в данный процесс;

•сохраненные значения аппаратной составляющей (когда выполнение процесса приостановлено).

Рассмотрим более подробно такие атрибуты системной составляющей, как реальный и эффективный идентификаторы пользователя-владельца. В ОС Unix формированием процесса считается запуск исполняемого файла на выполнение. Исполняемым считается файл, имеющий установленный соответствующий бит исполнения в правах доступа к нему, при этом файл может содержать либо исполняемый код, либо набор команд для командного интерпретатора. Каждый пользователь системы имеет свой идентификатор (UID — User ID). Каждый файл имеет своего владельца, т.е. для каждого файла определен UID пользователя-владельца. В системе имеется возможность разрешать запуск файлов, которые не принадлежат конкретному пользователю. Большинство команд ОС Unix представляют собой исполняемые файлы, принадлежащие системному администратору (суперпользователю). Таким образом, при запуске файла определены фактически два пользователя: пользователь-владелец файла и пользователь, запустивший файл (т.е. пользователь-владелец процесса). И эта информация хранится в контексте процесса, как реальный идентификатор — идентификатор владельца процесса, и эффективный идентификатор — идентификатор владельца файла. А дальше возможно следующее: можно подменить права процесса по доступу к файлу с реального идентификатора на эффективный идентификатор. Соответственно, если пользователь системы хочет изменить свой пароль доступа к системе, хранящийся в файле, который принадлежит лишь суперпользователю и только им может модифицироваться, то этот пользователь запускает процесс *passwd*, у которого эффективный идентификатор пользователя — это идентификатор суперпользователя (UID = 0), а реальным идентификатором будет UID данного пользователя. И в этом случае права рядового пользователя заменятся на права администратора, поэтому пользователь сможет сохранить новый пароль в системной таблице (в соответствующем файле).

Следуя второй трактовке, ***процессом*** называется объект, порожденный системным вызовом *fork()*. Этот системный вызов обеспечивает создание копии текущего процесса. Выше уже упоминалось определение системного вызова, повторим его. Под ***системным вызовом*** понимается средство ОС, предоставляемое пользователям (а точнее, процессам), посредством которого процессы могут обращаться к ядру операционной системы за выполнением тех или иных функций. При этом выполнение системных вызовов происходит в привилегированном режиме (поскольку непосредственную обработку системных вызовов производит ядро), даже если сам процесс выполняется в пользовательском режиме. Что касается реализации системных вызовов, то в одних случаях системный вызов считается специфическим прерыванием, в других случаях — как команда обращения к операционной системе.

2.2.2 Базовые средства управления процессами в ОС Unix

2.2.2.1 fork() – создание нового процесса

Для порождения новых процессов в UNIX существует единая схема, с помощью которой создаются все процессы, существующие в работающем экземпляре ОС UNIX, за исключением первых двух процессов (0-го и 1-го). Для создания нового процесса в операционной системе UNIX используется системный вызов *fork()*.

Рассмотрим, что происходит при обращении к системному вызову *fork()*. При обращении процесса к данному системному вызову, операционная система создает копию текущего процесса, т.е. появляется еще один процесс, тело которого полностью идентично

исходному процессу. Это означает, что система заносит в таблицу процессов новую запись, тем самым новый порождённый процесс получает уникальный идентификатор. Для этого нового процесса в системе создается контекст, большая часть содержимого которого идентична контексту родительского процесса, в частности, тело порожденного процесса содержит копии сегментов кода и данных его родителя.

Сыновний процесс наследует от родительского процесса большую часть контекста, а именно:

- *окружение* — при формировании процесса ему передается некоторый набор параметров-переменных, используя которые, процесс может взаимодействовать с операционным окружением (интерпретатором команд и т.д.);
- *файлы, открытые в процессе-отце*, за исключением тех, которым было запрещено передаваться процессам-потомкам с помощью задания специального параметра при открытии. (Речь идет о том, что в системе при открытии файла с ним ассоциируется некоторый атрибут, который определяет правила передачи этого открытого файла сыновним процессам. По умолчанию открытые в «отце» файлы можно передавать «потомкам», но можно изменить значение этого параметра и блокировать передачу открытых в процессе-отце файлов.);
- способы обработки сигналов;
- разрешение переустановки эффективного идентификатора пользователя;
- разделяемые ресурсы процесса-отца;
- текущий рабочий и домашний каталоги;
- и т.д.

Отметим, что при вызове fork() сыновний процесс не наследует следующие составляющие контекста родительского процесса:

- идентификатор процесса (*PID*);
- идентификатор родительского процесса (*PPID*);
- сигналы, ждущие доставки в родительский процесс;
- время посылки ожидающего сигнала, установленное системным вызовом *alarm()*;
- блокировки файлов, установленные родительским процессом.

```
#include <sys/types.h>
#include <unistd.h>

pid_t fork(void);
```

По завершении системного вызова *fork()* каждый из процессов — родительский и порожденный, — получив управление, продолжает выполнение с одной и той же инструкции одной и той же программы, а именно, с той точки, где происходит возврат из системного вызова *fork()*. Вызов *fork()* в случае успешного завершения возвращает сыновнему процессу значение 0, а родительскому процессу — *PID* порожденного процесса. Это принципиально важно для различения сыновнего и родительского процессов, так как сегменты кода у них идентичны. Таким образом, у программиста имеется возможность разделить путь выполнения инструкций в этих процессах. В случае неудачного завершения (т.е. если сыновний процесс не был порожден, например, по причине отсутствия свободного места в таблице процессов), системный вызов *fork()* возвращает –1, а код ошибки устанавливается в переменной *errno*.

Рассмотрим **пример** (Рис. 84). Пусть в системе обрабатывается процесс с идентификатором 2757. В некоторый момент времени этот процесс обращается к системному вызову *fork()*, в результате чего в системе появляется новый процесс, который, предположим, имеет идентификатор 2760. Сразу оговоримся, что сыновний процесс может получить совершенно произвольный идентификатор, отличный от нуля и единицы (обычно

система выделяет новому процессу первую свободную запись в таблице процессов). По выходу из системного вызова *fork()* процесс 2757 продолжит свое выполнение с первой команды из then-блока, а сыновний процесс 2760 — с первой команды из else-блока. Далее эти процессы ведут себя независимо с точки зрения системного управления процессами: в частности, порядок их обработки на процессоре в общем случае пользователю неизвестен и зависит от той или иной реализованной в системе стратегии планирования времени процессора.

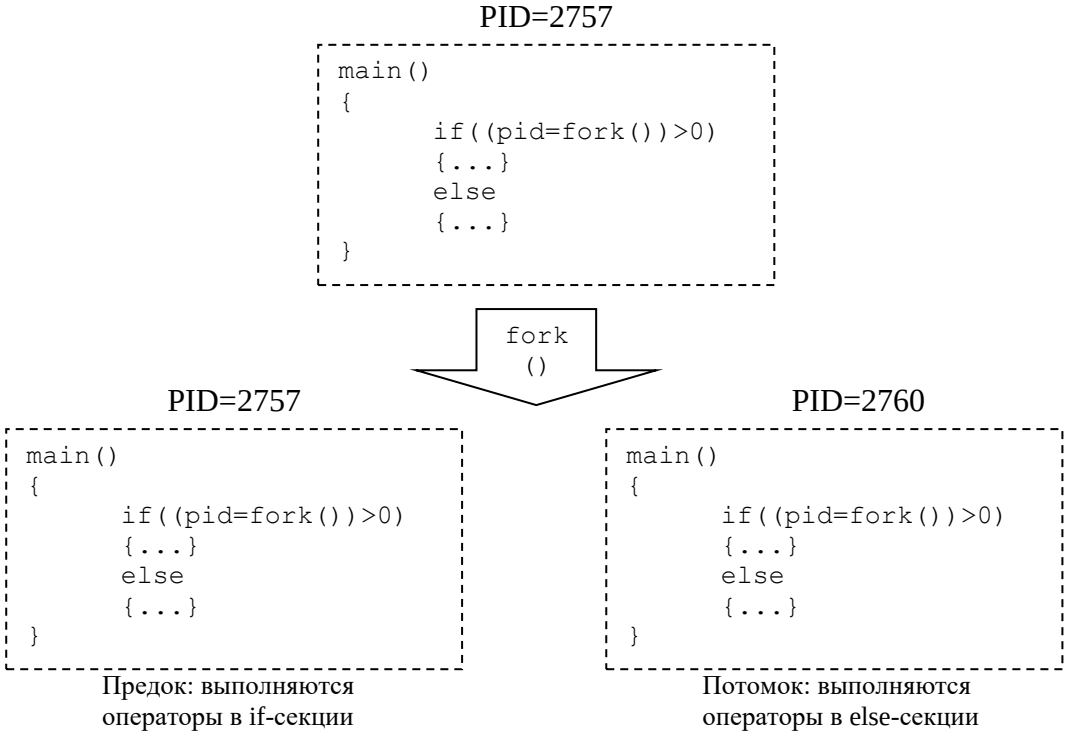


Рис. 84. Пример использования системного вызова *fork()*.

Рассмотрим еще один **пример**. В данном случае используются дополнительно два системных вызова: *getpid()* для получения идентификатора текущего процесса и *getppid()* для получения идентификатора родительского процесса. Итак, данный процесс при запуске печатает на экране идентификаторы себя и своего отца, затем производит обращение к системному вызову *fork()*, после чего и данный процесс, и его потомок снова печатают идентификаторы. Соответственно, на экране в случае успешной обработки всех системных вызовов будут напечатаны три строки, из которых две будут одинаковые.

```
int main(int argc, char **argv)
{
     /* печать PID  текущего процесса и PID процесса-предка */
     printf("PID=%d; PPID=%d \n", getpid(), getppid());
     /* создание нового процесса */
     fork();
     /* с этого момента два процесса функционируют параллельно и
     независимо */
     /* оба процесса печатают PID текущего процесса и PID
     процесса-предка */
     printf("PID=%d; PPID=%d \n", getpid(), getppid());
}
```

Следует отметить, что в случае, если родительский процеес сформировав сыновний завершится ло обращения сына к printf, то в соответсвующнй строке вывода у сыновнего процесса PPID будет равен 1 (pid init процесса).

2.2.2.2 exec() – семейство системных вызовов смены тела процесса

Редко бывает, когда в процессе происходит обращение лишь к системному вызову *fork()*. Обычно к нему происходит обращение в связке с одним из семейства системных вызовов *exec()*. Последние обеспечивают смену тела текущего процесса. В это семейство входят вызовы, у которых в названии префиксная часть обычно представлена как *exec*, а суффиксная часть служит для уточнения сигнатуры того или иного системного вызова. В качестве иллюстрации приведем определение системного вызова *execl()*.

```
#include <unistd.h>
int execl(const char *path, char *arg0, ..., char *argn, 0);
```

Параметр *path* указывает на имя файла программы, подлежащей исполнению. Параметры *arg0*, …, *argn* являются аргументами программы, передаваемыми ей при вызове (это те параметры, которые будут содержаться в массиве *argv* при входе в программу). При неудачном завершении возвращается -1, а в переменной *errno* устанавливается код ошибки.

Ниже представлены прототипы функций семейства *exec()*:

```
#include <unistd.h>

int execl(const char *path, char *arg0,…);
int execlp(const char *file, char *arg0,…);
int execle(const char *path, char *arg0,…, const char **env);
int execv(const char *path, const char **arg);
int execvp(const char *file, const char **arg);
int execve(const char *path, const char **arg, const char
**env);
```

Первый параметр во всех вызовах задает имя файла программы, подлежащей исполнению. Этот файл должен быть исполняемым файлом и пользователь-владелец процесса должен иметь право на исполнение данного файла. Для функций с суффиксом «p» в названии имя файла может быть кратким, при этом при поиске нужного файла будет использоваться переменная окружения PATH. Далее передаются аргументы командной строки для вновь запускаемой программы, которые отобразятся в ее массив *argv* — в виде списка аргументов переменной длины для функций с суффиксом «l» либо в виде вектора строк для функций с суффиксом «v». В любом случае, в списке аргументов должно присутствовать как минимум 2 аргумента: имя программы, которое отобразится в элемент *argv[0]*, и значение NULL, завершающее список. В функциях с суффиксом «e» имеется также дополнительный аргумент, описывающий переменные окружения для вновь запускаемой программы – это массив строк вида *name=value*, завершенный значением NULL.

Концептуально все системные вызовы семейства *exec()* работают следующим образом. Через параметры вызова передается указание на имя некоторого исполняемого файла, а также набор аргументов, которые передаются внутрь при запуске этого исполняемого файла. При выполнении данных системных вызовов происходит замена тела текущего процесса на тело, образованное в результате загрузки исполняемого файла, и управление передается на точку входа в новое тело.

Рассмотрим небольшой **пример** (Рис. 85). Запускается процесс (ему ставится в соответствие идентификатор 2757), который обращается к системному вызову *execl()*, для

смены своего тела телом команды **ls -l**, которая отображает содержимое текущего каталога. Реализация данной команды хранится, соответственно, в файле /bin/ls. После успешного завершения системного вызова *execl(),* процесс (с тем же идентификатором 2757) будет содержать реализацию команды **ls**, и управление в нем будет передано на точку входа (т.е. запустится функция *main()*).

Итак, семейство системных вызовов *exec()* производит замену тела вызывающего процесса, после чего данный процесс начинает выполнять другую программу, передавая управление на точку ее входа. Возврат к первоначальной программе происходит только в случае ошибки при обращении к *exec()*, т.е. если фактической замены тела процесса не произошло.

Заметим, что выполнение «нового» тела происходит в рамках уже существующего процесса, т.е. после вызова *exec()* сохраняется идентификатор процесса, и идентификатор родительского процесса, таблица дескрипторов файлов (за исключением, быть может, файлов, открытых в специальном режиме), приоритет и большая часть других атрибутов процесса. Фактически происходит замена сегмента кода и сегмента данных. Изменяется следующая системная информация, которая должна корректироваться при смене тела процесса:

- режимы обработки сигналов: для сигналов, которые перехватывались, после замены тела процесса будет установлена обработка по умолчанию, т.к. в новой программе могут отсутствовать указанные функции-обработчики сигналов
- эффективные идентификаторы владельца и группы могут измениться, если для новой выполняемой программы установлен *s*-бит
- перед началом выполнения новой программы могут быть закрыты некоторые файлы, ранее открытые в процессе. Это касается тех файлов, для которых при помощи системного вызова *fcntl()* был установлен флаг *close-on-exec*. Соответствующие файловые дескрипторы будут помечены как свободные.

При обращении к системным вызовам семейства *exec()* сохраняются основные атрибуты текущего процесса (в частности, идентификатор процесса, идентификатор родительского процесса, приоритет и др.), а также сохраняются все открытые в текущем процессе файлы (за исключением, быть может, файлов, открытых в специальном режиме). С другой стороны, изменяются режимы обработки сигналов, эффективные идентификаторы владельца и группы и прочая системная информация, которая должна корректироваться при смене тела процесса.

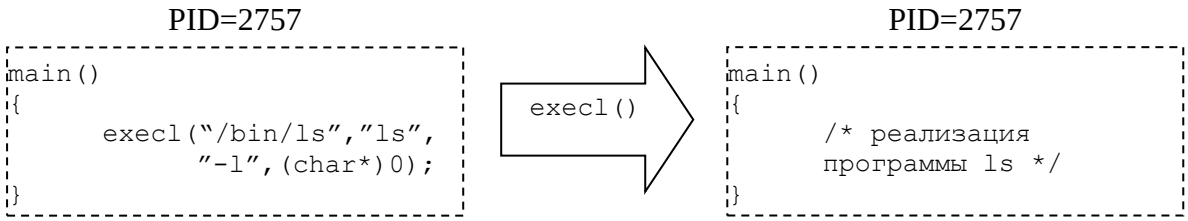


Рис. 85. Пример использования системного вызова *execl()*.

Приведем ряд примеров для иллюстрации применения различных вызовов семейства *exec()*.

Пример. Если обращение к системному вызову будет неуспешным, то функция *printf()* отобразит на экране соответствующий текст.

```
#include <unistd.h>
```

```
int main(int argc, char **argv)
{
     ...
     /*тело программы*/
     ...
     execl("/bin/ls", "ls", "-l", (char*)0);
     /* или execlp("ls", "ls", "-l", (char*)0); */
     printf("это напечатается в случае неудачного обращения к
предыдущей функции, к примеру, если не был найден файл ls \n");
     ...
}
```

Пример. **Вызов С-компилятора**. В данном случае второй параметр — вектор из указателей на параметры строки, которые будут переданы в вызываемую программу. Как и ранее, первый указатель — имя программы, последний — нулевой указатель. Эти вызовы удобны, когда заранее неизвестно число аргументов вызываемой программы.

```
int main(int argc, char **argv)
{
     char *pv[]={"cc", "-o", "ter", "ter.c", (char*)0};
     ...
     /*тело программы*/
     ...
     execv ("/bin/cc", pv);
     ...
}
```

Итак, мы рассмотрели по отдельности системные вызовы *fork()* и *exec()*. Чрезвычайно полезным является использование *fork()* совместно с системным вызовом *exec()*. Как отмечалось выше, системный вызов *exec()* используется для запуска исполняемого файла в рамках существующего процесса. Ниже (Рис. 86) приведена общая схема использования связки *fork()* — *exec()*. В данном случае родительский процесс (PID = 2757) порождает своего потомка посредством обращения к системному вызову *fork()*, после чего в отцовском процессе управление переходит на else-блок. В то же время в сыновнем процессе (PID = 2760) управление передается на первую инструкцию then-блока, где происходит обращение к системному вызову *execl()*. После чего тело сыновнего процесса меняется на тело команды **ls**.

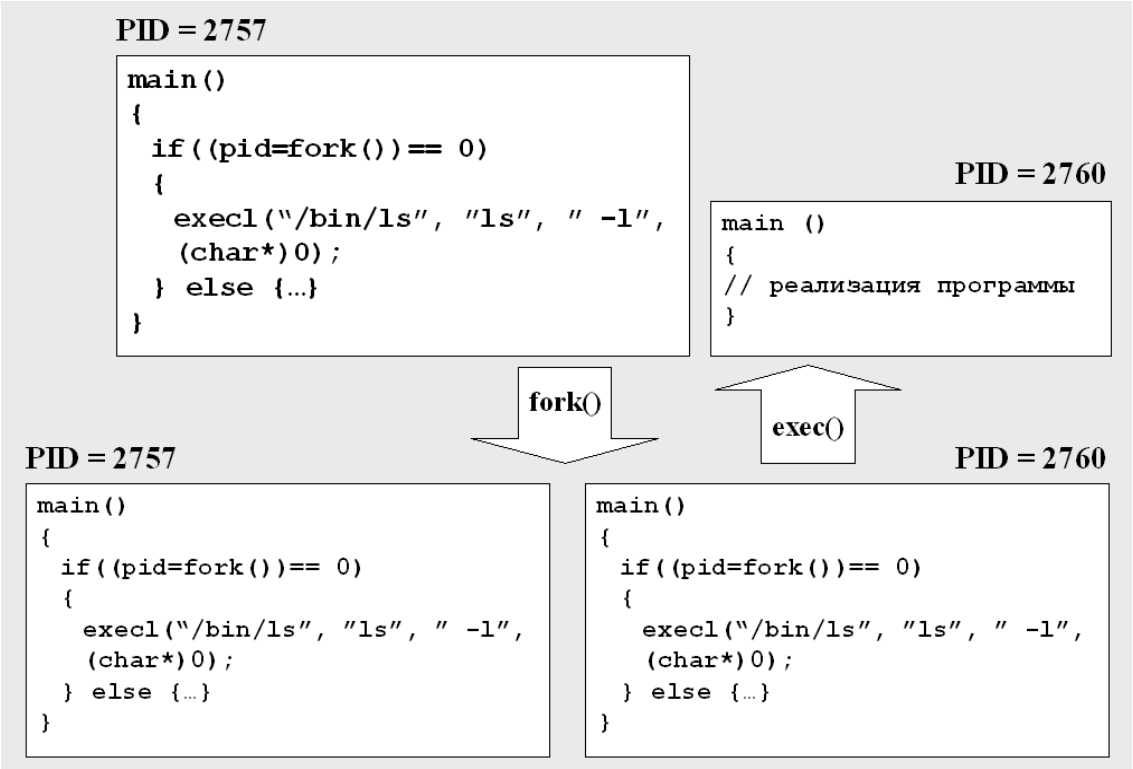


Рис. 86. Пример использования схемы *fork-exec*.

Рассмотрим еще один **пример**. Программа порождает три процесса, каждый из которых запускает программу **echo** посредством системного вызова *exec()*. Данный пример демонстрирует важность проверки успешного завершения системного вызова *exec()*, в противном случае возможно исполнение нескольких копий исходной программы. В нашем случае, если все вызовы *exec()* проработают неуспешно, то копий программ будет восемь. Если все вызовы *exec()* будут успешными, то после последнего вызова *fork()* будет существовать четыре копии процесса. В каком порядке они пойдут на выполнение предсказать трудно.

```
int main(int argc, char **argv)
{
     if(fork() == 0)
     {
          execl("/bin/echo","echo","это","сообщение один",NULL);
          printf("ошибка");
     }
     if(fork() == 0)
     {
          execl("/bin/echo","echo","это","сообщение два",NULL);
          printf("ошибка");
     }
     if(fork() == 0)
     {
          execl("/bin/echo","echo","это","сообщение три",NULL);
          printf("ошибка");
     }
     printf("процесс-предок закончился");
}
```

Результат работы может быть следующим:

```
процесс-предок закончился
это сообщение три
это сообщение один
это сообщение два
```

Теперь рассмотрим системные вызовы, которые сопутствуют базовым системным вызовам управления процессами в ОС Unix.

2.2.2.3 Завершение процесса

Прежде всего, речь пойдет о ***завершении процесса***. В случае, если процесс не трассируется (трассировка – специальный режим работы процесса, при котором родитель является главным, а потомк подчиненным), то он может может завершиться по возниконовению одного из двух, нижеуказанных событий.

1. Приход в процесс сигнала (сигнал можно считать программным аналогом прерывания, и речь о сигналах пойдет ниже при обсуждении вопросов взаимодействия процессов). Во многих случаях приход сигнала в процесс инициирует его завершение.
2. Вторая причина связана с обращением к системному вызову завершения процесса. При этом обращение может быть явным, когда в теле программы встречается обращение к системному вызову *_exit()*, или неявным, если происходит выполнение оператора *return* языка С внутри функции main() или выход на закрывающую скобку функции main(). В последнем случае компилятор заменит действие оператора return обращением к системному вызову *_exit()*.

В случае «завершения» трассируемого процесса сценарии обработки этих событий другие и будут рассмотрены позже (при рассмотрении системного вызова ptrace). Далее рассматривается последовательность действий, связанных с реальным завершение работы процесса.

```
#include <unistd.h>

void _exit(int status);
```

Системный вызов *_exit()* никогда не завершается неудачно, поэтому для него не предусмотрено возвращаемого значения. С помощью единственного параметра *status* процесс может передать породившему его процессу информацию о статусе своего завершения - т.н. программный код завершения процесса. **Принято, хотя и не является обязательным правилом, что возврат нулевого значения сигнализирует об успешном завершении процесса; ненулевые значения трактуются как ошибочное завершение (в частности, процесс может возвращать некий код ошибки)**.

```
void exit(int status);
```

Дополнительно отметим наличие библиотечной функции exit(), основное отличие которого от системного вызова _exit это корректный сброс (разгрузка) библиотечных кэш-буферов перед закрытием и, в конечном итоге, обращение к системному вызову _exit().

Рассмотрим, что происходит с процессом и в системе при обращении к системному вызову *_exit()*. Очевидно, что сиюминутно процесс не может завершиться, поэтому процесс переходит в переходное состояние — т.н. состояние «***зомби»***. При этом выполняется целая совокупность действий в системе, связанных с завершением процесса» **(отдельно случай трассировки):**:

1. корректно освобождаются ресурсы (закрываются все открытые дескрипторы файлов, освобождаются сегмент кода и сегмент данных процесса и пр.);
2. освобождается большая часть контекста процесса, однако сохраняется запись в таблице процессов и та часть контекста, в которой хранится статус завершения процесса и статистика его выполнения;
3. все сыновние процессы, чьи родители завершились, усыновляются процессом с номером 1;
4. процессу-предку от данного завершаемого процесса передается сигнал SIGCHLD, обычно он, по умолчанию, игнорируется.

Таким образом, каждый процесс может быть либо в состоянии замби или быть работающим. Информация о завершившемся процессе (в состоянии зомби) окончательно удаляется из системы (освобождается запись в таблице процессов и пр.) после того, как прямой родитель или усыновивший его процесс Init (PID 1) обрабатывают факт завершения сыновнего процесса посредством обращения к системному вызову wait().

2.2.2.4 wait() – ждем «завершения» процесса

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *status);
```

Результатом обращении к системному вызову *wait()* может быть:

1. если к моменту обращения к этому системному вызову какие-то сыновние процессы находятся в состоянии зомби, то процесс получит PID одного из этих процессов и данный процесс удаляется из системы;
2. если у процесса имеются только работающие сыновние процессы, то при обращении к указанному системному вызову отцовский процесс будет блокирован до перехода любого из своих сыновних процессов в состояние зомби;
3. если у процесса нет сыновних процессов (как работающих, так и зомби) то при обращении к системному вызову *wait()*, будет получен код ответа -1 (с указанием в переменной errno кода ошибки, соответствующему отсутствию сыновьих процессов);
4. если при выполнении системного вызова wait в данный процесс придет сигнал, то wait будет прерван с получением кода ответа -1 (см. сигналы);

Кроме того, в случае, если системный вызов wait() вернул PID завершенного процесса, то через параметр *status* дополнительно передается указатель на целочисленную переменную, в которой система возвращает процессу информацию о причине завершения сыновнего процесса. Данный параметр содержит в младшей части индикатор причины завершения процесса-потомка, устанавливаемый ядром ОС Unix (системный код завершения процесса: либо код стандартного завершения через *_exit()*, либо завершение/остановка по причине прихода сигнала). В старшей части - код завершения процесса-потомка (***пользовательский код завершения процесса***, передаваемый в качестве параметра системному вызову *_exit() при нормальном завершении,* либо номер сигнала, приход которого в сыновний процесс вызвал его завершение). С подробностями использования системного вызова wait можно ознакомиться в документации.

Необходимо сделать замечание, касающееся системного вызова *wait()*. Данный системный вызов не всегда отрабатывает на завершении сыновнего процесса. В случае если отцовский процесс производит трассировку сыновнего процесса, то посредством

системного вызова *wait()* можно фиксировать факт *приостановки* сыновнего процесса, причем сыновний процесс после этого может быть продолжен (т.е. не всегда он должен завершиться, чтобы отцовский процесс получил информацию о сыне). С другой стороны, имеется возможность изменить режим работы системного вызова *wait()* таким образом, чтобы отцовский процесс не блокировался в ожидании завершения одного из потомков, а сразу получал соответствующий код ответа.

И, наконец, отметим, что после передачи информации родительскому процессу о статусе завершения все структуры, связанные с процессом-зомби, освобождаются, и запись о нем удаляется из таблицы процессов. Таким образом, переход в состояние зомби необходим именно для того, чтобы процесс-предок мог получить информацию о судьбе своего завершившегося потомка, независимо от того, вызвал он wait() до или после его завершения.

Что же происходит с процессом-потомком, если его предок вообще не обращался к *wait()* и/или завершился раньше потомка? Как уже говорилось, при завершении процесса отцом для всех его потомков становится процесс с идентификатором 1. Он и осуществляет системный вызов *wait()*, тем самым, освобождая все структуры, связанные с потомками-зомби.

Часто используется сочетание функций *fork()* — *wait()*, если процесс-сын предназначен для выполнения некоторой программы, вызываемой посредством функции *exec()*. Фактически, это предоставляет процессу-родителю возможность контролировать окончание выполнения процессов-потомков.

Рассмотрим **пример использования системного вызова *wait()***. Ниже приводится текст программы, которая последовательно запускает программы, имена которых указаны при вызове.

```
#include<stdio.h>

int main(int argc, char **argv)
{
     int i;
     for (i=1; i<argc; i++)
     {
          int status;
          if(fork() > 0)
          {
               /* процесс-предок ожидает сообщения
               от процесса-потомка о завершении */
               wait(&status);
               printf("process-father\n");
               continue;
          }
          execlp(argv[i], argv[i], 0);
          exit(1); /* означает ошибка в execlp */
     }
}
```

Пусть существуют три исполняемых файла *print1*, *print2*, *print3*, каждый из которых только печатает текст *first*, *second*, *third* соответственно, а код выше приведенного примера находится в исполняемом файле с именем *file*. Тогда результатом работы команды

```
file print1 print2 print3
```

будет:

```
first
process-father
second
process-father
third
process-father
```

Рассмотрим еще один **пример**. В данном примере процесс-предок порождает два процесса, каждый из которых запускает команду **echo**. Далее процесс-предок ждет завершения своих потомков, после чего продолжает выполнение. В данном случае *wait()* вызывается в цикле три раза: первые два ожидают завершения процессов-потомков, последний вызов вернет неуспех, ибо ждать более некого.

```
int main(int argc, char **argv)
{
     if ((fork()) == 0) /*первый процесс-потомок*/
     {
          execl("/bin/echo","echo","this is","string 1",0);
          exit(1); /* означает ошибка в execlp */
     }
     if ((fork()) == 0) /*второй процесс-потомок*/
     {
          execl("/bin/echo","echo","this is","string 2",0);
          exit(1); /* означает ошибка в execlp */
     }
     /*процесс-предок*/
     printf("process-father is waiting for children\n");
     while(wait() != -1);
     printf("all children terminated\n");
     exit(0); /* нормальное завершение процесса */

}
```

2.2.3 Жизненный цикл процесса. Состояния процесса

Каждая ОС характеризуется жизненным циклом процессов, которые реализуются в системе. Рассмотрим обобщенную и несколько упрощенную схему жизненного цикла процессов в ОС Unix (Рис. 87). Можно выделить целую совокупность состояний, в которых может находиться процесс.

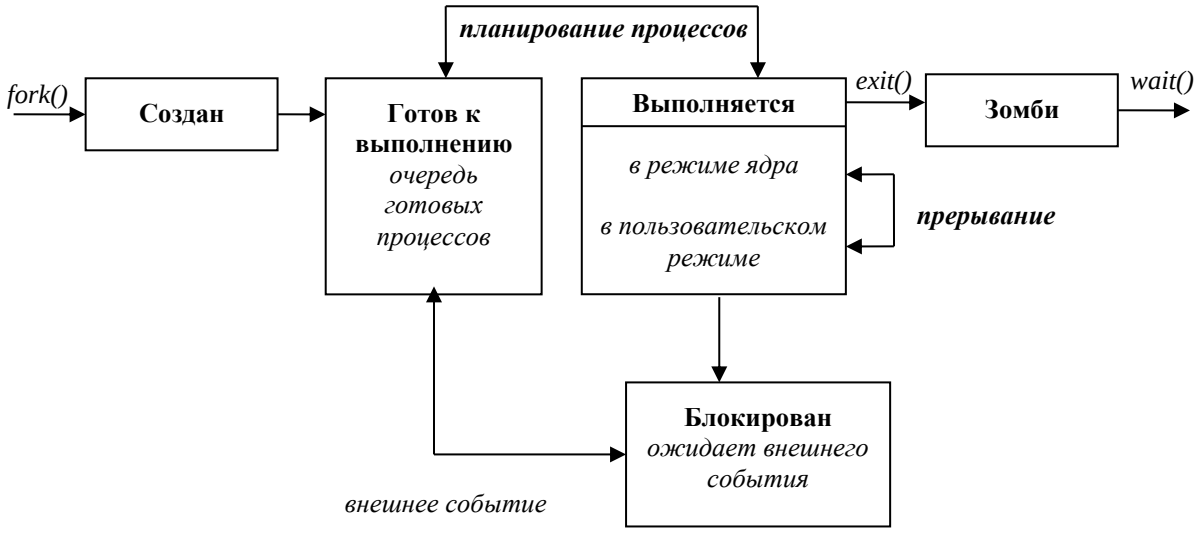


Рис. 87. Жизненный цикл процессов ОС Unix.

Начальное состояние — это состояние создания процесса: после обращения к системному вызову *fork()* создается новый процесс в системе (еще раз отметим, что иных способов создать процесс в ОС Unix не существует). Из этого состояния процесс практически сразу попадает в очередь процессов, готовых к выполнению. Из этой очереди по решению планировщика процесс может переходить в состояние выполнения, а затем обратно в состояние готовности к выполнению (например, в случае исчерпания кванта времени обработки на процессоре). Переходы между этими двумя состояниями основываются на динамических средствах планирования ОС Unix. Эти средства основаны на динамическом приоритете процесса: чем дольше процесс выполняется, тем ниже становится его приоритет, а, с другой стороны, чем дольше процесс находится в состоянии готовности к выполнению, тем выше становится приоритет процесса.

В состоянии выполнения процесс может работать как в пользовательском режиме, так и в режиме ядра (супервизора). В режиме ядра процесс работает при обращении к системному вызову, когда ядро ОС выполняет (на своих ресурсах) некоторые действия для конкретного процесса.

Из режима выполнения процесс может перейти как в состояние готовности к выполнению, так и в состояние блокировки. В состоянии блокировки процесс находится, ожидая завершения некоторого действия – например, ожидая завершения взаимодействия с внешним окружением. Соответственно, при возникновении соответствующего события процесс переходит из состояния блокировки в состояние готовности к выполнению.

Посредством обращения к системному вызову *_exit()* процесс переходит из состояния выполнения в состояние «зомби» (состояние завершения существования процесса), а после получения отцовским процессом информации о завершении данного процесса он завершает свой жизненный цикл.

Итак, мы рассмотрели упрощенную модель. Главным упрощением в ней можно считать отсутствие свопинга — отсутствие откачки процесса во внешнюю память. Ещё одно упрощение – реальные современные Unix-системы оперируют с нитями, и это тоже вносит свои особенности в жизненный цикл процесса.

2.2.4 Формирование процессов 0 и 1

Все механизмы взаимодействия процессов в ОС Unix унифицированы и основываются на связке системных вызовов *fork-exec*. Абсолютно все процессы в ОС Unix создаются по приведенной схеме, но существуют два процесса с номерами 0 и 1, которые

являются исключениями из данного правила. Эти два системных процесса существуют в течение всего времени работы системы.

Рассмотрим детально, как формируются данные процессы, но для этого необходимо разобраться, что происходит в системе при включении компьютера (т.е. в момент начальной загрузки Unix). Практически во всех компьютерах имеется область памяти, способная постоянно хранить информацию, — т.н. **постоянное запоминающее устройство** (**ПЗУ**). В этой области памяти постоянно находится программа, которая называется **аппаратный загрузчик компьютера**. При включении компьютера схемы управления запускают работу компьютера с адреса точки входа в этот аппаратный загрузчик. Данный загрузчик в общем случае имеет информацию о перечне и приоритетах системных устройств компьютера, которые априори могут содержать операционную систему. (Системное устройство – это блок-ориентированное устройство прямого доступа, на котором может размещаться ОС.) Приоритет определяет тот порядок, в котором аппаратный загрузчик по списку осуществляет перебор устройств в поисках программного загрузчика операционных систем. (Например, бывает полезно сделать floppy-диск наиболее приоритетным для загрузки ОС, так как в нормальном режиме аппаратный загрузчик этот диск не обнаружит, и загрузка будет осуществляться с жёсткого диска, а в случае «гибели» жёсткого диска мы как раз воспользуемся возможностью загрузки с floppy-диска.) Аппаратный загрузчик «знает» структуру системного устройства. Обычно в нулевом блоке системного устройства находится т.н. **программный загрузчик**, который может содержать информацию о наличии в различных разделах системного устройства различных операционных систем. Раздел системного устройства — это последовательность подряд идущих блоков (выделенная на внешнем запоминающем устройстве), внутри которых используется виртуальная нумерация этих блоков, т.е. каждый раздел начинается с нулевого блока. Соответственно, если операционных систем несколько, то программный загрузчик может предложить пользователю компьютера выбрать, какую систему загружать. После этого программный загрузчик обращается к соответствующему разделу данного системного устройства и из нулевого блока выбранного раздела считывает загрузчик конкретной операционной системы, после чего начинает работать программный загрузчик конкретной ОС. Этот загрузчик, в свою очередь, «знает» структуру раздела, структуру файловой системы и находит в соответствующей файловой системе файл, который должен быть запущен в качестве ядра операционной системы.

Что касается Unix-систем, то программный загрузчик ОС осуществляет поиск, считывание в память и запуск на исполнение файла */unix*, который содержит исполняемый код ядра ОС Unix. Рассмотрим теперь действия ядра при запуске.

Сначала происходит инициализация аппаратных и программных компонентов системы. Эта инициализация включает установку начальных параметров в аппаратных интерфейсах: устанавливаются системные часы (для генерации прерываний), формируется диспетчер оперативной памяти, устанавливаются средства защиты оперативной памяти. Затем, исходя из параметров настройки операционной системы, осуществляется формирование системных программных структур данных (в частности, создается таблица процессов, устанавливается размер КЭШ-буфера). После этого ядро создает нулевой процесс. При инициализации этого процесса резервируется память под его контекст и формируется нулевая запись в таблице процессов. Отметим, что здесь мы оперируем определением процесса в ОС Unix: ядро формирует нулевую запись в таблице процессов и более ничего, — это и есть ***создание нулевого процесса***. Этому нулевому процесс в общем случае соответствует ядру ОС (это процесс ядра), но он имеет особенность – отсутствие в контексте процесса сегмента кода (отметим, что это вовсе не означает, что нулевой процесс не имеет кода). Это означает, что нулевая запись таблицы процессов ссылается на контекст, в котором отсутствует ссылка на сегмент кода процесса.

Итак, основными отличиями нулевого процесса являются следующие моменты:

1. Данный процесс не имеет кодового сегмента — это просто структура данных, используемая ядром, и процессом его называют потому, что он каталогизирован в таблице процессов.
2. Он существует в течение всего времени работы системы (это чисто системный процесс), и считается, что он активен, когда работает ядро ОС.

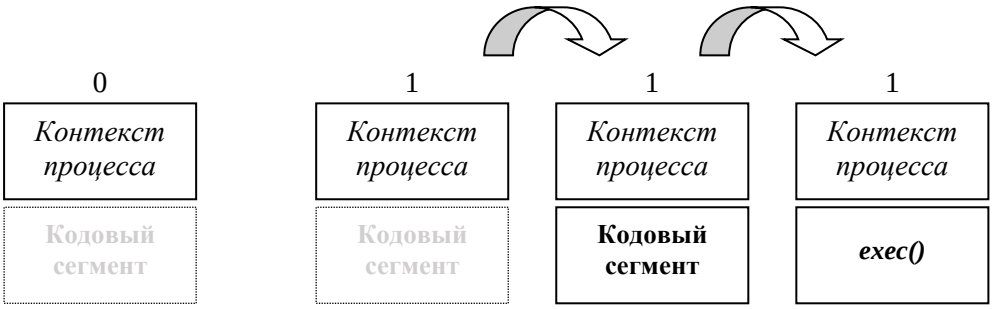


Рис. 88. Формирование нулевого и первого процессов.

Следующим этапом ядро начинает формирование первого процесса, который также создается нестандартным образом. При его создании выполняются следующие действия (Рис. 88). Ядро осуществляет копирование нулевой записи (в таблице процессов) в первую. После чего для первого процесса выделяется пространство оперативной памяти. В эту память загружается программа исполнения системного вызова *exec(),* после чего внутри первого процесса происходит обращение к этому системному вызову с параметром */etc/init*. Таким образом, можно отметить, что сам первый процесс формируется нестандартным путем, но тело его в конце формируется уже «правильным» образом посредством вызова *exec()*.

Итак, в итоге в рамках первого процесса сформирован процесс **init**, который существует в системе также на протяжении всего ее функционирования. Процесс **init** поддерживает соответствующие режимы работы системы: однопользовательский либо многопользовательский. Эти режимы характеризуются параметрами, которые определяются на стадии загрузки ядра и инициализации системы. Соответственно, система опознает один из подключенных терминалов как системную консоль. Если система работает в *однопользовательском* режиме, то происходит подключение интерпретатора команд к системной консоли (консоль регистрируется с корневыми привилегиями и доступ по каким-либо другим линиям связи невозможен). Если же режим *многопользовательский*, то процесс **init** обращается к системной таблице терминалов, хранящей все терминальные устройства, которые могут быть в системе, и для каждого готового к работе терминала из этого перечня он запускает процесс **getty** (Рис. 89). Процесс **getty** — это процесс, который обеспечивает работу конкретного терминала (т.е. сеанс работы пользователя). Заметим, что процесс **init** создает процесс **getty** уже стандартным способом, и после вообще все процессы создаются лишь по схеме *fork-exec*.

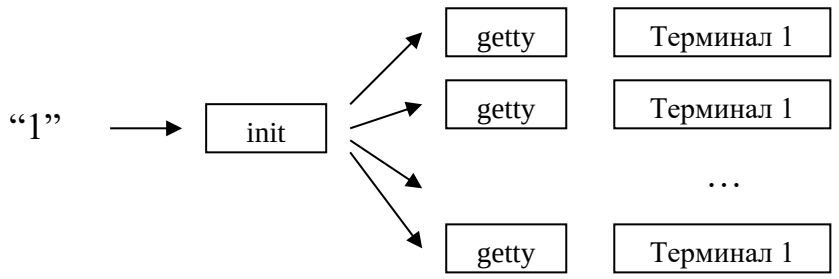


Рис. 89. Инициализация системы: создание многопользовательской среды.

В свою очередь, процесс **getty** печатает на экране приглашение ввести логин (Рис. 90). После того, как пользователь вводит логин, процесс **getty** загружает на свое место процесс подтверждения пароля **login**. Соответственно, процесс **login** запрашивает ввода пароля, после чего проверяет его. Если введенный пароль оказывается верным, начинается сеанс работы с пользователем. Сеанс работы с пользователем, как и сеанс работы всей системы, определяется конфигурационной информацией. Эта информация может храниться в зашифрованном виде в файле регистрации паролей *passwd*. Каждая запись этого файла содержит имя зарегистрированного пользователя, пароль, учётную информацию и некоторые настройки работы пользователя. В частности, к этим настройкам относится т.н. домашний каталог, т.е. тот каталог файловой системы, который станет текущим, когда пользователь начинает сеанс работы. В этой же записи файла *passwd* содержится информация о том интерпретаторе команд (**shell**), который должен быть загружен в начале сеанса работы пользователя. То есть, с одной стороны, система позволяет варьировать интерпретаторы команд для каждого из пользователей, а, с другой стороны, в качестве интерпретатора команд можно запустить любую программу (это может быть, например, программа, проверки целостности файловой системы).

С точки зрения интерпретатора команд, сеанс работы пользователя представляется в виде обменов с файлом (т.е. в виде операций чтения и записи). Работа пользователя с системой заканчивается закрытием файла – подачей EOF (end of file) (код EOF вводится путём нажатия комбинации клавиш Ctrl+D на клавиатуре). После получения от пользователя EOF интерпретатор завершает свою работу. Процесс **init** фиксирует эту ситуацию и снова запускает процесс **getty.**

Такова общая схема функционирования Unix-систем. Она строится на использовании нулевого и первого процессов. Это единственные процессы в системе, которые создаются нестандартным образом (без обращения к системному вызову *fork()*).

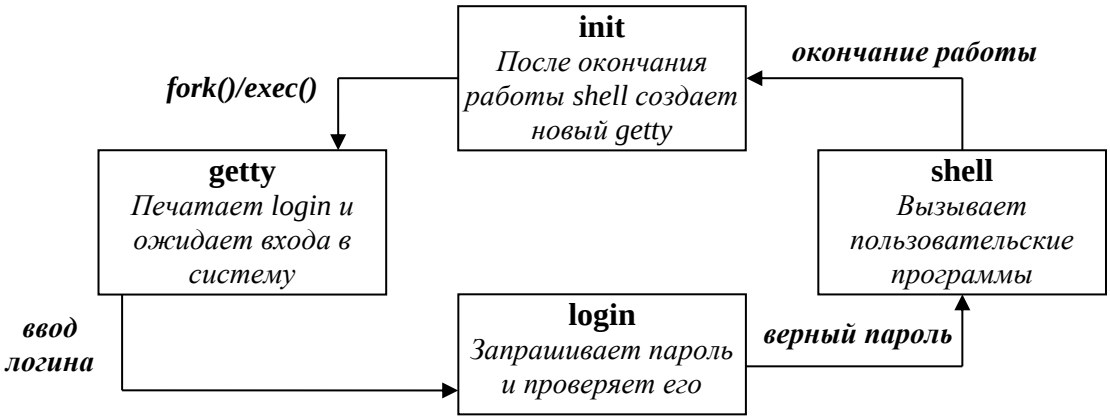


Рис. 90. Схема работы пользователя с ОС Unix.

2.3 Реализация легковесных процессов Unix (нити)

Легковесные процессы Unix или нити (далее **нити**) являются ресурсом, локализировнным в рамках процесса (полновесного процесса). Т.е. они создаются и могут взаимодействовать между собой только в рамках процесса, в котором они созданы. Нити, созданные в разных процессах друг друга «не видят» и, в общем случае, не могут осуществлять взаимодействие. Нити, созданные «внутри» процесса, разделяют глобальные переменные и системные ресуры, выделенные процессу. Стек и регистры (включая счетчик команд) локализованы в рамкак каждой нити.

2.3.1 Создание/запуск нити

int pthread_create(pthread_t *tid, const pthread_attr_t *attr, void* (*start)(void *), void *arg)

- tid – pthread указатель на область памяти, в которую будет размещен идентификатор созданной нити типа pthread_t. Заметим, что этот тип «скрыт» в реализации нитей и не обязательно является производным от целого (может быть, например, структурой). В связи с чем операции над значениями идентификаторов нитей реализуются через специальные фанции (pthread_equal, pthread_self – см. ниже).
- attr – используется для задания свойств (или атрибутов) нити. Не углубляясь в подробности, будем передавать в этом аргументе пустой указатель (NULL), что соответствует созданию нити со значениями свойств по умолчанию.
- start – указатель на функцию с фиксированным прототипом. Созданная нить начинает свое выпонения со входа на фукцию, указатель на которую передан в данном параметре. В качестве аргумента, этой функции передается значение параметра void *arg, таким образом имеется возможность «параметризовать» создание и запуск экземпляров нитей, имеющих один и тот же код. Возвращаемое значения данной функции void*.
- void *arg - см. описание предыдущего параметра.

Функция возвращает код ответа: 0 – в случае успешного создания и запуска нити. И значение #0 – в случае возникновения ошибки (заметим, что в этом случае ERRNO остается неизменным).

2.3.2 Операции с идентификатором нити

В общем случае идентификатор нити есть значение предопределенного типа pthread_t (#include pthread.h), которое не обязательно является производной целого типа. К примеру, значение идентификатора нити может являться структурой. Конкретное решение зависит от реализации конкретной Unix-системы. Так, например, представления типа pthread_t Linux 3.2.0 использует тип long int, Solaris 10 — unsigned int, а FreeBSD 8.0 и Mac OS X 10.6.8 в качестве типа pthread_t используют указатель на структуру pthread. В связи с чем для операций сравнения идентификаторов рекомендуется использовать специальную функцию:

int pthread_equal(pthread_t tid1, pthread_t tid2)

- tid1, tid2 - сравниваемые идентификаторы нитей;
- возвращает #0, если идентификаторы равны и 0, в противном случае.

Для получения идентификатолра нити можно использовать функцию:

pthread_t pthread_self(void)

- возвращает идентификатор вызывающей функцию нити.

2.3.3 Завершение нити

Завершение нити можно подразделить на две разновидности: завершение нити и всего процесаа; завершение только нити. Рассмотрим каждую из обозначенных разновидностей.

2.3.3.1 Завершение нити и всего процесса.

Завешение работы всего процесса произойдет в случае исполнени в одной из нитей системного вызова _exit (или библиотечной функции exit). Также произойдет завершение всего процесса, если в момент выпонения кода нити придет сигнал, вызывающий завершение работы процесса.

2.3.3.2 Завершение отдельной нити

Завершение работы отдельной нити произойдет в следующих случаях:

- Нить может вызвать функцию pthread_exit.
- Принудительное завершение нити из другой нити в том же процессе.
- Нить может завершиться и вернуть управление в другую нить

Функция pthread_exit

void pthread_exit(void *val_ptr)

- val_ptr – параметр «ответа», через него можно передать нити, ожидающей завершения данной, информацию о завершении («код завершения»).

- происходит завершение вызвавшей функцию нити.
Функция всегда завершается успешно, поэтому возвращаемое значение void.

Также, нить будет завершена в случае выхода из функции – «точки входа» в нить (той, указатель на которую был передан через pthread_create при создании данной нити). Выход может быть осуществлен посредством выполнения return (возможно с параметром кода ответа).

В случае, если завершения данной нити ожидает другая нить, то она получит значение параметра «ответа». Подробнее об этом ниже.

Завершение и передача управления в другую нить

int pthread_join(pthread_t tid, void** val_ptr)

- tid - идентификатор нити;
- val_ptr - в переменную val_ptr будет размещен код «ответа» завершенной нити (указатель на код ответа);
- возвращает 0 в случае успеха, код ошибки — в случае неудачи.

- обратившаяся к функции нить будет заблокирована до тех пор, пока нить с идентификатором tid не завершится (выполнение pthread_exit, выход из «вызывающей» функции или принудительное завершение). Только одна нить может дожидаться завершения заданной нити.

Принудительное завершение нити

int pthread_cancel(pthread_t tid)

- tid - идентификатор завершаемой нити;
- возвращает 0 в случае успеха, код ошибки — в случае неудачи.

- целевой нити с идентификатором tid передается воздействие, вызывающее действия, аналогичные обращению к функции `pthread_exit` с параметром `PTHREAD_CANCELED`. В принципе, целевая нить может перехватить это воздействие и обработать его (в частности, проигнорировать).

2.3.4 Примеры программирования нитей

Цель: демонстрация базового набора функционала для работы с нитями в Unix-системах**:** порождение нити, идентификатор нити, передача параметров в точку входа, работа с общими(разделяемыми) данными; синхронизация выполнения основной программы и порождённой нити с помощью механизма ожидания окончания выполнения.

Программа 1.

Основная программа: инициализирует общую переменную нулём, создает несколько параллельно работающих нитей, порождая их в цикле с передачей порядкового номера в качестве параметра в точку входа. Выводит на устройство вывода (на экран по умолчанию) системный идентификатор каждой созданной нити.

Каждая порождённая нить: выводит на устройство вывода (на экран по умолчанию) порядковый номер (полученный в качестве параметра) + свой идентификатор + текущее значение общей переменной + значение локальной переменной, увеличивает значение переменных на 1.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define NUM_THREADS 3
/*общая переменная, используемая нитями */
int shared_counter = 0;

/*функция – прототип нити */
void* thr_fn(void *arg)
{
	/*локальная переменная*/
	int local = 0;
	/*пауза для "перемешивания" выполнения нитей*/
	sleep(abs(5 - (int)arg));
```

```
        /*выводим информацию о нити (порядковый номер и идентификатор в 16ричном виде), текущее значение общей и локальной переменных (и увеличиваем на 1) */
        printf("***Thread #%d with ID %#x: shared value is %d, local value is %d\n", (int)arg, pthread_self(), shared_counter++, local++);
        return NULL;
}

int main()
{
        pthread_t th_id;
        int i, err;
        for (i = 0; i<NUM_THREADS; i++) {
                /*ссоздаем нить, используем последний аргумент для передачи порядкового номера по значению! */
                err = pthread_create(&th_id, NULL, thr_fn, (void *)i);
                printf("MAIN: created thread with ID %#x\n", th_id);
        }
        /*пауза - дать возможность всем нитям отработать и вывести информацию о себе*/
        sleep(10);
        return 0;
}
```

Возможный вывод на экран:

```
MAIN: created thread with ID 0xda8f2640
MAIN: created thread with ID 0xda0f1640
MAIN: created thread with ID 0xd98f0640
***Thread #2 with ID 0xd98f0640: shared value is 0, local value is 0
***Thread #1 with ID 0xda0f1640: shared value is 1, local value is 0
***Thread #0 with ID 0xda8f2640: shared value is 2, local value is 0
```

Рекомендованные задания.

1) Запустить программу, реализующую пример на компьютере.

2) Провести и запустить на компьютере модификации программы, реализующей пример, в части:

- добавления контроля и обработки всех возможных ошибок;
- сделать паузу для перемешивания выполнения порождённых нитей случайной;

Обратить внимание: при использовании некоторых компиляторов (например, gcc) для поддержки нитей может потребоваться указание дополнительной опции компилятора при сборке программы: “-lpthread” или “-pthread”. Подробности следует уточнить в инструкции компилятора.

Программа 2.

Основная программа: создаёт нить, дожидается её окончания, выводит на устройство вывода (на экран по умолчанию) системный идентификатор нити и время в

секундах, которое заняло выполнялнение нити. Порождённая нить: после начала работы засыпает на несколько секунд (от 0 до 10), после чего завершается.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <pthread.h>
/*функция – прототип нити */
void* thr_fn(void *arg)
{
	/*time(0) возвращает количество секунд с 01.01.1970 до
момента вызова */
	sleep(time(0) % 10); /*засыпаем на 0..10 секунд */
	pthread_exit(NULL);
}

int main()
{
	pthread_t th_id;
	void *pret;
	/*создаём нить и запоминаем время создания */
	pthread_create(&th_id, NULL, thr_fn, NULL);
	time_t tm_start = time(0);
	/*ожидаем завершения ранее созданной нити */
	pthread_join(th_id, &pret);

	printf("Thread %#x execution time =  %u\n", th_id, time(0) -
	tm_start);

	return 0;
}
```

Рекомендованные задания.

1) Запустить программу, реализующую пример на компьютере.
2) Провести и запустить на компьютере модификации программы, реализующей пример, в части:

- добавления контроля и обработки всех возможных ошибок;
- добавить вариант с ненулевым кодом завершения нити и распечатать информацию о коде завершения в основной программе.

Обратить внимание: при использовании некоторых компиляторов (например, gcc) для поддержки нитей может потребоваться указание дополнительной опции компилятора при сборке программы: “-lpthread” или “-pthread”. Подробности следует уточнить в инструкции компилятора.

2.4 Планирование

2.5 Взаимодействие процессов

2.5.1 Разделяемые ресурсы и синхронизация доступа к ним

Одной из важных проблем, которые появились в современных операционных системах, является проблема взаимодействия процессов.

Будем говорить, что процессы называются ***параллельными***, если их выполнение хотя бы частично перекрывается по времени. Т.е. можно говорить, что все процессы, находящиеся в буфере обрабатываемых процессов, являются параллельными, т.к. в той или иной степени времена их выполнения перекрываются друг с другом. Не стоит забывать, что, говоря о параллельных процессах, речь идет лишь о ***псевдопараллелизме***, поскольку реально на процессоре может исполняться только один процесс.

Параллельные процессы могут быть независимыми и взаимодействующими. ***Независимые процессы*** используют независимые множества ресурсов; т.е. множества ресурсов, которые принадлежат независимым процессам, в пересечении дают пустое множество. Альтернативой независимым процессам являются взаимодействующие процессы. ***Взаимодействующие процессы*** совместно используют ресурсы, и выполнение одного процесса может оказывать влияние на результат другого процесса, участвующего в этом взаимодействии.

Совместное использование ресурса ВС двумя и более параллельными процессами, когда каждый из процессов некоторое время владеет этим ресурсом, называется ***разделением ресурса***. Разделению подлежат как аппаратные, так и программные (виртуальные) ресурсы. Разделяемый ресурс, который в каждый момент времени может быть доступен только одному из взаимодействующих процессов, называется ***критическим ресурсом***. (Таковыми ресурсами могут быть как внешнее устройство, так и некая переменная, значение которой может изменяться разными процессами.) Соответственно, часть программы (фактически набор операций), в рамках которой осуществляется работа с критическим ресурсом, называется ***критической секцией (или критическим интервалом)***.

При организации многопроцессного взаимодействия и использования разделяемых ресурсов, возникает целый ряд новых проблем, по сравнению с однопроцессным программированием. Главным образом, эти проблемы связаны с обеспечением корректного доступа к разделяемым ресурсам. Выделяют две важнейшие задачи ОС: распределение ресурсов между процессами и организация корректного доступа (т.е. организация защиты ресурсов, выделенных определенному процессу, от неконтролируемого доступа со стороны других процессов). В частности, под организацией корректного доступа может пониматься требование, декларирующее необходимость обеспечения независимости работы параллельных процессов от порядка и интенсивности доступа этих процессов к разделяемым ресурсам.

Рассмотрим соответствующий **пример** (Рис. 91). Пусть имеется некоторая общая переменная (разделяемый ресурс) **in** и два процесса, которые работают с этой переменной. Пусть в некоторый момент времени процесс **A** присвоил переменной **in** значение X. Затем в некоторый момент процесс **B** присвоил значение Y этой же переменной **in**. Далее оба процесса читают эту переменную, и в обоих случаях процессы прочтут значение Y. То есть в этом случае символ, считанный процессом **A**, был потерян, а символ, считанный процессом **B**, был выведен дважды. Результат выполнения процессов здесь зависит от того, в какой момент осуществляется переключение процессов, и от того, какой конкретно процесс будет выбран следующим для выполнения. Таким образом, требование независимости работы параллельных процессов нарушается. Такие ситуация, когда

процессы конкурируют за разделяемый ресурс, называются ***гонкой процессов*** (***race conditions***).

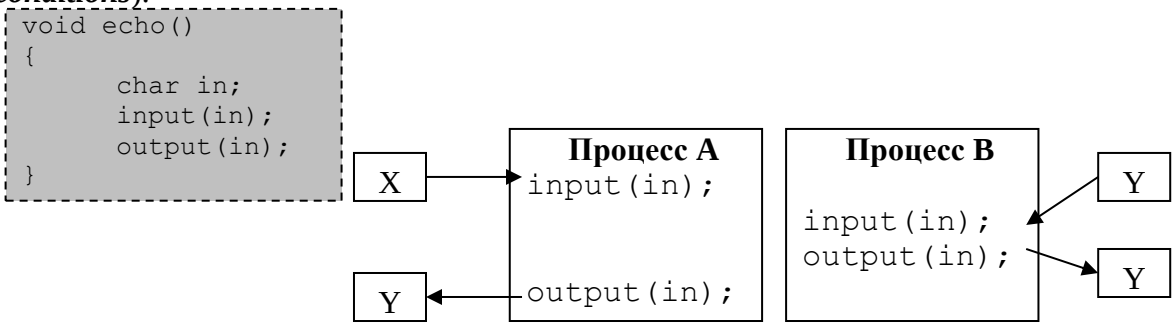


Рис. 91. Пример гонки процессов.

Единственный способ избежать гонок при использовании разделяемых ресурсов — контролировать доступ к любым разделяемым ресурсам в системе. При этом необходимо организовать **взаимное исключение** — т.е. такой способ работы с разделяемым ресурсом, при котором постулируется, что в тот момент, когда один из процессов работает с разделяемым ресурсом, все остальные процессы не могут иметь к нему доступ. Проблему организации взаимного исключения можно сформулировать в более общем виде: задача взаимного исключения сводится к тому, чтобы не допускать ситуации, когда два процесса одновременно находятся в критических секциях, связанных с одним и тем же ресурсом.

Заметим, что вопрос организации взаимного исключения актуален не только для взаимосвязанных процессов, совместно использующих определенные ресурсы для обмена информацией. Выше отмечалось, что возможна ситуация, когда процессы, не подозревающие о существовании друг друга, используют глобальные ресурсы системы, такие как устройства ввода/вывода, принтеры и т.п. В этом случае имеет место конкуренция за ресурсы, доступ к которым также должен быть организован по принципу взаимного исключения.

Для организации модели взаимного исключения используются различные модели синхронизации. Прежде чем рассматривать эти модели, необходимо отметить те проблемы, которые могут возникать при организации взаимного исключения, — это ***блокировки*** и ***тупики***.

Блокировка — это ситуация, когда доступ к разделяемому ресурсу одного из взаимодействующих процессов не обеспечивается из-за активности других, более приоритетных процессов. Отметим следующее. Рассмотрим основанную на приоритетах модель доступа к разделяемому ресурсу, когда более приоритетный запрос на обращение к ресурсу будет обработан быстрее, чем менее приоритетный. И пусть в этой модели работают два процесса, у которых приоритеты доступа к критическому ресурсу разные. Тогда, если более приоритетный процесс будет «часто» выдавать запросы на обращение к ресурсу, может возникнуть ситуация, когда второй процесс будет «вечно» (или достаточно долго) ожидать обработки каждого своего запроса, т.е. этот менее приоритетный процесс будет блокирован.

Тупик, или ***deadlock***, — это ситуация, когда (из-за некорректной организации доступа и разделения ресурсов) конкурирующие за критический ресурс процессы вступают в клинч — происходит взаимоблокировка. Рассмотрим *пример тупиковой ситуации* (Рис. 92).

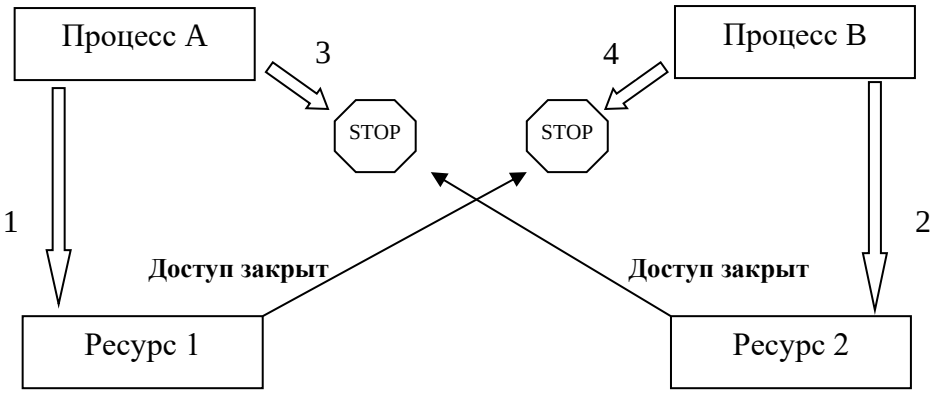


Рис. 92. Пример тупиковой ситуации (deadlock).

Предположим, что есть два процесса **A** и **B**, а также пара критических ресурсов. Пусть в некоторый момент времени процесс **A** вошел в критическую секцию работы с ресурсом 1. Это означает, что доступ любого другого процесса к данному ресурсу будет блокирован (пока процесс **A** не закончит с ним работать). Пусть также в это время процесс **B** войдет в критическую секцию ресурса 2. И этот ресурс также будет блокирован для доступа другим процессам (пока процесс **B** не закончит с ним работать). Пусть процесс **A**, не выходя из критической секции ресурса 1, пытается захватить ресурс 2. Но последний уже захвачен процессом **B**, и процесс **A** блокируется. Аналогично, пусть процесс **B**, не освобождая ресурс 2, пытается захватить ресурс 1 и также блокируется. Это пример простейшего тупика. В этой ситуации каждый из процессов ожидает освобождения недостающего ресурса, но оба ресурса никогда не будут освобождены, и процессы никогда не смогут выполнить необходимые действия. Соответственно, решением в данном случае может быть перезапуск системы или уничтожение обоих или одного из процессов.

2.5.2 Способы организации взаимного исключения

В этом разделе речь пойдет о средствах, позволяющих организовать работу с критическими ресурсами, т.е. организовать такой способ работы с разделяемым ресурсом, при котором в любой момент времени к нему может иметь доступ не более одного процесса. В настоящий момент известно множество средств организации взаимного исключения, среди которых мы рассмотрим **семафоры Дейкстры**, **мониторы Хоара** и **аппарат передачи сообщений**.

Семафоры Дейкстры — это формальная модель организации доступа, предложенная в середине 60-х гг. голландским ученым Дейкстрой. Эта модель базируется на следующей концепции. Имеется специальный тип данных — т.н. ***семафор***. Переменные типа ***семафор*** могут принимать целочисленные значения. Над этими переменными определены следующие **атомарные (неделимые) операции:** опустить семафор **down(**S**)** (или **P(**S**)**) и поднять семафор **up(**S**)** (или **V(**S**)**). Оригинальные обозначения **P** и **V**, данные Дейкстрой и получившие широкое распространение в литературе, являются сокращениями голландских слов *proberen* — проверить и *verhogen* — увеличить.

Операция **down(S)** проверяет значение семафора **S** и, если оно больше нуля, то уменьшает его на **1**. Если же это не так, процесс блокируется, причем связанная с заблокированным процессом операция **down** считается незавершенной.

Операция **up(S)** увеличивает значение семафора на **1**. При этом если в системе присутствуют процессы, блокированные ранее при выполнении **down** на этом семафоре, то один из них разблокируется и завершает выполнение операции **down**, т.е. вновь уменьшает значение семафора. Увеличение значения семафора и, возможно, разблокирование одного из процессов и уменьшение значения **являются атомарной, неделимой операцией.** Выбор процесса для разблокирования никак не оговаривается.

Ещё раз отметим, что операции **up** и **down** являются атомарными (неделимыми), т.е. их выполнение не может быть прервано прерыванием.

Для иллюстрации рассмотренного механизма приведем следующий пример. Рассмотрим некий универсам. Вход в торговый зал магазина возможен лишь для посетителей, имеющих тележку. В магазине имеется N тележек. Итак, в начальный момент (когда магазин открывается) имеется N свободных тележек. Каждый очередной посетитель берет тележку и проходит в зал. Так продолжается, пока не появится (N+1)-ый посетитель, которому тележки уже не хватает. Он войти не может и ждет свободной тележки перед входом в торговый зал. Если приходят еще покупатели, то они также ожидают свободной тележки. Поскольку рассматриваемый формализм, как упоминалось выше, ничего не говорит о выборе очередного заблокированного процесса, то будем считать, что прибывающие в магазин покупатели не становятся в очередь, а стоят в неком «беспорядке» (толпой). Как только один из покупателей с тележкой покидает торговый зал, происходит операция **up** – появляется одна свободная тележка. Эту тележку берет один из ожидающих посетителей и проходит в торговый зал. Это означает, что один из заблокированных клиентов разблокировался и продолжил работу, остальные же продолжают ждать в заблокированном состоянии.

Если тележка была бы одна, то это было бы иллюстрацией организации доступа в режиме взаимного исключения, т.е. в любой момент времени в торговом зале может оказаться лишь один покупатель. Это пример т.н. ***двоичного семафора*** — семафора, максимальное значение которого равно 1. Этот тип семафоров и обеспечивает взаимное исключение.

В приведенном ниже (Рис. 93) **примере** двоичного семафора рассмотрены два процесса, каждый из которых имеет критическую секцию. За счет использования двоичного семафора обеспечивается безопасная работа в критической секции любого из процессов, т.е. если один из них вошел в критическую секцию, то гарантируется, что второй при попытке также войти в свою критическую секцию будет блокирован до тех пор, пока первый не покинет оную.

процесс 1

```
int semaphore;
...
down(semaphore);
/* критическая секция
процесса 1 */
...
up(semaphore);
...
```

процесс 2

```
int semaphore;
...
down(semaphore);
/* критическая секция
процесса 2 */
...
up(semaphore);
...
```

Рис. 93. Пример использования двоичного семафора для организации взаимного исключения.

Заметим, что требование атомарности операций **down** и **up** накладывает ограничения на реализацию семафоров Дейкстры, и зачастую это сложная задача. Существуют различные программные реализации этих операций, но в них атомарность не всегда присутствует.

Таким образом, семафоры Дейкстры — это низкоуровневые средства синхронизации, для корректной практической реализации которых необходимо наличие специальных атомарных семафорных машинных команд.

Рассмотрим теперь модель синхронизации, в которой, в частности, предпринята попытка обойти требование аппаратной поддержки атомарности упомянутых выше операций. Идея ***монитора*** была впервые сформулирована в 1974 г. Хоаром. В отличие от

семафора, монитор является высокоуровневой конструкцией (можно говорить, что это конструкция уровня языка программирования), реализация которой поддерживается системой программирования (компилятором). Монитор — это специализированный модуль, включающий в себя совокупность процедур и функций, а также структуры данные, с которыми работают эти процедуры и функции. При этом данный модуль обладает следующими свойствами:

1. структуры данных монитора доступны только через обращения к процедурам или функциям этого монитора (т.е. монитор представляет собой некоторый аналог объекта в объектно-ориентированных языках и реализует инкапсуляцию данных);
2. считается, что процесс занимает (или входит в) монитор, если он вызывает одну из процедур или функций монитора;
3. в каждый момент времени внутри монитора может находиться не более одного процесса.

Если процесс пытается попасть в монитор, в котором уже находится другой процесс, то (в зависимости от используемой стратегии) он либо получает отказ, либо блокируется, становясь в очередь. Таким образом, чтобы защитить разделяемые структуры данных, их достаточно поместить внутрь монитора вместе с процедурами, представляющими критические секции для обработки этих структур данных.

Монитор представляет собой конструкцию языка программирования и компилятору известно о том, что входящие в него процедуры и данные имеют особую семантику, поэтому первое условие может проверяться еще на этапе компиляции, кроме того, код для процедур монитора тоже может генерироваться особым образом, чтобы удовлетворялось третье условие. Поскольку организация взаимного исключения в данном случае возлагается на компилятор, количество программных ошибок, связанных с организацией взаимного исключения, сводится к минимуму

Бытовой иллюстрацией монитора может служить кабина таксофонного аппарата.

Повторим, что монитор — это языковая конструкция с централизованным управлением (в отличие от семафоров, которые не обладают централизацией). Семафоры и мониторы являются, главным образом, средствами организации работы в однопроцессорных системах либо в многопроцессорных системах с общей памятью. Для многопроцессорных систем с распределенной памятью эти средства не очень подходят. Для них в настоящий момент наиболее часто используется механизм ***передачи сообщений***.

Сразу отметим, что механизм передачи сообщений является универсальным в том смысле, что он предоставляет, как средства организации взаимодействия между процессами, так и средства синхронизации. Механизм ***передачи сообщений*** основан на двух функциональных примитивах: **send** (отправить сообщение) и **receive** (принять сообщение). Данные операции можно разделить по трем характеристикам: модель синхронизации, адресация и формат сообщения.

Синхронизация. Операции посылки/приема сообщений могут быть ***блокирующими*** и ***неблокирующими***. Рассмотрим различные комбинации.

Блокирующий send: процесс-отправитель будет заблокирован до тех пор, пока посланное им сообщение не будет получено.

Блокирующий receive: процесс-получатель будет заблокирован до тех пор, пока не будет получено соответствующее сообщение.

Соответственно, неблокирующие операции, как следует из названия, происходят без блокировок.

Итак, комбинируя различные операции send и receive, мы получаем 4 различных модели синхронизации.

Адресация может быть ***прямой*** или ***косвенной***. При прямой адресации указывается конкретный адрес получателя и/или отправителя (например, когда получатель ожидает сообщения от конкретного отправителя, игнорируя сообщения других отправителей).

В случае косвенной адресации не указывается адрес конкретного получателя при отправке или адрес конкретного отправителя при получении; сообщение «бросается» в некоторый общий пул, в котором могут быть реализованы различные стратегии доступа (FIFO, LIFO и т.д.). Этим пулом может выступать очередь сообщений (FIFO) или почтовый ящик, в котором может быть реализована любая модель доступа. Бытовым примером косвенной адресации может служить модель обслуживающей системы (сотруднику ателье всё равно, чей пиджак он чистит).

Итак, повторимся, что механизм передачи сообщений совмещает два средства: средство передачи информации и средство синхронизации. Этот аппарат является базовым средством организации взаимодействия процессов в многопроцессорных системах с распределенной памятью.

Механизм передачи сообщений реализуется на базе интерфейсов передачи сообщений MPI. На основе этих интерфейсов строятся почти все кластерные системы (т.е. системы с распределенной памятью), а также MPI может работать и в системах с общей памятью.

2.5.3 Классические задачи синхронизации процессов

Классические задачи синхронизации процессов отражают разные модели взаимодействия и демонстрируют использование механизма семафоров для организации такого взаимодействия.

2.5.3.1 «Обедающие философы»

Пусть существует круглый стол (Рис. 94), за которым сидит группа философов: они пришли пообщаться и покушать. Кушают они спагетти, которое находится в общей миске, стоящей в центре стола. Для приема пищи они пользуются двумя вилками: одна в левой руке, другая — в правой. Вилки располагаются по одной между каждыми двумя философами. Каждый из философов некоторое время размышляет, затем берет две вилки и ест спагетти, затем кладёт вилки на стол и опять размышляет, и так далее. Каждый из них ведет себя независимо от других. Философы должны совместно использовать имеющиеся у них вилки (ресурсы). Задача состоит в организации доступа к вилкам.

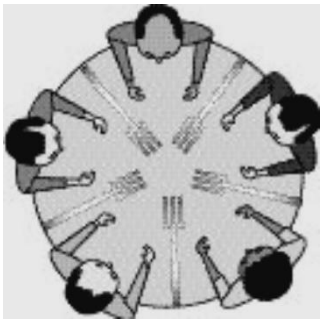


Рис. 94. Обещающие философы.

Итак, данная задача иллюстрирует модель доступа равноправных процессов к общему ресурсу, и ставится вопрос, как организовать **корректную** работу такой системы.

Рассмотрим простейшее решение данной задачи, использующее семафоры. Когда один из философов хочет есть, он берет вилку слева от себя, если она в наличии, а затем — вилку справа от себя. Закончив есть, он возвращает обе вилки на свои места. Данный алгоритм может быть представлен следующим способом.

```
#define N 5 /* количество философов */
```

```
void Philosopher(int i) /* i – номер философа от 0 до 4 */
{
     while(TRUE)
     {
          Think();/*философ думает */
          TakeFork(i); /* взятие левой вилки */
          TakeFork((i + 1) % N); /* взятие правой вилки */
          Eat();/* философ ест */
          PutFork(i); /* освобождение левой вилки */
          PutFork((i + 1) % N); /* освобождение правой вилки */
     }
}
```

Функция take_fork(i) описывает поведение философа по захвату вилки: он ждет, пока указанная вилка не освободится, и забирает ее.

Однако вышеобозначенное решение может привести к тупиковой ситуации. Что произойдет, если все философы захотят есть в одно и то же время? Каждый из них получит доступ к своей левой вилке и будет находиться в состоянии ожидания второй вилки до бесконечности, т.е. возникнет ситуация тупика. Другим решением может быть алгоритм, который обеспечивает доступ к вилкам только четырем из пяти философов. В этом случае всегда среди четырех философов по крайней мере один будет иметь доступ к двум вилкам. Данное решение не имеет тупиковой ситуации. Алгоритм решения может быть представлен следующим образом.

```
#define N 5 /* количество философов */
#define LEFT (i-1)%N /* номер левого соседа для i-го философа */
#define RIGHT (i+1)%N /*номер правого соседа для i-го философа
*/
/* состояния философов:
«думает»,
«голоден»,
«кушает»
*/
#define THINKING 0
#define HUNGRY 1
#define EATING 2

/*
массив состояний каждого из философов, инициализированный нулями
*/
int state[N]  = {0};

/* семафор для доступа в критическую секцию */
semaphore mutex = 1;
/*
массив семафоров по одному на каждого из философов,
инициализированный нулями
*/
semaphore s[N] = {0};

/* Процесс-философ (i = 0..N-1) */
void Philosopher(int i)
{
```

```
     while(TRUE)
     {
          Think();
          /* философ берёт обе вилки или блокируется */
          TakeForks(i);
          Eat();
          PutForks(i);
     }
}

/* получение вилок */
void TakeForks(int i)
{
     /* вход в критическую секцию */
down(&mutex);
     state[i] = HUNGRY;
     Test(i);
     /* выход из критической секции */
up(&mutex);
     down(&s[i]);
}

/* освобождение вилок */
void PutForks(int i)
{
     /* вход в критическую секцию */
down(&mutex);
     state[i] = THINKING;
     Test(LEFT);
     Test(RIGHT);
     /* выход из критической секции */
up(&mutex);
}
/* функция проверки возможности получения вилок –
проверяется состояние соседей данного философа */
void Test(int i)
{
     if(state[i] == HUNGRY &&
          state[LEFT] != EATING &&
          state[RIGHT] != EATING)
     {
          state[i] = EATING;
          up(&s[i]);
     }
}
```

В этом решении каждый философ живет по аналогичному циклическому распорядку: размышляет некоторое время, затем берет вилки, кушает, кладет вилки. Рассмотрим процедуру получения вилок (TakeForks). Опускается семафор mutex, который используется для синхронизации входа в критическую секцию. Внутри критической секции меняем состояние философа (помечаем его состоянием HUNGRY). Затем предпринимается попытка начать есть (вызывается функция Test). Функция Test проверяет, что если *i*-ый философ голоден, а его соседи в данный момент не едят (т.е. правая и левая вилки

свободны), то этот философ начинает прием пищи (состояние EATING), а его семафор поднимается (заметим, что изначально этот семафор инициализирован нулем). После этого мы возвращаемся обратно в функцию TakeForks, в которой далее происходит выход из критической секции (поднимаем семафор mutex), а затем опускаем семафор этого философа. Если внутри функции Test философу удалось начать прием пищи, то семафор поднят, и операция down обнулит его, не блокируясь. Если же функция Test не изменит состояние философа, а также не поднимет его семафор, то операция down (в функции TakeForks) в этой точке заблокируется до тех пор, пока оба соседа не освободят вилки.

Внутри функции освобождения вилок PutForks сначала происходит опускание семафора mutex, происходит вход в критическую секцию. Затем меняется статус философа (на статус THINKING), после чего проверяем его соседей: если любой из них был заблокирован лишь из-за того, что наш *i*-ый философ забрал его вилку, то мы его разблокируем, и он начинает прием пищи. После этого происходит выход из критической секции путем подъема семафора mutex.

Заметим, что использование *механизма взаимоисключающего нахождения внутри критической секции* (за счет семафора mutex) гарантирует, что не возникнет ситуация, когда два процесса, соответствующие соседним философам, будут так спланированы на обработку на процессоре, что функция Test в каждом из них проработает и разрешит каждому из них начать прием пищи (что, конечно же, является ошибкой). Если же этого механизма не будет, то возможно, что один из процессов-соседей входит в Test, делает проверку на возможность начала приема пищи. Проверка дает истинное значение, управление переходит к первой команде внутри if-блока. После этого происходит смена процесса на процессоре, управление получает сосед этого философа. Тот тоже делает проверку внутри функции Test и также получает положительный ответ, и управление переходит к первой инструкции if-блока. Дальнейшая работа будет некорректной.

2.5.3.2 «Читатели и писатели».

Представим произвольную систему резервирования ресурса. Например, это может быть система резервирования места в гостинице. В данной системе существует два типа процессов для работы с информацией. Одни процессы могут читать информацию, а другие — ее изменять, корректировать. Соответственно, возникает все тот же вопрос, как организовать корректную совместную работу этих процессов. Это означает, что в любой момент времени читать данные могут любое количество процессов-читателей, но если процесс-писатель начал свою работу, то все остальные процессы (и читатели, и писатели) будут блокированы на входе в систему. Задача заключается в планировании работы такой системы.

Рассмотрим модельную реализацию данной задачи для следующей стратегии: будем считать, что наиболее приоритетными являются читающие процессы. То есть процесс-писатель будет ожидать момента, когда все желающие процессы-читатели окончат свои действия в системе и покинут ее.

/* семафор для доступа в критическую секцию - контроль за доступом к «rc» (разделямый ресурс) */
semaphore mutex = 1;

/* семафор для доступа к базе данных */
semaphore db = 1;

/* количество читателей внутри хранилища */

```
int rc = 0;

/* процесс-читатель */
void Reader(void)
{
        down(&mutex); /* получить эксклюзивный доступ к «rc»*/

        rc = rc + 1; /* еще одним читателем больше */
        /* если это первый читатель - заблокировать доступ к базе */
        if  (rc == 1)  down(&db);

        up(&mutex); /*освободить ресурс rc */
        ReadDataBase(); /* доступ к данным */
        down(&mutex); /*получить эксклюзивный доступ к «rc»*/
        rc = rc – 1; /* теперь одним читателем меньше */

        /*если это последний читатель - разблокировать доступ к базе */
        if(rc == 0)  up(&db);
        up(&mutex); /*выход из критической секции  rc*/

        UseDataRead(); /* некритическая секция */
}

/* процесс-писатель */
void Writer(void)
{
        ThinkUpData(); /* некритическая секция */
        down(&db); /* получить эксклюзивный доступ к данным*/
        WriteDataBase(); /* записать данные */
        up(&db); /* отдать эксклюзивный доступ */
}
```

В приведенном решении **процесс-читатель** в каждом цикле своей работы входит в критическую секцию работы с rc (опускает семафор mutex), увеличивает счетчик читателей, находящихся в хранилище, на 1. Затем проверяет, что является ли он первым читателем (т.е. в данный момент он единственный клиент в хранилище). Если первый, то он опускает семафор *db*, тем самым, препятствуя писателям войти в систему.

Если же семафор *db* уже был опущен, то это означает, что в данный момент в хранилище присутствует писатель, и этот первый читатель заблокируется на этой операции, ожидая выхода писателя из системы. (Заметим, что эта блокировка происходит внутри критической секции, поэтому остальные читатели будут блокироваться на закрытии семафора mutex.) После этого происходит выход из критической секции (открываем семафор mutex), чтение информации из хранилища. Затем производятся обратные действия по выходу из хранилища, которые также происходят внутри критической секции.

Итак, на выходе мы уменьшаем число читателей в хранилище, и если некоторый читатель является последним клиентом в библиотеке, то происходит открытие семафора db, разрешая работу писателям (которые к этому моменту могли быть заблокированы на входе). В конце цикла работы читатель обрабатывает полученные данные из хранилища, после чего цикл повторяется. Писатель в начале каждого цикла своей работы подготавливает данные для сохранения, затем пытается войти в хранилище, опуская семафор db. Если в хранилище кто-то есть, то он будет ожидать, пока последний клиент

(независимо от того, читатель это или писатель) не покинет хранилище. После этого он производит корректировку данных в хранилище и покидает его, поднимая семафор db.

На первый взгляд здесь реализована корректная стратегия приоритетности читателя перед писателем. Однако возможна следующая ситуация. Рассмотрим следующую последовательность. Пусть, R1 - читатель, а W1 и W2 – два писателя. Рассмотрим возможную ситуацию:

1. W1 начинает работу, опускает семафор db и переходит к записи данных.
2. Пока W1 пишет данные, в системе появляются первый читатель R1 и новый писатель W2, которые доходят до команды down(&db) и блокируются в ней (поскольку ранее W1 опустил db перед началом своей записи).
3. Теперь W1 завершает запись и поднимает семафор db. Тогда, один из процессов (**любой!**), который заблокирован в операции down (либо R1, либо W2) будет разблокирован. Какой конкретно процесс будет разблокирован – неизвестно. Т.е. возможна ситуация разблокировки W2, что нарушит первоначальное требование приоритетности читателей.

Рассмотрим уточнение нашей задачи и коррекцию программного кода. Все процессы писатели, которые вошли в систему **до входа первого процесса читателя, должны быть обработаны.**

Дополнительно введем в код следующие глобальные компоненты:

- wc - счетчик процессов-писателей, которые вошли в систему до первого процесса-читателя (инициирован 0);
- mutw - семафор для организации критической секции для работы с wc (инициирован 1);
- mutflag - симафор (инициирован 0), для блокирования первого читателя, до тех пор пока не обработаются все писатели, которые вошли в систему до первого;
- r_block - признак (флаг) блокировки первого процесса-читателя (1 заблокирован; изначально инициализирован 0) в случае необходимости ожидания завершения вошедших ранее процессов-писателей.

```
/* семафор для доступа в критическую секцию - контроль за
доступом к «rc» (разделямый  ресурс) */
semaphore mutex = 1;
/* количество читателей внутри хранилища */
int rc = 0;

/* количество писателей до входа первого читателя */
int wc = 0;
semaphore mutw = 1; /* семафор для работы с wc */


/* семафор для пропуска wc писателей,   */
/* вошедших до входа первого читателя   */
semaphore mutflag = 0;

/*блокировка первого читателя: 1 – заблокирован; 0 - нет*/
int r_block = 0;
```

```
/* семафор для доступа к базе данных */
semaphore db = 1;


/* процесс-читатель */
void Reader(void)
{
          down(&mutex); /* получить эксклюзивный доступ к «rc»*/
          rc = rc + 1; /* еще одним читателем больше */

          /* если это первый читатель, нужно заблокировать
          эксклюзивный доступ к базе */
          if(rc == 1){

               /*ждем выход писателей, вошедших до 1-го чит.*/
               down(&mutw);
               if( wc != 0){  r_block = 1; up(&mutw);
                              down(mutflag); /* блок. 1-го чит.*/
                              r_block = 0;

               } else up(&mutw);

               /* в системе писателей нет*/

               down(&db);     /* блокировка базы */
          };
          up(&mutex); /*освободить ресурс rc */

          ReadDataBase(); /* доступ к данным */

          down(&mutex); /*получить эксклюзивный доступ к «rc»*/
          rc = rc - 1; /* теперь одним читателем меньше */

          /*если это был последний читатель, разблокировать
          эксклюзивный доступ к базе данных */
          if(rc == 0)up(&db);

          up(&mutex); /*освободить разделяемый ресурс rc*/

          UseDataRead(); /* некритическая секция */
}

/* процесс-писатель */
void Writer(void)
{


          /* w_flag == 1 - писатель вошел после первого читателя
          int w_flag = 1;

          down(&mutex);
          /* подсчет писателей до прихода первого читателя*/
          if( (rc == 0)) {
```

```
            /* если читателей еще нет */
            down(&mutw); wc++ ;
            w_flag = 0; /* писатель вошел до 1-го читателя*/
            up(&mutw);
        };
        up(&mutex);

        /* время в это может войти первый читатель (rc == 1),
        но он еще не  успел заблокировать данные */
        ThinkUpData(); /* подготовка данных для сохранения */

        down(&db); /* получить эксклюзивный доступ к данным*/
        WriteDataBase(); /* записать данные */
        up(&db); /* отдать эксклюзивный доступ */


        down(&mutw);

            if(w_flag == 0)

                /* вошел до первого читателя*/
                wc--;

            if((r_block == 1)&&(wc == 0))
                /* писатели были и закончились*/
                /* разблокировка первого читателя */
                up(&mutflag);

        up(&mutw);

}
```

Процесс-читатель. Если вошел первый читатель, и к этому моменту в системе есть процессы-писатели (wc #0), то он должен дождаться завершения работы этих писателей. которые вошли туду до первого процесса-читателя – счетчик таких процессов-писателей). В этом случае процесс-читатель устанавливает признак блокировки (r_block = 1) и блокируется на семафоре mutflag (ожидает завершения всех вошедших до первого читателя процессов-писателея). После разблокировки, когда все процессы-писатели завершат работу, устанавливается признак (r_block = 0).

Процесс-писатель. Если читателей в системе нет, увеличивает счетчик писателей wc, вошедших в систему до первого читателя на 1, а также устанавливает w_flag = 0 (это означает, что при завершении данного писателя мы должны будем вычесть из wc единичку, т.е данный писатель вошел в систему до первого читателя). В противном случае данный писатель заблокируется на входе в критическую секцию, организованную семафором mutex (означает, то, что, либо в систему вошел первый процесс-читатель и началось ожидание доработки всех предшествующих писателей, либо он заблокируется на доступе к данным на семафоре db).

Далее, после всех необходимых разблокировок происходит доработка завершения писателя. Если процесс-писатель вошел до первого читателя признак w_flag == 0 уменьшаем значение wc. Признаком того, что первый процесс-читатель вошел и ждет выхода писателей является r_block равное 1. Если при этом счетчик wc равен 0 (все писатели, вошедшие в систему до первого читателя, вышли), то мы разблокируем первого

читателя up(&muflag). Если же значение r_block равен 0, это означает, что в системе нет заблокированного первого процесса-читателя.

Данное решение также не является идиальным. Есть вариант некорректной работы и данного решения. Задача- попробовать самостоятельно найти некорректность и варианты ее ликвидации.

2.5.3.3 «Спящий парикмахер».

Рассмотрим парикмахерскую, в которой работает один парикмахер, имеется одно кресло для стрижки и несколько кресел в приемной для посетителей, ожидающих своей очереди. Если в парикмахерской нет посетителей, парикмахер засыпает прямо на своем рабочем месте. Появившийся посетитель должен его разбудить, в результате чего парикмахер приступает к работе. Если в процессе стрижки появляются новые посетители, они должны либо подождать своей очереди, либо покинуть парикмахерскую, если в приемной нет свободного кресла для ожидания (т.е. это стратегия обслуживания с отказами). Задача состоит в том, чтобы корректно запрограммировать поведение парикмахера и посетителей.

Данная задача является иллюстрацией модели клиент-сервер с ограничением на длину очереди клиентов.

Рассмотрим реализацию данной модели. Понадобится 3 семафора: *customers* — подсчитывает количество посетителей, ожидающих в очереди, *barbers* — статус парикмахера (0 – спит/занят, 1 –готов к работе)[1] и *mutex* — используется для синхронизации доступа к разделяемой переменной *waiting*. Переменная *waiting*, как и семафор *customers*, содержит количество посетителей, ожидающих в очереди. Эта переменная используется в программе для того, чтобы иметь возможность проверить, имеется ли свободное кресло для ожидания, и при этом не заблокировать процесс, если кресла не окажется. Заметим, что как и в предыдущем примере, эта переменная является разделяемым ресурсом, и доступ к ней охраняется семафором *mutex*.

```
/* количество стульев в комнате ожидания */
#define CHAIRS 5
/* наличие посетителей, ожидающих парикмахера */
semaphore customers = 0;
/* состояние парикмахера: спит/занят или готов к работе (0 или
1)*/
semaphore barbers = 0;
/* семафор для доступа в критическую секцию - контроль за
доступом к переменной waiting */
semaphore mutex = 1;
/* количество ожидающих посетителей */
int waiting = 0;

/* Брадобрей */
void Barber(void)
{
     while(TRUE)
     {
          /* если customers == 0, т.е. посетителей нет, то
          заблокируемся до появления посетителя */
```

[1] В принципе возможно расширение задачи для случая N парикмахеров, в этом случае, с незначительными коррекциями программ, barbers – количество свободных парикмахеров.

```
            down(&customers);
            down(&mutex); /* получаем доступ к waiting */
            /* уменьшаем кол-во ожидающих клиентов */
            waiting = waiting - 1;
            up(&barbers); /* парикмахер готов к работе */
            up(&mutex); /* освобождаем ресурс waiting */
            CutHair(); /* процесс стрижки */
      }
}

/* Посетитель */
void Customer(void)
{
      down(&mutex); /* получаем доступ к waiting */
      if(waiting < CHAIRS) /* есть место для ожидания */
      {
            /* увеличиваем кол-во ожидающих клиентов */
            waiting = waiting + 1;
            /* если парикмахер спит, это его разбудит */
            up(&customers);
            up(&mutex); /* освобождаем ресурс waiting */
            /* если парикмахер занят, переходим в состояние
            ожидания, иначе - занимаем парикмахера */
            down(&barbers);
            /* занять место и перейти к стрижке */
            GetHaircut();
      }
      else
      {
            /*нет свободного кресла для ожидания - придется уйти */
            up(&mutex);
      }
}
```

Процесс-парикмахер сначала опускает семафор *customers*, уменьшив тем самым количество ожидающих посетителей на 1. Если в комнате ожидания никого нет, то он «засыпает» в своем кресле, пока не появится клиент, который его разбудит. Затем парикмахер входит в критическую секцию, уменьшает счетчик ожидающих клиентов, поднимает семафор *barbers*, сигнализируя клиенту о своей готовности его обслужить, а потом выходит из критической секции. После описанных действий он начинает стричь волосы посетителю.

Посетитель парикмахерской входит в критическую секцию. Находясь в ней, он проверяет, есть ли свободные места в зале ожидания. Если нет, то он просто уходит (покидает критическую секцию, поднимая семафор *mutex*). Иначе он увеличивает счетчик ожидающих процессов и поднимает семафор customers. Если же этот посетитель является единственным в данный момент клиентом брадобрея, то он этим действием разбудит брадобрея. После этого посетитель выходит из критической секции и «захватывает» брадобрея (опуская семафор *barbers*). Если же этот семафор опущен, то клиент будет дожидаться, когда брадобрей его поднимет, известив тем самым, что готов к работе. В конце клиент обслуживается (*GetHaircut*).

3 Межпроцессное взаимодействия в ОС Unix

3.1 Базовые средства реализации взаимодействия процессов в ОС Unix

Сразу необходимо отметить, что во всех иллюстрациях организаций взаимодействия процессов будем рассматривать полновесные процессы, т.е. те «классические» процессы, которые представляются в виде обрабатываемой в системе программы, обладающей эксклюзивными правами на оперативную память, а также правами на некоторые дополнительные ресурсы.

Если посмотреть на проблемы взаимодействия процессов, то можно выделить две группы взаимодействия. Первая группа — это взаимодействие процессов, функционирующих под управлением одной ОС на одной локальной ЭВМ. Вторая группа взаимодействия — это взаимодействие в пределах сети. В зависимости от того, к какой группе относится тот или иной механизм, он будет обладать соответствующими свойствами и особенностями.

Рассмотрим **взаимодействие в рамках локальной ЭВМ (под управлением одной ОС)**. Прежде всего, возникает общая для обеих упомянутых групп **проблема именования взаимодействующих процессов**, которая заключается в ответе на вопрос, как, т.е. посредством каких механизмов, взаимодействующие процессы смогут «найти друг друга». В рамках взаимодействия под управлением одной ОС можно выделить две основные группы решений данной задачи (Рис. 95).

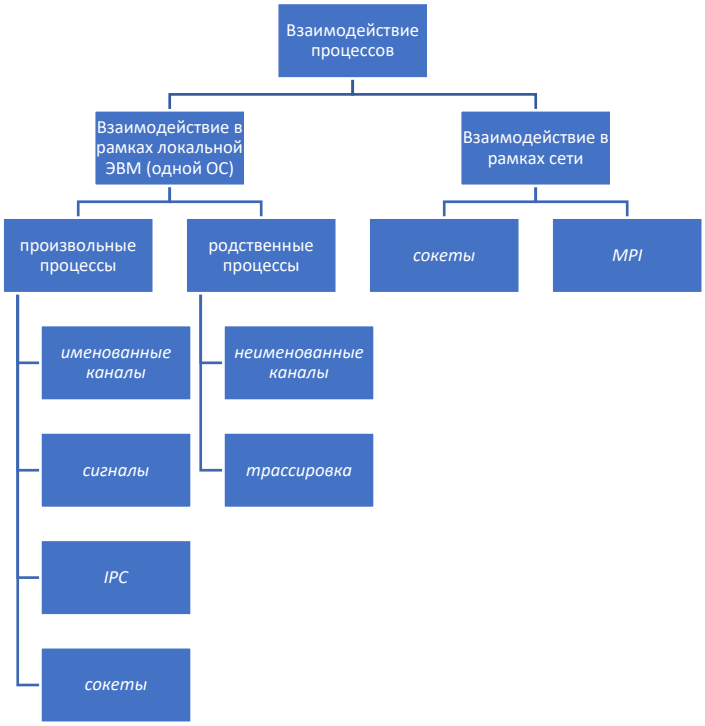


Рис. 95. Способы организации взаимодействия процессов.

Первая группа решений основана на **взаимодействии родственных процессов**. При взаимодействии родственных процессов, т.е. процессов, связанных некоторой иерархией родства, ОС обычно предоставляет возможность наследования сыновними процессами некоторых характеристик родительских процессов. И именно за счет наследования различных характеристик возможно реализовать то самое именование. К примеру, в ОС

Unix можно передавать по наследству от отца сыну дескрипторы открытых файлов. В данном случае именование будет *неявным*, поскольку не указываются непосредственно имена процессов.

Другим решением в рамках данной группы взаимодействующих родственных процессов является взаимодействие по цепочке предок–потомок, причем известно, кто из процессов является предком, а кто — потомком. В этом случае существует возможность процессу-предку обращаться к своему потомку посредством явного именования. В качестве имени, например, может выступать идентификатор процесса (PID). А потомок, зная имя предка, может также к нему обратиться.

Так или иначе, но данная группа реализаций взаимодействия родственных процессов основана на том факте, что некоторая необходимая для взаимодействия информация может быть передана по наследству.

Следующая группа — это **взаимодействие произвольных процессов** в рамках одной локальной машины. Очевидно, что в этом случае отсутствует факт наследования, и поэтому для решения проблемы именования логично использовать следующие механизмы. Во-первых, *прямое именование*, когда процессы для указания своих партнеров по взаимодействию используют уникальные имена партнеров (например, используя идентификаторы процессов или же по-иному: PID привязывается к некоторому новому уникальному имени, и обращение при взаимодействии происходит с использованием системы этих новых имен). Во-вторых, это может быть взаимодействие посредством *общего ресурса*. Но в этом случае встает проблема именования этих общих ресурсов.

Итак, мы рассмотрели модели взаимодействия процессов в рамках локальной машины. ОС Unix предоставляет целый спектр механизмов взаимодействия по каждой из указанных групп. В частности, для взаимодействия родственных процессов могут быть использованы такие механизмы, как ***неименованные каналы*** и ***трассировка***.

Неименованный канал — это некоторый ресурс, наследуемый сыновьями процессами, причем этот механизм может быть использован для организации взаимодействия произвольных родственников (т.е., условно говоря, можно организовать неименованный канал между «сыном» и его «племянником», и т.п.).

Неименованные каналы — пример симметричного взаимодействия, т.е., несмотря на то, что ресурс неименованного канала передается по наследству, взаимодействующие процессы в общем случае, абстрагируясь от семантики программы, имеют идентичные права.

Другой моделью взаимодействия является несимметричная модель, которую иногда называют ***модель «главный–подчиненный»***. В этом случае среди взаимодействующих процессов можно выделить процессы, имеющие больше полномочий, чем у остальных. Соответственно, у главного процесса (или процессов) есть целый спектр механизмов управления подчиненными процессами.

Для организации взаимодействия произвольных процессов система предоставляет целый спектр средств взаимодействия, среди которых преобладают средства симметричного взаимодействия (т.е. процессам при взаимодействии предоставляются равные права).

Именованные каналы — это ресурс, принадлежащий взаимодействующим процессам, посредством которого осуществляется взаимодействие. При этом не обязательно знать имена процессов-партнеров по взаимодействию.

Передача ***сигналов*** — это средство оказания воздействия одним процессом на другой процесс в общем случае (в частности, одним из процессов в этом виде взаимодействия может выступать процесс операционной системы). При этом используются непосредственные имена процессов.

Система ***IPC*** (Inter-Process Communication) предоставляет взаимодействующим процессам общие разделяемые ресурсы (среди которых ниже будут рассмотрены ***общая память***, ***массив семафоров*** и ***очередь сообщений***), посредством которых осуществляется

взаимодействие процессов. Отметим, что система IPC является некоторым альтернативным решением именованным каналам.

Аппарат ***сокетов*** — унифицированное средство организации взаимодействия. На сегодняшний момент сокеты — это не столько средства ОС Unix, сколько стандартизированные средства межмашинного взаимодействия. В аппарате сокетов именование осуществляется посредством связывания конкретного процесса (его идентификатора PID) с конкретным сокетом, через который и происходит взаимодействие.

Итак, мы перечислили некоторые средства взаимодействия процессов в рамках одной локальной машины (точнее сказать, в рамках ОС Unix), но это лишь малая часть существующих в настоящий момент средств организации взаимодействия.

Второй блок организации взаимодействия — это **взаимодействие в пределах сети**. В данном случае ставится задача организовать взаимодействие процессов, находящихся на разных машинах под управлением различных операционных систем. Та же проблема именования процессов в рамках сети решается достаточно просто.

Пусть у нас есть две машины, имеющие сетевые имена A и B. Пусть на этих машинах работают процессы P_1 и P_2 соответственно. Тогда, чтобы именовать процесс в сети, достаточно использовать связку «сетевое имя машины + имя процесса на этой машине». В нашем примере это будут пары ($A–P_1$) и ($B–P_2$).

Но тут встает следующая проблема. В рамках сети могут взаимодействовать машины, находящиеся под управлением операционных систем различного типа (т.е. в сети могут оказаться Windows-машины, FreeBSD-машины, Macintosh-машины и пр.). И система именования должна быть построена так, чтобы обеспечить возможность взаимодействия произвольных машин, т.е. это должно быть стандартизованным (унифицированным) средством. На сегодняшний день наиболее распространенными являются ***аппарат сокетов*** и ***система MPI***.

Аппарат сокетов можно рассматривать как базовое средство организации взаимодействия. Этот механизм лежит на уровне протоколов взаимодействия. Он предполагает для обеспечения взаимодействия использование т.н. ***сокетов***, и взаимодействие осуществляется между сокетами. Конкретная топология взаимодействующих процессов зависит от задачи (можно организовать общение одного сокета со многими, можно установить связь один–к–одному и т.д.). В конечном счете, именование сокетов также зависит от топологии: в одном случае необходимо знать точные имена взаимодействующих сокетов, в другом случае имена некоторых сокетов могут быть произвольными (например, в случае клиент–серверной архитектуры обычно имена клиентских сокетов могут быть любыми).

Система MPI (***интерфейс передачи сообщений***) также является достаточно распространенным средством организации взаимодействия в рамках сети. Эта система иллюстрирует механизм передачи сообщений, речь о котором шла выше (см. раздел 2.5.2). Система MPI может работать на локальной машине, в многопроцессорных системах с распределенной памятью (т.е. может работать в кластерных системах), в сети в целом (в частности, в т.н. GRID-системах).

Далее речь пойдет о конкретных средствах взаимодействия процессов (как в ОС Unix, так и в некоторых других).

3.1.1 Сигналы

В ОС Unix присутствует т.н. аппарат ***сигналов***, позволяющий одним процессам оказывать воздействия на другие процессы. Сигналы могут рассматриваться как средство уведомления процесса о наступлении некоторого события в системе. В некотором смысле аппарат сигналов имеет аналогию с аппаратом прерываний, поскольку последний есть также уведомление системы о том, что в ней произошло некоторое событие. Прерывание вызывает определенную детерминированную последовательность действий системы, точно

так же приход сигнала в процесс вызывает в нем определенную последовательность действий.

Инициатором отправки сигнала процессу может быть как процесс, так и сама ОС. Сигналы, посылаемые ОС, уведомляют о наступлении некоторых строго предопределенных ситуаций (как, например, завершение порожденного процесса, попытка выполнить недопустимую машинную инструкцию, попытка недопустимой записи в канал и т.п.), при этом каждой такой ситуации сопоставлен свой сигнал. Для иллюстрации приведем следующий пример. Пусть в ходе выполнения некоторого процесса произошло деление на ноль, вследствие чего в системе происходит прерывание, управление передается операционной системе. ОС «видит», что это прерывание «деление на ноль», и отправляет сигнал процессу, в теле которого произошла данная ошибка. Дальше процесс реагирует на получение сигнала, но об этом чуть позже.

Инициатором посылки сигнала может выступать другой процесс. В качестве примера можно привести следующую ситуацию. Пользователь ОС Unix запустил некоторый процесс, который в некоторый момент времени зацикливается. Чтобы снять этот процесс со счета, пользователь может послать ему сигнал об уничтожении (например, нажав на клавиатуре комбинацию клавиш Ctrl+C, а это есть команда интерпретатору команд послать код сигнала SIGINT). В данном случае процесс интерпретатора команд пошлет сигнал пользовательскому процессу.

Аппарат сигналов является механизмом асинхронного взаимодействия, момент прихода сигнала процессу заранее неизвестен. Однако процесс может предвидеть возможность получения того или иного сигнала и установить определенную реакцию на его приход. В этом плане сигналы можно рассматривать как программный аналог аппаратных прерываний.

Так же, как и аппарат прерываний, имеющий фиксированное количество различных прерываний, Unix-системы имеют фиксированный набор сигналов. Перечень сигналов, реализованных в конкретной операционной системе, обычно находится в файле **<signal.h>**. В этом файле перечисляется набор пар «имя сигнала — его целочисленное значение». Ниже приведено несколько примеров (следует заметить, что в разных версиях UNIX имена сигналов могут различаться).

2 - SIGINT /*прерывание*/
3 - SIGQUIT /*аварийный выход*/
9 - SIGKILL /*уничтожение процесса*/
14 - SIGALRM /*прерывание от таймера*/
18 – SIGCHLD /* процесс-потомок завершился */

При получении процессом сигнала возможны три типа реакции на него (Рис. 96). Во-первых, это ***обработка сигнала по умолчанию***. В подавляющем большинстве случаев обработка сигнала по умолчанию означает завершение процесса. В этом случае системным кодом завершения процесса становится номер пришедшего сигнала.

Во-вторых, процесс может ***перехватывать обработку пришедшего сигнала***. Если процесс получает сигнал, то вызывается функция, принадлежащая телу процесса, которая была специальным образом зарегистрирована в системе как ***обработчик сигнала***. Следует отметить, что часть реализованных в ОС сигналов можно перехватывать, а часть сигналов перехватывать нельзя. Примером неперехватываемого сигнала может служить сигнал **SIGKILL** (код 9), предназначенный для безусловного уничтожения процесса. А упомянутый выше сигнал **SIGINT** (код 2) перехватить можно.

В-третьих, сигналы можно ***игнорировать***, т.е. приход некоторых сигналов процесс может проигнорировать. Как и в случае с перехватываемыми сигналами, часть сигналов можно игнорировать (например, **SIGINT**), а часть — нет (например, **SIGKILL**).

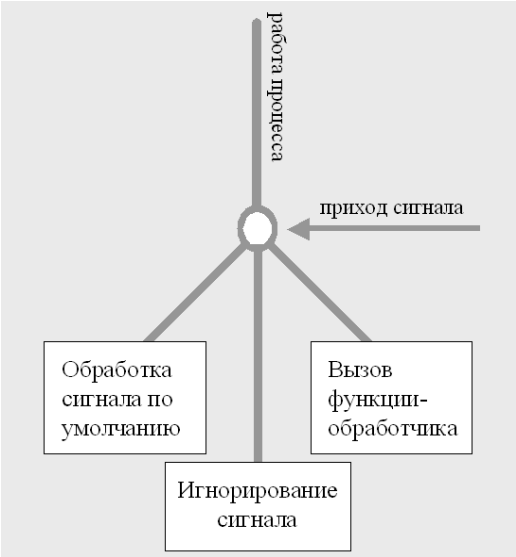


Рис. 96. Реакции на получение процессом сигнала.

Если в процесс одновременно доставляется несколько различных сигналов, то порядок их обработки не определен. Если же обработки ждут несколько экземпляров одного и того же сигнала, то ответ на вопрос, сколько экземпляров будет доставлено в процесс – все или один – зависит от конкретной реализации ОС.

Отдельного рассмотрения заслуживает ситуация, когда сигнал приходит в момент выполнения системного вызова. Обработка такой ситуации в разных версиях UNIX реализована по-разному, например, обработка сигнала может быть отложена до завершения системного вызова; либо системный вызов автоматически перезапускается после его прерывания сигналом; либо системный вызов вернет –1, а в переменной **errno** будет установлено значение **EINTR**.

Для отправки сигнала процессу в ОС Unix имеется системный вызов **kill()**.

```
#include <sys/types.h>
#include <SIGNAL.H>
INT KILL (PID_T PID, INT SIG);
```

Первый параметр вызова – идентификатор процесса, которому посылается сигнал (в частности, процесс может послать сигнал самому себе). Существует также возможность одновременно послать сигнал нескольким процессам. Например, если значение этого параметра есть 0, сигнал будет передан всем процессам, которые принадлежат той же группе, что и процесс, посылающий сигнал, за исключением процессов с идентификаторами 0 и 1.

Во втором параметре передается номер посылаемого сигнала. Если этот параметр равен 0, то будет выполнена проверка корректности обращения к **kill()** (в частности, существование процесса с идентификатором **pid**), но никакой сигнал в действительности посылаться не будет.

Если процесс-отправитель не обладает правами привилегированного пользователя, то он может отправить сигнал только тем процессам, у которых реальный или эффективный идентификатор владельца процесса совпадает с реальным или эффективным идентификатором владельца процесса-отправителя.

При успешном завершении системный вызов **kill()** возвращает 0, иначе возвращается -1.

Чтобы установить реакцию процесса на приходящий сигнал, используется системный вызов *signal()*.

```
#include <signal.h>
```

```
void (*signal  ( int sig, void (*disp) (int))) (int);
```

Аргумент ***sig*** определяет номер сигнала, реакцию на приход которого надо установить. Второй аргумент ***disp*** определяет новую реакцию на приход указанного сигнала. То есть ***disp*** — это либо определенная пользователем функция-обработчик сигнала, либо одна из констант: **SIG_DFL** и **SIG_IGN**. Первая из них указывает, что необходимо установить для данного сигнала обработку по умолчанию, т.е. стандартную реакцию системы, а вторая — что данный сигнал необходимо игнорировать. При успешном завершении системный вызов возвращает указатель на предыдущий обработчик данного сигнала (он может использоваться процессом, например, для восстановления прежней реакции на сигнал).

Как видно из прототипа вызова **signal()**, определенная пользователем функция-обработчик сигнала должна принимать один целочисленный аргумент (в нем будет передан номер обрабатываемого сигнала), и не возвращать никаких значений.

Отметим, что механизм сигналов является достаточно ресурсоемким, ибо отправка сигнала представляет собой системный вызов, а доставка сигнала - прерывание выполнения процесса-получателя. Вызов функции-обработчика и возврат требует операций со стеком. Если мы успешно установили в качестве обработчика сигнала свою функцию, то при возникновении сигнала выполнение процесса прерывается, фиксируется точка возврата и управление в процессе передается данной функции, при этом в качестве фактического целочисленного параметра передается номер пришедшего сигнала (тем самым возможно использование одной функции в качестве обработчика нескольких сигналов). Соответственно, при выходе из функции-обработчика управление передается в точку возврата и процесс продолжает свою работу.

Стоит обратить внимание на то, что возможны и достаточно часто происходят ситуации, когда сигнал приходит во время вызова процессом некоторого системного вызова. В этом случае последующие действия зависят от реализации системы: системный вызов либо «дорабатывает» до конца, либо повторно перезапускается после окончания обработки сигнала, либо прерывается с ошибочным кодом возврата (обычно -1) с занесением в переменную *errno* соответствующго кода ошибки.

Рассмотрим ряд примеров.

Пример. Перехват и обработка сигнала.

В данном примере при получении сигнала SIGINT (что соответствует нажатию CTRL+C) четыре раза вызывается специальный обработчик. На пятый же раз происходит обработка сигнала обработчиком по умолчанию, в результате чего процесс завершается.

```
#include <sys/types.h>
#include <signal.h>
#include <stdio.h>
int count = 0;
/* функция – пользовательский обработчик сигнала */
void SigHndlr (int s)
{
     printf("\n I got SIGINT %d time(s) \n",++ count);
     if (count == 5) signal (SIGINT, SIG_DFL);/* ставим
     обработчик сигнала по умолчанию */
}

int main(int argc, char **argv)
{
     /* установка реакции на сигнал */
     signal (SIGINT, SigHndlr);
```

```
     while (1); /*"тело программы" */
     return 0;
}
```

Пример. Удаление временных файлов при завершении программы. При разработке программ нередко приходится создавать временные файлы, которые позже удаляются. Если произошло непредвиденное событие, такие файлы могут остаться не удаленными. Ниже приведена программа, которая и в случае «дорабатывания» до конца, и в случае получения сигнала SIGINT перед завершением удаляет созданный ею временный файл.

```
#include <unistd.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
const char * tempfile = "abc";

void SigHndlr (int s)
{
     /* уничтожение временного файла в случае прихода сигнала
     SIGINT. В случае, если такой файл не существует (еще не
     создан или уже удален), вызов вернет -1 */
     unlink(tempfile);
}

int main(int argc, char **argv)
{
     signal (SIGINT, SigHndlr); /*установка реакции на сигнал */
     …
     creat(tempfile, 0666); /*создание временного файла*/
     …
     /*уничтожение временного файла в случае нормального
     функционирования процесса */
     unlink(tempfile);
     return 0;
}
```

В данном примере для создания временного файла используется системный вызов **creat()**:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int creat(const char *pathname, mode_t mode);
```

А системный вызов **unlink()** удаляет имя и файл, на который оно ссылается.

```
#include <unistd.h>
int unlink(const char *pathname);
```

Пример. Программа «будильник». Существуют задачи, в которых необходимо прервать выполнение процесса по истечении некоторого времени. В данном примере средствами ОС "заводится" будильник, который будет поторапливать ввести некоторое имя. Системный вызов **alarm():**

```
#include <unistd.h>
unsigned int alarm(unsigned int seconds);
```

инициализирует отложенное появление сигнала **SIGALRM** - процесс запрашивает ядро отправить ему самому сигнал по прошествии определенного времени**.**

```
#include <unistd.h>
#include <signal.h>
#include <stdio.h>

void alrm(int s) /*обработчик сигнала SIG_ALRM */
{
     printf("\n жду имя \n");
     alarm(5); /* заводим будильник */
}

int main(int argc, char **argv)
{
     char s[80];
     /* установка обработчика alrm на приход сигнала SIG_ALRM */
     signal(SIGALRM, alrm);
     alarm(5); /* заводим будильник */
     printf("Введите имя \n");
     for (;;)
     {
          printf("имя:");
          if (gets(s) != NULL) break; /*ожидаем ввода имени */
     };
     printf("OK! \n");
     return 0;
}
```

В данном примере происходит установка обработчика сигнала SIGALRM (функция **alrm()**). Затем происходит обращение к системному вызову *alarm()*, который заводит будильник на 5 единиц времени. Поскольку продолжительность единицы времени зависит от конкретной реализации системы, то мы будем считать в нашем примере, что происходит установка будильника на 5 секунд. Это означает, что по прошествии 5 секунд процесс получит сигнал SIGALRM. Далее выводится запрос "*Введите имя*" и управление передается бесконечному циклу *for*, выход из которого возможен лишь при вводе непустой строки текста. Если же по истечении 5 секунд ввода так и не последовало, то приходит сигнал SIGALRM, управление передается обработчику alrm, который печатает на экран напоминание о необходимости ввода имени, а затем снова устанавливает будильник на 5 секунд. Далее управление возвращается в функцию *main* в бесконечный цикл, и последовательность действий повторяется. Отметим, что если в момент выполнения системного вызова возникает событие, связанное с сигналом, то система прерывает выполнение системного вызова и возвращает код ответа, равный «-1».

3.1.1.1 Пример. Двухпроцессный вариант программы “Будильник”.

```
#include <signal.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

void alr(int s)
{
     printf("\n Быстрее!!! \n");
}

int main(int argc, char **argv)
{
     char s[80];
     int pid;
     /* установка обработчика alr на приход сигнала SIGALRM */
     signal(SIGALRM, alr);
     if  (pid = fork())
     {
          /* отцовский процесс */
          for (;;)
          {
               /*приостанавливаем процесс на 5 секунд */
               sleep(5);
               /*отправляем сигнал SIGALRM процессу- сыну */
               kill(pid, SIGALRM);
          }
     }
     else
     {
          /* сыновний процесс */
          printf("Введите имя \n");
          for (;;)
          {
               printf("имя:");
               if (gets(s) != NULL) break; /*ожидаем ввода
               имени*/
          }
          printf("OK!\n");
          /* убиваем зациклившегося отца */
          kill(getppid(), SIGKILL);
     }
     return 0;
}
```

В данном случае программа реализуется в двух процессах. Как и в предыдущем примере, имеется функция реакции на сигнал **alr()**, которая выводит на экран сообщение-запрос. В основной программе мы также указываем **alr()** как реакцию на сигнал **SIGALRM**. После этого мы запускаем сыновний процесс, и отцовский процесс (бесконечный цикл) “засыпает” на 5 единиц времени, после чего отправляет сыновнему процессу сигнал **SIGALRM** с помощью системного вызова *kill()*. Первым параметром

данному системному вызову передается идентификатор сыновнего процесса (PID), который был получен после вызова *fork()*.

Сыновний процесс запрашивает ввод имени, а дальше в бесконечном цикле ожидает ввода строки текста до тех пор, пока не получит непустую строку. При этом он периодически получает от отцовского процесса сигнал SIGALRM, вследствие чего выводит на экран напоминание. После получения непустой строки он печатает на экране подтверждение успешности ввода (“OK!”), посылает процессу-отцу сигнал SIGKILL и завершается. Послать сигнал безусловного завершения отцовскому процессу необходимо, поскольку после завершения сыновнего процесса тот будет некорректно слать сигнал SIGALRM (возможно, что идентификатор процесса-сына потом получит совершенно иной процесс со своей логикой работы, а процесс-отец так и будет слать на его PID сигналы SIGALRM).

3.1.2 Надежные сигналы.

Отметим одну особенность реализации сигналов в ранних версиях UNIX на базе signal: каждый раз при получении сигнала его диспозиция (т.е. действие при получении сигнала) сбрасывается на действие по умолчанию, т.о. если процесс желал многократно обрабатывать сигнал своим собственным обработчиком, он должен был каждый раз при обработке сигнала заново устанавливать реакцию на него. Позднее эта реализация подверглась критике за недостаточную надежность. Хотя и существует возможность заново установить реакцию на сигнал в функции-обработчике, возможна ситуация, когда между моментом вызова пользовательского обработчика некоторого сигнала и моментом, когда он восстановит нужную реакцию на этот сигнал, будут получены еще несколько экземпляров того же сигнала. В этом случае другие экземпляры не будет перехвачены, так как на момент их прихода для данного сигнала действует реакция по умолчанию.

Поэтому в более современных версиях UNIX (начиная с BSD UNIX 4.2 и System V.4) помимо обновления поведения на базе signal (без сброса обработчика по приходу каждого сигнала), также была реализована дополнительная модель так называемых надежных сигналов, которая вошла и в стандарт POSIX. В этой модели при перехватывании сигнала ядро не меняет его диспозицию, тем самым появляется гарантия перехвата всех экземпляров сигнала. Кроме того, чтобы избежать нежелательных эффектов при рекурсивном вызове обработчика для множества экземпляров одного и того же сигнала, ядро блокирует доставку других экземпляров того же сигнала в процесс до тех пор, пока функция-обработчик не завершит свое выполнение.

В модели надежных сигналов также появилась возможность на время блокировать доставку того или иного вида сигналов в процесс. Отличие блокировки сигнала от игнорирования в том, что пришедшие экземпляры сигнала не будут потеряны, а произойдет лишь откладывание их обработки на тот период времени, пока процесс не разблокирует данный сигнал. Таким образом процесс может оградить себя от прерывания сигналом на тот период, когда он выполняет какие-либо критические операции. Для реализации механизма блокирования вводится понятие **сигнальной маски**, которая описывает, какие сигналы из посылаемых процессу блокируются. Процесс наследует свою сигнальную маску от родителя при порождении[2], и имеет возможность изменять ее в процессе своего выполнения.

Рассмотрим системные вызовы для работы с сигнальной маской процесса. Сигнальная маска описывается битовым полем типа **`sigset_t`**. Для управления сигнальной маской процесса служит системный вызов:

```
#include <signal.h>
```

[2] Несмотря на это, как уже говорилось, сами сигналы, ожидающие своей обработки родительским процессом на момент порождения потомка, в том числе и блокированные, не наследуются потомком

```
int sigprocmask(int how, const sigset_t *set, sigset_t
*old_set);
```

Значения аргумента **how** влияют на характер изменения маски сигналов:

SIG_BLOCK – к текущей маске добавляются сигналы, указанные в наборе **set**;

SIG_UNBLOCK – из текущей маски удаляются сигналы, указанные в наборе **set**;

SIG_SETMASK – текущая маска заменяется на набор **set**.

Если в качестве аргумента **set** передается NULL-указатель, то сигнальная маска не изменяется, значение аргумента **how** при этом игнорируется. В последнем аргументе возвращается прежнее значение сигнальной маски до изменения ее вызовом **sigprocmask()**. Если процесс не интересуется прежним значением маски, он может передать в качестве этого аргумента NULL-указатель.

Если один или несколько заблокированных сигналов будут разблокированы посредством вызова **sigprocmask()**, то для их обработки будет использована диспозиция сигналов, действовавшая до вызова **sigprocmask()**. Если за время блокирования процессу пришло несколько экземпляров одного и того же сигнала, то ответ на вопрос о том, сколько экземпляров сигнала будет доставлено – все или один – зависит от реализации конкретной ОС.

Существует ряд вспомогательных функций, используемых для того, чтобы сформировать битовое поле типа **sigset_t** нужного вида:

- Инициализация битового набора - очищение всех битов:
  ```
  #include <signal.h>
  int sigemptyset(sigset_t *set);
  ```
- Противоположная предыдущей функция устанавливает все биты в наборе:
  ```
  #include <signal.h>
  int sigfillset(sigset_t *set);
  ```
- Две следующие функции позволяют добавить или удалить флаг, соответствующий сигналу, в наборе:
  ```
  #include <signal.h>
  int sigaddset(sigset_t *set, int signo);
  int sigdelset(sigset_t *set, int signo);
  ```
  В качестве второго аргумента этим функциям передается номер сигнала
- Приведенная ниже функция проверяет, установлен ли в наборе флаг, соответствующий сигналу, указанному в качестве параметра:
  ```
  #include <signal.h>
  int sigismember(sigset_t *set, int signo);
  ```
  Этот вызов возвращает 1, если в маске **set** установлен флаг, соответствующий сигналу **signo**, и 0 в противном случае.

Чтобы узнать, какие из заблокированных сигналов ожидают доставки, используется функция **sigpending()**:

```
#include <signal.h>
int sigpending(sigset_t *set);
```

Через аргумент этого вызова возвращается набор сигналов, ожидающих доставки.

3.1.2.1 Пример. Работа с сигнальной маской.

В данном примере анализируется сигнальная маска процесса, и выдается сообщение о том, заблокирован ли сигнал **SIGINT**, и ожидает ли такой сигнал доставки в процесс. Для того, чтобы легче было увидеть в действии результаты данных операций, предусмотрена возможность добавить этот сигнал к сигнальной маске процесса и послать этот сигнал самому себе.

```
#include <signal.h>
#include <sys/types.h>
#include <stdlib.h>
#include <stdio.h>
int main(int argc, char **argv)
{
     sigset_t sigset;
     int fl;
     sigemptyset(&sigset);
     printf("Добавить SIGINT к текущей маске? (yes - 1, no -
     0) \n");
     scanf("%d", &fl);
     if (fl)
     {
          sigaddset(&sigset, SIGINT);
          sigprocmask(SIG_BLOCK, &sigset, NULL);
     }
     printf("Послать SIGINT? (yes - 1, no - 0)\n");
     scanf("%d", &fl);
     if (fl) kill(getpid(), SIGINT);
     /* получаем сигнальную маску процесса. Так как второй
     аргумент NULL, то первый аргумент игнорируется */
     if (sigprocmask(SIG_BLOCK, NULL, &sigset) == -1)
     {
          printf("Ошибка при вызове sigprocmask()\n");
          return -1;
     }
     else if (sigismember(&sigset, SIGINT)) /*проверяем
наличие сигнала SIGINT в маске*/
     {
          printf("Сигнал SIGINT заблокирован! \n");
          sigemptyset(&sigset);
          /*узнаем сигналы, ожидающие доставки */
          if (sigpending(&sigset) == -1)
          {
               printf("Ошибка при вызове sigpending()\n");
               return -1;
          }
          /*проверяем наличие сигнала SIGINT в маске*/
          printf("Сигнал SIGINT %s\n", sigismember(&sigset,
          SIGINT) ? "ожидает доставки" : "не ожидает
          доставки");
```

```
        }
        else printf("Сигнал SIGINT не заблокирован. \n");
        return 0;
}
```

Для управления работой сигналов используется функция, аналогичная функции **signal()** в реализации обычных сигналов, но более мощная, позволяющая установить обработку сигнала, узнать ее текущее значение, приостановить получение сигналов:

```
#include <signal.h>

int sigaction(int sig, const struct sigaction *act,
struct *oact)
```

Аргументами данного вызова являются: номер сигнала, структура, описывающая новую реакцию на сигнал, и структура, через которую возвращается прежний метод обработки сигнала. Если процесс не интересуется прежней обработкой сигнала, он может передать в качестве последнего параметра NULL-указатель.

Структура **sigaction** содержит информацию, необходимую для управления сигналами. Ее полями являются :

```
void (*sa_handler) (int),
void (sa_sigaction) (int, siginfo_t*, void*),
sigset_t sa_mask,
int sa_flags
```

Здесь поле **sa_handler** — функция-обработчик сигнала, либо константы **SIG_IGN** или **SIG_DFL**, говорящие соответственно о том, что необходимо игнорировать сигнал или установить обработчик по умолчанию. В поле **sa_mask** указывается набор сигналов, которые будут добавлены к маске сигналов на время работы функции-обработчика. Сигнал, для которого устанавливается функция-обработчик, также будет заблокирован на время ее работы. При возврате из функции-обработчика маска сигналов возвращается в первоначальное состояние. В последнем поле указываются флаги, модифицирующие доставку сигнала. Одним из них может быть флаг **SA_SIGINFO**. Если он установлен, то при получении этого сигнала будет вызван обработчик **sa_sigaction**, ему помимо номера сигнала также будет передана дополнительная информация о причинах получения сигнала и указатель на контекст процесса.

Итак, «надежные» сигналы являются более мощным средством межпроцессного взаимодействия нежели обычные сигналы. В частности, здесь ликвидированы такие недостатки, как необходимость восстанавливать функцию-обработчик после получения сигнала, имеется возможность отложить получение сигнала на время выполнения критического участка программы, имеются большие возможности получения информации о причине отправления сигнала.

3.1.2.2 Пример. Использование надежных сигналов.

При получении сигнала **SIGINT** четырежды вызывается установленный обработчик, а в пятый раз происходит обработка по умолчанию.

```
#include <signal.h>
#include <stdlib.h>
#include <stdio.h>

int count = 1;
```

```
struct sigaction action, sa;

void SigHandler(int s)
{
     printf("\nI got SIGINT %d time(s)\n", count++);
     if (count == 5)
     {
          /*изменяем указатель на функцию-обработчик
          сигнала*/
          action.sa_handler = SIG_DFL;
          /* изменяем обработчик для сигнала SIGINT */
          sigaction(SIGINT, &action, &sa);
     }
}

int main(int argc, char **argv)
{
     sigset_t sigset;
     sigemptyset(&sigset); /* инициализируем набор сигналов
     */
     sigaddset(&sigset, SIGINT); /*добавляем в набор сигналов
     бит, соответствующий сигналу SIGINT*/
     /*устанавливаем новую сигнальную маску*/
     if (sigprocmask(SIG_UNBLOCK, &sigset, NULL) == -1)
     {
          printf("sigprocmask() error\n");
          return -1;
     }
     /* инициализируем указатель на функцию-обработчик
     сигнала*/
     action.sa_handler = SigHandler;
     /* изменяем обработчик по умолчанию для сигнала SIGINT
     */
     sigaction(SIGINT, &action, &sa);
     while(1); /* бесконечный цикл */
     return 0;
}
```

3.1.3 Неименованные каналы

Неименованный канал (или ***программный канал***) представляется в виде области памяти, управляемой операционной системой, которая осуществляет выделение взаимодействующим процессам частей из этой области памяти для совместной работы, т.е. эта область памяти является разделяемым ресурсом.

Для доступа к неименованному каналу система ассоциирует с ним два файловых дескриптора. Один из них предназначен для чтения информации из канала, т.е. с ним можно ассоциировать файл, открытый только на чтение. Другой дескриптор предназначен для записи информации в канал. Соответственно, с ним может быть ассоциирован файл, открытый только на запись.

Дисциплина доступа к информации, записанной в программный канал, - FIFO, т.е. информация, первой записанная в канал, будет прочитана из канала первой. Это означает, что для данных файловых дескрипторов неприменимы системные вызовы перемещения файлового указателя. Предельный размер канала, который может быть выделен процессам, декларируется параметрами настройки операционной системы.

Для создания неименованного канала служит системный вызов *pipe()*.

```
#include <unistd.h>

int pipe(int *fd)
```

Аргументом данного системного вызова является указатель на массив *fd* из двух целочисленных элементов. Если системный вызов *pipe()* прорабатывает успешно, то он возвращает код ответа, равный нулю, а массив будет содержать два открытых файловых дескриптора. Соответственно, в fd[0] будет содержаться дескриптор чтения из канала, а в fd[1] — дескриптор записи в канал. После этого с данными файловыми дескрипторами можно использовать всевозможные средства работы с файлами, поддерживающие стратегию FIFO, т.е. любые операции работы с файлами, за исключением тех, которые касаются перемещения файлового указателя.

Однако следует четко понимать ***различия между обычным файлом и каналом***. Основные отличительные свойства канала, следующие:

- В отличие от файла, к неименованному каналу невозможен доступ по имени, т.е. единственная возможность использовать канал – это те файловые дескрипторы, которые с ним ассоциированы
- В отличие от файлов неименованный канал существует в системе, до тех пор, пока существуют процессы, его использующие. То есть для существования канала необходим процесс, который его создал либо получил в наследство. Отметим, что канал может быть доступен как для процесса, который его создал, так и для процессов, которые унаследовали открытые файловые дескрипторы, ассоциированные с этим каналом. За счёт этой возможности наследования неименованный канал превращается в средство взаимодействия родственных процессов.
- Канал – это структура данных, которая реализует модель последовательного доступа к данным (FIFO), т.е. данные из канала можно прочитать только в той же последовательности, в какой они были записаны в канал. Это означает, что для файловых дескрипторов, ассоциированных с каналом, не определена операция **`lseek()`** (при попытке обратиться к этому вызову произойдет ошибка).

Неименованные каналы в общем случае *предназначены для синхронизации и организации взаимодействия родственных процессов*, осуществляющегося за счет передачи по наследству ассоциированных с каналом файловых дескрипторов. Но иногда встречаются вырожденные случаи использования неименованного канала в рамках одного процесса.

Пример. Использование неименованного канала. В приведенном ниже примере производится копирование текстовой строки с использованием канала. Этот пример является «вырожденным»: он иллюстрирует случай использования канала в рамках одного процесса – фактически осуществляется посылка данных самому себе.

```
int main(int argc, char **argv)
{
```

```
        char *s = "channel";
        char buf[80];
        int pipes[2];

        pipe(pipes);
        write(pipes[1], s, strlen(s) + 1);
        read(pipes[0], buf, strlen(s) + 1);
        close(pipes[0]);
        close(pipes[1]);
        printf("%s\n", buf);
        return 0;
}
```

В приведенном примере имеется текстовая строка *s*, которую хотим скопировать в буфер *buf*. Для этого дополнительно декларируется массив *pipes*, в котором будут храниться файловые дескрипторы, ассоциированные с каналом. После обращения к системному вызову *pipe()* элемент *pipes[1]* хранит открытый файловый дескриптор, через который можно писать в канал, а *pipes[0]* — файловый дескриптор, через который можно читать из канала. Затем происходит обращение к системному вызову *write()*, чтобы скопировать содержимое строки *s* в канал, а после этого идет обращение к системному вызову *read()*, чтобы прочитать данные из канала в буфер *buf*. Потом закрываем дескрипторы и печатаем содержимое буфера на экран.

Кроме того, существует ряд отличий при организации операций чтения и записи в канал.

При чтении из канала:

- При чтении из пустого канала процесс блокируется до момента появления в канале данных. Блокировка происходит лишь при условии, что есть хотя бы один открытый дескриптор записи в канал. Если закрывается последний дескриптор записи в данный канал, то в канал помещается код конца файла EOF. В этом случае процесс, осуществляющий чтение, может выбрать из канала все оставшиеся данные и признак конца файла, благодаря которому блокирования при чтении в этом случае не происходит. Соответственно, если заблокированы два и более процесса на чтение, то порядок разблокировки определяется конкретной реализацией.
- Если из канала читается порция данных меньшая, чем находящаяся в канале, то эта порция считывается по стратегии FIFO, а оставшаяся порция непрочитанных данных остается в канале.
- Если делается попытка прочитать больше данных, чем имеется в канале, то будет прочитано (т.е. изъято из канала) доступное количество данных.

При записи в канал:

- Если процесс пытается записать большее число байтов, чем доступное свободное пространство канала (но не превышающее предельный размер канала), то записывается возможное количество данных, после чего процесс, осуществляющий запись, блокируется до тех пор, пока в канале не появится достаточное количество места для завершения операции записи.
- Если же процесс пытается записать в канал порцию данных, превышающую предельный размер канала, то будет записано доступное количество данных, после чего процесс заблокируется до появления в канале свободного места любого размера (пусть даже и всего 1 байт), затем процесс разблокируется, вновь производит запись на доступное место в канале, и если данные для записи еще не исчерпаны, вновь блокируется до появления свободного места и т.д., пока не будут записаны все данные, после чего происходит возврат из вызова **`write()`**.

- Если процесс пытается осуществить запись в канал, с которым не ассоциирован ни один дескриптор чтения, то он получает сигнал **SIGPIPE** (тем самым ОС уведомляет его о недопустимости такой операции).

В стандартной ситуации (при отсутствии переполнения) система гарантирует атомарность операции записи, т. е. при одновременной записи нескольких процессов в канал их данные не перемешиваются.

Процесс может избежать блокирований при работе с каналом, изменив для канала режим блокировки с использованием системного вызова fcntl(). В неблокирующем режиме управление будет сразу возвращено процессу с возможной установкой соотвествующего кода ошибки (в зависимости от типа операции и возможных переполнений).

В общем случае возможна многонаправленная работа процессов с каналом, т.е. возможна ситуация, когда с одним и тем же каналом взаимодействуют два и более процесса, и каждый из взаимодействующих каналов пишет информацию в канал и читает из канала. Но традиционной схемой организации работы с каналом является однонаправленная организация, когда канал связывает два (в большинстве случаев) или несколько взаимодействующих процессов, каждый из которых может либо читать, либо писать в канал.

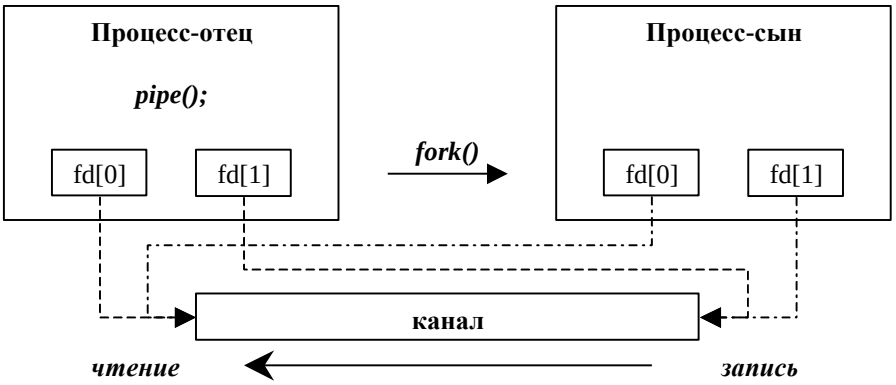


Рис. 97. Схема взаимодействия процессов с использованием неименованного канала.

Пример. Схема организации взаимодействия процессов с использованием канала (Рис. 97)**.** Схема всегда такова: некоторый родительский процесс внутри себя порождает канал, после этого идут обращения к системным вызовам *fork()* — создается дерево процессов. При этом за счет того, что при порождении процесса открытые файловые дескрипторы наследуются, сыновний процесс также обладает файловыми дескрипторами, ассоциированными с каналом, который создал его предок. За счет этого можно организовать взаимодействие родственных процессов.

В следующем примере организуется неименованный канал между отцовским и сыновним процессами, причем процесс-отец будет писать в канал, а процесс-сын — читать из него.

```
#include <sys/types.h>
#include <unistd.h>

int main(int argc, char **argv)
{
     int fd[2];
     pipe(fd);
     if (fork())
```

```
            {/*процесс-родитель*/
                 close(fd[0]); /* закрываем ненужный дескриптор
                 */
                 write (fd[1], …);
                 …
                 close(fd[1]);
                 …
            }
            else
            {
                 /*процесс-потомок*/
                 close(fd[1]); /* закрываем ненужный дескриптор
                 */
                 while(read (fd[0], …))
                 {
                      …
                 }
                 …
            }
        }
```

В рассмотренном примере после создания канала посредством системного вызова *pipe()* и порождения сыновнего процесса посредством системного вызова *fork()*, отцовский процесс закрывает дескриптор, открытый на чтение из канала, потом производит различные действия, среди которых он пишет некоторую информацию в канал, после чего закрывает дескриптор записи в канал, и, наконец, после некоторых действий завершается. Процесс-сын первым делом закрывает дескриптор записи в канал, а после этого циклически считывает некоторые данные из канала. Стоит обратить внимание, что закрытие дескриптора чтения в канал в отцовском процессе можно не делать, т.к. при завершении процесса все открытые файловые дескрипторы будут автоматически закрыты. Но в сыновнем процессе закрытие дескриптора записи в канал обязательно: в противном случае, поскольку сыновний процесс читает данные из канала *циклически* (до получения кода конца файла), он не сможет получить этот код конца файла, а потому он зациклится. А код конца файла не будет помещен в канал, потому что при закрытии дескриптора записи в канал в отцовском процессе с каналом все еще будет ассоциирован открытый дескриптор записи сыновнего процесса.

Пример. Реализация конвейера. Приведенный ниже пример основан на том факте, что при порождении процесса в ОС Unix он заведомо получает три открытых файловых дескриптора: ***дескриптор стандартного ввода*** (этот дескриптор имеет нулевой номер), ***дескриптор стандартного вывода*** (имеет номер 1) и ***дескриптор стандартного потока ошибок*** (имеет номер 2). Обычно на стандартный ввод поступают данные с клавиатуры, а стандартный вывод и поток ошибок отображаются на дисплей монитора. Интерпретатор команд UNIX позволяет организовывать цепочки команд, когда стандартный вывод одной команды поступает на стандартный ввод другой команды, и такие цепочки называются конвейером команд. В конвейере могут участвовать две и более команды.

В данном примере реализуется конвейер команд **ls|wc**, в котором команда **ls** осуществляет печать списка файлов в рабочей директории, а команда **wc** выводит некоторые статистические характеристики входного потока (количество байт, строк и слов), что позволяет в итоге получить количество файлов в текущей директории.

```
int main(int argc, char **argv)
{
     int fd[2];
     pipe(fd); /* организовываем канал */
```

```
    if(fork()> 0)
    {     /* ПРОЦЕСС- отец */

          /* отождествим стандартный вывод с файловым
          дескриптором канала, предназначенным для записи */
          dup2(fd[1],1);
          /* закрываем файловый дескриптор канала,
          предназначенный для записи */
          close(fd[1]);
          /* закрываем файловый дескриптор канала,
          предназначенный для чтения */
          close(fd[0]);
          /* запускаем программу ls */
          execlp("ls","ls",0);
    }

    /* ПРОЦЕСС-сын */
    /*отождествляем стандартный ввод с файловым дескриптором
    канала, предназначенным для чтения */
    dup2(fd[0],0);
    /* закрываем файловый дескриптор канала, предназначенный для
    чтения */
    close(fd[0]);
    /* закрываем файловый дескриптор канала, предназначенный для
    записи */
    close(fd[1]);
    /* запускаем программу wc */
    execl("/usr/bin/wc","wc",0);
}
```

В приведенной программе создаётся канал, затем порождается сыновний процесс. Далее отцовский процесс обращается к системному вызову *dup2()*, который закрывает файл, ассоциированный с файловым дескриптором 1 (т.е. стандартный вывод), и ассоциирует файловый дескриптор 1 с файлом, ассоциированным с дескриптором fd[1]. Таким образом, теперь через первый дескриптор стандартный вывод будет направляться в канал. После этого файловые дескрипторы fd[0] и fd[1] нам более не нужны, мы их закрываем, а в родительском процессе остается ассоциированным с каналом файловый дескриптор с номером 1. После этого происходит обращение к системному вызову *execlp()*, который запустит команду ls, у которой выходная информация будет писаться в канал.

В сыновнем процессе производятся аналогичные действия, только здесь идет работа со стандартным вводом, т.е. с нулевым файловым дескриптором. И в конце запускается команда *wc*, у которой входная информация будет поступать из канала. Тем самым мы запустили конвейер этих команд: синхронизация этих процессов будет происходит за счет реализованной в механизме неименованных каналов стратегии FIFO.

Пример. «Пинг-понг» (совместное использование сигналов и каналов – для синхронизации используется канал и аппарат сигналов). В данном примере обеспечивается корректная организация двунаправленной работы, когда каждый из взаимодействующих процессов может и читать из канала, и писать в канал.

Итак, пусть есть два процесса (родительский и сыновний), которые через канал будут некоторое предопределенное число раз перекидывать «мячик»-счетчик, подсчитывающий количество своих бросков в канал. Извещение процесса о получении управления (когда он может взять «мячик» из канала, увеличить своё значение на 1 и снова бросить в канал) будет происходить на основе механизма сигналов. Таким образом, на

канал возлагается роль среды двусторонней передачи информации, для синхронизации используется канал и аппарат сигналов.

```
#include <signal.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

#define MAX_CNT 100
int target_pid, cnt;
int fd[2];
int status;

void SigHndlr(int s)
{
     /* в обработчике сигнала происходит и чтение, и запись */
     if(cnt < MAX_CNT)
     {
          read(fd[0], &cnt, sizeof(int));
          printf("%d\n", cnt);
          cnt++;
          write(fd[1], &cnt, sizeof(int));
          /* посылаем сигнал второму: пора читать из канала */
          kill(target_pid, SIGUSR1);
     }
     else if(target_pid == getppid())
     {
          /* условие окончания игры проверяется потомком */
          printf("Child is going to be terminated\n");
          close(fd[1]);
          close(fd[0]);
          /* завершается потомок */
          exit(0); /* нормальное завершение */

     }
     else
          kill(target_pid, SIGUSR1);
}

int main(int argc, char **argv)
{
     pipe(fd);
     signal(SIGUSR1, SigHndlr);
     cnt = 0;

     if(target_pid = fork())
     {
          /* В родительском процессе: */
          write(fd[1], &cnt, sizeof(int)); /* старт. синхр.*/
```

```
        /* ждем завершения потомка; если при этом происходит
    поступление сигнала, то системный вызов wait прерывается с
    кодом завершения -1, поэтому используется цикл */
        while(wait(&status)== -1);

        printf("Завершение родительского процесса\n");
        close(fd[1]);
        close(fd[0]);
        return 0;
    }
    else
    {
        /* Сыновний процесс – ждем отца: */
        read(fd[0], &cnt, sizeof(int));
        target_pid = getppid();

        /* сыновний процесс начинает пинг-понг */
        write(fd[1], &cnt, sizeof(int));
        kill(target_pid, SIGUSR1);
        for(;;);
    }
}
```

Для синхронизации взаимодействующих процессов используется сигнал SIGUSR1. Обычно в операционных системах присутствуют сигналы, которые не ассоциированы с событиями, происходящими в системе, и которые процессы могут использовать по своему усмотрению. Количество таких пользовательских сигналов зависит от конкретной реализации. В приведенном примере реализован следующий принцип работы: процесс получает сигнал SIGUSR1, берет счетчик из канала, увеличивает его на 1 и снова помещает в канал, после чего посылает своему напарнику сигнал SIGUSR1. Далее действия повторяются, пока счетчик не возрастет до некоторой фиксированной величины MAX_CNT, после чего происходят завершения процессов.

В качестве счетчика в данной программе выступает целочисленная переменная *cnt*. Рассмотрим функцию-обработчик сигнала SIGUSR1. В ней проверяется, не превзошло ли значение *cnt* величины MAX_CNT. Если нет, то из канала читается новое значение *cnt*, происходит печать нового значения, после этого увеличивается на 1 значение *cnt*, и оно помещается в канал, а напарнику посылается сигнал SIGUSR1 (посредством системного вызова *kill()*).

Если же значение *cnt* оказалось не меньше MAX_CNT, то начинаются действия по завершению процессов, при этом первым должен завершиться сыновний процесс. Для этого проверяется идентификатор процесса-напарника (target_pid) на равенство идентификатору родительского процесса (значению, возвращаемому системным вызовом *getppid()*). Если это так, то в данный момент управление находится у сыновнего процесса, который и инициализирует завершение. Он печатает сообщение о своем завершении, закрывает дескрипторы, ассоциированные с каналом, и завершается посредством обращения к библиотечной функции *exit()*. Если же указанное условие ложно, то в данный момент управление находится у отцовского процесса, который сразу же передает его сыновнему процессу, посылая сигнал SIGUSR1, при этом ничего не записывая в канал, поскольку у сына уже имеется значение переменной *cnt*.

В самой программе (функции main) происходит организация канала, установка обработчика сигнала SIGUSR1 и инициализация счетчика нулевым значением. Затем происходит обращение к системному вызову *fork()*, значение которого присваивается переменной целевого идентификатора target_pid. Если мы находимся в родительском процессе, то в этой переменной будет находиться идентификатор сыновнего процесса.

После этого отцовский процесс начинает ожидать завершения сыновнего процесса посредством обращения к системному вызову *wait()*. Дождавшись завершения, отцовский процесс выводит сообщение о своем завершении, закрывает дескрипторы, ассоциированные с каналом, и завершается.

Если же системный вызов *fork()* возвращает нулевое значение, то это означает, что в данный момент мы находимся в сыновнем процессе, поэтому первым делом переменной target_pid присваивается значение идентификатора родительского процесса посредством обращения к системному вызову *getppid()*. После чего процесс пишет в канал значение переменной *cnt*, посылает отцовскому процессу сигнал SIGUSR1, тем самым, начиная «игру», и входит в бесконечный цикл.

Пример. «Пинг-понг» (совместное использование сигналов и каналов – для синхронизации используется только аппарат сигналов).

```
#include <signal.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

#define MAX_CNT 100
int target_pid, cnt;
int fd[2];
int status;

void SigHndlr(int s)
{
	if (cnt < MAX_CNT)
	{
		read(fd[0], &cnt, sizeof(int));
		printf("pid=%d cnt=%d \n", target_pid, cnt);
		cnt++;
		write(fd[1], &cnt, sizeof(int));
		kill(target_pid, SIGUSR1);
	}
	else
		if (target_pid == getppid())
		{
			printf("Child is going to be terminated\n");
			close(fd[1]); close(fd[0]);
			exit(0); /* нормальное завершение */

		}
		else
			kill(target_pid, SIGUSR1);
}


void Init(int s)
{
    target_pid = getppid();
    write(fd[1], &cnt, sizeof(int));
    kill(target_pid, SIGUSR1);
```

```
}

int main(int argc, char **argv)
{
      pipe(fd);
      signal (SIGUSR2, Init);
      signal (SIGUSR1, SigHndlr);

      cnt = 0;

      if ((target_pid = fork()) > 0)
      {
         /* отсылаем сыну сигнал для старта */
         kill(target_pid, SIGUSR2);

         while(wait(&status)== -1);

         printf("Parent is going to be terminated\n");
         close(fd[1]); close(fd[0]);
         return 0;
      }
      else
      {
         for(;;);
      }
}
```

3.1.4 Именованные каналы

Рассмотренные выше программные каналы имеют важное ограничение: так как доступ к ним возможен только посредством дескрипторов, возвращаемых при порождении канала, необходимым условием взаимодействия процессов через канал является передача этих дескрипторов по наследству при порождении процесса. Именованные каналы (FIFO-файлы) расширяют свою область применения за счет того, что подключиться к ним может любой процесс в любое время, в том числе и после создания канала. Это возможно благодаря наличию у них имен**.**

FIFO-файл (именованный канал) представляет собой специальный тип файла в файловой системе UNIX, который обладает всеми атрибутами файла, такими как имя владельца, права доступа и размер. Особенностью этих файлов является их организация по стратегии FIFO (т.е. невозможны операции, связанные с перемещением файлового указателя).

Таким образом, файлы FIFO могут использоваться для организации взаимодействия процессов, при этом в отличие от неименованных каналов эти файлы могут существовать независимо от процессов, взаимодействующих через них.

Для создания файлов FIFO в различных реализациях используются разные системные вызовы, одним из которых может являться *mkfifo()*[3].

```
#include <sys/types.h>
#include <sys/stat.h>
```

[3] Для создания файла FIFO в UNIX System V.3 и ранее используется системный вызов *mknod()*, а в BSD UNIX и System V.4 — вызов *mkfifo()* (этот вызов поддерживается и стандартом POSIX).

```
int mkfifo (char *pathname, mode_t mode);
```

В этом вызове первый аргумент представляет собой имя создаваемого канала, во втором аргументе указываются режимы открытия, устанавливаются права доступа к каналу для владельца, группы и прочих пользователей, и кроме того, устанавливается флаг, указывающий на то, что создаваемый объект является именно FIFO-файлом (в разных версиях ОС он может иметь разное символьное обозначение – **S_IFIFO** или **I_FIFO**).

После создания именованного канала любой процесс может установить с ним связь посредством системного вызова *open()*. При этом действуют следующие правила:

- если процесс открывает FIFO-файл для чтения, он блокируется до тех пор, пока какой-либо процесс не откроет тот же канал на запись;
- если процесс открывает FIFO-файл на запись, он будет заблокирован до тех пор, пока какой-либо процесс не откроет тот же канал на чтение;
- процесс может избежать такого блокирования, указав в вызове *open()* специальный флаг (в разных версиях ОС он может иметь разное символьное обозначение — O_NONBLOCK или O_NDELAY). В этом случае в ситуациях, описанных выше, вызов *open()* сразу же вернет управление процессу.

Правила работы с именованными каналами (в частности, особенности операций чтения-записи) полностью аналогичны правилам работы с неименованным каналами.

Ниже рассматривается пример, где один из процессов является сервером, предоставляющим некоторую услугу, другой же процесс, который хочет воспользоваться этой услугой, является клиентом. Клиент посылает серверу запросы на предоставление услуги, а сервер отвечает на эти запросы.

Пример. Модель «клиент-сервер». Процесс-сервер запускается на выполнение первым, создает именованный канал (FIFO-файл), открывает его на чтение в блокирующем режиме и входит в цикл, пытаясь прочесть что-либо. Процесс-клиент подключается к каналу с известным ему именем, записывает в него свой идентификатор, а затем закрывает файловый дескриптор, ассоциированный с данным FIFO-файлом. Прочитав идентификатор клиента, сервер выходит из цикла, печатает этот идентификатор, а затем закрывает дескриптор, ассоциированный с данным FIFO-файлом, и уничтожает сам файл.

```
/* процесс-сервер*/
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int main(int argc, char **argv)
{
     int fd;
     int pid;
     /*создали специальный файл FIFO с открытыми для всех
правами доступа на чтение и запись*/
     mkfifo("fifo", S_IFIFO | 0666);
     /* открыли канал на чтение*/
     fd = open("fifo", O_RDONLY);
     while  (read (fd, &pid, sizeof(int)) >0){
```

```
            printf("Server %d  got message from %d !\n",
            getpid(), pid);
            …
        }
        close(fd);
        unlink("fifo");/*уничтожили именованный канал*/
        return 0;
}



/* процесс-клиент*/
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>
int main(int argc, char **argv)
{
        int fd;
        int pid = getpid( );
        /* открыли существующий канал на запись*/
        fd = open("fifo", O_WRONLY);
        write(fd, &pid, sizeof(int));
        close(fd);
        return 0;
}
```

3.1.5 Нелокальные переходы

Рассмотрим некоторые дополнительные возможности по организации управления ходом процесса в UNIX, а именно возможность передачи управления в точку, расположенную вне данной функции.

Как известно, оператор **goto** позволяет осуществлять безусловный переход только внутри одной функции. Это ограничение связано с необходимостью сохранения целостности стека: в момент входа в функцию в стеке отводится место, называемое стековым кадром, где записываются адрес возврата, фактические параметры, отводится место под автоматические переменные. Стековый кадр освобождается при выходе из функции. Соответственно, если при выполнении безусловного перехода процесс минует тот фрагмент кода, где происходит освобождение стекового кадра, и управление непосредственно перейдет в другую часть программы (например, в объемлющую функцию), то фактическое состояние стека не будет соответствовать текущему участку кода, и тем самым стек подвергнется разрушению.

Однако такое ограничение в некоторых случаях создает большое неудобство: например, в случае возникновения ошибки в рекурсивной функции, после обработки

ошибки имеет смысл перейти в основную функцию, которая может находиться на несколько уровней вложенности выше текущей. Поскольку такой переход невозможно осуществить ни оператором **return**, ни оператором **goto**, программист будет вынужден создавать какие-то громоздкие структуры для обработки ошибок на каждом уровне вложенности.

Возможность передавать управление в точку, находящуюся в одной из вызывающих функций, предоставляется двумя системными вызовами, реализующими механизм нелокальных переходов:

```
#include <setjmp.h>
int setjmp(jmp_buf env);
void longjmp(jmp_buf env, int val);
```

Вызов **setjmp()** используется для регистрации некоторой точки кода, которая в дальнейшем будет использоваться в качестве пункта назначения для нелокального перехода, а вызов **longjmp()** – для перехода в одну из ранее зарегистрированных конечных точек.

При обращении к вызову **setjmp()**, происходит сохранение параметров текущей точки кода (значения счетчика адреса, позиции стека, регистров процессора и реакций на сигналы). Все эти значения сохраняются в структуре типа **jmp_buf**, которая передается вызову **setjmp()** в качестве параметра. При этом вызов **setjmp()** возвращает 0.

После того, как нужная точка кода зарегистрирована с помощью вызова **setjmp()**, управление в нее может быть передано при помощи вызова **longjmp()**. При этом в качестве первого параметра ему указывается та структура, в которой были зафиксированы атрибуты нужной нам точки назначения. После осуществления вызова **longjmp()** процесс продолжит выполнение с зафиксированной точки кода, т.е. с того места, где происходит возврат из функции **setjmp()**, но в отличие от первого обращения к **setjmp()**, возвращающим значением **setjmp()** станет не 0, а значение параметра **val** в вызове **longjmp()**, который произвел переход.

Отметим, что если программист желает определить в программе несколько точек назначения для нелокальных переходов, каждая из них должна быть зарегистрирована в своей структуре типа **jmp_buf**. С другой стороны, разумеется, на одну и ту же точку назначения можно переходить из разных мест программы, при этом, чтобы различить, из какой точки был произведен нелокальный переход, следует указывать при переходах разные значения параметра **val**. В любом случае, при вызове **longjmp()** значение параметра **val** не должно быть нулевым (даже если оно есть 0, то возвращаемое значение **setjmp()** будет установлено в 1). Кроме того, переход должен производиться только на такие точки, которые находятся в коде одной из вызывающих функций для той функции, откуда осуществляется переход (в том числе, переход может быть произведен из функции-обработчика сигнала). При этом в момент перехода все содержимое стека, используемое текущей функцией и всеми вызывающими, вплоть до необходимой, освобождается.

3.1.5.1 Пример. Использование нелокальных переходов.

```
#include <signal.h>
#include <setjmp.h>
jmp_buf env;
void abc(int s)
{
     …
     longjmp(env,1);     /*переход  - в точку *** */
}
```

```
int main(int argc, char **argv)
{
        …
        /* запоминается данная точка процесса - *** */
        if (setjmp(env) == 0)
        {
                signal(SIGINT,abc);   /* установка реакции на
                сигнал */
                …
                /*цикл обработки данных после вызова функции
                setjmp()*/
        }
        else
        {
                …
                /* цикл обработки данных после возврата из
                обработчика сигнала */
        }
        ...
}
```

3.1.6 Трассировка процессов – модель межпроцессного взаимодействия «главный-подчиненный»

Достаточно часто при организации многопроцессной работы необходимо наличие возможности, когда один процесс является главным по отношению к другим процессам (трассировка процессов). В частности, это необходимо для организации средств отладки, когда есть процесс-отладчик и отлаживаемый процесс. Для механизма отладки полезно, чтобы отладчик мог в произвольные моменты времени останавливать отлаживаемый процесс и, когда отлаживаемый процесс остановлен, осуществлять действия по его отладке: просматривать содержимое тела процесса, при необходимости корректировать тело процесса и т.д. Также является полезным возможность установки контрольных точек в отлаживаемом процессе. Очевидно, что полномочия процесса-отладчика по отношению к отлаживаемому процессу являются полномочиями **главного**, т.е. отладчик может осуществлять управление, в то время как отлаживаемый процесс может лишь **подчиняться**.

В UNIX трассировка возможна только между родственными процессами: процесс-родитель может вести трассировку только непосредственно порожденных им потомков, при этом трассировка начинается только после того, как процесс-потомок дает разрешение на это.

Далее схема взаимодействия процессов путем трассировки такова: выполнение отлаживаемого процесса-потомка приостанавливается всякий раз при получении им какого-либо сигнала, а также при выполнении вызова **`exec()`**. Если в это время отлаживающий процесс осуществляет системный вызов **`wait()`**, этот вызов немедленно возвращает управление. В то время, как трассируемый процесс находится в приостановленном состоянии, процесс-отладчик имеет возможность анализировать и изменять данные в адресном пространстве отлаживаемого процесса и в пользовательской составляющей его контекста. Далее, процесс-отладчик возобновляет выполнение трассируемого процесса до следующей приостановки (либо, при пошаговом выполнении, для выполнения одной инструкции).

Для организации взаимодействия «главный–подчиненный» ОС Unix предоставляет системный вызов *ptrace()*.

```
#include <sys/ptrace.h>
int ptrace(int cmd, int pid, int addr, int data);
```

В этом системном вызове параметр *cmd* обозначает код выполняемой команды, *pid* — идентификатор процесса-потомка (который мы хотим трассировать), *addr* — некоторый адрес в адресном пространстве процесса-потомка, и, наконец, *data* — слово информации.

Посредством системного вызова *ptrace()* можно решать две задачи. С одной стороны, с помощью этого вызова подчиненный процесс может разрешить родительскому процессу проводить свою трассировку: для этого в качестве параметра *cmd* необходимо указать команду PTRACE_TRACEME. С другой стороны, с помощью этого же системного вызова процесс отладчик может манипулировать отлаживаемым процессом. Для этого используются остальные значения параметра *cmd*.

Системный вызов *ptrace()* позволяет выполнять следующие действия:

1. читать данные из сегмента кода и сегмента данных отлаживаемого процесса;
2. читать некоторые данные из контекста отлаживаемого процесса (в частности, имеется возможность чтения содержимого регистров);
3. осуществлять запись в сегмент кода, сегмент данных и в некоторые области контекста отлаживаемого процесса (в т.ч. модифицировать содержимое регистров). Следует отметить, что производить чтение и запись данных (а также осуществлять большинство управляющих команд над отлаживаемым процессом) можно лишь тогда, когда трассируемый процесс приостановлен;
4. продолжать выполнение отлаживаемого процесса с прерванной точки или с предопределенного адреса сегмента кода;
5. исполнять отлаживаемый процесс в пошаговом режиме. ***Пошаговый режим*** — это режим, обеспечиваемый аппаратурой компьютера, который вызывает прерывание после исполнения каждой машинной команды отлаживаемого процесса (т.е. после исполнения каждой машинной команды процесс приостанавливается).

Рассмотрим основные коды операций (параметр *cmd*) системного вызова *ptrace()*.

cmd = PTRACE_TRACEME — **ptrace()** с таким кодом операции сыновний процесс вызывает в самом начале своей работы, позволяя тем самым трассировать себя. Все остальные обращения к вызову **ptrace()** осуществляет процесс-отладчик.

cmd = PTRACE_PEEKDATA — чтение слова из адресного пространства отлаживаемого процесса по адресу **addr**, **ptrace()** возвращает значение этого слова.

cmd = PTRACE_PEEKUSER — чтение слова из контекста процесса. Речь идет о доступе к пользовательской составляющей контекста данного процесса, сгруппированной в некоторую структуру, описанную в заголовочном файле **<sys/user.h>**. В этом случае параметр **addr** указывает смещение относительно начала этой структуры. В этой структуре размещена такая информация, как регистры, текущее состояние процесса, счетчик адреса и так далее. **ptrace()** возвращает значение считанного слова.

cmd = PTRACE_POKEDATA — запись данных, размещенных в параметре **data**, по адресу **addr** в адресном пространстве процесса-потомка.

cmd = PTRACE_POKEUSER — запись слова из **data** в контекст трассируемого процесса со смещением **addr**. Таким образом можно, например, изменить счетчик адреса трассируемого процесса, и при последующем возобновлении трассируемого процесса его выполнение начнется с инструкции, находящейся по заданному адресу.

cmd = PTRACE_GETREGS, PTRACE_GETFREGS — чтение регистров общего назначения (в т.ч. с плавающей точкой) трассируемого процесса и запись их значения по адресу **data**.

cmd = PTRACE_SETREGS, PTRACE_SETFREGS — запись в регистры общего назначения (в т.ч. с плавающей точкой) трассируемого процесса данных, расположенных по адресу **data** в трассирующем процессе.

cmd = PTRACE_CONT — возобновление выполнения трассируемого процесса. Отлаживаемый процесс будет выполняться до тех пор, пока не получит какой-либо сигнал, либо пока не завершится.

cmd = PTRACE_SYSCALL, PTRACE_SINGLESTEP — эта команда, аналогично **PTRACE_CONT**, возобновляет выполнение трассируемой программы, но при этом произойдет ее остановка после того, как выполнится одна инструкция. Таким образом, используя **PTRACE_SINGLESTEP**, можно организовать пошаговую отладку. С помощью команды **PTRACE_SYSCALL** возобновляется выполнение трассируемой программы вплоть до ближайшего входа или выхода из системного вызова. Идея использования **PTRACE_SYSCALL** в том, чтобы иметь возможность контролировать значения аргументов, переданных в системный вызов трассируемым процессом, и возвращаемое значение, переданное ему из системного вызова.

cmd = PTRACE_KILL — завершение выполнения трассируемого процесса.

3.1.6.1 Пример. Общая схема использования механизма трассировки.

Рассмотрим типовую схему организации трассировки. Будем рассматривать взаимодействие родительского процесса-отладчика с подчиненным сыновним процессом (Рис. 98).

```
...
     if ((pid = fork()) == 0)
     {
          /* сыновний процесс разрешает трассировать себя */
          ptrace(PTRACE_TRACEME, 0, 0, 0);
          /* замещается телом процесса, который необходимо
          трассировать */
          exec("трассируемый процесс", 0);
     }
     else
     {/* это процесс, управляющий трассировкой */

          /* процесс приостанавливается до тех пор, пока от
          трассируемого процесса не придет сообщение о том, что он
          приостановился */
          wait((int ) 0);
          for(;;)
          {
               /* возобновляем выполнение трассируемой программы
               на 1 команду*/
               ptrace(PTRACE_SINGLESTEP, pid, 0, 0);
               /* процесс приостанавливается до тех пор, пока от
               трассируемого процесса не придет сообщение о том,
               что он приостановился */
               wait((int ) 0);
               …
               /* теперь выполняются нужные действия над
               трассируемым процессом */
               ptrace(cmd, pid, addr, data);
```

```
                ...
            }
        }
```

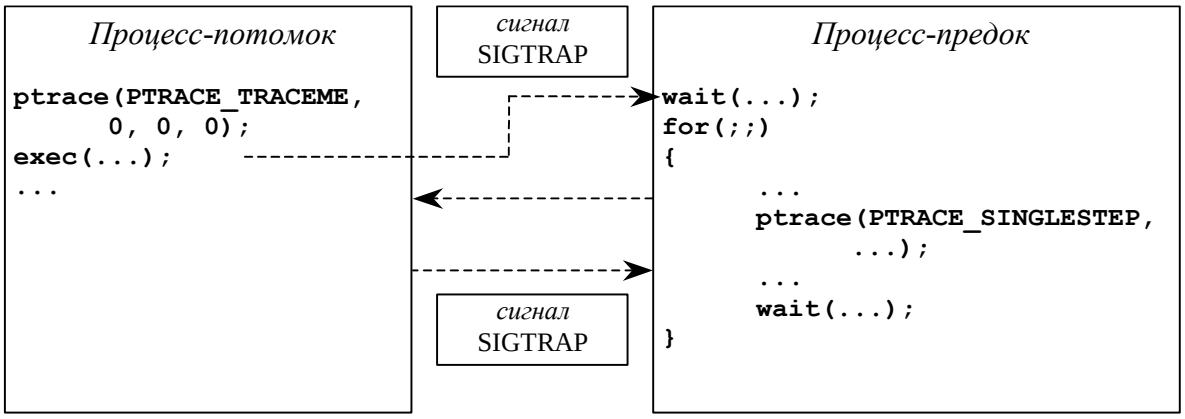


Рис. 98. Общая схема трассировки процессов.

Отцовский процесс формирует сыновний процесс и ожидает его завершения посредством обращения к системному вызову *wait()*. Сыновний процесс подтверждает право родителя его трассировать (обращаясь к системному вызову *ptrace()* с кодом cmd = PTRACE_TRACEME и нулевыми оставшимися аргументами). После чего он меняет свое тело на тело процесса, которое необходимо отлаживать (посредством обращения к одному из системных вызовов *exec()*). После смены тела данный процесс приостановится на точке входа и к нему приходит сигнал SIGTRAP, информацию о котором получает родительский процесс (через обращение к системному вызову wait()). Именно с этого момента начинается отладка: отлаживаемый процесс загружен, он готов к отладке и находится в начальной точке процесса. Дальше родительский трассирующий процесс может делать все те действия, которые ему необходимы по отладке: запустить процесс с точки останова, читать содержимое различных переменных, устанавливать контрольные точки и т.п.

Отладчики бывают двух типов: ***адресно-кодовыми*** и ***символьными***. Адресно-кодовые отладчики оперируют адресами тела отлаживаемого процесса, в то время как символьные отладчики позволяют оперировать объектами языка, т.е. переменными и операторами языка.

Механизм организации контрольной точки в адресно-кодовом отладчике достаточно простой. Пусть нам необходимо по некоторому адресу А установить контрольную точку, т.е. организовать так, чтобы при приходе управления в эту точку программы процесс всегда приостанавливался, и управление передавалось процессу-отладчику. В отладчике имеется таблица контрольных точек, в каждой строке которой присутствует адрес некоторой контрольной точки и оригинальный код (содержимое) отлаживаемого процесса, взятый по данному адресу. Для установки контрольной точки по адресу А необходимо тем или иным способом остановить отлаживаемый процесс (либо он останавливается при входе, либо отладчик посылает ему соответствующий сигнал). Затем отладчик читает из сегмента кода машинное слово по адресу А (посредством обращения к системному вызову *ptrace()*), которое он записывает в соответствующую строку таблицы контрольных точек, тем самым, сохраняя оригинальное содержимое тела трассируемого процесса. Далее по адресу А в сегмент кода записывается команда, которая вызывает прерывание, и, соответственно, приход предопределенного события (сигнала). Примером может служить команда деления на ноль. После этого запускаем отлаживаемый процесс на исполнение.

Итак, трассируемый процесс исполняется, и управление, наконец, передается на машинное слово по адресу А. Происходит деление на ноль. Соответственно, происходит прерывание, система передает сигнал. И отладчик через системный вызов *wait()* получает код возврата и «понимает», что в сыновнем процессе случилось деление на ноль. Отладчик посредством системного вызова *ptrace()* читает адрес останова в контексте сыновнего процесса. Далее анализируется причина останова. Если причиной останова явилось деление на ноль, то возможны две ситуации: либо это действительно деление на ноль как ошибка, либо это деление на ноль как контрольная точка. Для идентификации этой ситуации отладчик обращается к таблице контрольных точек и ищет там адрес останова подчиненного процесса. Если в данной таблице указанный адрес встретился, то это означает, что отлаживаемый процесс пришел на контрольную точку (иначе деление на ноль отрабатывается как ошибка).

Находясь в контрольной точке, отладчик может производить различные манипуляции с трассируемым процессом (читать данные, устанавливать новые контрольные точки и т.п.). Далее встает вопрос, как корректно продолжить подчиненный процесс. Для этого производятся следующие действия. В сегмент кода по адресу А записывается оригинальное машинное слово (которое берётся из таблицы контрольных точек). После этого системным вызовом *ptrace()* включаем пошаговый режим выполнения процесса. И выполняем одну команду (которую только что записали по адресу А). Из-за включенного режима пошаговой отладки подчиненный процесс снова остановится. Затем отладчик выключает режим пошаговой отладки и запускает процесс с текущей точки.

Для ***организации контрольных точек в символьных отладчиках*** необходима информация, собранная на этапах компиляции и редактирования связей. Если с компилятором связан символьный отладчик, то компилятор формирует некоторую специализированную базу данных, в которой находится информация по всем именам, используемым в программе. Для каждого имени определены диапазоны видимости и существования этого имени, его тип (статическая переменная, автоматическая переменная, регистровая переменная и т.п.). А также данная база содержит информацию обо всех операторах (диапазон начала и конца оператора и т.п.).

Предположим, необходимо просмотреть содержимое некоторой переменной *v*. Для этого трассируемый процесс должен быть остановлен. По адресу останова можно определить, в какой точке программы произошел останов. На основе информации об этой точке программы можно (обратившись к содержимому базы данных) определить то пространство имен, которое доступно из этой точки. Если интересующая нас переменная *v* оказалась доступна, то работа продолжается: происходит обращение к базе данных и определяется тип данной переменной. Если тип переменной *v* — статическая переменная, то в соответствующей записи будет адрес, по которому размещена данная переменная (этот адрес станет известным на этапе редактирования связей). И с помощью *ptrace()* отладчик берет содержимое по этому адресу. Также из базы данных берется информация о типе переменной (char, float, int и т.п.), и на основе этой информации пользователю соответствующим образом отображается значение указанной переменной *v*.

Рассмотрим теперь случай, когда переменная *v* – автоматическая переменная или формальный параметр. Такие переменные обычно размещаются в блоке на фиксированном смещении относительно вершины стека (т.е. для этих переменных в качестве адреса фиксируется смещение от вершины стека). Чтобы прочитать содержимое переменной подобного типа, необходимо обратиться к контексту процесса, считать значение регистра-указателя стека (адрес вершины стека), после чего к содержимому этого регистра необходимо прибавить смещение и по получившемуся адресу обратиться к соответствующему сегменту.

Если переменная *v* — регистровая переменная, то происходит обращение к базе данных, считывается номер регистра, затем идет обращение к сегменту кода и считывается содержимое нужного регистра.

При записи значений в переменные происходит та же последовательность действий.

Если необходимо установить контрольную точку на оператор, то через базу данных определяется диапазон адресов оператора, определяется начальный адрес, а дальше производятся действия по описанной выше схеме.

3.1.6.2 Пример. Трассировка процессов.

```
/* Процесс-сын: */
int main(int argc, char **argv)
{
     /* деление на ноль – здесь процессу будет послан сигнал
     SIGFPE – floating point exception */
     return argc/0;
}

/* Процесс-родитель: */
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <signal.h>
#include <sys/ptrace.h>
#include <sys/user.h>
#include <sys/wait.h>

int main(int argc, char *argv[])
{
     pid_t pid;
     int status;
     struct user_regs_struct REG;
     if  ((pid = fork()) == 0)
     {
          /*находимся в процессе-потомке, разрешаем
          трассировку*/
          ptrace(PTRACE_TRACEME, 0, 0, 0);
          execl("son", "son", 0);   /* замещаем тело процесса
          */
          /* здесь процесс-потомок будет остановлен с
          сигналом SIG_TRAP, ожидая команды продолжения
          выполнения от управляющего процесса*/
     }
     /* в процессе-родителе */
     while (1)
     {
          /* ждем, когда отлаживаемый процесс приостановится
          */
          wait(&status);
          /*читаем содержимое регистров отлаживаемого
          процесса */
          ptrace(PTRACE_GETREGS, pid, &REG, &REG);
          /* выводим статус отлаживаемого процесса, номер
          сигнала, который его остановил и значения
          прочитанных регистров  */
```

```
            printf("signal = %d, status = %#x, EIP=%#x
            ESP=%#x\n", WSTOPSIG(status), status, REG.eip,
            REG.esp);
            if (WSTOPSIG(status) != SIGTRAP)
            {
                if (!WIFEXITED(status))
                {
                    /* завершаем выполнение трассируемого
                    процесса */
                    ptrace (PTRACE_KILL, pid, 0, 0);
                }
                break;
            }
            /* разрешаем выполнение трассируемому процессу */
            ptrace (PTRACE_CONT, pid, 0, 0);
        }
    }
```

3.2 Система межпроцессного взаимодействия IPC (Inter-Process Communication)

Все рассмотренные выше средства взаимодействия процессов не обладают достаточной универсальностью и имеют те или иные недостатки: так, сигналы несут в себе слишком мало информации и не могут использоваться для передачи сколь либо значительных объемов данных; неименованный канал должен быть создан до того, как порождается процесс, который будет осуществлять коммуникацию; именованный канал, хотя и лишен этого недостатка, требует одновременной работы с ним обоих процессов, участвующих в коммуникации. Поэтому практически все UNIX-системы поддерживают более мощные и развитые средства межпроцессного взаимодействия.

Система IPC (известная так же, как IPC System V) позволяет обеспечивать взаимодействие произвольных процессов в пределах локальной машины. Для этого она предоставляет взаимодействующим процессам возможность использования ***общих***, или ***разделяемых***, ресурсов. Эти ресурсы могут существовать в системе с момента создания до момента их принудительного удаления либо в течение сеанса работы операционной системы (вне зависимости от наличия процессов, которые их используют). Такие разделяемые ресурсы можно подразделить на три типа:

- ***Очередь сообщений*** — это разделяемый ресурс, позволяющий организовывать очереди сообщений: один процесс может в эту очередь положить сообщение, а другой процесс — прочитать его. Данный механизм имеет возможность блокировок, поэтому его можно использовать и как средство передачи информации между взаимодействующими процессами, и как средство их синхронизации.
- ***Массив семафоров*** — ресурс, представляющий собой массив из N элементов, где N задается при создании данного ресурса, и каждый из элементов является семафором **IPC** (а **не семафором Дейкстры**: семафор Дейкстры так или иначе является формализмом, не опирающимся ни на какую реализацию, а семафор IPC — конкретной программной реализацией семафора в ОС). Семафоры IPC предназначены, в первую очередь, для использования в качестве средств организации синхронизации.
- ***Общая***, или ***разделяемая***, ***память***, которая представляется процессу как указатель на область памяти, которая является общей для двух и более процессов. Т.е. внутри процесса некоторый указатель можно установить на начало данной области и работать далее с этой областью, как с массивом. Все изменения, которые сделает данный

процесс, будут видны другим процессам. Разделяемая память IPC почти не обладает никакими средствами синхронизации (т.е. существует очень слабо развитый механизм взаимных блокировок, но мы на нем не будем останавливаться).

Для организации совместного использования разделяемых ресурсов необходим некоторый ***механизм именования ресурсов***, используя который взаимодействующие процессы смогут работать с данным конкретным ресурсом. Для этих целей используется система ключей: при создании ресурса с ним ассоциируется ***ключ*** — целочисленное значение. Жесткого ограничения к выбору этих ключей нет. Т.е. при создании ресурса, (например, общей памяти) процесс-создатель ассоциирует с ним конкретный ключ — например, 25. Теперь все процессы, желающие работать с той же общей памятью, должны, во-первых, обладать соответствующими правами, а во-вторых, заявить, что они желают работать с ресурсом общая память с ключом 25. И они будут с ней работать. Но тут возможны коллизии из-за случайного совпадения ключей различных ресурсов.

Во избежание коллизий система предлагает некоторую унификацию именования IPC-ресурсов. Для генерации уникальных ключей в системе имеется библиотечная функция *ftok()*.

```
#include <sys/types.h>
#include <sys/ipc.h>
key_t ftok(char *filename, char proj);
```

Первый параметр данной функции — полное имя существующего файла. Второй параметр — это уточняющая информация (значение символьной переменной; в частности, этот параметр может использоваться для поддержания разных версий программы).

Итак, функция *ftok()* обращается к указанному файлу, считывает его атрибуты и, используя значение второго аргумента, генерирует уникальный ключ (уникальное целое число). Стоит особо обратить внимание, что первый параметр должен указывать на существующий файл, и, что не менее важно, при генерации ключа используются атрибуты файла. Это означает, что если один процесс для генерации ключа ссылается на некоторый файл, создает ресурс, потом этот файл уничтожается и создается другой файл с тем же именем (но, скорее всего, с другими атрибутами), то другой процесс, желающий получить ключ к созданному ресурсу, не сможет этого сделать, т.к. функция *ftok()* будет генерировать иное значение.

Каждый ресурс IPC имеет атрибут владельца. Владельцем считается тот процесс (его идентификация), который создал данный ресурс, и, соответственно, удалить данный ресурс может только владелец. Но стоит отметить, что существует механизм передачи прав владения от процесса процессу.

С каждым ресурсом, так же как и с файлами, связаны три категории прав (права владельца, группы и остальных пользователей). Но в каждой категории имеются лишь права на чтение и запись: права на выполнение нет.

Ресурсы IPC могут существовать без процессов, их создавших. Это означает, что созданный разделяемый ресурс будет существовать до тех пор, пока его явно не удалят либо до перезапуска системы.

Итак, имеется три группы IPC-ресурсов, с каждой из которой связан свой набор функций по работе с конкретным типом ресурса. Но функциональная структура работы идентична. В частности, среди этих наборов можно выделить функции, имеющие суффикс *get* (а префиксная часть представляет собою аббревиатуру имени ресурса). Среди параметров данных функций присутствует ключ, о котором шла речь выше, а также некоторые флаги. Соответственно, эти функции в зависимости от флагов позволяют работать либо в режиме создания ресурса, либо в режиме подключения к существующему IPC-ресурсу, ассоциированному с указываемым ключом.

```
<ResourceName>get(key, ..., flags);
```

Параметр *key* – ключ, который мы хотим ассоциировать с разделяемым ресурсом.

Параметр флаги (*flags*) задаёт права доступа к ресурсу и режимы работы с ресурсом. Этот параметр может быть комбинацией различных флагов. Основные из них приведены ниже.

- **IPC_PRIVATE** — данный флаг определяет создание частного IPC-ресурса, доступного только процессу, который его создал, не доступного остальным процессам. Т.е. функция *get* при наличии данного флага всегда открывает новый ресурс, к которому никто другой не может подключиться. Данный флаг позволяет использовать разделяемый ресурс между родственными процессами (поскольку дескриптор созданного ресурса передается при наследовании сыновним процессам).
- **IPC_CREAT** — если данного флага нет среди параметров функции *get*, то это означает, что процесс хочет подключиться к существующему ресурсу. В этом случае, если такой ресурс существует и права доступа позволяют к нему обратиться, то процесс получит дескриптор ресурса и продолжит работу. Иначе функция *get* вернет код ошибки -1, а переменная *errno* будет содержать код ошибки. Если же при вызове функции *get* данный флаг установлен, то функция работает на создание или подключение к существующему ресурсу. В данном случае возможно возникновение ошибки из-за нехватки прав доступа в случае существования ресурса. Но при установленном флаге встает вопрос, кто является владельцем ресурса, и, соответственно, кто его должен удалять (поскольку каждый процесс либо подключается к существующему ресурсу, либо создает новый).
- Для разрешения данной проблемы используется следующий флаг.
  IPC_EXCL — используя данный флаг в паре с флагом IPC_CREAT, функция *get* будет работать только на создание нового ресурса. Если же ресурс будет уже существовать, то функция *get* вернет -1, а переменная *errno* будет содержать код соответствующей ошибки.

Ниже приводится список некоторых ошибок, возможных при вызове функции *get*, возвращаемых в переменной *errno*:

- **ENOENT** — ресурс не существует, и не указан флаг IPC_CREAT;
- **EEXIST** — ресурс существует, и установлены флаги IPC_CREAT | IPC_EXCL;
- **EACCESS** — не хватает прав доступа на подключение.

3.2.1 Очередь сообщений IPC

Система предоставляет возможность создания некоторого функционально расширенного аналога канала, причём главное отличие заключается в том, что сообщения в очереди сообщений IPC типизированы. Каждое сообщение помимо своей содержательной части имеет атрибут ***тип сообщения*** (неотрицательное целое значение). Тогда очередь сообщений можно рассматривать с двух позиций: во-первых, как сквозную очередь (когда все сообщения, вне зависимости от типа, рассматриваются как единая очередь), а, во-вторых, как суперпозицию очередей однотипных сообщений (Рис. 99). При этом способ интерпретации допускает одновременно различные типы интерпретации. Непосредственный выбор интерпретации определяется в момент считывания сообщения из очереди.

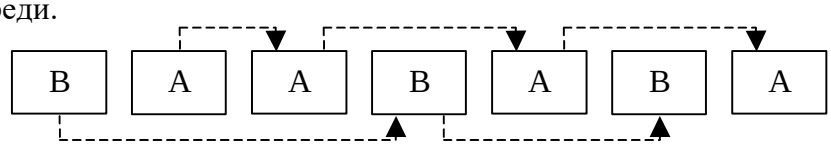


Рис. 99. Очередь сообщений IPC.

Для организации работы с очередью предусмотрен набор функций. Во-первых, это уже упомянутая функция создания/доступа к очереди сообщений.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/message.h>

int msgget(key_t key, int msgflag);
```

У данной функции два параметра: ключ и флаги. В случае успешного выполнения функция возвращает положительный дескриптор очереди, иначе возвращается -1.

Рассмотрим теперь функции отправки и приема сообщений. Для ***отправки сообщений*** служит функция *msgsnd()*.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgsnd(int msqid, const void *msgp, size_t msgsz,
     int msgflg);
```

Первый аргумент — идентификатор очереди, полученный в результате вызова *msgget()*. Второй аргумент — указатель на буфер, содержащий реальные данные и тип сообщения, подлежащего посылке в очередь, в третьем аргументе указывается размер буфера. В качестве буфера используется структура, содержащая следующие поля:

- **long msgtype** — тип сообщения (только положительное длинное целое);
- **char msgtext[]** — данные (тело сообщения).

В заголовочном файле **`<sys/msg.h>`** определена константа **`MSGMAX`**, описывающая максимальный размер тела сообщения. При попытке отправить сообщение, у которого число элементов в массиве **`msgtext`** превышает это значение, системный вызов вернет –1.

Последний аргумент функции — это флаги. Среди разнообразных флагов можно выделить те, которые определяют режим блокировки при отправке сообщения. Если флаг равен 0, то вызов будет блокироваться, если для отправки недостаточно системных ресурсов. Можно установить флаг IPC_NOWAIT, который позволяет работать без блокировки: тогда, в случае возникновения ошибки при отправке сообщения, вызов вернет -1, а переменная *errno* будет содержать соответствующий код ошибки.

Для ***получения сообщений*** используется функция *msgrcv()*.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgrcv(int msqid, void *msgp, size_t msgsz,
          long msgtyp, int msgflg);
```

Первые три аргумента аналогичны аргументам предыдущего вызова: дескриптор очереди, указатель на буфер, куда следует поместить данные, и максимальный размер (в байтах) тела сообщения, которое можно туда поместить. Буфер, используемый для приема сообщения, должен иметь структуру, описанную выше.

Четвертый аргумент указывает тип сообщения, которое процесс желает получить. Если значение этого аргумента есть 0, то будет получено сообщение любого типа (т.е. идет работа со сквозной очередью). Если значение аргумента msgtyp больше 0, из очереди будет извлечено сообщение указанного типа. Если же значение аргумента msgtyp отрицательно,

то тип принимаемого сообщения определяется как наименьшее значение среди типов, которые меньше модуля msgtyp. В любом случае, при доступе используется стратегия FIFO.

Последним аргументом является комбинация (побитовое сложение) флагов. Если среди флагов не указан IPC_NOWAIT, и в очереди не найдено ни одного сообщения, удовлетворяющего критериям выбора, процесс будет заблокирован до появления такого сообщения. (Однако, если такое сообщение существует, но его длина превышает указанную в аргументе msgsz, то процесс заблокирован не будет, и вызов сразу вернет -1; сообщение при этом останется в очереди). Если же флаг IPC_NOWAIT указан, и в очереди нет ни одного необходимого сообщения, то вызов сразу вернет -1.

Процесс может также указать флаг MSG_NOERROR: в этом случае он может прочитать сообщение, даже если его длина превышает указанную емкость буфера. Тогда в буфер будет записано первые msgsz байт тела сообщения, а остальные данные отбрасываются.

В случае успешного завершения функция возвращает количество успешно прочитанных байтов в теле сообщения.

Следующая группа функций — это функции ***управления ресурсом*** – очередью сообщений. Эти функции обеспечивают в общем случае изменение режима функционирования ресурса, в т.ч. и удаление ресурса.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgctl(int msqid, int cmd, struct msgid_ds *buf);
```

Данная функция используется для получения или изменения управляющих параметров, связанных с очередью, и уничтожения очереди. Ее аргументы — идентификатор ресурса, команда, которую необходимо выполнить, и структура, описывающая управляющие параметры очереди. Тип msgid_ds представляет собой структуру, в полях которой хранятся права доступа к очереди, статистика обращений к очереди, ее размер и т.п.

Возможные значения аргумента cmd:

- **IPC_STAT** — скопировать структуру, описывающую управляющие параметры очереди по адресу, указанному в параметре buf;
- **IPC_SET** — заменить структуру, описывающую управляющие параметры очереди, на структуру, находящуюся по адресу, указанному в параметре buf;
- **IPC_RMID** — удалить очередь. Удалить очередь может только процесс, у которого эффективный идентификатор пользователя совпадает с владельцем или создателем очереди, либо процесс с правами привилегированного пользователя.

Пример. Использование очереди сообщений. В приведенном ниже примере участвуют три процесса: основной процесс, процесс A и процесс B. Основной процесс считывает со стандартного устройства ввода текстовую строку. Если она начинается на букву A, то эта строка посылается процессу A, если на B — то процессу B, если на Q — то обоим процессам (в этом случае основной процесс ждет некоторое время, затем удаляет очередь сообщений и завершается). Процессы A и B считывают из очереди адресуемые им сообщения и распечатывают их на экране. Если пришедшее сообщение начинается с буквы Q, то процесс завершается.

```
/* ОСНОВНОЙ ПРОЦЕСС */
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/message.h>
```

```
#include <stdio.h>
/* декларация структуры сообщения */
struct
{
     long mtype;          /* тип сообщения */
     char Data[256];      /* сообщение */
} Message;

int main(int argc, char **argv)
{
     key_t key;
     int msgid;
     char str[256];

     /* получаем уникальный ключ, однозначно определяющий
     доступ к ресурсу данного типа */
     key = ftok("/usr/mash",'s');
     /* создаем новую очередь сообщений,
     0666 определяет права доступа */
     msgid = msgget(key, 0666 | IPC_CREAT | IPC_EXCL);
     /* запускаем вечный цикл */
     for(;;)
     {
          gets(str); /* читаем из стандартного ввода строку */
          /* и копируем ее в буфер сообщения */
          strcpy(Message.Data, str);
          /* анализируем первый символ прочитанной строки */
          switch(str[0])
          {
               case 'a':
               case 'A':
                    /* устанавливаем тип 1 для ПРОЦЕССА A*/
                    Message.mtype = 1;
                    /* посылаем сообщение в очередь */
                    msgsnd(msgid, (struct msgbuf*) (&Message),
                                   strlen(str)+1, 0);
                    break;
               case 'b':
               case 'B':
                    /* устанавливаем тип 2 для ПРОЦЕССА A*/
                    Message.mtype = 2;
                    msgsnd(msgid, (struct msgbuf*) (&Message),
                                   strlen(str)+1, 0);
                    break;
               case 'q':
               case 'Q':
                    Message.mtype = 1;
                    msgsnd(msgid, (struct msgbuf*) (&Message),
                                   strlen(str)+1, 0);
                    Message.mtype = 2;
                    msgsnd(msgid, (struct msgbuf*) (&Message),
                                   strlen(str)+1, 0);
                    /* ждем получения сообщений
```

```
                    процессами A и B */
                    sleep(10);[4]
                    /* уничтожаем очередь */
                    msgctl(msgid, IPC_RMID, NULL);
                    exit(0);
               default:
                    /* игнорируем остальные случаи */
                    break;
          }
     }
}

/* ПРИНИМАЮЩИЙ ПРОЦЕСС A (процесс B будет аналогичным) */
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/message.h>
#include <stdio.h>
struct
{
     long mtype;          /* тип сообщения */
     char Data[256];      /* сообщение */
} Message;

int main(int argc, char **argv)
{
     key_t key;
     int msgid;

     /* получаем ключ по тем же параметрам */
     key = ftok("/usr/mash",'s');
     /*подключаемся к очереди сообщений */
     msgid = msgget(key, 0666);
     /* запускаем вечный цикл */
     for(;;)
     {
          /* читаем сообщение с типом 1 для ПРОЦЕССА A */
          msgrcv(msgid, (struct msgbuf*) (&Message), 256, 1, 0);[5]
          printf("%s", Message.Data);
          if(Message.Data[0] == 'q' || Message.Data[0] == 'Q')
               break;
     }
     return 0;
}
```

Благодаря наличию типизации сообщений, очередь сообщений предоставляет возможность мультиплексировать сообщения от различных процессов, при этом каждая пара взаимодействующих через очередь процессов может использовать свой тип сообщений, и таким образом, их данные не будут смешиваться.

---

[4] В данном случае это решение не совсем корректно, поскольку делается предположение, что процессы A и B за 10 единиц времени должны получить последние сообщения. Но для простоты решения мы опускаем проблемы синхронизации.

[5] В этом случае возможна некорректная работа, если процессы A и/или B запускаются раньше основного процесса. В этом случае они обратятся к пока еще не созданному ресурсу, что приведет к ошибке.

Пример. Очередь сообщений. Модель «клиент-сервер». В приведенном ниже примере имеется совокупность взаимодействующих процессов. Эта модель несимметричная: один из процессов назначается сервером, и его задачей становится обслуживание запросов остальных процессов-клиентов. В данном примере сервер принимает запросы от клиентов в виде сообщений (из очереди сообщений) с типом 1. Тело сообщения-запроса содержит идентификатор клиентского процесса (PID), который выслал данный запрос. Для каждого запроса сервер генерирует ответ, которое также посылает через очередь сообщений, но посылаемое сообщение будет иметь тип, равный идентификатору процесса-адресата. В свою очередь, клиентский процесс будет брать из очереди сообщений сообщения с типом, равным его идентификатору.

```
/* СЕРВЕР */
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char **argv)
{
     struct
     {
          long mestype;
          char mes[100];
     } messageto;

     struct
     {
          long mestype;
          long mes;
     } messagefrom;

     key_t key;
     int mesid;

     key = ftok("example", 'r');
     mesid = msgget (key, 0666 | IPC_CREAT | IPC_EXCL );
     while(1)
     {
          msgrcv(mesid, &messagefrom, sizeof(messagefrom) -
                                        sizeof(long), 1, 0);
          messageto.mestype = messagefrom.mes;
          strcpy(messageto.mes, "Message for client");
          msgsnd (mesid, &messageto, sizeof(messageto) -
                                        sizeof(long),0);
     }
}

/* КЛИЕНТ */
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
```

```
int main(int argc, char **argv)
{
     struct
     {
          long mestype;
          long mes;
     } messageto;

     struct
     {
          long mestype;
          char mes[100];
     } messagefrom;

     key_t key;
     int mesid;

     long pid = getpid();
     key = ftok("example", 'r');
     mesid = msgget (key, 0666);
     messageto.mestype = 1;
     messageto.mes = pid;
     msgsnd(mesid, &messageto, sizeof(messageto) -
                                   sizeof(long),0);
     msgrcv(mesid,&messagefrom, sizeof(messagefrom) -
                                   sizeof(long),pid,0);
     printf("%s", messagefrom.mes);
     return 0;
}
```

В серверном процессе декларируются две структуры для принимаемого (meassagefrom) и посылаемого (messageto) сообщений, а также ключ key и дескриптор очереди сообщений mesid. Затем сервер предпринимает традиционные действия: получает ключ, а по нему — дескриптор очереди сообщений. Затем он входит в бесконечный цикл, в котором и обрабатывает клиентские запросы. Каждая итерация цикла выглядит следующим образом. Из очереди выбирается сообщение с типом 1 (это сообщения с запросами от клиентов). Из тела этого сообщения считывается информация об идентификаторе клиента, и этот идентификатор сразу заносится в поле типа посылаемого сообщения. Затем сервер генерирует тело посылаемого сообщения, после чего отправляет созданное сообщение в очередь. На этом итерация цикла завершается.

Клиентский процесс имеет аналогичные декларации (за исключением того, что теперь посылаемое и принимаемое сообщения поменялись ролями). Далее клиент получает свой идентификатор процесса, записывает его в тело сообщения-запроса, которому устанавливает тип 1. После этого отправляет запрос в очередь, принимает из очереди ответ (сообщение с типом, равным его собственному идентификатору процесса) и завершается.

<u>В рассмотренных выше примерах предполагается, определеннй порядок начального запуска взаимодействующих процессов.</u>

Рекомендации.

1. Рассмотреть, в чем состоит проблема, требующая соблюдения порядка запуска процессов.

2. Предложить корректные решения.

3.2.2 Разделяемая память IPC

Механизм разделяемой памяти позволяет нескольким процессам получить отображение некоторых страниц из своей виртуальной памяти на общую область физической памяти (Рис. 100). Благодаря этому, данные, находящиеся в этой области памяти, будут доступны для чтения и модификации всем процессам, подключившимся к данной области памяти.

Процесс, подключившийся к разделяемой памяти, может затем получить указатель на некоторый адрес в своем виртуальном адресном пространстве, соответствующий данной области разделяемой памяти. После этого он может работать с этой областью памяти аналогично тому, как если бы она была выделена динамически (например, путем обращения к *malloc()*), однако, как уже говорилось, разделяемая область памяти не уничтожается автоматически даже после того, как процесс, создавший или использовавший ее, перестанет с ней работать.

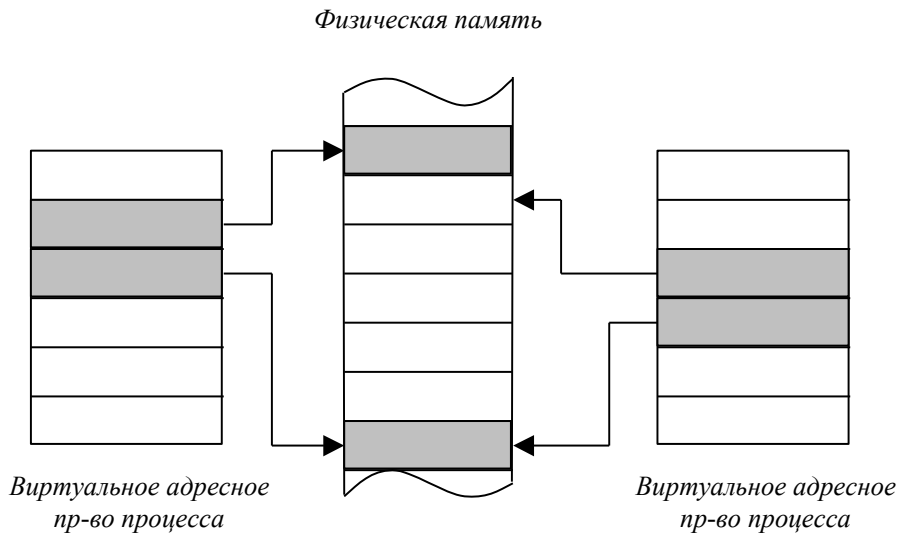


Рис. 100. Разделяемая память.

Рассмотрим набор системных вызовов для работы с разделяемой памятью. Для создания/подключения к ресурсу разделяемой памяти IPC используется функция *shmget()*.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmget (key_t key, int size, int shmflg);
```

Аргументы данной функции: *key* — ключ для доступа к разделяемой памяти; *size* задает размер области памяти, к которой процесс желает получить доступ. Если в результате вызова *shmget()* будет создана новая область разделяемой памяти, то ее размер будет соответствовать значению *size*. Если же процесс подключается к существующей области разделяемой памяти, то значение *size* должно быть не более ее размера, иначе вызов вернет -1. Заметим, что если процесс при подключении к существующей области разделяемой памяти указал в аргументе *size* значение, меньшее ее фактического размера, то впоследствии он сможет получить доступ только к первым *size* байтам этой области.

Третий параметр определяет флаги, управляющие поведением вызова. Подробнее алгоритм создания/подключения разделяемого ресурса был описан выше.

В случае успешного завершения вызов возвращает положительное число — дескриптор области памяти, в случае неудачи возвращается -1. Но наличие у процесса дескриптора разделяемой памяти не дает ему возможности работать с ресурсом, поскольку при работе с памятью процесс работает в терминах адресов. Поэтому необходима еще одна функция, которая присоединяет полученную разделяемую память к адресному пространству процесса, — это функция *shmat()*.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

char *shmat(int shmid, char *shmaddr, int shmflg);
```

При помощи этого вызова процесс подсоединяет область разделяемой памяти, дескриптор которой указан в *shmid*, к своему виртуальному адресному пространству. После выполнения этой операции процесс сможет читать и модифицировать данные, находящиеся в области разделяемой памяти, адресуя ее как любую другую область в своем собственном виртуальном адресном пространстве.

В качестве второго аргумента процесс может указать виртуальный адрес в своем адресном пространстве, начиная с которого необходимо подсоединить разделяемую память. Если в качестве значения этого аргумента передается 0, то это означает, что система сама может выбрать адрес начала разделяемой памяти. Передача конкретного адреса (положительного целого) в этом параметре имеет смысл лишь в определенных случаях, и это означает, что процесс желает связать начало области разделяемой памяти с конкретным адресом. В подобных случаях необходимо учитывать, что возможны коллизии с имеющимся адресным пространством.

Третий аргумент представляет собой комбинацию флагов. В качестве значения этого аргумента может быть указан флаг SHM_RDONLY, который указывает на то, что подсоединяемая область будет использоваться только для чтения. Реализация тех или иных флагов будет зависеть от аппаратной поддержки соответствующего свойства. Если аппаратура не поддерживает защиту памяти от записи, то при установке флага SHM_RDONLY ошибка, связанная с модификацией содержимого памяти, не сможет быть обнаружена (поскольку программным способом невозможно выявить, в какой момент происходит обращение на запись в данную область памяти).

Эта функция возвращает адрес (указатель) в виртуальном адресном пространстве процесса, начиная с которого будет отображаться присоединяемая разделяемая память. И с этим указателем можно работать стандартными средствами языка С. В случае неудачи вызов возвращает -1.

Соответственно, для открепления разделяемой памяти от адресного пространства процесса используется функция *shmdt()*.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmdt(char *shmaddr);
```

Данный вызов позволяет отсоединить разделяемую память, ранее присоединенную посредством вызова *shmat()*. Параметр shmaddr — адрес прикрепленной к процессу памяти, который был получен при вызове *shmat()*. В случае успешного выполнения функция вернет значение 0, в случае неудачи возвращается -1.

И, напоследок, рассмотрим функцию *shmctl()* управления ресурсом разделяемая память.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

Данный вызов используется для получения или изменения процессом управляющих параметров, связанных с областью разделяемой памяти, наложения и снятия блокировки на нее и ее уничтожения. Аргументы вызова — дескриптор области памяти, команда, которую необходимо выполнить, и структура, описывающая управляющие параметры области памяти. Тип **shmid_ds** описан в заголовочном файле **<sys/shm.h>**, и представляет собой структуру, в полях которой хранятся права доступа к области памяти, ее размер, число процессов, подсоединенных к ней в данный момент, и статистика обращений к области памяти.

Возможные значения аргумента **cmd**:

IPC_STAT – скопировать структуру, описывающую управляющие параметры области памяти по адресу, указанному в параметре **buf**

IPC_SET – заменить структуру, описывающую управляющие параметры области памяти, на структуру, находящуюся по адресу, указанному в параметре **buf**. Выполнить эту операцию может процесс, у которого эффективный идентификатор пользователя совпадает с владельцем или создателем очереди, либо процесс с правами привилегированного пользователя, при этом процесс может изменить только владельца области памяти и права доступа к ней.

IPC_RMID – удалить разделяемую память - помечает для удаления. происходит пока только тогда, когда последний процесс, работавший с разделяемой памятью, не "отвяжется". Ресурс может только процесс, у которого эффективный идентификатор пользователя совпадает с владельцем или создателем очереди, либо процесс с правами привилегированного пользователя.

SHM_LOCK, SHM_UNLOCK – блокировать или разблокировать область памяти. Выполнить эту операцию может только процесс с правами привилегированного пользователя.

Это единственные средства синхронизации в данном ресурсе, их реализация должна поддерживаться аппаратурой.

Пример. Схема работы с общей памятью в рамках одного процесса. В данном примере процесс создает ресурс разделяемая память, размером в 100 байт (и с соответствующими флагами), присоединяет ее к своему адресному пространству; при этом указатель на начало данной области сохраняется в переменной *shmaddr*. Дальше процесс производит различные манипуляции, а перед своим завершением он удаляет данную область разделяемой памяти. В данном примере считается, что **putm()** и **waitprocess()** – некие пользовательские функции, определенные в другом месте.

```
int main(int argc, char **argv)
{
     key_t key;
     char *shmaddr;

     key = ftok("/tmp/ter", 'S');
     shmid = shmget(key, 100, 0666 | IPC_CREAT | IPC_EXCL);
     shmaddr = shmat(shmid, NULL, 0);
     /* работаем с разделяемой памятью, как с обычной */
     putm(shmaddr);
     waitprocess();
```

```
    shmctl(shmid, IPC_RMID, NULL);
    return 0;
}
```

3.2.3 Массив семафоров IPC

Семафоры представляют собой одну из форм IPC и обычно используются для синхронизации доступа нескольких процессов к разделяемым ресурсам, так как сами по себе другие средства IPC не предоставляют механизма синхронизации.

Как уже говорилось, семафор представляет собой особый вид числовой переменной, над которой определены две неделимые операции: уменьшение ее значения с возможным блокированием процесса и увеличение значения с возможным разблокированием одного из ранее заблокированных процессов. Объект System V IPC представляет собой массив (набор) семафоров.

Рассмотрим основные вызовы для работы с семафорами в UNIX System V.

3.2.3.1 Доступ к семафору

Для получения доступа к массиву семафоров (или его создания) используется системный вызов *semget()*:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semget (key_t key, int nsems, int semflag);
```

Первый параметр функции **semget()** – ключ для доступа к разделяемому ресурсу, второй - количество семафоров в создаваемом наборе (длина массива семафоров) и третий параметр – флаги, управляющие поведением вызова. Подробнее процесс создания разделяемого ресурса описан выше. Отметим семантику прав доступа к такому типу разделяемых ресурсов, как семафоры: процесс, имеющий право доступа к массиву семафоров по чтению, может проверять значение семафоров; процесс, имеющий право доступа по записи, может как проверять, так и изменять значения семафоров.

В случае успеха вызов **semget()** возвращает положительный дескриптор созданного разделяемого ресурса, в случае неудачи -1.

3.2.3.2 Операции над семафором

Используя полученный дескриптор, можно производить изменять значения одного или нескольких семафоров в наборе, а также проверять их значения на равенство нулю, для чего используется системный вызов **semop()**:

```
#include <sys/types.h>
#include<sys/ipc.h>
#include<sys/sem.h>

int semop (int semid, struct sembuf *semop, size_t nops)
```

Этому вызову передаются следующие аргументы:

semid – дескриптор массива семафоров;

semop – массив из объектов типа **struct sembuf**, каждый из которых задает одну операцию над семафором;

nops – длина массива **semop**. Количество семафоров, над которыми процесс может одновременно производить операцию в одном вызове **semop()**, ограничено константой **SEMOPM**, описанной в файле **<sys/sem.h>**. Если процесс попытается вызвать **semop()** с параметром **nops**, большим этого значения, этот вызов вернет неуспех.

Структура имеет **sembuf** вид:

```
struct sembuf {
     short sem_num;  /* номер семафора в векторе */
     short sem_op;   /* производимая операция */
     short sem_flg;  /* флаги операции */
}
```

Поле операции в структуре интерпретируется следующим образом. Пусть значение семафора с номером **sem_num** равно **sem_val**.

1. если значение операции не равно нулю:
   - оценивается значение суммы **sem_val + sem_op**.
   - если эта сумма больше либо равна нулю, то значение данного семафора устанавливается равным этой сумме: **sem_val = sem_val + sem_op**
   - если же эта сумма меньше нуля, то действие процесса будет приостановлено до тех пор, пока значение суммы **sem_val + sem_op** не станет больше либо равно нулю, после чего значение семафора устанавливается равным этой сумме: **sem_val = sem_val + sem_op**
2. Если код операции **sem_op** равен нулю:
   - Если при этом значение семафора (**sem_val**) равно нулю, происходит немедленный возврат из вызова
   - Иначе происходит блокирование процесса до тех пор, пока значение семафора не обнулится, после чего происходит возврат из вызова

Таким образом, ненулевое значение поля **sem_op** обозначает необходимость прибавить к текущему значению семафора значение **sem_op**, а нулевое – дождаться обнуления семафора. Итак, порядок работы с семафором можно записать в виде следующей схемы.

```
Если sem_op = 0 то
     если sem_val ≠ 0 то
               пока (sem_val ≠ 0) [процесс блокирован]
[возврат из вызова]
Если sem_op ≠ 0 то
     если sem_val + sem_op < 0 то
               пока (sem_val+sem_op< 0) [процесс блокирован]
     sem_val = sem_val + sem_op
```

Поле **sem_flg** в структуре **sembuf** содержит комбинацию флагов, влияющих на выполнение операции с семафором. В этом поле может быть установлен флаг **IPC_NOWAIT**, который предписывает соответствующей операции над семафором не блокировать процесс, а сразу возвращать управление из вызова **semop()**. Вызов **semop()** в такой ситуации вернет –1. Кроме того, в этом поле может быть установлен флаг **SEM_UNDO**, в этом случае система запомнит изменение значения семафора, произведенные данным вызовом, и по завершении процесса автоматически ликвидирует это изменение. Это предохраняет от ситуации, когда процесс уменьшил значение семафора, начав работать с ресурсом, а потом, не увеличив значение семафора обратно, по какой-либо причине завершился. В этом случае остальные процессы, ждущие доступа к ресурсу, оказались бы заблокированы навечно.

3.2.3.3 Управление состоянием массива семафоров

```
#include <sys/types.h>
#include<sys/ipc.h>
#include<sys/sem.h>

int semctl (int semid, int num, int cmd, union semun arg)
```

С помощью этого системного вызова можно запрашивать и изменять управляющие параметры разделяемого ресурса, а также удалять его.

Первый параметр вызова – дескриптор массива семафоров. Параметр **num** представляет собой индекс семафора в массиве, параметр **cmd** задает операцию, которая должна быть выполнена над данным семафором. Последний аргумент имеет тип **union semun** и используется для считывания или задания управляющих параметров одного семафора или всего массива, в зависимости от значения аргумента **cmd**. Тип данных **union semun** определен в файле **<sys/sem.h>** и выглядит следующим образом:

```
union semun {
     int val;  /* значение одного семафора */
     struct semid_ds *buf; /* параметры массива семафоров в
     целом */
     ushort    *array;   /* массив значений семафоров */
}
```

где **struct semid_ds** – структура, описанная в том же файле, в полях которой хранится информация о всем наборе семафоров в целом, а именно, количество семафоров в наборе, права доступа к нему и статистика доступа к массиву семафоров.

Приведем некоторые наиболее часто используемые значения аргумента **cmd**:

IPC_STAT – скопировать управляющие параметры набора семафоров по адресу **arg.buf**

IPC_SET – заменить управляющие параметры набора семафоров на те, которые указаны в **arg.buf**. Чтобы выполнить эту операцию, процесс должен быть владельцем или создателем массива семафоров, либо обладать правами привилегированного пользователя, при этом процесс может изменить только владельца массива семафоров и права доступа к нему.

IPC_RMID – удалить массив семафоров. Чтобы выполнить эту операцию, процесс должен быть владельцем или создателем массива семафоров, либо обладать правами привилегированного пользователя

GETALL, SETALL – считать / установить значения всех семафоров в массив, на который указывает **arg.array**

GETVAL – возвратить значение семафора с номером **num**. Последний аргумент вызова игнорируется.

SETVAL – установить значение семафора с номером num равным **arg.val**

В случае успешного завершения вызов возвращает значение, соответствующее конкретной выполнявшейся операции (0, если не оговорено иное), в случае неудачи – -1.

3.2.3.4 Пример - работа с разделяемой памятью и с семафорами.

В рассматриваемом примере моделируется двухпроцессная система, в которой первый процесс создает ресурсы разделяемая память и массив семафоров. Затем он начинает читать информацию со стандартного устройства ввода, считанные строки записываются в разделяемую память. Второй процесс читает строки из разделяемой памяти. Таким образом, мы имеем критический участок в момент, когда один процесс еще не дописал строку, а другой ее уже читает. Данная задача требует синхронизации, которая будет осуществляться на основе механизма семафоров. Стоит обратить внимание на то, что

с одним и тем же ключом одновременно создаются ресурсы двух разных типов (в случае использования ресурсов одного типа подобные действия некорректны).

```
1й процесс:
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <string.h>
#define NMAX   256
int main(int argc, char **argv)
{
     key_t key;
     int semid, shmid;
     struct sembuf sops;
     char *shmaddr;
     char str[NMAX];

     /* создаем уникальный ключ */
     key = ftok("/usr/ter/exmpl", 'S');

     /* Должен быть обеспечен порядок запуска процессов*/
     /* сначала первый, а затем второй           */

     /*создаем один семафор с определенными правами доступа*/
     semid = semget(key, 1, 0666 | IPC_CREAT | IPC_EXCL);
     /* создаем разделяемую память на 256 байт */
     shmid = shmget(key, NMAX, 0666 | IPC_CREAT | IPC_EXCL);
     /*  подключаемся к разделяемой памяти, в shaddr –
     указатель на буфер с разделяемой памятью */
     shmaddr = shmat(shmid, NULL, 0);
     /* инициализируем семафор значением 0 */
     semctl(semid,0,SETVAL, (int) 0);
     sops.sem_num = 0;
     sops.sem_flg = 0;
     do { /* запуск цикла */
          printf("Введите строку:");
          if (fgets(str, NMAX, stdin) == NULL)
          {
               /* окончание ввода */
               /* пишем признак завершения – строку "Q" */
               strcpy(str, "Q");
          }
          /*  в  текущий  момент  семафор  открыт  для  этого
          процесса */
          strcpy(shmaddr, str); /* копируем строку в разд.
          память */
          /* предоставляем второму процессу возможность войти
          */
          sops.sem_op = 3; /* увеличение семафора на 3 */
          semop(semid, &sops, 1);
          /* ждем, пока семафор будет открыт для 1го процесса
          -  для следующей итерации цикла */
```

```
            sops.sem_op = 0; /* ожидание обнуления семафора */
            semop(semid, &sops, 1);
        } while (str[0] != 'Q');
        /* в данный момент второй процесс уже дочитал из
        разделяемой памяти и отключился от нее – можно ее
        удалять*/
        shmdt(shmaddr) ; /* отключаемся от разделяемой памяти */
        shmctl(shmid, IPC_RMID, NULL); /* уничтожаем разделяемую
        память */
        semctl(semid, 0, IPC_RMID, (int) 0); /*уничтожаем
        семафор*/
        return 0;
}
```

2й процесс:

```
/* необходимо корректно определить существование ресурса,
если он есть - подключиться */
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <string.h>
#define NMAX   256

int main(int argc, char **argv)
{
        key_t key;
        int semid, shmid;
        struct sembuf sops;
        char *shmaddr;
        char str[NMAX];

        /* создаем тот же самый ключ */
        key = ftok("/usr/ter/exmpl",'S');
        /* аналогично предыдущему процессу - инициализация
        ресурсов */
        semid = semget(key, 1, 0666 );
        shmid = shmget(key, NMAX, 0666 );
        shmaddr = shmat(shmid, NULL, 0);

        sops.sem_num = 0;
        sops.sem_flg = 0;
        /* запускаем цикл */
        do
        {
            printf("Waiting… \n"); /* ожидание на семафоре */
            sops.sem_op = -2;
            /* будем ожидать, пока "значение семафора" +
            "значение sem_op" не станет положительным*/
            semop(semid, &sops, 1);/* теперь значение семафора
            равно 1 */
```

```
            /*критическая секция - работа с разделяемой памятью
            - в этот момент  первый процесс к разделяемой памяти
            доступа не имеет*/

            strcpy(str,  shmaddr);  /*  копируем  строку  из
            разд.памяти */
            if (str[0] == 'Q')
            {
                  /*   завершение   работы   -   освобождаем
                  разделяемую память */
                  shmdt(shmaddr);
            }
            /*после работы с текущей строкой- обнулим семафор*/
            sops.sem_op=-1;
            semop(semid, &sops, 1);
            printf("Read from shared memory: %s\n", str);
      } while (str[0] != 'Q');
      return 0;
}
```

Отметим, что данный пример демонстрирует два разных приема использования семафоров для синхронизации: первый процесс блокируется в ожидании обнуления семафора, т.е. для того, чтобы он мог войти в критическую секцию, значение семафора должно стать нулевым; второй процесс блокируется при попытке уменьшить значение семафора до отрицательной величины, для того, чтобы этот процесс мог войти в критическую секцию, значение семафора должно быть не менее 3. Обратим внимание, что в данном примере, помимо взаимного исключения процессов, достигается строгая последовательность действий двух процессов: они получают доступ к критической секции строго по очереди.

Следует особо отметить, что в начальном запуске данных процессов должен быть строго соблюден определенный порядок. Сначала должен быть запущен первый процесс, он должен создать разлеляемые ресурсы, инициализировать семафор, а затем уже должен быть запущен второй процесс.

Рекомендации.

1. Разобрать возможные проблемы, в случае, если не соблюден порядок запуска и работы процессов
2. Предложить решение, обеспечивающее указанный порядок.

3.3 Сокеты

Средства межпроцессного взаимодействия ОС UNIX, представленные в системе IPC, решают проблему взаимодействия процессов, выполняющихся в рамках одной операционной системы. Однако, очевидно, их невозможно использовать, когда требуется организовать взаимодействие процессов в рамках сети. Это связано как с принятой системой именования, которая обеспечивает уникальность только в рамках данной системы, так и вообще с реализацией механизмов разделяемой памяти, очереди сообщений и семафоров, – очевидно, что для удаленного взаимодействия они не годятся.

Следовательно, возникает необходимость в каком-то дополнительном механизме, позволяющем общаться двум процессам в рамках сети. Однако если разработчики программ будут иметь два абсолютно разных подхода к реализации взаимодействия процессов, в зависимости от того, на одной машине они выполняются или на разных узлах сети, им, очевидно, придется во многих случаях создавать два принципиально разных куска кода, отвечающих за это взаимодействие. Понятно, что это неудобно и хотелось бы в связи с этим иметь некоторый унифицированный механизм, который в определенной степени позволял бы абстрагироваться от расположения процессов и давал бы возможность использования одних и тех же подходов для локального и нелокального взаимодействия. Кроме того, как только мы обращаемся к сетевому взаимодействию, встает проблема многообразия сетевых протоколов и их использования. Понятно, что было бы удобно иметь какой-нибудь общий интерфейс, позволяющий пользоваться услугами различных протоколов по выбору пользователя.

Обозначенные проблемы был призван решить механизм, впервые появившийся в UNIX – BSD (4.2) и названный сокетами (**sockets**).

Сокеты представляют собой в определенном смысле обобщение механизма каналов, но с учетом возможных особенностей, возникающих при работе в сети. Кроме того, они предоставляют больше возможностей по передаче сообщений, например, могут поддерживать передачу экстренных сообщений вне общего потока данных. Общая схема работы с сокетами любого типа такова: каждый из взаимодействующих процессов должен на своей стороне создать и отконфигурировать сокет, после чего процессы должны осуществить соединение с использованием этой пары сокетов. По окончании взаимодействия сокеты уничтожаются.

Механизм сокетов чрезвычайно удобен при разработке взаимодействующих приложений, образующих систему «клиент-сервер». Клиент посылает серверу запросы на предоставление услуги, а сервер отвечает на эти запросы.

Схема использования механизма сокетов для взаимодействия в рамках модели «клиент-сервер» такова. Процесс-сервер запрашивает у ОС сокет и, получив его, присваивает ему некоторое имя (адрес), которое предполагается заранее известным всем клиентам, которые захотят общаться с данным сервером. После этого сервер переходит в режим ожидания и обработки запросов от клиентов. Клиент, со своей стороны, тоже создает сокет и запрашивает соединение своего сокета с сокетом сервера, имеющим известное ему имя (адрес). После того, как соединение будет установлено, клиент и сервер могут обмениваться данными через соединенную пару сокетов. Ниже мы подробно рассмотрим функции, выполняющие все необходимые действия с сокетами, и напишем пример небольшой серверной и клиентской программы, использующих сокеты.

3.3.1 Типы сокетов. Коммуникационный домен

Сокеты подразделяются на несколько типов в зависимости от типа коммуникационного соединения, который они используют. Два основных типа коммуникационных соединений и, соответственно, сокетов представляет собой соединение с использованием виртуального канала и датаграммное соединение.

Соединение с использованием виртуального канала – это последовательный поток байтов, гарантирующий надежную доставку сообщений с сохранением порядка их следования. Данные начинают передаваться только после того, как виртуальный канал установлен, и канал не разрывается, пока все данные не будут переданы. Примером соединения с установлением виртуального канала является механизм каналов в UNIX, аналогом такого соединения из реальной жизни также является телефонный разговор. Заметим, что границы сообщений при таком виде соединений не сохраняются, т.е. приложение, получающее данные, должно само определять, где заканчивается одно сообщение и начинается следующее. Такой тип соединения может также поддерживать

передачу экстренных сообщений вне основного потока данных, если это возможно при использовании конкретного выбранного протокола.

Датаграммное соединение используется для передачи отдельных пакетов, содержащих порции данных – датаграмм. Для датаграмм не гарантируется доставка в том же порядке, в каком они были посланы. Вообще говоря, для них не гарантируется доставка вообще, надежность соединения в этом случае ниже, чем при установлении виртуального канала. Однако датаграммные соединения, как правило, более быстрые. Примером датаграммного соединения из реальной жизни может служить обычная почта: письма и посылки могут приходить адресату не в том порядке, в каком они были посланы, а некоторые из них могут и совсем пропадать.

Поскольку сокеты могут использоваться как для локального, так и для удаленного взаимодействия, встает вопрос о пространстве адресов сокетов. При создании сокета указывается так называемый *коммуникационный домен*, к которому данный сокет будет принадлежать. Коммуникационный домен определяет конкретную модель именования (форматы адресов, правила их интерпретации), а также область взаимодействия процессов. Мы будем рассматривать два основных домена: для локального взаимодействия – домен **AF_UNIX** и для взаимодействия в рамках сети – домен **AF_INET** (префикс AF обозначает сокращение от «address family» – семейство адресов). В домене **AF_UNIX** формат адреса – это допустимое имя файла, в домене **AF_INET** адрес образуют комбинация IP адреса и номера порта (порт – виртуальная точка соединения, которая позволяет адресовать конкретный процесс извне).

Заметим, что фактически коммуникационный домен определяет также используемые семейства протоколов. Так, для домена **AF_UNIX** это будут внутренние протоколы ОС, для домена **AF_INET** – протоколы семейства TCP/IP. Современные системы поддерживают и другие коммуникационные домены.

Ниже приведен набор функций для работы с сокетами.

3.3.2 Создание и конфигурирование сокета

3.3.2.1 Создание сокета

```
#include <sys/types.h>
#include <sys/socket.h>
int socket (int domain, int type, int protocol)
```

Функция создания сокета так и называется – **socket().** У нее имеется три аргумента. Первый аргумент – domain – обозначает коммуникационный домен, к которому должен принадлежать создаваемый сокет. Для двух рассмотренных нами доменов соответствующие константы будут равны, как мы уже говорили, **AF_UNIX** и **AF_INET**. Второй аргумент – **type** – определяет тип соединения, которым будет пользоваться сокет (и, соответственно, тип сокета). Для двух основных рассматриваемых нами типов сокетов это будут константы **SOCK_STREAM** для соединения с установлением виртуального канала и **SOCK_DGRAM** для датаграмм[6]. Третий аргумент – **protocol** – задает конкретный протокол, который будет использоваться в рамках данного коммуникационного домена для создания соединения. Если установить значение данного аргумента в 0, система автоматически выберет подходящий протокол. В наших примерах мы так и будем

[6] Заметим, что данный аргумент может принимать не только указанные два значения, например, тип сокета **SOCK_SEQPACKET** обозначает соединение с установлением виртуального канала со всеми вытекающими отсюда свойствами, но при этом сохраняются границы сообщений; однако данный тип сокетов не поддерживается ни в домене **AF_UNIX**, ни в домене **AF_INET**, поэтому мы его здесь рассматривать не будем

поступать. Однако здесь для справки приведем константы для протоколов, используемых в домене **AF_INET:**

IPPROTO_TCP – обозначает протокол TCP (корректно при создании сокета типа **SOCK_STREAM**)

IPPROTO_UDP – обозначает протокол UDP (корректно при создании сокета типа **SOCK_DGRAM**)

Функция **socket()** возвращает в случае успеха положительное целое число – дескриптор сокета, которое может быть использовано в дальнейших вызовах при работе с данным сокетом. Заметим, что дескриптор сокета фактически представляет собой файловый дескриптор, а именно, он является индексом в таблице файловых дескрипторов процесса, и может использоваться в дальнейшем для операций чтения и записи в сокет, которые осуществляются подобно операциям чтения и записи в файл (подробно эти операции будут рассмотрены ниже).

В случае если создание сокета с указанными параметрами невозможно (например, при некорректном сочетании коммуникационного домена, типа сокета и протокола), функция возвращает –1.

3.3.2.2 Связывание

Для того чтобы к созданному сокету мог обратиться какой-либо процесс извне, необходимо связать с этим сокетом некоторое имя (адрес). Как мы уже говорили, формат адреса зависит от коммуникационного домена, в рамках которого действует сокет, и может представлять собой либо путь к файлу, либо сочетание IP-адреса и номера порта. Но в любом случае связывание сокета с конкретным адресом осуществляется одной и той же функцией `bind`:

```
#include <sys/types.h>
#include <sys/socket.h>
int bind (int sockfd, struct sockaddr *myaddr, int addrlen)
```

Первый аргумент функции – дескриптор сокета, возвращенный функцией **socket()**; второй аргумент – указатель на структуру, содержащую адрес сокета, третий аргумент задает реальный размер структуры, на которую указывает **myaddr.**

Для домена **AF_UNIX** формат структуры `sockaddr` описан в **<sys/un.h>** и выглядит следующим образом:

```
#include <sys/un.h>
struct sockaddr_un {
     short sun_family; /* == AF_UNIX */
     char sun_path[108];
};
```

Для домена **AF_INET** формат структуры описан в **<netinet/in.h>** и выглядит следующим образом:

```
#include <netinet/in.h>
struct sockaddr_in {
     short sin_family;        /* == AF_INET */
     u_short sin_port;        /*port number */
     struct in_addr sin_addr; /* host IP address */
     char sin_zero[8];        /* not used */
};
/* IP address */
struct in_addr {
     unsigned long    s_addr; /* address */
};
```

Отметим, что так как значения **sin_port** и **`s_addr`** в данных структурах являются многобайтными полями и при взаимодействии они передаются между компьютерами разных архитектур, то для правильной работы протоколов они должны быть определены в едином формате, независящем от порядка следования байт в машиином слове конкретного компьютера. Поэтому для этих полей стандартом задаётся использование так называемого *сетевого порядке следования байт* (альтернативное определение - big-endian формат), т.е. от старшего байта к младшему

Для примера, 4х байтовое значение (0xD4C3B2A1 в 16-ричном представлении) - его размещение в памяти по адресу N на компьютерах разных архитектур будет следующим:

Big-endian (сетевой порядок или от старшего к младшему)

адрес байта памяти:	N	N+1	N+2	N+3
значение:	D4	C3	B2	A1

Архитектура Little-endian (от младщего к старшему)

байт памяти:	N	N+1	N+2	N+3
значение:	A1	B2	C3	D4

Другой пример, если в 2-х поряд расположенных 16-ти разрядных словах хранится строка из 4-х символов: UNIX, то рассматривая его содержимое, как последовательность байтов, мы в первом случае получим UNIX, а во втором получим NUXI. Данный пример получил название **проблема NUXI**.

Для организации корректной передачи подобных многобайтовых значений используются библиотечные функции преобразования, которые, при необходимости, конвертируют свои аргументы к нужному формату (порядку следования байт). Традиционно применяются следующие функции:

```
#include <arpa/inet.h>
/*или <netinet/in.h> (в некоторых реализациях требуется использовать оба файла) */
```

uint32_t htonl(uint32_t hostint32)
- на входе 32-разрядное целое текущей архитектуры, возвращает 32-разрядное целое, приведённое к сетевому порядку байт (**h**ost-**to**-**n**etwork);

uint32_t ntohl(uint32_t netint32)
- на входе 32-разрядное целое в сетевом порядке байт, возвращает 32-разрядное целое, приведённое к формату текущей архитектуры (**n**etwork-**to**-**h**ost).

Функции **htons** и **ntohs** реализуют аналогичные преобразования для 16-разрядных целых.

Важно отметить, что если мы имеем дело с доменом **`AF_UNIX`** и адрес сокета представляет собой имя файла, то при выполнении функции **`bind()`** система в качестве побочного эффекта создает файл с таким именем. Поэтому для успешного выполнения **`bind()`** необходимо, чтобы такого файла не существовало к данному моменту. Это следует учитывать, если мы «зашиваем» в программу определенное имя и намерены запускать нашу программу несколько раз на одной и той же машине – в этом случае для успешной работы **`bind()`** необходимо удалять файл с этим именем перед связыванием. Кроме того, в процессе создания файла, естественно, проверяются права доступа пользователя, от имени которого производится вызов, ко всем директориям, фигурирующим в полном путевом

имени файла, что тоже необходимо учитывать при задании имени. Если права доступа к одной из директорий недостаточны, вызов **bind()** завершится неуспешно.

В случае успешного связывания **bind()** возвращает 0, в случае ошибки – -1.

3.3.3 Предварительное установление соединения.

3.3.3.1 Сокеты с установлением соединения. Запрос на соединение.

Различают **сокеты с предварительным установлением соединения**, когда до начала передачи данных устанавливаются адреса сокетов отправителя и получателя данных – такие сокеты соединяются друг с другом и остаются соединенными до окончания обмена данными; и **сокеты без установления соединения**, когда соединение до начала передачи данных не устанавливается, а адреса сокетов отправителя и получателя передаются с каждым сообщением. Если тип сокета – виртуальный канал, то сокет ***должен*** устанавливать соединение, если же тип сокета – датаграмма, то, как правило, это сокет без установления соединения, хотя последнее не является требованием. Для установления соединения служит следующая функция:

```
#include <sys/types.h>
#include <sys/socket.h>
int connect (int sockfd, struct sockaddr *serv_addr,
     int addrlen);
```

Первый аргумент – дескриптор сокета, созланный в данном процессе (в том, который обращается к connect()), второй аргумент – указатель на структуру, содержащую адрес сокета, с которым производится соединение, в формате, который мы обсуждали выше, и третий аргумент содержит реальную длину этой структуры. Функция возвращает 0 в случае успешного установления соединения с сервером и –1 в случае неудачи (например тогда, когда сервер еще «не готов» к установлению соединений с клиентами - не выполнил обращение к системному вызову listen), при этом код ошибки можно посмотреть в переменной **errno.** Обратившийся к connect клиентский процесс может быть заблокирован до тех пор пока сервер не обработает запрос данного процесса на соединение (не выполнит системный вызов accept, которые обработает запрос на соединение с данным клиентом).

Заметим, что в рамках модели «клиент-сервер» клиенту, вообще говоря, не важно, какой адрес будет назначен его сокету, так как никакой процесс не будет пытаться непосредственно установить соединение с сокетом клиента.

В случае взаимодействия в рамках домена **AF_INET** клиент может не вызывать предварительно функцию **bind()**, в этом случае при вызове **connect()** система автоматически выберет приемлемые значения для локального адреса клиента. При работе в домене **AF_UNIX** клиентское приложение само должно позаботиться о связывании сокета.

3.3.3.2 Сервер: прослушивание сокета и подтверждение соединения.

Следующие два вызова используются сервером только в том случае, если используются сокеты с предварительным установлением соединения.

```
#include <sys/types.h>
#include <sys/socket.h>
int listen (int sockfd, int backlog);
```

Этот вызов используется процессом-сервером для того, чтобы сообщить системе о том, что он готов к обработке запросов на соединение, поступающих на данный сокет. До тех пор, пока процесс – владелец сокета не вызовет **listen(),** все запросы на соединение с данным сокетом (вызовы connect()) будут возвращать ошибку. Первый аргумент функции

– дескриптор сокета. Второй аргумент, **backlog,** задаёт ограничение на размер очереди запросов на соединение. Семантика параметра **backlog зависит от конкретной операционной системы - в ранних Unix системах данный параметр регламентировал количество открытых соединений с клиентами в т.ч. посредством вызовов connect(), в современных системах параметр регламентирует внутреннее сетевое взаимодействие (мы не рассматриваем детализацию в данном материале).**

Конкретное соединение устанавливается при помощи вызова **accept()**:

```
#include <sys/types.h>
#include <sys/socket.h>
int accept (int sockfd, struct sockaddr *addr,
     int *addrlen);
```

Этот вызов применяется сервером для удовлетворения поступившего клиентского запроса на соединение с сокетом, который сервер к тому моменту уже прослушивает (т.е. предварительно была вызвана функция **listen()**). Вызов **accept()** извлекает первый запрос из очереди запросов, ожидающих соединения, и устанавливает с ним соединение. Если к моменту вызова **accept()** очередь запросов на соединение пуста, процесс, вызвавший **accept()**, блокируется до поступления запросов.

Когда запрос поступает и соединение устанавливается, **accept()** создает новый сокет, который будет использоваться для работы с данным соединением, и возвращает дескриптор этого нового сокета, соединенного с сокетом клиентского процесса. При этом первоначальный сокет продолжает оставаться в состоянии прослушивания. Через новый сокет осуществляется обмен данными, в то время как старый сокет продолжает обрабатывать другие поступающие запросы на соединение (напомним, что именно первоначально созданный сокет связан с адресом, известным клиентам, поэтому все клиенты могут слать запросы только на соединение с этим сокетом). Это позволяет процессу-серверу поддерживать несколько соединений одновременно. Обычно это реализуется путем порождения для каждого установленного соединения отдельного процесса-потомка, который занимается собственно обменом данными только с этим конкретным клиентом, в то время как процесс-родитель продолжает прослушивать первоначальный сокет и порождать новые соединения.

Во втором параметре передается указатель на структуру, в которой возвращается адрес клиентского сокета, с которым установлено соединение, а в третьем параметре возвращается реальная длина этой структуры. Благодаря этому сервер всегда знает, куда ему в случае надобности следует послать ответное сообщение. Если адрес клиента нас не интересует, в качестве второго аргумента можно передать **NULL**.

3.3.4 Прием и передача данных

Собственно для приема и передачи данных через сокет используются три пары функций.

```
#include <sys/types.h>
#include <sys/socket.h>
int  send(int  sockfd,  const  void *msg, int len,
     unsigned  int flags);
int recv(int sockfd, void *buf, int len,
     unsigned int flags);
```

Эти функции используются для обмена только через сокет с предварительно установленным соединением. Аргументы функции **send(): sockfd** – дескриптор

сокета, через который передаются данные, **msg** и **len** - сообщение и его длина. Если сообщение слишком длинное для того протокола, который используется при соединении, оно не передается и вызов возвращает ошибку **EMSGSIZE.** Если же сокет окажется переполнен, т.е. в его буфере не хватит места, чтобы поместить туда сообщение, выполнение процесса блокируется до появления возможности поместить сообщение. Функция **send()** возвращает количество переданных байт в случае успеха и -1 в случае неудачи. Код ошибки при этом устанавливается в **errno.** Аргументы функции **recv()** аналогичны: **sockfd** – дескриптор сокета, **buf** и **len** – указатель на буфер для приема данных и его первоначальная длина. В случае успеха функция возвращает количество считанных байт, в случае неудачи -1[7].

Последний аргумент обеих функций – **flags** – может содержать комбинацию специальных опций. Нас будут интересовать две из них:

MSG_OOB – этот флаг сообщает ОС, что процесс хочет осуществить прием/передачу экстренных сообщений

MSG_PEEK – данный флаг может устанавливаться при вызове **recv()**. При этом процесс получает возможность прочитать порцию данных, не удаляя ее из сокета, таким образом, что последующий вызов **recv()** вновь вернет те же самые данные.

Другая пара функций, которые могут использоваться при работе с сокетами с предварительно установленным соединением – это обычные **read()** и **write()**, в качестве дескриптора которым передается дескриптор сокета.

И, наконец, пара функций, которая может быть использована как с сокетами с установлением соединения, так и с сокетами без установления соединения:

```
#include <sys/types.h>
#include <sys/socket.h>
int sendto(int sockfd, const void *msg, int len,
     unsigned  int flags, const struct sockaddr *to, int tolen);
int recvfrom(int sockfd, void *buf, int len,
     unsigned int flags, struct sockaddr *from, int *fromlen);
```

Первые 4 аргумента у них такие же, как и у рассмотренных выше. В последних двух в функцию **sendto()** должны быть переданы указатель на структуру, содержащую адрес получателя, и ее размер, а функция **recvfrom()** в них возвращает соответственно указатель на структуру с адресом отправителя и ее реальный размер. Отметим, что перед вызовом **recvfrom()** параметр **fromlen** должен быть установлен равным первоначальному размеру структуры **from.** Здесь, как и в функции **accept**, если нас не интересует адрес отправителя, в качестве **from** можно передать **NULL.**

3.3.5 Завершение работы с сокетом

Если процесс закончил весь прием либо передачу данных, ему следует закрыть соединение. Это можно сделать с помощью функции **shutdown():**

```
# include <sys/types.h>
# include <sys/socket.h>
int shutdown (int sockfd, int mode);
```

Помимо дескриптора сокета, ей передается целое число, которое определяет режим закрытия соединения. Если **mode=0,** то сокет закрывается для чтения, при этом все дальнейшие попытки чтения будут возвращать **EOF**. Если **mode=1**, то сокет закрывается

[7] Отметим, что, как уже говорилось, при использовании сокетов с установлением виртуального соединения границы сообщений не сохраняются, поэтому приложение, принимающее сообщения, может принимать данные совсем не теми же порциями, какими они были посланы. Вся работа по интерпретации сообщений возлагается на приложение.

для записи, и при осуществлении в дальнейшем попытки передать данные будет выдан код ошибки (-1). Если **mode=2**, то сокет закрывается и для чтения, и для записи.

Аналогично файловому дескриптору, дескриптор сокета освобождается системным вызовом **close()**. При этом, если до этого не был вызван **shutdown()**, соединение будет закрыто. Таким образом, в принципе, если по окончании работы с сокетом мы собираемся закрыть соединение и по чтению, и по записи, можно было бы сразу вызвать **close()** для дескриптора данного сокета, опустив вызов **shutdown()**. Однако, если у дескриптора остаются дубликаты, **close()** закроет соединение только после закрытия последнего из них, в то время как **shutdown()** закрывает соединение немедленно.

3.3.6 Резюме: общая схема работы с сокетами

Механизм сокетов включает в свой состав достаточно разнообразные средства, позволяющие организовывать взаимодействие различной топологии. В частности, имеется возможность организации взаимодействия с ***предварительным установлением соединения***. Данная модель ориентирована на организацию клиент-серверных систем, когда организуется один серверный узел процессов (обратим внимание, что не процесс, а именно узел процессов), который принимает сообщения от клиентский процессов и их как-то обрабатывает. Общая схема подобного взаимодействия представлена ниже (Рис. 101). Заметим, что тип сокета в данном случае не важен, т.е. можно использовать как сокеты, являющиеся виртуальными каналами, так и дейтаграммные сокеты.

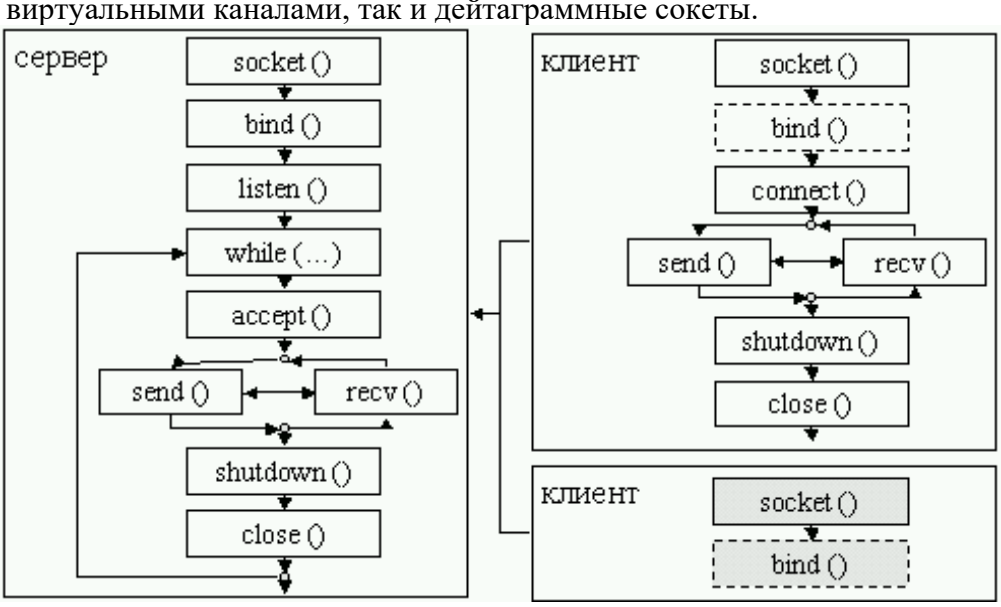


Рис. 101. Схема работы с сокетами с предварительным установлением соединения.

В данной модели можно выделить две группы процессов: процессы серверной части и процессы клиентской части. На стороне сервера открывается основной сокет. Поскольку необходимо обеспечить возможность другим процессам обращаться к серверному сокету по имени, то в данном случае необходимо связывание сокета с именем (вызов *bind()*). Затем серверный процесс переводится в *режим прослушивания* (посредством системного вызова *listen()*): это означает, что данный процесс может принимать запросы на соединение с ним от клиентских процессов.

Каждый клиентский процесс создает свой сокет. Заметим, что на стороне клиента связывать сокет необязательно (поскольку сервер может работать с любым клиентом, в частности, и с «анонимным» клиентом). Затем клиентский процесс может передать серверному процессу сообщение, что он с ним хочет соединиться, т.е. передать запрос на

соединение (посредством системного вызова *connect()*). В данном случае возможны альтернативы.

Во-первых, может оказаться, что клиент обращается к *connect()* до того, как сервер перешел в режим прослушивания. В этом случае клиент получает отказ с уведомлением, что сервер в данный момент не прослушивает сокет.

Во-вторых, клиент может обратиться к *connect()*, а у серверного процесса в данный момент сформирована очередь необработанных запросов, и эта очередь пока не насыщена. В этом случае запрос на соединение встанет в очередь, и клиентский процесс заблокируется и будет ожидать обслуживания.

Итак, данная схема организована таким образом, что к одному серверному узлу может быть множество соединений. Для организации этой модели имеется системный вызов *accept()*, который работает следующим образом. При обращении к данному системному вызову, если в очереди имеется необработанная заявка на соединение, создается новый локальный сокет, который связывается с клиентом. После этого клиент может посылать сообщения серверу через этот новый сокет. А сервер, в свою очередь, получая через данный сокет сообщения от клиента, «понимает», от какого клиента пришло это сообщение (т.е. клиент получает некоторое внутрисистемное именование, даже если он не делал связывание своего сокета). И, соответственно, в этом случае сервер может посылать ответные сообщения клиенту.

Завершение работы состоит из двух шагов. Первый шаг заключается в отключении доступа к сокету посредством системного вызова *shutdown()*. Можно закрыть сокет на чтение, на запись, на чтение-запись. Тем самым системе передается информация, что сокет более не нужен. С помощью этого вызова обеспечивается корректное прекращение работы с сокетом (в частности, это важно при организации виртуального канала, который должен гарантировать доставку посланных данных). Второй шаг заключается в закрытии сокета с помощью системного вызова *close()* (т.е. закрытие сокета как файла).

Далее эту концептуальную модель можно развивать. Например, сервер может порождать сыновний процесс для каждого вызова *accept()*, и тогда все действия по работе с данным клиентом ложатся на этот сыновний процесс. На родительский процесс возлагается задача прослушивание «главного» сокета и обработка поступающих запросов на соединение. Вот почему речь идет не об одном процессе сервере, а о серверном узле, в котором может находиться целый набор процессов.

Теперь рассмотрим модель сокетов ***без предварительного соединения*** (Рис. 102). В этой модели используются лишь дейтаграммные сокеты. В отличие от предыдущей модели, которая была иерархически организованной, то эта модель обладает произвольной организацией взаимодействия. Это означает, что в данной модели у каждого взаимодействующего процесса имеется сокет, через который он может получать информацию от различных источников, в отличие от предыдущей модели, где имеется «главный» известный всем клиентам сокет сервера, через который неявно передается управляющая информация (заказы на соединение), а затем с каждым клиентом связывается один локальный сокет. Этот механизм позволяет серверу взаимодействовать с клиентом, не зная его имя явно. В текущей модели ситуация симметричная: любой процесс через свой сокет может послать информацию любому другому сокету. Это означает, что механизм отправки имеет соответствующую адресную информацию (т.е. информацию об отправителе и получателе).

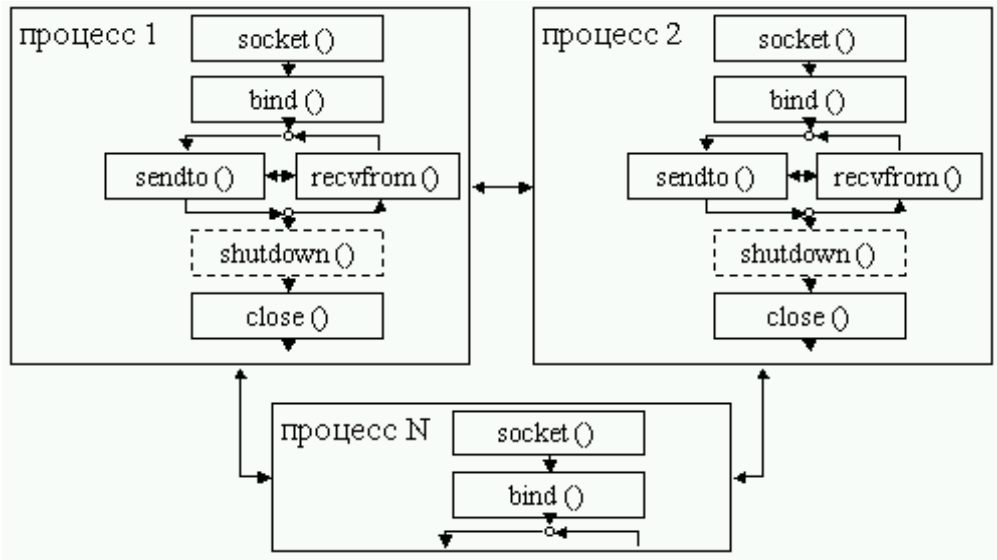


Рис. 102. Схема работы с сокетами без предварительного установления соединения.

Поскольку в данной модели используются дейтаграммные сокеты, то необходимость обращаться к вызову *shutdown()* отпадает: в этой модели сообщения проходят (и уходят) одной порцией данных, поэтому можно сразу закрывать сокет посредством вызова *close()*.

3.3.7 Примеры использования сокетов

3.3.7.1 Сетевое взаимодействие через сокеты, с установлением соединения.

Цель: разработка распределенной клиент/серверной программы, использующей сокеты для организации взаимодействия в домене **`AF_INET`.**

Прикладная задача. Разработать программную реализацию клиент/сервера, взаимодействующего с клиентами в пространтве домена **AF_INET.**

Клиент – создает свой сокет и через него запрашивает соедиенение к серверу по заданному адресу. После успешного подтверждения соединения формирует текстовую строку, содержащую PID своего процесса, которую отправляет на сервер. Дожидается получения информации в виде текстовой строки от Сервера, выводит ее на стандартное устройство вывода. Далее закрытие сокета.

Сервер – создает свой основной сокет и открывает его на прослушивание (прием запросов клиентов на установление соединения). Далее, Сервер, циклически подтвержает запросы на соединение с каждым Клиентом. Для каждого подтвержденного соединения Сервер формирует отдельный процесс в котором будет осуществляться подтвержденное взаимодействие с Клиентом.

Особенности программной реализации.

Для обеспечения возможности самостоятельного запуска демонстрационного примера сетевого взаимодействия в рамках одной локальной ВС, в качестве IP адреса может использоваться зарезервированная IP-константа 127.0.0.1 – означающая, что взаимодействие будет осущетствляться в пределах одного (данного) компьютера. В случае Сервера означает, что к нему могут обращаться клиенты, находящиеся в рамках той же ВС, что Сервер. Это позволяет избежать использования фактического IP адреса Сервера.

INADDR_ANY – в качестве IP адреса Сервера используется предопределенная константа, использование которой обеспечивает «привязку» ко всем IP адресам данного сервера, включая 127.0.0.1 .

inet_addr() – функция преобразования IP адреса из формата текстовой строки в 32-х разрядный формат.

Код программной реализации Сервера

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#define PORTNUM 50500
#define BUFLEN 80
#define QUESIZE 5  /* условный размер очереди запросов на соединение */


int main(int argc, char **argv)
{
        struct sockaddr_in      own_addr, /* основной сокет*/
                                party_addr;/* «клиентский» сокет – создается после выполнения accept() */

        int sockfd, newsockfd;
        int party_len, len;
        char buf[BUFLEN];

        /* создаем основной сокет */
        sockfd = socket(AF_INET, SOCK_STREAM, 0);

        /* заполнение данных для именования основного сокета */

        memset(&own_addr, 0, sizeof(own_addr));
        /*Заполнение массива кодом заданного символа
                #include < string.h >
                void *memset (void *destination, int c, size_t n);
                destination – указатель на заполняемый массив
                c – код символа для заполнения
                n – размер заполняемой части массива в байтах
                Функция возвращает указатель на заполняемый массив*/

        own_addr.sin_family = AF_INET;
        own_addr.sin_addr.s_addr = INADDR_ANY; /*константа INADDR_ANY обеспечивает возможность привязки сокета ко всем IP адресам компьютера, включая выделенный IP 127.0.0.1*/

        own_addr.sin_port = htons(PORTNUM); /* функция ntohs возвращает свой 16-разрядный аргумент, приведённый к сетевому порядку байт */
```

```
/*связываем основной сокет, используем приведение типа между указателями на фактическую и формальную структуры данных */

bind(sockfd, (struct sockaddr *) &own_addr, sizeof(own_addr));


/* Разрешаем обработку запросов на соединение, QUESIZE - условный размер очереди запросов*/
listen(sockfd, QUESIZE);

while (1) {
	/*подготавливаем очищенную структуру данных*/
	memset(&party_addr, 0, sizeof(party_addr));
	party_len = sizeof(party_addr);

	/*создание соединения – при подключении нового клиента система заполняет party_addr информацией о клиенте (IP адрес, порт)*/
	newsockfd = accept(sockfd, (struct sockaddr *)&party_addr, &party_len);

	/*обрабатываем соединение нового клиента в отдельном процессе*/
	if (!fork()) {
		/*сын - обрабатывает запрос и посылает ответ*/
		close(sockfd);   /* этот сокет сыну не нужен*/
		len = recv(newsockfd, &buf, BUFLEN, 0);
		/*печатаем информацию о клиенте и переданную им строку*/
		printf("received string %s from client %s:%d \n", buf, inet_ntoa(party_addr.sin_addr), ntohs(party_addr.sin_port));

		/*посылаем клиенту ответ в виде длины полученной строки */
		sprintf(buf, "%d", strlen(buf));
		send(newsockfd, buf, strlen(buf) + 1, 0);

		/*закрываем сокет */
		close(newsockfd);

		return 0;
	}
	/*отец-закрывает новый сокет, продолжает прослушивать старый*/
	close(newsockfd);
}
}
```

Код программной реализации Клиента

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
```

```
#include <string.h>
#include <unistd.h>
#define PORTNUM 50500
#define BUFLEN 80

int main(int argc, char **argv)
{
	struct sockaddr_in addr;
	int sockfd;
	char buf[BUFLEN];

	/*создание сокета*/
	sockfd = socket(AF_INET, SOCK_STREAM, 0);
	/*заполнение структуры данных для подключения к серверу*/
	memset(&addr, 0, sizeof(addr));
	addr.sin_family = AF_INET;

	/*Для обеспечения возможности самостоятельного запуска демонстрационного примера сетевого взаимодействия в рамках одной локальной ВС, в качестве IP адреса Сервера будет использоваться выделенный IP 127.0.0.1*/
	addr.sin_addr.s_addr = inet_addr("127.0.0.1"); /* преобразование строки в формат IP */

	addr.sin_port = htons(PORTNUM);

	/*подключение к серверу*/
	connect(sockfd, (struct sockaddr *) &addr, sizeof(addr));

	/*отсылка данных серверу в виде строки-числа(собственный pid)*/
	sprintf(buf, "%d", getpid());
	send(sockfd, buf, strlen(buf) + 1, 0);
	/*получение ответа от сервера в виде строки*/
	recv(sockfd, &buf, BUFLEN, 0);
	printf("received: %s \n", buf);

	close(sockfd);
}
```

Рекомендации.

1) Запустить программу, реализующую пример на компьютере.
2) Провести и запустить на компьютере модификации программы, реализующей пример, в части:

- добавления контроля и обработки всех возможных ошибок как в коде клиента, так и в коде сервера;
- реализовать альтернативные средства закрытия сокета.

3) При наличии технической возможности (наличие локальной сети и 2х компьютеров) реализовать схему клиент-сервера в рамках реальной сети: при этом потребуется использование фактических IP адресов компьютеров.

3.3.7.2 Взаимодействие без предварительного установления соединения.

Пример для демонстрации обмена данными в схеме без предварительного установления соединения на примере двух (равноправных) процессов. Реализация приведена одновременно для домена **AF_INET** (основной код программы), и домена **AF_UNIX** (альтернативный код в строках-комментариях //).
Первый процес создаёт свой сокет с фиксированным именем, известным второму процессу («целевой» сокет). Дожидается поступления данных в свой сокет, печатает их и информацию о сокете-отправителе. Посылает сокету-отправителю свой pid и завершается. Второй процесс создаёт свой сокет, посылает в целевой сокет свой pid. Дожидается ответа в свой сокет, печатает его и завершается.

Код программной реализации Процесса1

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
//#include <sys/un.h>
#include <netinet/in.h>
#include <string.h>
//#define SADDRESS "socket"
#define N_PORT 50500
#define BUFLEN 32

int main(int argc, char **argv) {
	int sockfd, len;
	char buf[BUFLEN];
	//struct sockaddr_un addr;
	struct sockaddr_in addr;

	/*инициализация структур данных*/
	memset(&addr, 0, sizeof(addr));
	//addr.sun_family = AF_UNIX;
	addr.sin_family = AF_INET;
	/*именование своего сокета*/
	//strcpy(addr.sun_path, SADDRESS);
	addr.sin_addr.s_addr = inet_addr("127.0.0.1");
	addr.sin_port = htons(N_PORT+1);

	/*создание сокета*/
	// sockfd = socket(AF_UNIX, SOCK_DGRAM, 0);
	sockfd = socket(AF_INET, SOCK_DGRAM, 0);

	/* связывание сокета */
	//bind(sockfd, (struct sockaddr *) &addr, sizeof(addr.sun_family)
+ strlen(addr.sun_path));
	bind(sockfd, (struct sockaddr *) &addr, sizeof(addr));

	/*получение данных*/
	len = sizeof(addr);
	recvfrom(sockfd, buf, BUFLEN, 0, (struct sockaddr *) &addr,
&len);
	//printf("process %s from socket %s \n", buf, addr.sun_path);
```

```
	printf("from process %s \n", buf);
	/*отсылка данных*/
	sprintf(buf, "%d", getpid());
	sendto(sockfd, buf, strlen(buf) + 1, 0, (struct sockaddr *)
&addr, len);

	/*освобождение сокета*/
	close(sockfd);
	return 0;
}
```

Код программной реализации Процесса2

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
//#include <sys/un.h>
#include <netinet/in.h>
#include <stdio.h>
#include <string.h>
//#define SADDRESS "socket"
#define N_PORT 50500
#define BUFLEN 32

int main(int argc, char **argv) {
	int sockfd;
	char buf[BUFLEN];
	//struct sockaddr_un addr;
	struct sockaddr_in addr;

	/*инициализация структур данных*/
	memset(&addr, 0, sizeof(addr));
	//addr.sun_family = AF_UNIX;
	addr.sin_family = AF_INET;
	/*именование своего сокета*/
	//strcpy(addr.sun_path, "my_socket");
	addr.sin_addr.s_addr = inet_addr("127.0.0.1");
	addr.sin_port = htons(N_PORT);

	/*создание сокета*/
	//sockfd = socket(AF_UNIX, SOCK_DGRAM, 0);
	sockfd = socket(AF_INET, SOCK_DGRAM, 0);

	/*связывание сокета*/
	//bind(sockfd, (struct sockaddr *) &addr, sizeof(addr.sun_family)
+ strlen(addr.sun_path));
	bind(sockfd, (struct sockaddr *) &addr, sizeof(addr));

	/*заполнение данных для именования целевого сокета*/
```

```
	//strcpy(addr.sun_path, SADDRESS);
	addr.sin_port = htons(N_PORT + 1);

	/*отсылка данных в виде строки-собственного pid*/
	sprintf(buf, "%d", getpid());
	//sendto(sockfd, buf, strlen(buf) + 1, 0, (struct sockaddr *)
&addr, sizeof(addr.sun_family) + strlen(SADDRESS));
	sendto(sockfd, buf, strlen(buf) + 1, 0, (struct sockaddr *)
&addr, sizeof(addr));

	/*получение ответа без сохранения информации об отправителе*/
	recvfrom(sockfd, buf, BUFLEN, 0, NULL, 0);
	printf("answer: %s\n", buf);

	/*освобождение сокета*/
	close(sockfd);
	return 0;
}
```

Рекомендации.

1) Запустить оба варианта программы (и в домене **AF_UNIX**, и в **AF_INET**), реализующую пример, на компьютере.
2) Провести и запустить на компьютере модификации программы, реализующей пример, в части:

- добавления контроля и обработки всех возможных ошибок в коде обоих процессов;
- обеспечения описанной логики взаимодействия процессов вне зависимости от порядка их запуска.

3.4 Взаимодействие нитей в ОС Unix

Кратко рассмотрим организацию взаимодействия нитей в ОС Unix. Современные библиотеки управления нитями предусматривают предоставление широкого спектра механизмов, обеспецивающих взаимодействия нитей, в частности:

- мьютексы (организация взаимного исключения)
- циклические блокировки (ожидание в цикле, без приостановки нитей)
- барьеры (координация действий выполняющихся нитей)
- условные перменные (уведомление нитей о наступления событий без риска гонки процессов)

В настоящем курсе мы кратко рассмотрим основные средства организации и использования мьютексов – инструмента, предназначенного для решения задач синхронизации доступа к разделяемым ресурсам на основе организации взаимного исключения.

3.4.1 Создание, инициализация, концепции мьютекса

По аналогии с семафорами, мьютекс это переменная специального типа **pthread_mutex_t**, который определяется в файле **pthread.h**. Функциональность мьютекса идентична двоичным семафором. При использовании мьютекс может находиться в двух состояниях – ***открытом*** или ***закрытом***.
Для использования мьютекса необходимо объявление переменной соответствующего типа (**pthread_mutex_t**) и ее инициализация.

Дли инициализации можно воспользоваться функцией:

int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr)

- mutex - указатель на переменную типа мьютекс.
- attr - атрибуты мьютекса мы не рассматриваем, используем NULL (атрибуты по умолчанию).

Данная функция может инициализировать как мьютексы (переменные типа мьютекс) определенные как в статической или автоматической памяти, так и в динамической памяти. При успешном вызове возвращается 0, иначе код ошибки #0.

В случае, если мьютекс объявлен как автоматическая или статическая переменная, то его можно инициализировать сразу при объявлении:

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER ,

где PTHREAD_MUTEX_INITIALIZER макрос, определенный в файле **pthread.h**.

В случае, если мьютекс определен в динамической памяти, то после завершения работы с ним перед освобождением динамической памяти, следует обратиться к функции удаления мьютекса:

int pthread_mutex_destroy(pthread_mutex_t *mutex)

В случае создания мьютекса в статической или автоматической памяти обращение к функции destroy по завершении работы с мьютексом не нужно.

3.4.2 Операции над мьютексами

После создания и инициализации мьютекс находится в состоянии ***открыт.*** «Захватить» или блокировать мьютекс (перевести его в состояние ***закрыт)*** можно посредством обращения к функции:

int pthread_mutex_lock(pthread_mutex_t *mutex)

- происходит попытка захватить мьютекс. В случае если при обращении к функции мьютекс был закрыт другой нитью (уже находится в состоянии закрыт), нить обратившаяся к функции pthread_mutex_lock будет заблокирована до те пор, пока мьютекс не будет разблокирован. Функция pthread_mutex_lock возвращает 0 в случае успеха , код ошибки #0 в случае неуспешного обращения.

Неблокирующей альтернативой функции pthread_mutex_lock является функция:

int pthread_mutex_trylock(pthread_mutex_t *mutex)

- при обращении нити к данной фикции, если мьютекс был в состоянии ***открыт***, то нить «захватывает» данный мьютекс (он переходит в состояние закрыт и функция возвращает 0). Если же при обращении к функции мьютекс был в состоянии ***закрыт*** (другой нитью), то блокирования нити не произойдет, а она получит код ответа EBUSY. Обращение к функции в случае, если мьютекс уже закрыт данной нитью, вызовет ошибку (возможно зависание).

Для перевода закрытой нити в состояние ***открыт*** используется функция:

int pthread_mutex_unlock(pthread_mutex_t *mutex)

функция корректно прорабатывает, если к ней обращается нить, которая перевела данный мьютекс в состояние ***закрыт*** в этом случае код ответа 0**.** В противном случае (обращение к открытому мьютексу или закрытому другим процессом) обращение считается ошибочным – код ответа #0.

3.4.3 Простой пример синхронизации нитей

В данном примере порождается 10 одновременно работающих нитей, каждая из которых модифицирует общий разделяемый ресурс – строку: каждая нить определяет текущую длину строки и добавляет символ (с последовательно возрастающим значением, начиная с ‘A’) в конец строки. Т.к. при одновременной работе нитей над одним ресурсом возможно «вклинивание» одной нити в работу другой (что может привести к затиранию символа, добавленного одной нитью, другой), то операцию модификации разделяемого ресурса необходимо производить в режиме взаимного исключения (критической секции) – для этого используется статический мьютекс.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define N_SIZE 10
/*разделяемый ресурс*/
char str[N_SIZE + 1];
/*объявление и инициализация мьютекса, состояние "открыт"*/
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

/*прототип нити */
void* fThread(void * arg)
{
	pthread_mutex_lock(&mutex); /*вход в критическую секцию*/
	/*добавляем символ к разделяемой строке*/
	int iPos = strlen(str);
	str[iPos] = 'A' + iPos;
	pthread_mutex_unlock(&mutex); /*выход из критической секции*/
```

```
	return NULL;
}

int main()
{
	pthread_t th_id;
	int i;
	/*инициализация 0 всего массива символов*/
	memset(str, 0, N_SIZE+1);
	/*порождение N_SIZE параллельно работающих нитей, каждая из
	которых добавляет символ к разделяемой строке*/
	for (i = 0; i<N_SIZE; i++) {
		pthread_create(&th_id, NULL, fThread, NULL);
	}
	sleep(20); /*пауза для ожидания завершения всех нитей*/
	printf("%s\n", str); /*печать итоговой строки*/
	return 0;
}
```

Рекомендованные задания:

1) Запустить программу, реализующую пример на компьютере.
2) Провести и запустить на компьютере модификации программы, реализующей пример, в части:

- добавления контроля и обработки всех возможных ошибок;
- использования функции **pthread_mutex_trylock;**

3.4.4 Читатели и писатели с использованием нитей

Рассмотрим реализацию классической задачи читатели-писатели с приоритетом читателей на основе механизма мьютексов. Ниже приведена реализация тел читателей и писателей, соответсвующая приведённым в п. 2.5.3 алгоритмам.

Для наглядности добавлена функция main, в которой запрашивается число порождаемых читателей и писателей, которые затем запускаются чередованием. При порождении/завершении любой нити печатается её TID, тип (читатель/писатель) и прошедшее время секундах относительно начала работы с нитями в функции main.

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#include <pthread.h>

time_t tmstart;
/*разделяемая переменная - число одновременно работающих читателей*/
int rc = 0;
/*определение и инициализация мьютексов, состояние "открыт"*/
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t db = PTHREAD_MUTEX_INITIALIZER;

/*эмуляция работы нити – пауза в течение nsec секунд*/
```

```
void do_some_work(int nsec) { sleep(nsec); }

/*реализация нити читателя – Версия 1*/
void* reader(void * arg)
{
	pthread_mutex_lock(&mutex);
	rc++;
	if (rc == 1) pthread_mutex_lock(&db);
	pthread_mutex_unlock(&mutex);
	do_some_work(5);
	pthread_mutex_lock(&mutex);
	rc--;
	if (rc == 0) pthread_mutex_unlock(&db);
	pthread_mutex_unlock(&mutex);
	printf("---Читатель %#x завершился на %d секунде\n",
		pthread_self(), time(0) - tmstart);
	return NULL;
}

/*реализация нити писателя*/
void* writer(void * arg)
{
	pthread_mutex_lock(&db);
	do_some_work(12);
	pthread_mutex_unlock(&db);
	printf("---Писатель %#x завершился на %d секунде\n",
		pthread_self(), time(0) - tmstart);
	return NULL;
}

int main()
{
	int nr, nw, bFlag;
	pthread_t th_id;
	printf(">>Введите число читателей и писателей:\n");
	scanf("%u %u", &nr, &nw);
	tmstart = time(0); /*нулевая точка отсчёта в секундах*/
	bFlag = !(nr>0); /*значение 0-порождаем читателя, 1-писателя*/
	/*цикл порождения читателей и писателей чередованием*/
	while (nr>0 || nw>0) {
		pthread_create(&th_id, NULL, bFlag ? writer : reader, NULL);
		printf("+++%s %#x запущен на %d секунде\n", bFlag ?
			"Писатель" : "Читатель", th_id, time(0) - tmstart);
		/*выбираем тип нити для запуска на следующей итерации
			чередованием(если не исчерпаны)*/
		bFlag = bFlag ? (nw--, nr == 0) : (nr--, nw);
	}
	for (;;) sleep(5); /*бесконечный цикл ожидания*/
	return 0;
}
```

Необходимо отметить, что данная реализация читателей (функции `reader`) **не является полностью корректной**, т.к. открытие мьютекса db может осуществляться другой нитью, чем та, которая перевела его в состояние закрыт – согласно описанию функций `pthread_mutex_lock()`, `pthread_mutex_unlock()` закрытие и открытие мьютекса может осуществляться только в одной и той же нити.
Для решения данной проблемы в программу можно добавить неблокирующее ожидание в нить, которая закрыла мьютекс, и поднять его при опустошении текущей очереди читателей. Таким образом, более корректная реализация функции `reader` может выглядеть следующим образом:

```
/*реализация нити читателя – корректная версия*/
void* reader(void * arg)
{
	int flag = 0;
	pthread_mutex_lock(&mutex);
	rc++;
	if (rc == 1) {
		flag = 1; /*флаг – данная нить захватила мьютекс*/
		pthread_mutex_lock(&db);
	}
	pthread_mutex_unlock(&mutex);
	do_some_work(5);

	pthread_mutex_lock(&mutex);
	rc--;

	while (flag) {
		/*цикл ожидания работает только в нити (первом читателе),
		которая захватила мьютекс db */

		if (rc == 0) {
			/*освобождение мьютекса db захватившей его нитью */
			pthread_mutex_unlock(&db);
			flag = 0;
		}
		else { /* ожидание окончания текущей очереди читателей для
			корректного(в этой же нити) освобождения db*/

			/*перепроверка состояния очереди читателей раз в
			секунду, неблокирующее ожидание */
			pthread_mutex_unlock(&mutex);
			sleep(1);
			pthread_mutex_lock(&mutex);
		}
	}
	pthread_mutex_unlock(&mutex);

	printf("---Читатель %#x завершился на %d секунде\n",
		pthread_self(), time(0) - tmstart);
	return NULL;
}
```

Рекомендованные задания.

1) Запустить программу, реализующую пример на компьютере.
2) Провести и запустить на компьютере модификации программы, реализующей пример, в части:

- добавления контроля и обработки всех возможных ошибок;
- завершения main по завершению всех нитей;
- модификации программы для обеспечения приоритета нитей-писателей.

4 Файловые системы

4.1 Основные концепции

Под ***файловой системой*** (ФС) будем понимать часть операционной системы, представляющую собой совокупность организованных наборов данных, хранящихся на внешних запоминающих устройствах, и программных средств, гарантирующих именованный доступ к этим данным и их защиту.

С точки зрения пользователя, файловая система является первым виртуальным ресурсом (который появился в операционных системах), достаточно понятным и достаточно просто используемым во время работы пользователя за компьютером. Если сравнить ФС с другим виртуальным ресурсом — например, виртуальной памятью, то рядовому пользователю ПК может быть совсем не понятным, зачем нужен механизм виртуальной памяти. Появление ФС кардинально изменило взгляд на использование вычислительных систем. Почти сразу с момента использования вычислительной техники возникла проблема размещения данных во внешней памяти. Необходимость поддержки этого размещения обуславливалось несколькими причинами. Во-первых, была тривиальная необходимость сохранения данных: время «одноразовых» решений задач (когда требовалось, грубо говоря, лишь вычислить значение некой формулы) прошло достаточно быстро. Появились задачи, требующие больших объемов начальных данных, которые, в свою очередь, являлись результатом решения другой задачи. И эти данные надо было где-то сохранять, причем, сохранять без наличия программ, которые их используют. Как следствие, возникла проблема эффективности доступа к этим данным. Вторая необходимая проблема — это само сохранение информации (и программ, и данных). Имеется в виду, сам факт долгосрочного хранения информации.

С точки зрения аппаратной поддержки можно выделить следующие этапы развития. Одним из первых внешних запоминающих устройств (ВЗУ) была магнитная лента. Магнитная лента — это устройство последовательного доступа, информация на котором хранится в виде записей (фиксированного или переменного размера). Запись структурно состоит из последовательности содержательной информации, ограниченной маркерами начала и конца записи. Для доступа к информации необходимо иметь номер соответствующей записи на ленте. Соответственно, если пользователь хотел сохранять данные на магнитной ленте, то ему было необходимо знать магнитную ленту как носитель (по номеру или по расположению в хранилище, и т.п.) и номер своей записи на этой ленте. Отметим, что относительно эффективная работа с лентой может быть достигнута лишь при персональном использовании: в случае, когда одновременно с одной лентой работают два пользователя, возникают достаточно большие накладные расходы (в частности, частое перематывание ленты на начало), что может привести даже к ее порче (разрыву). Решение проблемы корректности организации данных на магнитной ленте лежало на пользователе: если пользователь некорректно организовал запись своих данных, то он мог тем самым испортить и свои данные, и чужие записи на ленте.

На следующем этапе развития появились устройства прямого доступа (барабаны и магнитные диски), что естественно сказалось на адресации данных на носителе. Например, чтобы получить доступ к информации на диске, достаточно знать номер диска, номер поверхности, номер цилиндра и номер сектора. Но каждое устройство прямого доступа имело и свои особенности — в частности, размеры блока: у одних дисков блоки имели размер 256 байт, у других — 512 байт, и т.д. И эти особенности пользователь должен был учитывать при работе с данными устройствами. Чтобы разместить свой файл на диске, пользователь должен был разбить этот файл на блоки (в зависимости от конкретного устройства хранения), найти на диске свободные блоки, чтобы в них разместить весь свой файл, сохранить файл и запомнить координаты и последовательность блоков, в которых

был сохранен файл. Заметим, что диски ориентированы на массовое использование, т.е. предполагается работа с диском двух и более пользователей, что накладывало дополнительные трудности на корректное размещение данных на диске — задачу, совершенно нетривиальную для рядового пользователя.

Подобный подход к хранению данных продлился примерно до середины 60-х — начала 70-х годов, когда в машинах второго поколения появился программный компонент операционной системы, который получил название *файловая система*. Повторимся, ***файловая система*** — это компонент операционной системы, обеспечивающий **корректный именованный доступ** к данным пользователя. Данные в файловой системе представляются в виде файлов, каждый из которых имеет имя. Главными характеристиками в определении файловой системы являются **именованный доступ** и **корректная работа**. Последнее означает, что файловая система обеспечивает корректное управление свободным и занятым пространством на ВЗУ (заметим, что не обязательно на физическом устройстве: в качестве ВЗУ может выступить и виртуальное устройство), а также защиту информации от несанкционированного доступа. Большинство современных файловых систем обеспечивают корректную организацию распределенного доступа к одному и тому же файлу (когда с ним могут работать два и более пользователя). Это не означает, что система будет отвечать за корректную семантику данных внутри файла: гарантируется, что система обеспечит корректный доступ пользователей к файлу с точки зрения системной организации. Также многие современные файловые системы поддерживают возможность синхронизации доступа к информации.

4.1.1 Структурная организация файлов

С точки зрения структурной организации файлов имеется целый спектр различных подходов. Существует некоторая установившаяся систематизация методов структурной организации файлов. Рассмотрим модели в соответствии с хронологией их появления.

Первой моделью файла явилась модель файла как **последовательности байтов**. В этом случае содержимое файла представляется как неинтерпретируемая информация (или интерпретируемая примитивным образом). Задача интерпретации данных ложится на пользователя. Данная модель файла наиболее распространена на сегодняшний день: большинство широко используемых файловых систем поддерживают возможность представлять содержимое файла как последовательности байтов, а это означает, что программные интерфейсы, обеспечивающие доступ к содержимому файла, позволяют считывать и записывать произвольные порции данных.

Следующие модели представляют файл как **последовательность записей переменной и постоянной длины**. Первая из этих моделей является аналогом магнитной ленты. Соответственно, эта организация файла и рассчитана на работу с магнитными лентами. В этом случае возникают проблемы, такие как коррекция данных в середине файла, вследствие чего меняется размер файла, и появляется необходимость сдвигать «хвост» файла на ленте.

Модель файла как последовательности записей постоянной длины является также аппаратно-ориентированной: она является аналогом перфокарты. Перфокарта представляет собою картонный листок прямоугольной формы, на котором изображены двенадцать строк по 80 позиций в каждой. Каждая позиция соответствует одному биту информации. Соответственно, перфокарта может хранить лишь фиксированное количество данных, поэтому для отображения колоды перфокарт в файл подходит модель файла как последовательности записей фиксированного размера (каждая запись являлась образом одной перфокарты). Данная модель имеет следующие недостатки. Во-первых, из-за того, что каждая запись имеет фиксированный размер, возникает внутренняя фрагментация: т.е. если хотя бы один байт занят в записи, то занят и весь объем записи. Также остаются проблемы, возникающие при необходимости вставить или удалить запись из середины файла.

И, наконец, модель **иерархической организации файла**. В данной модели организация файла имеет сложную логическую структуру, позволяющую организовывать динамическую работу с данными. Одной из наиболее распространенных структур является дерево, в узлах которого расположены записи. Каждая запись состоит из двух полей: поле ключа и поле данных. В качестве ключа может выступать номер записи. Данная модель является удобной для редактирования файла, но, с другой стороны, требует достаточной сложной реализации.

Еще одной исторической характеристикой файлов были режимы доступа, отражавшие организацию внешних устройств. Были файлы прямого доступа и файлы последовательного доступа. Режим доступа задавался на этапе создания файла. В современных файловых системах эти режимы не используются. Зато актуальны режимы доступа с точки зрения разрешения или запрета определенные операций: возможно иметь доступ только по чтению, только по записи или по чтению-записи информации в файл.

4.1.2 Атрибуты файлов

Каждый файл обладает фиксированным набором параметров, характеризующих свойства и состояния файла, причем и долговременное (стратегическое), и оперативное состояния. Совокупность этих параметров называют ***атрибутами файла***. В набор атрибутов может входить достаточно большое количество параметров, и состав атрибутов зависит от конкретной реализации системы. Среди атрибутов часто можно встретить следующие параметры: имя файла, права доступа, персонификация (создатель/владелец), тип файла, размер записи (блока), размер файла, указатель чтения/записи, время создания, время последней модификации, время последнего обращения, предельный размер файла и т.п.

Под ***именем файла*** понимается последовательность символов, используя которую организуется именованный доступ к данным файла. В одних файловых системах имя файла воспринимается в качестве атрибута, другие ФС разделяют файл (его содержимое), имя и отдельно набор атрибутов.

Следующим немаловажным атрибутом являются ***права доступа***. Данный атрибут характеризует возможность доступа к содержимому файла различным категориям пользователей. Структура категорий пользователей, по которой организуется доступ, зависит от конкретной операционной системы. В частности, существуют операционные системы, в которых прав доступа нет: файлы доступны любому пользователю системы.

Следующий атрибут — ***персонификация*** — связан с предыдущим. Соответственно, данный атрибут содержит информацию о принадлежности файла. В общем случае здесь может находиться несколько параметров: например, информация о создателе файла, а также информация о владельце файла. Зачастую эти параметры совпадают, но возможны ситуации, когда они отличаются.

Тип файла — информация о способе организации файла и интерпретации его содержимого. Говоря о способе организации, можно привести пример файловой системы ОС Unix, которая поддерживает разные типы файлов. Среди прочих имеются т.н. ***файлы устройств***, соответствующие тем устройствам, которые обслуживает данная ОС; и через эти файлы устройств происходит фактически обращение к драйверам устройств. Совсем иначе организованы ***регулярные файлы***, которые могут хранить различную информацию (текстовую, графическую и пр.). О различных способах организации речь пойдет ниже.

Если речь идет об интерпретации, то она может быть ***явной*** и ***неявной***, т.е. возможно указание, как интерпретировать содержимое файла. Например, можно указать, является ли данный файл исполняемым или неисполняемым. Исполняемый файл можно запустить как процесс, в отличие от неисполняемого. Таким образом, атрибут типа файла может содержать многоуровневую комплексную информацию.

Размер записи (или ***размер блока***). В системе имеется возможность указать, что данный файл организован в виде последовательности блоков данного размера, при этом

размер определяется пользователем (пользовательским процессом). Размер может быть ***стационарным***, когда при создании файла указывается фиксированный размер блоков, и ***нестационарным***, когда размер блока задается каждый раз при открытии файла.

Размер файла. Данный атрибут имеет достаточно простой смысл; заметим, что обычно размер файла задается в байтах.

Указатель чтения/записи — это указатель, относительно которого происходит чтение или запись информации. В общем случае с каждым файлом ассоциируются два указателя (и на чтение, и на запись), хотя бывают файловые системы, в которых используется единый указатель чтения/записи. Соответственно, операции чтения/записи оперируют данными, следующими за указателями.

Среди прочих атрибутов файла возможны атрибуты, отражающие системную и статистическую информацию о файле: например, ***время последней модификации, время последнего обращения***, ***предельный размер файла*** и т.д. Еще одной важной группой атрибутов являются атрибуты, хранящие информацию о размещении содержимого файла, т.е. где в файловой системе организовано хранение данных файла, и как оно организовано.

4.1.3 Основные правила работы с файлами. Типовые программные интерфейсы

Практически все файловые системы при организации работы с файлами действуют по схожим сценариям, которые в общем случае состоят из трех основных блоков действий.

Первый этап – это **начало работы с файлом** (или **открытие файла**). В большинстве систем для процессов, желающих работать с файлом, имеются операции открытия файла. Посредством данной операции процесс передает файловой системе запрос на работу с конкретным файлом. Получив запрос, файловая система производит соответствующие проверки возможности (в т.ч. на наличие полномочий) работы с файлом; в случае успеха выделяет внутри себя необходимые ресурсы для работы процесса с указанным файлом. В частности, для каждого открытого файла создается т.н. ***файловый дескриптор*** – системная структура данных, содержащая информацию об актуальном состоянии открытого файла (режимы, позиции указателей и т.п.). Файловый дескриптор, как системная структура данных, может размещаться как в адресном пространстве процесса, так и в пространстве памяти операционной системы. Соответственно, при открытии файла процесс получает либо номер файлового дескриптора, либо указатель на начало данной структуры. Все последующие операции с содержимым файла происходят с указанием именно файлового дескриптора, и эти операции образуют следующий блок действий.

Второй блок действий образуют **операции по работе с содержимым** файла (чтение и запись), также **операции**, **изменяющие атрибуты** файла (режимы доступа, изменение указателей чтения/записи и т.п.).

Последний этап — это **закрытие файла**: уведомление системе о закрытии процессом файлового дескриптора (а не файла). Подчеркнем, что процесс прекращает работу не с файлом, а с конкретным файловым дескриптором, поскольку даже в рамках одного процесса можно открыть один и тот же файл два и более раза, и на каждое открытие будет предоставлен новый файловый дескриптор. После операции закрытия операционная система выполняет необходимые действия по корректному завершению работы с файловым дескриптором, а если закрывается последний открытый дескриптор — то осуществляет корректное завершение работы с файлом: в частности, по необходимости освобождаются системные ресурсы, в т.ч. выполняется разгрузка кэш-буферов файловых обменов и т.д.

Структурно каждая операционная система предлагает унифицированный набор интерфейсов, посредством которых можно обращаться к системным вызовам работы с файлами. Обычно этот набор содержит следующие основные функции:

- **open** — открытие/создание файла;

– **close** — закрытие;
– **read/write** — чтение/запись (относительно положения указателя чтения/записи соответственно);
– **delete** — удаление файла из файловой системы;
– **seek** — позиционирование указателя чтения/записи;
– **read_attributes/write_attributes** — чтение/модификация некоторых атрибутов файла (в файловых системах, рассматривающих имя файла не как атрибут, возможна дополнительная функция переименования файла — **rename**).

Практически все файловые системы включают в свой состав некоторый специальный компонент, посредством которого можно установить соответствие между именем файла и его атрибутами. Используя это соответствие, можно получить информацию о размещении данных в файловой системе и организовать доступ к данным. И этим компонентом является ***каталог***. Итак, ***каталог*** — это системная структура данных файловой системы, в которой находится информация об именах файлов, а также информация, обеспечивающая доступ к атрибутам и содержимому файлов. Каталоги являются специальным видом файлов.

Рассмотрим типовые модели организации каталогов (в соответствии с хронологическим порядком их появления).

Первой исторической моделью является **одноуровневая файловая система** (**система с одноуровневым каталогом**). В файловой системе данного типа (Рис. 103) присутствует единственный каталог, в котором перечислены всевозможные имена файлов, находящихся в данной системе. Этот каталог устроен простым способом: для каждого файла хранится информация об его имени, расположении первого блока и размере файла. Эта простота влечет за собой и простоту доступа к информации файлов, но эта модель не предполагает многопользовательской работы. В данном случае возможны коллизии имен (когда возникают попытки создания файлов с одним именем). Данная модель в настоящее время используется в бытовой технике, которая выполняет фиксированный набор действий.

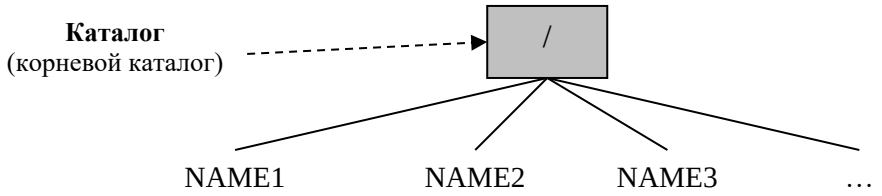


Рис. 103. Модель одноуровневой файловой системы.

Следующей моделью является **двухуровневая файловая система** (Рис. 104). Данная модель предполагает работу нескольких пользователей: файловая система этого типа позволяет группировать файлы по принадлежности тому или иному пользователю. Эта модель лучше одноуровневой (в частности, не возникают коллизии имен файлов разных пользователей), но и она обладает недостатками: зачастую неудобно и даже нежелательно расположение всех файлов одного пользователя в одном месте (в одном каталоге). В частности, остается проблема коллизии имен для файлов одного пользователя.

Отметим, что двух-, трех- и вообще N-уровневые (N – фиксированное) модели остаются актуальными и по сей день. Объясняется это тем, что они, в первую очередь, достаточно просты по своей структуре и организации работы с ними. Если мы посмотрим на простейший мобильный телефон, имеющий несколько уровней меню, в них обычно реализованы именно подобные файловые системы.

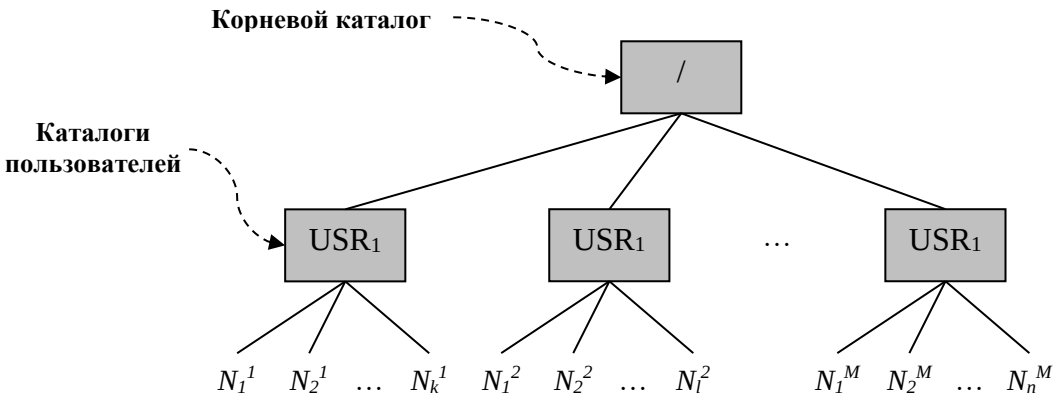


Рис. 104. Модель двухуровневой файловой системы.

И, наконец, последняя модель, которую мы рассмотрим, — **иерархическая файловая система** (Рис. 105). Современные многопользовательские файловые системы основываются на использовании иерархических структур данных, в частности, на использовании деревьев.

Вся информация в иерархической файловой системе представляется в виде дерева, имеющего корень. Это т.н. ***корневая файловая система***. В узлах дерева, отличных от листьев, находятся каталоги, которые содержат информацию о размещенных в них файлах. Иерархические файловые системы обычно имеют специальный тип файлов-каталогов. Т.е. каталог представляется не как отдельная выделенная структура данных, а как файл особого типа. Листом дерева может быть либо файл-каталог, либо любой файл файловой системы.

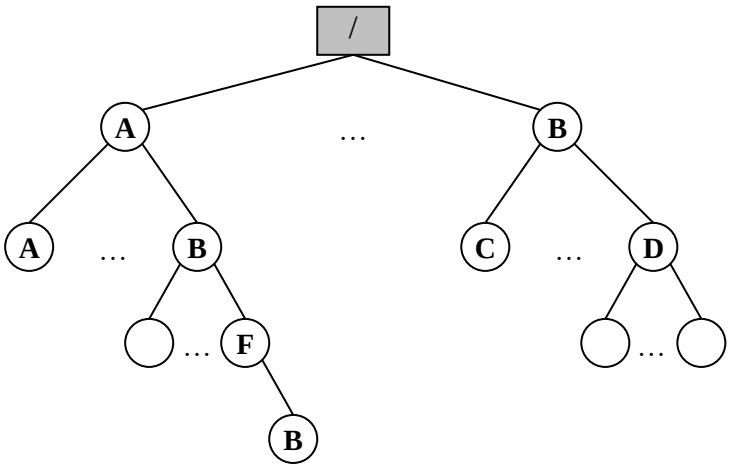


Рис. 105. Модель иерархической файловой системы.

Иерархическая (или древообразная) организация файловой системы предоставляет возможность использования уникального именования файлов. Оно основывается на том, что в дереве существует единственный путь от корня до любого узла. Приведенная схема именования (от корня до конкретного узла дерева) является принципиальной схемой именования файлов в иерархических файловых системах. При этом обычно используются следующие характеристики. ***Текущий каталог*** — это каталог, на работу с которым в данный момент настроена файловая система. Текущим каталогом может стать любой каталог файловой системы, и обозревание файлов в файловой системе происходит относительно этого каталога. Файлы, находящиеся непосредственно в текущем каталоге, доступны «просто» по имени. Таким образом, ***имя файла*** — это имя файла оносительно текущего каталога, а ***полное имя файла*** — это перечень всех имен файлов от корня до узла с данным файлом. Признаком полного имени обычно является присутствие специального

префиксного символа, обозначающего корневой каталог (например, в ОС Unix в качестве корневого каталога выступает символ “/”).

Иерархическая файловая система позволяет использовать т.н. ***относительные имена файлов*** — это путь от некоторого каталога до данного файла. Для данного способа именования необходимо указать явно или неявно каталог, относительно которого строится это именование. Например, если существует файл с полным именем **/A/B/C/D**, то относительно каталога **B** файл будет иметь имя **C/D**. Чтобы использовать это относительное имя, необходимо либо явно задать каталог **B** (по сути это означает задание полного имени), либо сделать каталог **B** текущим.

Иерархические файловые системы обычно используют еще одну характеристику — т.н. ***домашний каталог***. Суть его заключается в том, что для каждого зарегистрированного в системе пользователя (или для всех пользователей) задается полное имя каталога, который должен стать текущим каталогом при входе пользователя в систему.

4.1.4 Подходы в практической реализации файловой системы

Рассмотрим некоторые подходы в практической реализации файловой системы. Снова вернемся к понятию ***системного устройства*** — устройства, на котором, как считается аппаратурой компьютера, должна присутствовать операционная система. Почти в любом компьютере можно определить некоторую цепочку внешних устройств, которые при загрузке компьютера могут рассматриваться как системные устройства. В этой цепочке имеется определенный приоритет. Во время старта вычислительная система (компьютер) перебирает данную цепочку в порядке убывания приоритета до тех пор, пока не обнаружит готовое к работе устройство. Система предполагает, что на этом устройстве имеется необходимая системная информация и пытается загрузить с него операционную систему. Например, допустим, что компьютер сконфигурирован таким образом, что первым системным устройством является флоппи-дисковод, вторым — дисковод оптических дисков (CDROM), а третьим — жесткий диск. Мы поместили во флоппи-дисковод дискету и включили компьютер. Аппаратный загрузчик обращается к первому системному устройству, и поскольку в дисководе присутствует дискета, то считается, что дисковод готов к работе. Тогда происходит попытка загрузки операционной системы с флоппи-диска, и если это будет неуспешно, то будет выведено сообщение об ошибке загрузки системы. Если же дискеты во флоппи-дисководе не будет, но будет находиться диск в CDROM, то точно так же будет предпринята попытка загрузить операционную систему, но уже с оптического диска. Если же не будет ни флоппи-дискеты, ни оптического диска, то загрузчик попытается загрузить операционную систему с жесткого диска. Обычно в штатном режиме загрузка происходит с жесткого диска, но в ситуации, например, краха системы и невозможности загрузиться с жесткого диска приведенная модель позволяет загрузить систему со съемного носителя и произвести некие действия по восстановлению работоспособности поврежденной системы.

В приведенной модели работа аппаратного загрузчика основана на предположении о том, что любое системное устройство имеет некоторую предопределенную структуру (Рис. 106). В начальном блоке системного устройства располагается **основной программный загрузчик** (MBR — Master Boot Record). В качестве основного программного загрузчика может выступать как загрузчик конкретной операционной системы, так и некоторый унифицированный загрузчик, имеющий информацию о структуре системного диска и способный загружать по выбору пользователя одну из альтернативных операционных систем.

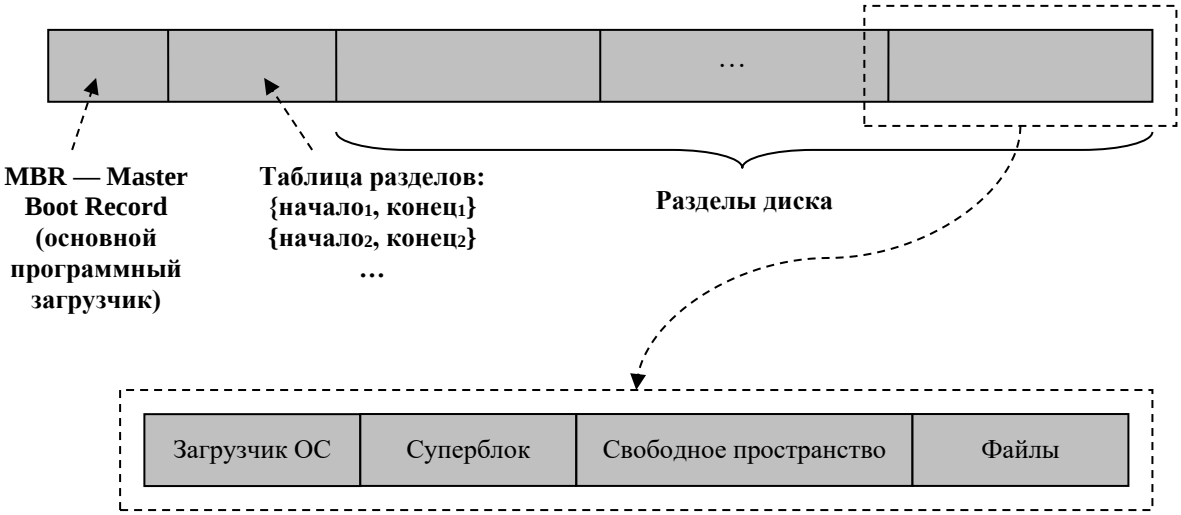


Рис. 106. Структура «системного» диска.

В общем случае после блока основного программного загрузчика на диске следует последовательность блоков, в которых находится т.н. ***таблица разделов***. В современных жестких дисках имеется возможность разбивать все физическое пространство диска на некоторые области, которые называются ***разделами*** (***partition***). Внутри каждого раздела может быть помещена в общем случае своя операционная система. Соответственно, границы каждого раздела (его конец и начало) регистрируются в указанной таблице разделов. Еще одно важное применение данной таблицы связано с тем, что современные диски имеют настолько большие емкости, что для адресации произвольной точки диска не хватает разрядной сетки процессора. И за счет косвенной адресации (адресации относительно начала раздела) использование подобной таблицы позволяет решить данную проблему.

Логическая структура раздела имеет следующий вид. В начальном блоке раздела находится ***загрузчик*** конкретной ***операционной системы***. Все остальное пространство раздела обычно занимает файловая система. Зачастую в файловой системе часть пространства выделяется т.н. ***суперблоку***, в котором хранятся настройки (размеры блоков, режимы работы и т.п.) и информация об актуальном состоянии (информация о свободных и занятых блоках и т.п.) файловой системы. Все оставшееся пространство файловой системы состоит из свободных и занятых блоков, т.е. блоков, способных хранить системные и пользовательские данные.

Прежде, чем продолжить изучение способов организации файловых систем, хотелось бы остановиться и еще раз просмотреть, что происходит при **загрузке компьютера**. При включении компьютера управление передается аппаратному загрузчику, который просматривает (согласно приоритетам) список системных устройств, определяет готовое к работе устройство и передает управление основному программному загрузчику этого устройства. Последний является программным компонентом и может загрузить конкретную операционную систему, а может являться мультисистемным загрузчиком, способным предложить пользователю выбрать, какую из операционных систем, расположенных в различных разделах диска, загрузить. В одном разделе может находиться, например, ОС Microsoft Windows XP, в другом — Linux, в третьем — FreeBSD, и т.д. Данный мультисистемный загрузчик владеет информацией, какая операционная система в каком разделе диска находится. После того, как пользователь сделал свой выбор, загрузчик по таблице разделов определяет координаты соответствующего раздела и передает управление загрузчику операционной системы указанного раздела. Соответственно, загрузчик операционной системы производит непосредственную загрузку этой ОС.

Говоря об **иерархии блоков**, у многих создается впечатление, что такие понятия, как, например, блоки файла, блоки файловой системы, блоки устройств и т.п., обозначают

одно и то же, что, в общем случае, неверно. Большинство современных операционных систем поддерживают целую иерархию блоков, используемую при организации работы с блок-ориентированными устройствами. В основе этой иерархии лежат блоки физического устройства, т.е. это те порции данных, которыми можно совершать обмен с данным физическим устройством. Более того, размер блока физического устройства зависит от конкретного устройства. Соответственно, детали этого уровня иерархии скрываются следующим уровнем абстракции — блоками виртуального диска. Следующий уровень иерархии — уровень блоков файловой системы. Эти блоки используются при организации структуры файловой системы. Размер блока файловой системы, равно как и виртуального диска, является стационарной характеристикой, определяемой при настройке системы, и в динамике эта характеристика не меняется. И, наконец, последний уровень представляют блоки файла. Размер данных блоков может определить пользователь при открытии или создании файла (как об этом говорилось выше). Отметим, что размер блока, в конечном счете, влияет на эффективность работы, которая будет несколько выше, если размеры всех блоков будут хотя бы кратны друг другу.

4.1.5 Модели реализации файлов

Первой тривиальной и самой эффективной с точки зрения минимизации накладных расходов является **модель непрерывных файлов** (Рис. 107). Данная модель подразумевает размещение каждого файла в непрерывной области внешнего устройства. Эта организация достаточно простая: для обеспечения доступа к файлу среди атрибутов должны присутствовать имя, блок начала и длина файла. Но тут возникают следующие проблемы. Во-первых, внутренняя фрагментация (хотя это проблема почти всех блок-ориентированных устройств). В качестве иллюстрации можно привести следующее: если необходимо хранить всего один байт, то для этого будет выделен целый блок, который, по сути, будет пустым. Во-вторых, это фрагментация между файлами (эта проблема обсуждалась при рассмотрении моделей организации работы оперативной памяти). Но данная система имеет и некоторые достоинства, и немаловажное из них — отсутствие фрагментации файла по диску: поскольку файл хранится в единой непрерывной области диска, то при считывании файла головка жесткого диска совершает минимальное количество механических движений, что означает более высокую производительность системы. Соответственно, при реализации данной модели должна решаться важная проблема, возникающая при модификации файла, в частности, при увеличении его содержательной части. Чтобы несколько упростить эту задачу, зачастую используют такой прием: при создании файла к запрошенному объему добавляют некоторое количество свободного пространства (например, 10% от запрошенного объема), но этот же прием ведет к увеличению внутренней фрагментации. Еще одной немаловажной проблемой является фрагментация свободного пространства. Файловые системы, реализующие данную модель хранения файла, являются деградирующими: в ходе эксплуатации фрагментация свободного пространства увеличивается, и в итоге на диске имеется множество мелких свободных участков, имеющих очень большой суммарный объем, но из-за своего небольшого размера эти участки использовать не представляется возможным. Для разрешения этой проблемы необходимо запускать процесс компрессии (дефрагментации), который занимается тем, что сдвигает файлы с учетом зарезервированного для каждого файла «запаса», «прижимая» их друг к другу. Эта операция трудоемкая, продолжительная и опасная, поскольку при перемещении файла возможен сбой.

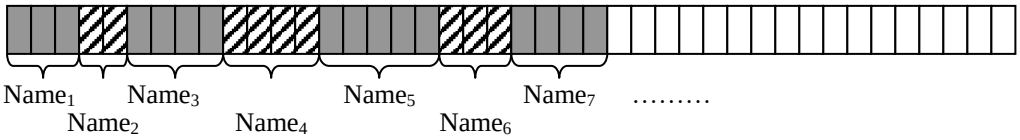


Рис. 107. Модель непрерывных файлов.

Следующей моделью является **модель файлов, имеющих организацию связанного списка** (Рис. 108). В этой модели файл состоит из блоков, каждый из которых включает в себя две составляющие: данные, хранимые в файле, и ссылка на следующий блок файла. Эта модель также является достаточно простой, достаточно эффективной при организации последовательного доступа, а также эта модель решает проблему фрагментации свободного пространства (за исключением блочной фрагментации). С другой стороны, обозначенная модель не предполагает прямого доступа: чтобы обратиться к i-ому блоку, необходимо последовательно просмотреть все предыдущие. Также эта модель предполагает фрагментацию файла по диску, т.е. содержимое файла может быть рассредоточено по всему дисковому пространству, а это означает, что при последовательном считывании содержимого файла добавляется значительная механическая составляющая, снижающая эффективность доступа. И еще одним недостатком данной модели является то, что в каждом блоке присутствует и системная, и пользовательская информация. Возможна ситуация, что при необходимости считать данные, объемом в один логический блок, реально происходит чтение двух блоков.

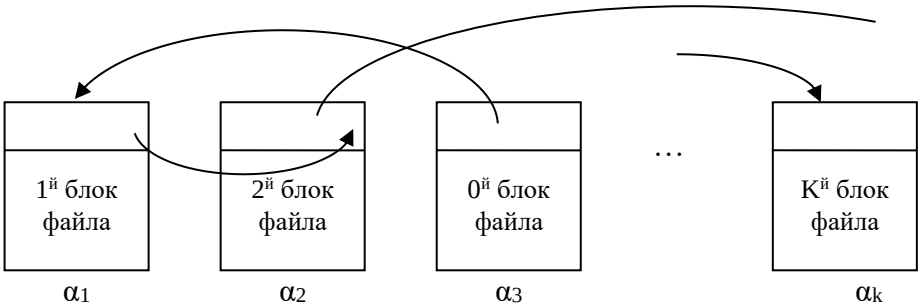


Рис. 108. Модель файлов, имеющих организацию связанного списка.

Другой подход иллюстрирует модель, основанная на использовании т.н. **таблицы размещения файлов** (***File Allocation Table — FAT***, Рис. 109). В этой модели операционная система создает программную таблицу, количество строк в которой совпадает с количеством блоков файловой системы, предназначенных для хранения пользовательских данных. Также имеется отдельный каталог (или система каталогов), в котором для каждого имени файла имеется запись, содержащая номер начального блока. Соответственно, таблица размещения имеет позиционную организацию: i-ая строка таблицы хранит информацию о состоянии i-ого блока файловой системы, а, кроме того, в ней указывается номер следующего блока файла. Чтобы получить список блоков файловой системы, в которых хранится содержимое конкретного файла, необходимо по имени в указанном каталоге найти номер начального блока, а затем, последовательно обращаясь к таблице размещения и извлекая из каждой записи номер следующего блока, возможно построение искомого списка. Данное решение выглядит эффективнее предыдущего. Во-первых, в этом решении весь блок используется полностью для хранения содержимого файла. Во-вторых, при открытии файла можно составить список блоков данного файла и, следовательно, осуществлять прямой доступ. Заметим, что для максимальной эффективности необходимо, чтобы эта таблица целиком размещалась в оперативной памяти, но для современных дисков, имеющих огромные объемы, данная таблица будет иметь большой размер

(например, для 60-ти гигабайтного раздела и блоков, размером 1 килобайт, потребуется 60 000 000*4(байта) = 240 мегабайт), что является одним из недостатков рассматриваемой модели. Другим недостатком является ограничение на размер файла в силу ограниченности длины списка блоков, принадлежащих данному файлу.

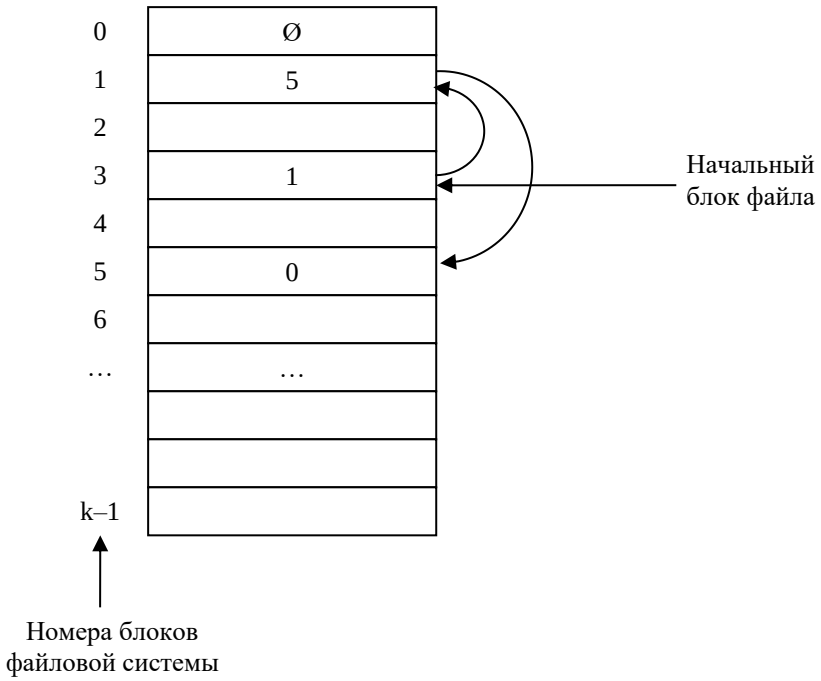


Рис. 109. Таблица размещения файлов (FAT).

Следующая модель — модель организации файловой системы с использованием т.н. ***индексных узлов*** (***дескрипторов***). Принцип модели состоит в том, что в атрибуты файла добавляется информация о номерах блоков файловой системы, в которых размещается содержимое файла. Это означает, что при открытии файла можно сразу получить перечень блоков. Соответственно, необходимость использования FAT-таблицы отпадает, зато, с другой стороны, при предельных размерах файла размер индексного дескриптора становится соизмеримым с размером FAT-таблицы. Для разрешения этой проблемы существует два принципиальных решения. Во-первых, это тривиальное ограничение на максимальный объем файла. Во-вторых, это построение иерархической организации данных о блоках файла в индексном дескрипторе. В последнем случае вводятся иерархические уровни представления информации: часть (первые N) блоков перечисляются непосредственно в индексном узле, а оставшиеся представляются в виде косвенной ссылки. Это решение имеет следующие преимущества. Нет необходимости в размещении в ОЗУ информации всей FAT-таблицы обо всех файлах системы. В памяти размещаются атрибуты, связанные только с открытыми файлами. При этом индексный дескриптор имеет фиксированный размер, а файл может иметь практически «неограниченную» длину.

4.1.6 Модели реализации каталогов

Существуют несколько подходов организации каталогов (Рис. 110). Во-первых, каталог может представляться в виде таблицы, у которой в одной колонке находятся имена файлов, а в остальных — все атрибуты. Эта модель хороша тем, что все необходимые данные оперативно доступны, но размер записи в таблице каталога, да и сама таблица, может быть большой (например, из-за большого числа атрибутов), что влечет за собой долгий поиск в каталоге. Другой подход заключается в том, что каталог также представляется в виде таблицы, в которой один столбец хранит имена, а в другом хранится ссылка на системную таблицу, содержащую атрибуты соответствующего файла. Этот

подход имеет дополнительное преимущество, заключающееся в том, что для разных типов файлов можно иметь различный набор атрибутов.

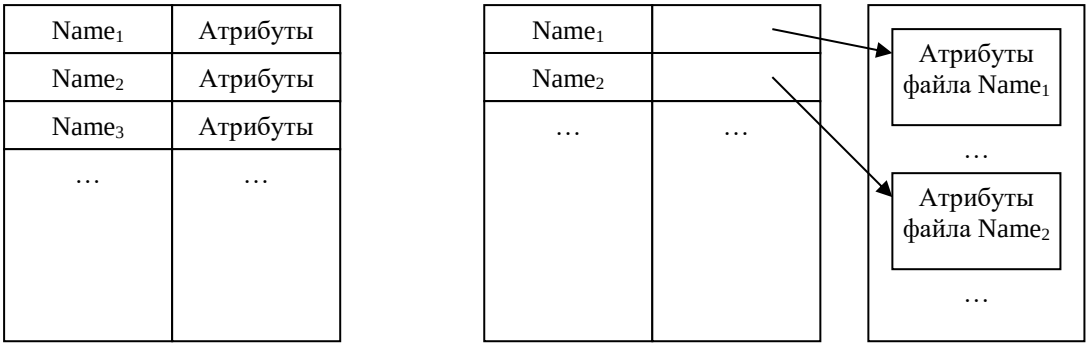


Рис. 110. Модели организации каталогов.

Одной из проблем при организации каталогов является проблема длины имени. Изначально эта проблема решалась достаточно просто: имя файла было ограничено шестью или восемью символами. В современных системах разрешается присваивать файлам достаточно длинные имена. И, как следствие, встает иная проблема: для эффективной работы с каталогом необходимо, чтобы информация в каталогах хранилась в сжатом виде, в т.ч. и имена файлов должны быть короткими. О некоторых решениях данной проблемы речь пойдет ниже при обсуждении файловой системы ОС Unix.

4.1.7 Соответствие имени файла и его содержимого

Еще один момент, на который стоит обратить внимание при рассмотрении организации файловых систем, — это проблема соответствия между именем файла и содержимым этого файла.

Как отмечалось выше, у любого файла есть имя как отдельная характеристика файла или же как один из атрибутов файла. В различных файловых системах по-разному решается указанная проблема соответствия имени и содержимого файла. Первый очевидный подход заключается в том, что устанавливается **взаимнооднозначное соответствие**, т.е. для каждого содержимого файла в системе существует единственное имя, ассоциированное с этим содержимым, и обратно — для каждого имени существует единственное содержимое.

На сегодняшний день почти все современные файловые системы позволяют нарушать это взаимнооднозначное соответствие путем предоставления возможности установления для одного и того же содержимого файла двух и более имен. При этом, существует несколько моделей организации данного подхода.

Первая модель — это симметричное (или равноправное) именование. В этом случае с одним и тем же содержимым ассоциируется группа имен, каждое из которых равноправное. Это означает, что посредством любого из имен, ассоциированных с данным содержимым, можно выполнить все операции, и они будут выполнены одинаково. Такая модель реализована в некоторых файловых системах посредством установления т.н. ***жесткой связи*** (Рис. 111). В этом случае среди атрибутов есть счетчик имен, ссылающихся на данное содержимое. Уничтожая файл с одним из этих имен, производится следующий порядок действий. Файловая система уменьшает счетчик имен на единицу; если счетчик обнулился, то в этом случае удаляется содержимое, иначе файловая система удаляет только указанное имя файла из соответствующего каталога. Заметим, что при такой организации древовидность файловой системы (древовидность размещения файлов) нарушается, поскольку имена, ассоциированные с одним и тем же содержимым, можно разместить в разных узлах дерева, т.е. произвольных каталогах файловой системы. Но сохраняется древовидность именования файлов.

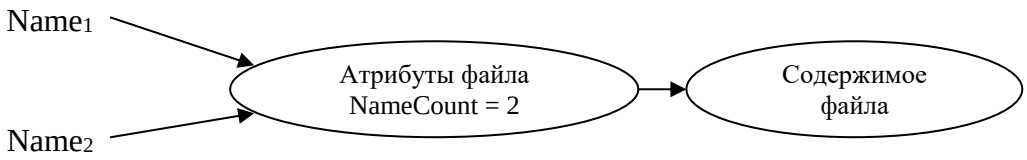


Рис. 111. Пример жесткой связи.

Следующая модель организации — это «мягкая» ссылка, или ***символическая связь*** (хотя здесь лучше бы подошло название *символьной* связи, Рис. 112). Данное именование является несимметричным, суть его заключается в следующем. Пусть существует содержимое файла, с которым жесткой связью ассоциировано некоторое имя ($Name_2$). К этому файлу теперь можно организовать доступ через файл-ссылку. Это означает, что создается еще один файл некоторого специального типа (типа файл-ссылка), в атрибутах которого указывается его тип и то, что он ссылается на файл с именем $Name_2$. Теперь можно работать с содержимым файла $Name_2$ посредством косвенного доступа через файл-ссылку. Но некоторые операции с файлом-ссылкой будут происходить иначе. Например, если вызывается операция удаления файла-ссылки, то удаляется именно файл-ссылка, а не файл с именем $Name_2$. Если же явно удалить файл $Name_2$, то в этом случае файл-ссылка окажется висячей ссылкой.

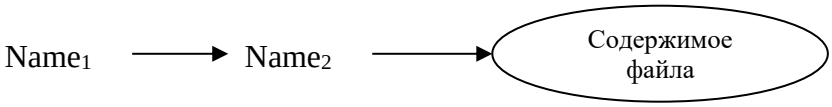


Рис. 112. Пример символической связи.

4.1.8 Координация использования пространства внешней памяти

С точки зрения организации использования пространства внешней памяти файловой системой, существует несколько аспектов, на которые необходимо обратить внимание. Первый момент связан с проблемой выбора размера блока файловой системы. Задача определения оптимального размера блока не имеет четкого решения. Если файловая система предоставляет возможность квотировать размер блока, то надо учитывать, что больший размер блока ведет к увеличению производительности файловой системы (поскольку данные файла оказываются локализованными на жестком диске, из чего следует, что при доступе снижается количество перемещений считывающей головки). Но недостатком является то, что чем больше размер блока, тем выше внутренняя фрагментация, а, следовательно, возникает неэффективность использования пространства ВЗУ (если, блок имеет размер, например, 1024 байт, а файл занимает 1 байт, то теряются 1023 байта). Альтернативой являются блоки меньшего размера, которые снижают внутреннюю фрагментацию, но при выборе меньшего размера блока повышаются накладные расходы при доступе к файлу в связи фрагментация файла по диску.

Еще одна проблема, на которую стоит обратить внимание, — это **проблема учета свободных блоков** файловой системы. Здесь тоже существует несколько подходов решения, среди которых нельзя выбрать наилучший: каждый из подходов имеет свои достоинства и недостатки.

Первый подход заключается в том, что вся совокупность свободных блоков помещается в единый список, т.е. номера свободных блоков образуют ***связный список***, который располагается в нескольких блоках файловой системы. Для более эффективной работы первый блок, содержащий начальную часть списка, должен располагаться в ОЗУ, чтобы файловая система могла к нему оперативно обращаться. Заметим, что список может достигать больших размеров, но размер списка не столь важен, поскольку по мере

использования свободных блоков этот список сокращается, при этом освобождающиеся блоки, хранившие указанный список, ничем не отличаются от других свободных блоков файловой системы, а значит, их можно использовать для хранения файловых данных.

Вторая модель основана на использовании ***битовых массивов***. В этом случае каждому блоку файловой системе ставится в соответствие двоичный разряд, сигнализирующий о незанятости данного блока. Для организации данной модели необходимо подсчитать количество блоков файловой системы, рассчитать количество разрядов массива, а также реализовать механизм пересчета номера разряда в номер блока и наоборот.

4.1.9 Квотирование пространства файловой системы

Как отмечалось выше, файловая система должна обеспечивать контроль использования двух видов системных ресурсов — это регистрация файлов в каталогах (т.е. контроль количества имен файлов, которое можно зарегистрировать в каталоге) и контроль свободного пространства (чтобы не возникла ситуация, когда один процесс заполнил все свободное пространство, тем самым не давая другим пользователям возможность сохранять свои данные). Для решения поставленных задач в файловой системе вводятся квотирование имен (т.е. числа) файлов и квотирование блоков (Рис. 113).

В общем случае модель квотирования может иметь два типа лимитов: ***жесткий*** и ***гибкий***. Для каждого пользователя при его регистрации в системе определяются два типа квот. ***Жесткий лимит*** — это количество имен в каталогах или количество блоков файловой системы, которое пользователь превзойти не может: если происходит превышение жесткого лимита, работа пользователя в системе блокируется. ***Гибкий лимит*** — это значение, которое устанавливается в виде лимита; с ним ассоциировано еще одно значение, называемое ***счетчиком предупреждений*** (гибкий лимит превышать можно, но после этого включается обратный счётчик предупреждений). При входе пользователя в систему происходит подсчет соответствующего ресурса (числа имен файлов либо количества используемых пользователем блоков файловой системы). Если вычисленное значение не превосходит гибкий лимит, то счетчик предупреждений сбрасывается на начальное значение, и пользователь продолжает свою работу. Если же вычисленное значение превосходит установленный гибкий лимит, то значение счетчика предупреждений уменьшается на единицу, затем происходит проверка равенства его значения нулю. Если значение счётчика равно нулю, то вход пользователя в систему блокируется, иначе (если значение больше нуля) пользователь получает предупреждение о том, что соответствующий гибкий лимит израсходован, после чего пользователь может работать дальше. Таким образом, система позволяет пользователю привести свое «файловое пространство» в порядок в соответствии с установленными квотами.

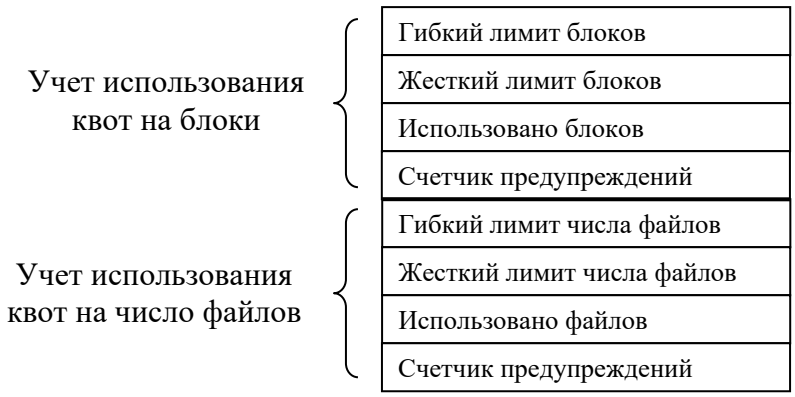


Рис. 113. Квотирование пространства файловой системы.

Рассмотренная модель имеет большую эффективность при использовании именно пары этих параметров. Если в системе реализован лишь гибкий лимит, то можно реализовать упоминавшуюся картину: пользовательский процесс может «забить» все свободное пространство файловой системы. Данную проблему решает жесткий лимит. Если же в системе реализована модель лишь жесткого лимита, то возможны ситуации, когда пользователь получает отказ от системы, поскольку он «неумышленно» превзошел указанную квоту (например, из-за ошибки в программе был сформирован очень большой файл).

4.1.10 Надежность файловой системы

Понятие надежности файловой системы включает в себя множество требований, среди которых, в первую очередь, можно выделить то, что системные данные файловой системы должны обладать избыточной информацией, которая позволяла бы в случае аварийной ситуации минимизировать ущерб (т.е. минимизировать потерю информации) от этих сбоев.

Минимизация потери информации при аварийных ситуациях может достигаться за счет использования различных систем ***архивирования***, или ***резервного копирования***. Архивирование может происходить как автоматически по инициативе некоторого программного робота, так и по запросу пользователя. Но целиком каждый раз копировать всю файловую систему неэффективно и дорого. И тут перед нами встает одна из проблем резервного копирования — минимизировать объем копируемой информации без потери качества. Для решения поставленной задачи предлагается несколько подходов. Во-первых, это ***избирательное копирование***, когда намеренно не копируются файлы, которые заведомо восстанавливаются. К таким файлам могут быть отнесены исполняемые файлы ОС, систем программирования, прикладных систем, поскольку считается, что в наличии есть дистрибутивные носители, с которых можно восстановить эти файлы (но файлы с данными копировать, конечно же, придется). Также можно не копировать исполняемые файлы, если для них имеется в наличии дистрибутив или исходных код, который можно откомпилировать и получить данный исполняемый файл. Также можно не копировать файлы пользователей определенных категорий (например, файлы студентов в машинном зале, которые имеют небольшие объемы, - их можно достаточно легко восстановить, переписав заново, но количество этих файлов огромно, что повлечет огромные накладные расходы при архивировании).

Следующая модель заключается в т.н. ***инкрементном архивировании***. Эта модель предполагает создание в первое архивирование полной копии всех файлов — это т.н. ***мастер-копия*** (***master-copy***). Каждая следующая копия будет включать в себя только те файлы, которые изменились или были созданы с момента предыдущего архивирования.

Также при архивировании могут использоваться дополнительные приемы, в частности, ***компрессия***. Но тут встает дилемма: с одной стороны сжатие данных при архивировании дает выигрыш в объеме резервной копии, с другой стороны компрессия крайне чувствительна к потере информации. Потеря или приобретение лишнего бита в сжатом архиве может повлечь за собой порчу всего архива.

Еще одна проблема, которая может возникнуть при резервном копировании, — это ***копирование на ходу***, когда во время резервного копирования какого-то файла пользователь начинает с ним работать (модифицировать, удалять и т.п.). Если для примера рассмотреть инкрементное архивирование, то мастер-копию стоит создать в полном отсутствии пользователей в системе (этот процесс зачастую занимает довольно продолжительное время). Но последующие копии вряд ли удастся создавать в отсутствии пользователей, поэтому необходимо грамотно выбирать моменты для архивирования: понятно, что если большая часть пользователей работает в дневное время суток, то подобные операции стоит проводить в ночные часы, когда в системе почти никто не работает.

Еще один полезный прием заключается в ***распределенном хранении резервных копий***. Всегда желательно иметь две копии, причем храниться они должны в совершенно разных местах, чтобы не могла возникнуть ситуация, когда пожар в офисе уничтожает компьютеры и все резервные копии, хранящиеся в этом офисе, иначе польза от резервного копирования может оказываться нулевой.

Среди **стратегий копирования** можно выделить ***физическое*** и ***логическое*** копирование. ***Физическое копирование*** заключается в поблочном копировании данных с носителя («один в один»). Понятно, что такой способ копирования неэффективен, поскольку копируются и свободные блоки. Следующей модификацией этого способа стало ***интеллектуальное*** физическое копирования лишь занятых блоков. Так или иначе, но данная стратегия имеет проблему обработки дефектных блоков: сталкиваясь при копировании с физически дефектным блоком, невозможно связать данный блок с конкретным файлом. Альтернативой физическому копированию является ***логическая архивация***. Эта стратегия подразумевает копирование не блоков, а файлов (например, файлов, модифицированных после заданной даты).

4.1.11 Проверка целостности файловой системы

Далее речь пойдет о моделях организации контроля и исправления ошибочных ситуаций, связанных с целостностью файловой системы. Обратим внимание, что будет рассматриваться целостность именно файловой системы, а не файлов. Если произошел сбой (например, сломался центральный процессор или оперативная память), то гарантированно потери будут, и эти потери будут двух типов. Во-первых, это потеря актуального содержимого одного или нескольких открытых файлов. Это проблема, но при соответствующей организации резервного копирования она разрешается. Вторая проблема связана с тем, что во время сбоя может нарушиться корректность системной информации. Вторая проблема более существенна и требует более тонких механизмов ее решения.

Для выявления непротиворечивости и исправления возможных ошибочных ситуаций файловая система использует избыточную информацию, т.е. данные тем или иным образом (явно или косвенно) дублируются. Далее рассмотрим организацию контроля целостности блоков файловой системы.

Рассмотрим модельный пример. В системе формируются две таблицы, каждая из которых имеет размеры, соответствующие реальному количеству блоков файловой системы. Одна из таблиц называется ***таблицей занятых блоков***, вторая — ***таблицей свободных блоков***. Изначально содержимое таблиц обнуляется.

На втором шаге система запускает процесс анализа блоков на предмет их незанятости. Для каждого свободного блока увеличивается на 1 соответствующая ему запись в таблице свободных блоков.

На следующем шаге запускается аналогичный процесс, но уже для анализа индексных узлов. Для каждого блока, номер которого встретился в индексном дескрипторе, увеличивается на 1 соответствующая ему запись в таблице занятых блоков.

На последнем шаге запускается процесс анализа содержимого этих таблиц и коррекции ошибочных ситуаций.

Рассмотрим, какие ситуации могут возникнуть, и посмотрим, как файловая система поступает в том или ином случае. Допустим, что рассматриваемая файловая система состоит из шести блоков.

Если при анализе таблиц для каждого номера блока сумма содержимого ячеек с данным номером дает 1, то считается, что система не выявила противоречий (Рис. 114).

0	1	2	3	4	5	
1	1	0	1	0	1	Таблица занятых блоков
0	0	1	0	1	0	Таблица свободных блоков

Рис. 114. Проверка целостности файловой системы. Непротиворечивость файловой системы соблюдена.

Если же находится блок, о котором нет информации ни в таблице свободных, ни в таблице занятых блоков (т.е. в соответствующих ячейках стоят нули), то считается, что этот блок потерян из списка свободных блоков (Рис. 115). Данная ситуация не катастрофическая, соответственно, не требует оперативного разрешения (т.е. может быть отложенной): информацию о данном блоке система может внести в таблицу свободных блоков спустя некоторое время.

0	1	2	3	4	5	
1	0	1	0	1	1	Таблица занятых блоков
0	0	0	1	0	0	Таблица свободных блоков

Рис. 115. Проверка целостности файловой системы. Зафиксирована пропажа блока.

Если в ходе анализа блок получается свободным, но индекс свободности его больше 1 (т.е. соответствующая ячейка таблицы свободных блоков хранит значение, большее 1), то считается, что нарушен список свободных блоков, и начинается процесс пересоздания таблицы свободных блоков (Рис. 116).

0	1	2	3	4	5	
1	0	0	0	1	1	Таблица занятых блоков
0	1	2	1	0	0	Таблица свободных блоков

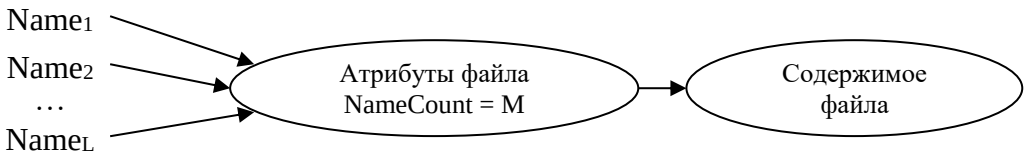
Рис. 116. Проверка целостности файловой системы. Зафиксировано дублирование свободного блока.

Если же возникает аналогичная ситуация, но уже для таблицы занятых блоков, то это означает, что данным файлом владеют несколько файлов, что является ошибкой (Рис. 117). Автоматически определить, какой из файлов ошибочно хранит ссылку на этот блок, не представляется возможным: необходимо анализировать содержимое этих файлов. Для разрешения данной проблемы файловая система может предпринять следующие действия. Пусть конфликтуют файлы с именами $Name_1$ и $Name_2$. Тогда файловая система сначала создает копии этих файлов (соответственно, с именами $Name_1^2$ и $Name_2^2$), затем удаляет файлы с исходными именами $Name_1$ и $Name_2$, запускает процесс переопределения списка свободных блоков и, наконец, обратно переименовывает эти копии с фиксацией факта их возможной некорректности.

0	1	2	3	4	5	
1	2	0	0	1	1	Таблица занятых блоков
0	0	1	1	0	0	Таблица свободных блоков

Рис. 117. Проверка целостности файловой системы. Зафиксировано дублирование занятого блока.

И, наконец, для проверки корректности файловой системы может выполняться проверка соответствия числа реального количества жестких связей тому значению, которое хранится среди атрибутов файла (Рис. 118). Если эти значения совпадают, то считается, что данный файл находится в корректном состоянии, иначе происходит коррекция атрибута-счетчика жестких связей.



if (NameCount != L) { NameCount = L; }

Рис. 118. Проверка целостности файловой системы. Контроль жестких связей.

4.2 Примеры реализаций файловых систем

В качестве примеров реализаций файловых систем рассмотрим файловые системы, реализованные в ОС Unix. Выбор наш объясняется тем, что создатели ОС Unix изначально выбрали удачную архитектуру файловой системы, причем эту файловую систему нельзя рассматривать в отрыве от самой ОС: ОС Unix строится на понятии файла как на одном из фундаментальных понятий (напомним, что вторым важным понятием является понятие процесса). Необходимо заметить, что, как утверждают авторы системы, архитектура файловой системы была заимствована и развита из ОС Multix (файловую систему, которую скорее можно назвать экспериментальной, и, как следствие, она не была массово распространена).

В качестве одного из главных достоинств ОС Unix является то, что именно эта система стала одной из первых распространенных систем, в которой была реализована иерархическая файловая система. Сама же операционная система строится на понятиях процесса и файла, т.е. все то, с чем мы работаем, является файлом, а это, в свою очередь, означает, что в системе реализованы унифицированные интерфейсы доступа и организации информации.

Еще одно важное достоинство, которое необходимо отметить у ОС Unix, — это то, что она стала одной из первых операционных систем, открытых для расширения набора команд системы. До ее создания практически все команды, которые были доступны пользователю, представлялись в виде набора жестких правил общения человека с системой (сравнимые с современными интерпретаторами команд); модифицировать этот набор правил не представлялось возможным. Если же требовалось внести коррективы в существующие команды, то необходимо было обратиться к разработчику операционной системы, и тот, по сути, создавал новую систему. В ОС Unix все исполняемые команды принадлежат одной из двух групп: команды, встроенные в интерпретатор команд (например, pwd, cd и т.д.), и команды, реализация которых представляется в виде исполняемых файлов, расположенных в одном из каталогов файловой системы (это либо

исполняемый бинарный файл, либо текстовый файл с командами для исполнения интерпретатором команд). Данный подход означает, что, варьируя правами доступа к файлам, уничтожая при необходимости или добавляя новые исполняемые файлы, пользователь способен самостоятельно выстраивать функциональное окружение, необходимое для решения своих задач.

Еще одним достоинством этой операционной системы является элегантная организация идентификации доступа и прав доступа к файлам (об этом речь пойдет ниже). Так или иначе, но обозначенные фундаментальные концепции лежат в основе современных операционных систем семейства Unix до сих пор.

4.2.1 Организация файловой системы ОС Unix. Виды файлов. Права доступа

Файл ОС Unix — это специальным образом именованный набор данных, размещенных в файловой системе. Файлы ОС Unix могут быть разных типов:

- ***обычный файл*** (***regular file***) — это те файлы, с которыми регулярно имеет дело пользователь в своей повседневной работе (например, текстовый файл, исполняемый файл и т.п.);
- ***каталог*** (***directory***) — файл данного типа содержит имена и ссылки на атрибуты, которые содержатся в данном каталоге;
- ***специальный файл устройств*** (***special device file***) — как отмечалось выше, каждому устройству, с которым работает ОС Unix, должен быть поставлен в соответствие файл данного типа. Через имя файла устройства происходит обращение к устройству, а через содержимое данного файла (которое достаточно специфично) можно обратиться к конкретному драйверу этого устройства;
- ***именованный канал***, или ***FIFO-файл*** (***named pipe***, ***FIFO file***) — о файлах этого типа шла речь выше при обсуждении базовых средств организации взаимодействия процессов в ОС Unix (см. 0);
- ***файл-ссылка***, или ***символическая связь*** (***link***) — файлы данного типа рассматривались выше при изучении вопросов соответствия имени файла и его содержимого (см. 4.1.7);
- ***сокет*** (***socket***) — файлы данного типа используются для реализации унифицированного интерфейса программирования распределенных систем (см. 3.3).

Так или иначе, но с файлом каждого из указанных типов возможно осуществлять работу (в той или иной степени) посредством стандартных интерфейсов работы с файлами. С каждым файлом также ассоциированы такие характеристики, как **права доступа к файлу**, которые регламентируют чтение содержимого файла, запись и исполнение файла. Подобная интерпретация прав доступа свойственна регулярным файлам, для других типов файлов интерпретация прав доступа отличается (например, для файлов-каталогов).

Права на доступ к файлу разделяются на три категории пользователей — это права пользователя-владельца файла, права группы, к которой принадлежит владелец файла, исключая этого владельца, и, наконец, права всех остальных пользователей системы (без указанной группы владельца). Соответственно, для каждой из категорий определяются вышеперечисленные права доступа.

При работе с каталогом его владельцу, члену соответствующей группы, и всем остальным пользователям может разрешается чтение, запись и выполнение. Однако, эти разрешения интерпретируются не так, как для обычных файлов. Разрешение на чтение из каталога означает, что разрешено открытие каталога и чтение из него. Разрешение на запись предоставляет возможность создавать и уничтожать файлы. Разрешение на выполнение свидетельствует о том, что система может выполнять поиск в каталоге с целью обнаружения какого-либо файла. Если вместо простого имени используется составное, то в каждом из указанных в нем каталогов выполняется поиск имени файла, стоящего следующим в составном имени.

4.2.2 Логическая структура каталогов

Одной из характеристик ОС Unix является характеристика, кажущаяся на первый взгляд достаточно странной: система рекомендует размещать системную и пользовательскую информацию по некоторым правилам. Вообще говоря, эти правила нежесткие, их можно нарушать, но обычно, следуя им, пользователь системы получает дополнительное удобство.

Прежде всего, необходимо отметить, что файловая система ОС Unix является иерархической древовидной файловой системой (Рис. 119), т.е. у нее есть корневой каталог /, из которого за счет каталогов разных уровней вложенности «вырастает» целое дерево **имен** файлов.

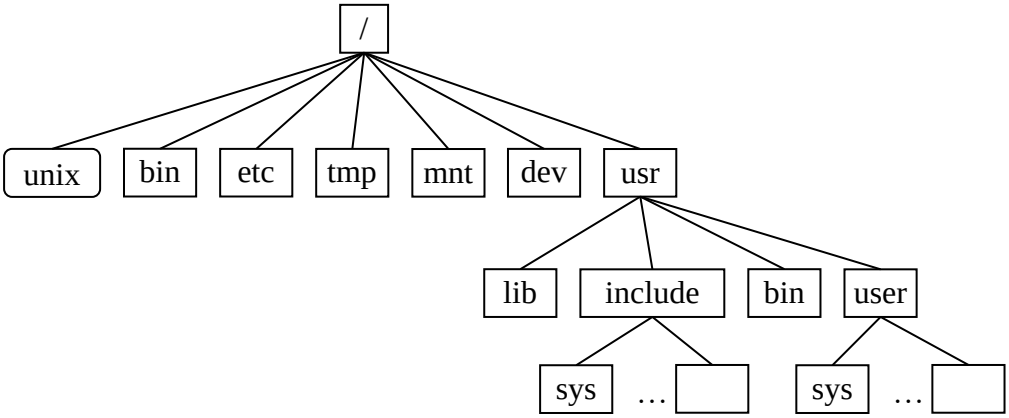


Рис. 119. Логическая структура каталогов.

Система предполагает, что в корневом каталоге всегда расположен некоторый файл, в котором размещается код ядра операционной системы. Сразу оговоримся, что мы рассматриваем некоторую модельную систему: в действительности файл с кодом ядра и упоминаемые в будущем каталоги могут иметь иное расположение в системе и другие имена. Вообще говоря, в корневом каталоге можно размещать любые файлы (с учетом прав доступа), но система предполагает наличие совокупности каталогов с предопределенными именами. Рассмотрим основные каталоги системы.

В каталоге **/bin** находятся команды общего пользования (точнее говоря, исполняемые файлы, реализующие указанные команды).

Каталог **/etc** содержит системные таблицы и команды, обеспечивающие работу с этими таблицами. В частности, в этом каталоге хранится таблица (файл) passwd, содержащую информацию о зарегистрированных в системе пользователях.

Каталог **/tmp** является каталогом временных файлов, т.е. в этом каталоге система и пользователи могут размещать свои файлы на некоторый ограниченный промежуток времени; при этом при перезагрузке системы нет гарантии, что файлы не будут удалены из этого каталога.

Каталог **/mnt** традиционно используют для монтирования различных файловых систем к данной системе. Операция монтирования в общих чертах заключается в том, что корень монтируемой файловой системы ассоциируют с данным каталогом (или с одним из его подкаталогов), после чего доступ к файлам подмонтированной системы осуществляется уже через этот каталог (т.н. ***точку монтирования***).

В каталоге **/dev** размещаются специальные файлы устройств, посредством которых осуществляется регистрация обслуживаемых в системе устройств и связь этих устройств с тем или иным драйвером. Соответственно, все устройства, с которыми работает операционная система, именуются посредством имен этих специальных файлов устройств.

Каталог **/usr** можно охарактеризовать как каталог пользовательской информации. Предполагается, что это каталог имеет свою специфичную структуру подкаталогов. В частности, каталог **/usr/lib** обычно содержит инструменты работы пользователей, не относящихся напрямую к взаимодействию с операционной системой (например, тут могут храниться системы программирования, С-компилятор, С-отладчик и т.п.). Еще одним достаточно важным каталогом является каталог **/usr/include**, который содержит файлы заголовков (или include-файлы) с расширением *.h, и именно в этом каталоге будет искать препроцессор С-компилятора соответствующие файлы заголовков, указанные в программе в угловых скобках. Каталог **/usr/bin** — это каталог команд, которые введены на данной вычислительной установке (например, тут могут храниться команды, связанные с непосредственной деятельностью организации). И, наконец, в каталоге **/usr/user** размещаются домашние каталоги зарегистрированных в системе пользователей.

Итак, мы рассмотрели основные аспекты логической структуры каталогов ОС Unix. Еще раз отметим, что, придерживаясь рекомендаций системы в плане размещения тех или иных файлов, легче и удобнее поддерживать систему «в порядке».

4.2.3 Внутренняя организация файловой системы: модель версии System V

Рассмотрение внутренней организации файловой системы мы начнем с модели файловой системы, реализованной в ОС Unix версии System V. Данная файловая система была реализована одной из первых в ОС Unix и имеет название **s5fs**.

Суперблок	Область индексных дескрипторов	Блоки файлов

Рис. 120. Структура файловой системы версии System V.

Данная файловая система имеет следующую структуру (Рис. 120). Файловая система занимает часть того раздела, в котором она находится (назовем его ***системным разделом***, чтобы отличать его от разделов с другими файловыми системами, которые можно примонтировать к данной системе), начиная с нулевого блока и заканчивая некоторым фиксированным блоком. Эта часть состоит из трех подпространств: суперблока, области индексных дескрипторов и блоков файлов.

Итак, первое подпространство — это ***суперблок***. Суперблок содержит следующую информацию:

- Тип файловой системы
- Размер файловой системы в логических блоках, включая сам суперблок, область ИД и блоки хранения данных
- Размер области индексных дескрипторов
- Число свободных блоков, доступных для размещения
- Число свободных ИД, доступных для размещения
- Флаги (модификации, режима монтирования…)
- Размер логического блока (512, 1024, 2048…)
- Список номеров свободных ИД
- Список адресов свободных блоков

Он содержит данные, определяющие статические параметры и характеристики данной файловой системы (например, информация о размере блока файла, информация о размере всей файловой системы в блоках или байтах или же информация о количестве

индексных дескрипторов в системе). Также суперблок хранит информацию об оперативном состоянии файловой системы. Суперблок является частью файловой системы, которая резидентно находится в оперативной памяти. Среди прочего суперблок хранит информацию о наличии свободных ресурсов файловой системы — наличии свободных блоков в рабочем пространстве файловой системы и наличие свободных индексных дескрипторов. Забегая вперед, отметим, что для этих целей используются соответственно массив номеров свободных блоков и массив индексных дескрипторов.

Следующее подпространство — это ***область индексных дескрипторов***. Индексные дескрипторы были описаны нами выше, мы их рассматривали как некоторые системные структуры данных фиксированного размера, содержащих комплексную информацию о размещении, актуальном состоянии и содержимом конкретного файла.

Последнее подпространство — это ***блоки файлов*** (если быть более точным, то данное пространство корректнее было бы назвать ***рабочим пространством*** файловой системы). Здесь размещаются блоки файлов (с содержимым этих файлов), а также системная информация, которая не поместилась в суперблоке и области индексных дескрипторов.

4.2.3.1 Работа с массивами номеров свободных блоков

Изначально номера всех свободных блоков файловой системы выстраиваются в единый связный список (Рис. 121). Первый элемент этого списка размещается в суперблоке и называется массивом номеров свободных блоков. Все последующие элементы списка располагаются в блоках области блоков файловой системы. Каждый элемент списка хранит номера свободных блоков, а также номер следующего элемента списка (номер блока файловой системы).

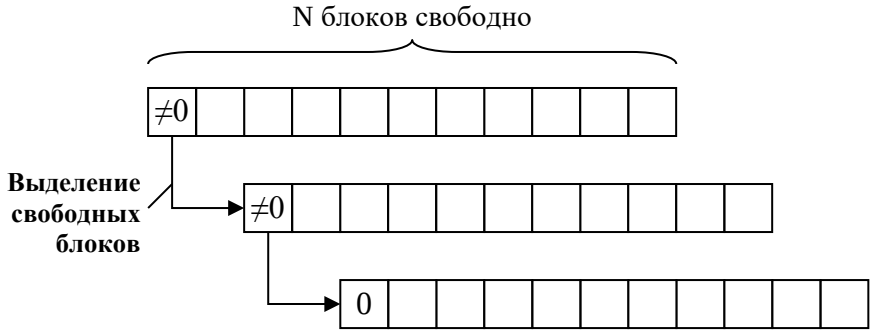


Рис. 121. Работа с массивами номеров свободных блоков.

Работа с массивом номеров свободных блоков достаточно проста. При запросе файловой системы на получение свободного блока происходит поиск в массиве номеров свободных блоков ячейки с содержательной (ненулевой) информацией, обнуление найденной ячейки, а номер найденного блока выдается в ответ на запрос. Если при запросе в массиве не найдено ячейки с ненулевым содержимым, то содержимое следующего элемента списка загружается в массив номеров свободных блоков, а номер блока, в котором размещался освободившийся элемент, возвращается файловой системе как ответ на запрос. Если же какой-то блок освобождается, то выполняются противоположные действия.

На первый взгляд может показаться, что хранение в блоках массива свободных блоков уменьшает рабочее пространство файловой системы (т.е. пользователь не сможет воспользоваться блоками, хранящими массив), но это не так: если представить граничную ситуацию, когда нет свободных блоков, - тогда нет и номеров свободных блоков, а значит, нет и блоков, хранящих эти номера, т.е. файловая система занята на 100%.

4.2.3.2 Работа с массивом свободных индексных дескрипторов

Массив номеров свободных индексных дескрипторов состоит из фиксированного количества элементов. Изначально данный массив заполнен номерами свободных индексных дескрипторов.

Если происходит освобождение индексного дескриптора (т.е. происходит удаление файла), то происходит обращение к данному массиву. Если в массиве есть свободные места, то происходит запись номера освободившегося индексного дескриптора в первое встретившееся свободное место массива, иначе номер дескриптора «забывается».

При создании файла происходят обратные действия. Идет обращение к массиву; если он не пуст, то из него изымается первый содержательный элемент, который представляет собой номер свободного индексного дескриптора. Если же при обращении к массиву оказалось, что он пуст, а в суперблоке присутствует информация о наличии свободных индексных дескрипторов, то система запускает процесс обновления рассматриваемого массива. Этот процесс обращается к области индексных дескрипторов, последовательно перебирает их и в зависимости от их содержимого делает однозначный вывод о занятости или свободности дескриптора. Номера свободных индексных дескрипторов процесс помещает в массив.

Рассмотренные массивы свободных блоков и свободных индексных дескрипторов исполняют роль специализированных КЭШей: происходит буферизация обращений к системе за свободным ресурсом.

4.2.3.3 Индексные дескрипторы. Адресация блоков файла

Выше уже отмечалось, что ***индексный дескриптор*** (Рис. 122) является системной структурой данных, содержащей атрибуты файла, а также всю оперативную информацию об организации и размещении данных. Система устроена таким образом, что между содержимым файла и его индексным дескриптором существует **взаимнооднозначное соответствие**. Заметим, что содержимое файла не обязательно размещается в рабочем пространстве файловой системы: существуют некоторые типы файлов, для которых содержимое хранится в самом индексном дескрипторе. Примером тут может послужить тип специального файла устройств.

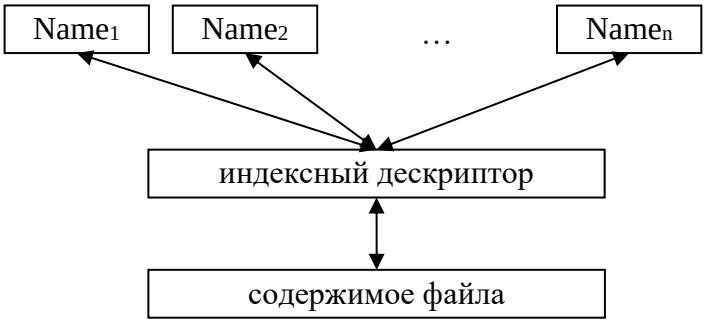


Рис. 122. Индексные дескрипторы.

Для каждого индексного дескриптора существует, по меньшей мере, одно имя, зарегистрированное в каталогах файловой системы. И еще раз повторимся: говоря о древовидности файловой системы, понимают древовидность не с точки зрения размещения файла, а с точки зрения размещения имен файлов. Следует отметить, что для типа файлов – каталог, возможно наличие только одного имени.

Индексный дескриптор хранит информацию о типе файла, правах доступа, информацию о владельце файла, размере файла в байтах, количестве имен, зарегистрированных в каталогах файловой системы и ссылающихся на данный индексный

дескриптор. В частности, признаком свободного индексного дескриптора является нулевое значение последнего из указанных атрибутов.

В индексном дескрипторе также собирается различная статистическая информация о времени создания, времени последней модификации, времени последнего доступа. И, наконец, в индексном дескрипторе находится массив номеров блоков файла.

Организация блоков файла — еще одна удачная особенность файловой системы ОС Unix System V. Структура организации блоков файла выглядит следующим образом. Массив номеров блоков файла состоит из 13 элементов. Первые 10 элементов используются для указания номеров первых десяти блоков файла, оставшиеся три элемента используются для организации косвенной адресации блоков. Так, одиннадцатый элемент ссылается на массив из N номеров блоков файла, двенадцатый — на массив из N ссылок, каждая из которых ссылается на массив из N номеров блоков файла, тринадцатый элемент используется уже для трехуровневой косвенной адресации блоков. В современных файловых системах Unix массив номеров блоков файла состоит из 15 элементов (12 – прямая адресация, 3 - косвенная).

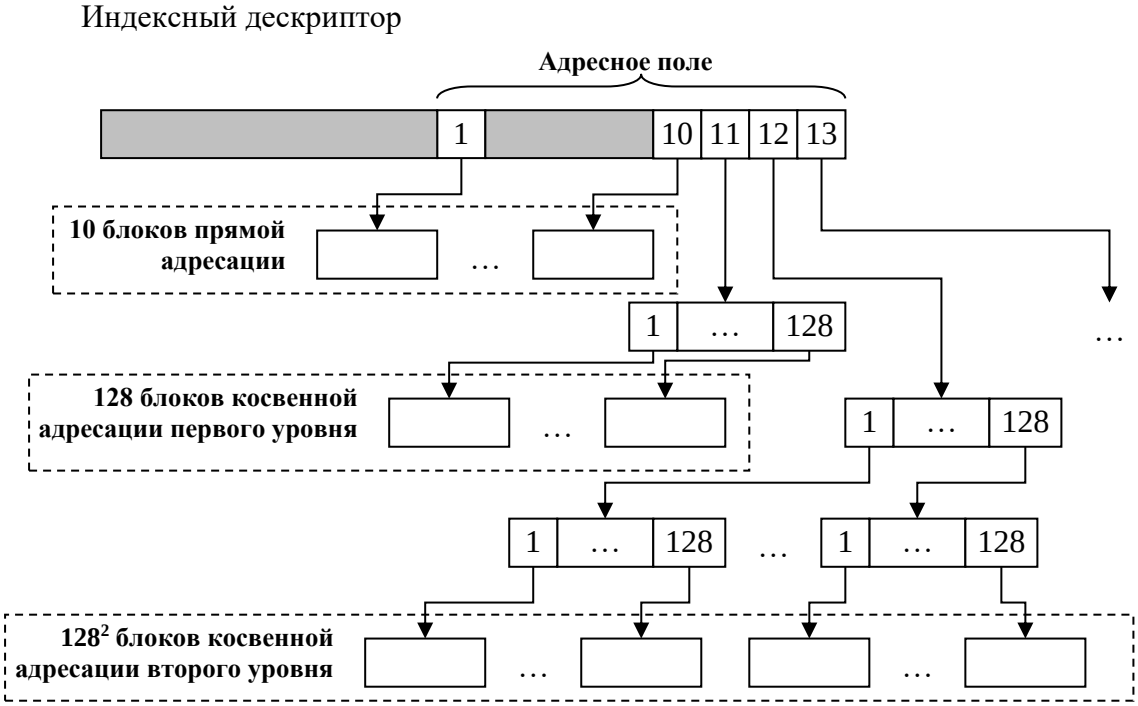


Рис. 123. Адресация блоков файла.

Рассмотрим **пример** системы (Рис. 123), в которой размер блока равен 512 байтам (т.е. 128 четырехбайтовым числам). Если количество блоков файла больше 10, то сначала используется косвенная адресация первого уровня. Суть ее заключается в том, что в одиннадцатом элементе хранится номер блока, состоящий в нашем случае из 128 номеров блоков файла, которые следуют за первыми десятью блоками. Иными словами, посредством одиннадцатого элемента массива адресуются 11-ый – 138-ой блоки файла. Если же блоков оказывается больше, чем 138, то начинает использоваться косвенность второго уровня, и для этих целей задействуют двенадцатый элемент массива. Этот элемент массива содержит номер блока, в котором (опять-таки для нашего примера) могут находиться до 128 номеров блоков, в каждом из которых может находиться до 128 номеров блоков файла. Когда размеры файла оказываются настолько большими, что для хранения номеров его блоков не хватает двойной косвенной адресации, используется тринадцатый элемент массива и косвенная адресация третьего уровня. Итак, в рассмотренной модели (в которой размер блока равен 512 байтам) максимальный размер файла может достигать

$(10+128+128^2+128^3)*512$ байт. На сегодняшний день файловые системы с таким размером блока не используются, наиболее типичны размеры блока 4, 8, и т.д. вплоть до 64 кбайт.

Обратим внимание, что рассмотренная модель адресации блоков файла является достаточно компактной и эффективной, поскольку для обращения к блоку файла с использованием тройной косвенности потребуется всего три обмена, а если учесть, что в системе реализована буферизация блочных обменов, то накладные расходы становятся еще меньше.

4.2.3.4 Файл-каталог

Каталог файловой системы версии System V — это файл специального типа; его содержимое так же, как и у регулярных файлов, находится в рабочем пространстве файловой системы и по организации данных ничем не отличается от организации данных регулярных файлов.

Файлы-каталоги (Рис. 124) имеют следующую структурную организацию. Каждая запись в нем имеет фиксированный размер: длина записи довольно сильно варьируется (будем считать, что длина записи 16 байт). Первые два байта хранят номер индексного дескриптора файла, а оставшиеся 14 байтов —имя файла (т.е. в нашей модели имя файла в системе ограничено 14 символами). При создании каталога он получает две предопределенные записи, которые невозможно модифицировать и удалять. Первая запись — это запись, для которой используется унифицированное имя “**.**”, интерпретируемая как ссылка на сам этот каталог. Соответственно, в этой записи указывается номер индексного дескриптора данного файла-каталога. Второй записью, для которой используется унифицированное имя “**..**”, является ссылка на родительский для данного файла каталог, и соответственно, в этой записи хранится номер индексного дескриптора родительского каталога.

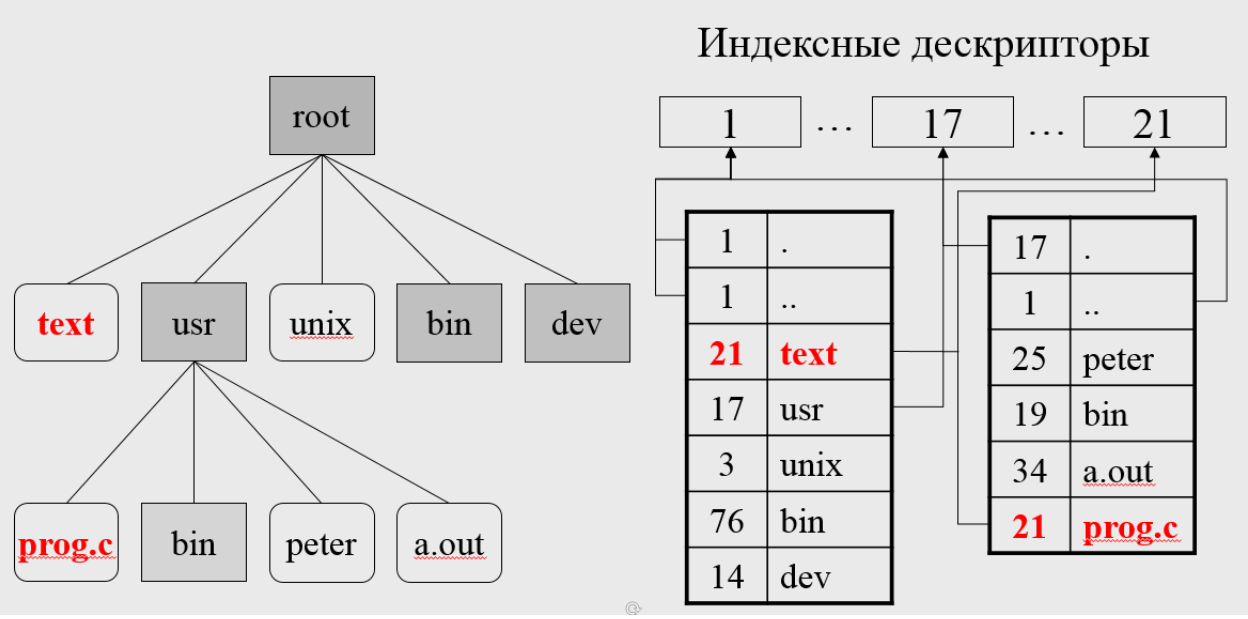


Рис. 124. Файл-каталог.

Отвлекаясь от файловой системы версии System V, отметим, что многие более развитые файловые системы ОС Unix поддерживают средства **установления связей** (Рис. 125) между индексным дескриптором и именами файла. Можно устанавливать как жесткие связи, так и символические связи. Жесткая связь позволяет с одним индексным дескриптором связать два и более равноправных имени. Соответственно, при удалении имени, участвующего в жесткой связи, сначала удаляется имя из каталога, затем

уменьшается счетчик жестких связей в индексном дескрипторе. В случае обнуления этого счетчика происходит удаление содержимого файла и освобождение данного индексного дескриптора.

Для организации символической связи создается файл специального типа — типа ссылка. Файл данного типа содержит полный путь к тому файлу, на который ссылается данный файл-ссылка. Используя такую косвенную адресацию, можно добраться до целевого файла. Такой подход иллюстрирует ассиметричное именование.

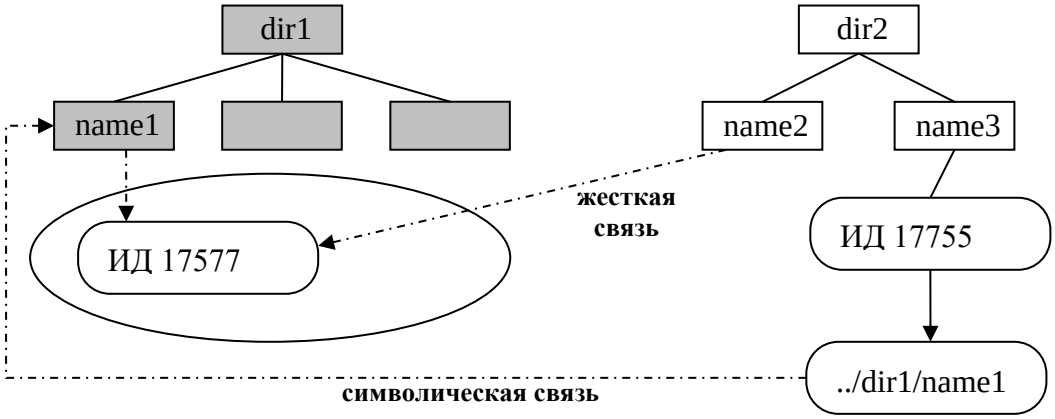


Рис. 125. Установление связей.

4.2.3.5 Достоинства и недостатки файловой системы модели System V

Среди **достоинств** рассматриваемой файловой системы стоит отметить, что данная система является иерархичной. Также надо отметить, что за счет использования системного кэширования оптимизирована работа с массивом свободных блоков и свободных индексных дескрипторов. И, наконец, в данной файловой системе найдено удачное решение организации блоков файлов за счет использования «нарастающей» косвенности адресации.

С другой стороны, данная система не лишена **недостатков**, большая часть которых является следствием ее достоинств. Первым недостатком является тот факт, что в суперблоке концентрируется ключевая информация файловой системы. Соответственно, потеря суперблока приводит к достаточно серьезным проблемам.

Следующая проблема опять-таки связана с концентрацией информации в суперблоке. Несмотря на то, что суперблок резидентно размещается в ОП, система периодически «сбрасывает» его копию на диск — это делается для того, чтобы при сбое минимизировать потери актуальной информации из суперблока. Это, в свою очередь, означает, что система регулярно обращается к одной и той же точке дискового пространства, и, соответственно, вероятность выхода из строя именно данной области диска со временем сильно увеличивается.

Следующий недостаток связан с фрагментацией блоков файла по диску. Здесь имеется в виду, что при интенсивной работе файловой системы (когда в ней со временем создается, модифицируется и уничтожается достаточно большое число файлов) складывается ситуация, когда блоки одного файла оказываются разбросанными по всему доступному дисковому пространству. В этом случае, если потребуется прочитать последовательные блоки файла (что бывает достаточно часто), то головка жесткого диска начинает совершать довольно много механических передвижений, что отрицательно сказывается на эффективности работы файловой системы.

И в заключение отметим такой недостаток как ограничение, накладываемое на длину имени файла (6, 8, 14 байт для представления имени — величины на сегодняшний день достаточно небольшие).

4.2.4 Внутренняя организация файловой системы: модель версии Fast File System (FFS) BSD

Разработчики файловой системы ***Fast File System*** (***FFS***), оставив основные положительные характеристики предыдущих файловых систем (в т.ч. и файловой системы версии System V), пошли по следующему пути (Рис. 126). Они представили раздел как последовательность дисковых цилиндров, которую разбили на порции фиксированного размера. В каждом из образовавшихся кластеров размещается копия суперблока, блоки файлов (которые мы назвали рабочим пространством файловой системы), информация об индексных дескрипторах, ассоциированных с данным кластером, а также информация о свободных ресурсах этого кластера. При этом разбиение устройства на кластеры происходит аппаратно-зависимо таким образом, чтобы суперблоки не оказывались на «опасно близком» расстоянии (например, на одной поверхности). Такой подход обеспечивает большую надежность файловой системы.

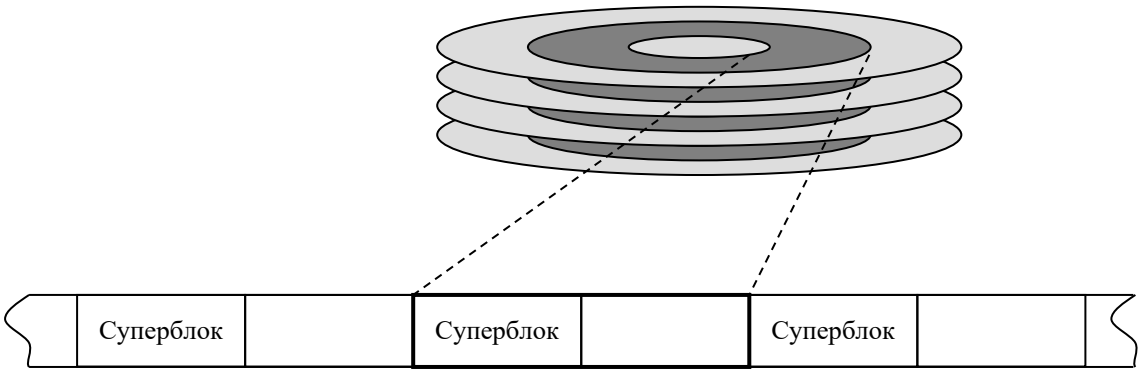


Рис. 126. Структура файловой системы версии FFS BSD.

4.2.4.1 Стратегии размещения

Работа системы основывается на трех концепциях. Первой концепцией является **оптимизация размещения каталога**. При создании каталога система осуществляет поиск кластера, наиболее свободного в данный момент с точки зрения использования индексных дескрипторов, т.е. ищутся кластеры, количество свободных индексных дескрипторов в которых превосходит некоторую среднюю величину, и среди найденных кластеров выбирается кластер с наименьшим количеством каталогов.

Следующей стратегией является **равномерность использования блоков данных**. Во время создания файла он делится на несколько частей. Часть файла, которая имела непосредственную адресацию из индексного дескриптора, по возможности размещается в том же кластере, что и индексный дескриптор. Оставшиеся части файла делятся на равные порции, которые файловая система размещает в отдельных кластерах. Перечисленные стратегии призваны для борьбы с фрагментацией файла по разделу: файл либо целиком размещается в одном кластере, либо размещается в нескольких кластерах, но тогда в них размещаются достаточно большие фрагменты подряд идущих блоков.

И, наконец, третья стратегия размещения — технологическое **размещение последовательных блоков файлов** (Рис. 127). Представим следующую ситуацию: пусть необходимо прочитать два последовательных блока с магнитного диска (будем считать, что эти блоки находятся на одной дорожке магнитного диска). Это означает, что данная задача требует двух последовательных обращений к системным вызовам. Соответственно, между окончанием физического считывания первого блока и началом физического считывания второго блока потратится некоторое время Δt на накладные расходы (в частности, вход и выход из системного вызова). Это время хоть и мало, но за данный промежуток диск успеет повернуться на угол $\omega * \Delta t$ (где ω — скорость вращения диска). Если следующий второй

блок расположен на диске непосредственно за первым, то за время Δt головка пропустит начало второго блока, и когда будет предпринята попытка физически прочесть второй блок, то придется ожидать полного оборота диска, что является относительно долгим по времени. Чтобы избежать подобных накладных расходов, связанных с необходимостью ожидать полного оборота диска, необходимо расположить второй блок с некоторым отступом от первого. В этом и заключается технологическое размещение блоков на диске.

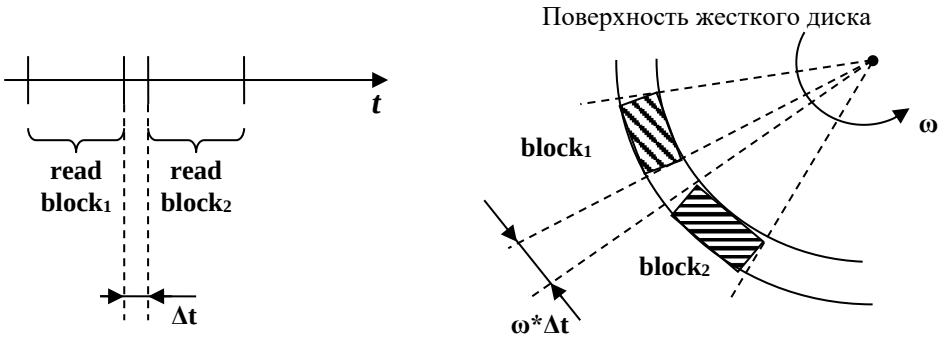


Рис. 127. Стратегия размещения последовательных блоков файлов.

4.2.4.2 Внутренняя организация блоков

Размер блока в файловой системе FFS может варьироваться в достаточно широком диапазоне: предельный размер блока — 64 Кбайт. Как отмечалось выше, проблема выбора оптимального размера блока достаточно сложна: и большие блоки, и маленькие имеют свои достоинства и недостатки, и от администратора системы требуются хорошие навыки, чтобы подобрать оптимальные для данной системы (решающей задачи конкретного типа) размеры блоков файловой системы.

Создатели рассматриваемой файловой системы пошли по пути увеличения размера блока. За счет этого 1) уменьшается фрагментация файла по диску и 2) уменьшаются накладные расходы при чтении подряд идущих данных файла (эффективнее считать за 1 раз большую порцию информации, чем два раза считать по «половинке»). Но главным недостатком крупных блоков — большая степень внутренней фрагментации. Для борьбы с внутренней фрагментацией в системе реализован еще один уровень структурной организации: каждый блок файловой системы поделен на фиксированное количество т.н. ***фрагментов*** (обычно число фрагментов в блоке кратно степени 2 — 2, 4, 8 и т.д.).

Размещение файла по блокам файловой системы строится на основе следующей концепции (Рис. 128). Считаем, что все блоки файла, за исключением последнего, целиком заполнены содержимым данного файла. Соответственно, номера этих блоков хранятся среди атрибутов файла. Последний блок выделен отдельно (кстати, он тоже может быть полностью заполнен): помимо его номера в атрибутах файла хранятся и номера занятых в нем фрагментов, принадлежащих данному файлу. Информация о блоках и фрагментах может быть представлена разными способами: например, двоичная маска, или же номером первого фрагмента в этом блоке, занятым данным файлом (количество фрагментов тогда можно вычислить на основании длины файла в байтах), и т.д.

Блоки	0				1				…	N			
Фрагменты	0	1	2	3	4	5	6	7	…				
Маска	0	0	0	0	0	1	1	1	…				

Рис. 128. Внутренняя организация блоков (блоки выровнены по кратности).

4.2.4.3 Выделение пространства для файла

Рассмотрим алгоритм выделения пространства для файлов на следующем примере. Будем считать, что блок файловой системы поделен на 4 фрагмента. Пусть в системе хранятся файлы petya.txt и vasya.txt (Рис. 129), для которых в соответствующих индексных дескрипторах хранится информация об их размерах и номерах блоков, принадлежащих файлам, в виде стартовых фрагментов. Соответственно, файл petya.txt расположен в нулевом (стартовый фрагмент № 00), первом (стартовый фрагмент № 04) и втором блоке (стартовый фрагмент № 08). Если учесть длину файла (5120 байт), то получается, что во втором блоке этот файл занимает 08 и 09 фрагменты. Файл vasya.txt расположен в третьем блоке (стартовый фрагмент № 12), четвертом (стартовый фрагмент № 16) и втором блоке (стартовый фрагмент № 10); при этом во втором блоке файлу принадлежит только 10-ый фрагмент (т.к. размер файла 4608 байт). Итак, очевидно, что данная система нарушает концепцию файловой системы ветви System V, в которой каждый блок мог принадлежать только одному файлу; в FFS последний блок можно разделять между различными файлами.

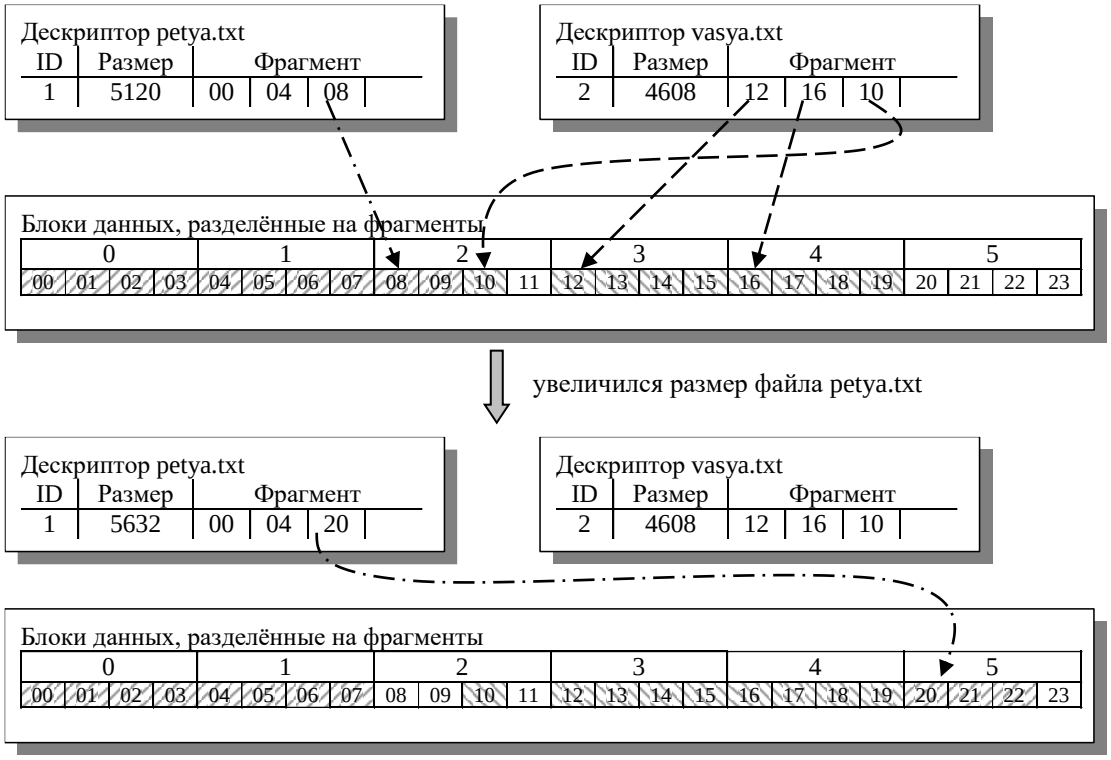


Рис. 129. Выделение пространства для файла.

Если, например, размер файла petya.txt увеличивается настолько, что конец файла не помещается в 08 и 09 фрагментах, то система начинает поиск блока с тремя **подряд**

идущими свободными фрагментами. (Соответственно, если размер файла увеличивается на большую величину, то сначала для него отводятся полностью свободные блоки, в которых файл занимает все фрагменты, а для размещения последних фрагментов ищется блок с соответствующим числом подряд идущих свободных фрагментов.) Когда система находит такой блок, то происходит перемещение последних фрагментов файла petya.txt в этот блок.

4.2.4.4 Структура каталога FFS

Каталог файловой системы FFS позволяет использовать имена файлов, длиной до 255 символов (Рис. 130). Каталог состоит из записей переменной длины. Начальная запись содержит номер индексного дескриптора, размер записи (т.е. ссылку на последний элемент записи) и длину имени файла, после этого следует дополненное до кратности в 4 байта имя файла (максимальная длина имени файла — 255 символов). Работа системы организована следующим образом: если происходит удаление файла из каталога, то освобождающееся пространство, занимаемое раньше записью данного файла, присоединяется к предыдущей записи. Это означает, что размер предыдущей записи увеличивается, но длина хранимого в ней имени не меняется (т.е. остается реальной). Удаление первой записи выражается в обнулении номера индексного дескриптора в этой записи. Такая модель позволяет при удалении файла практически не заботиться о высвобождаемом пространстве внутри файла-каталога: получаемые при удалении «дыры» ликвидируются не за счет компрессии, а за счет тривиального «склеивания» с предыдущей записью.

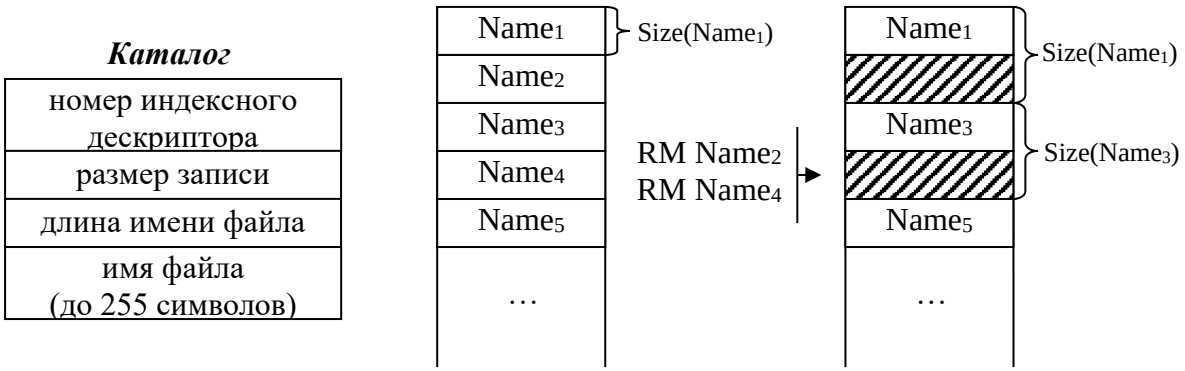


Рис. 130. Структура каталога FFS BSD.

Таким образом, система позволяет использовать имена файлов произвольной длины вплоть до 255 символов, что достаточно удобно для пользователя. Но такое преимущество оборачивается тем, что система несет накладные расходы как по использованию дискового пространства (в каталогах присутствует внутренняя фрагментация), так и по времени (осуществление поиска в системах с фиксированными размерами записей в большинстве случаев эффективнее, чем в системах с переменными размерами записей).

4.2.4.5 Блокировка доступа к содержимому файла

Организация файловой системы ОС Unix позволяет открывать и работать с одним и тем же файлом произвольному числу процессов. Более того, один и тот же файл может быть многократно открыт в рамках одного процесса. При этом система поддерживает модель синхронизации работы с файлами. Для этого используется системный вызов ***fcntl()*** (данный системный вызов предназначен вообще для организации управления работы с файлом), который может обеспечивать блокировку как файла в целом, так и отдельных областей внутри файла (т.е. сделать какую-то область файла недоступной для других процессов). Различают два типа блокировок: ***исключающие*** и ***распределенные***.

Исключающая блокировка (***exclusive lock***) — это «жесткая» блокировка: если произошла такая блокировка области, то любой другой процесс не сможет осуществить операции обмена с данной областью (в этом случае процесс либо будет приостановлен в ожидании разблокирования области, либо получит отказ, в зависимости от установленного режима работы). Данный вид блокировок является блокировкой с монополизацией: области с исключающими блокировками пересекаться не могут.

Альтернативой исключающей блокировке является ***распределенная блокировка*** (***shared lock***), или «мягкая», рекомендательная блокировка. Процесс может установить для области блокировку этого типа, а другие процессы при работе могут на нее не обращать внимания, т.е. при установленной блокировке всЁ равно разрешены чтение и запись информации из блокированной области. Для обеспечения корректной работы с файлом необходимо средство установки блокировки на той или иной области – для этого опять-таки используется системный вызов *fcntl()*. Области с рекомендательными блокировками могут пересекаться.

4.2.5 Организация обмена данными с файлами

В этом разделе мы рассмотрим механизм организации обмена данными с файлами, рассмотрим, что происходит в системе, когда один и тот же файл открывается в системе одновременно несколькими процессами и в каждом из них, возможно, по нескольку раз.

Для организации операций обмена в ОС Unix используются системные таблицы и структуры, часть которых ассоциирована с каждым процессом (т.е. они располагаются в адресном пространстве процесса), а часть — с самой ОС.

Таблица открытых файлов (***ТОФ***) создается в адресном пространстве процесса. Каждая запись этой таблицы соответствует открытому в процессе файлу. Говоря о номере дескриптора открытого в процессе файла (т.н. ***файлового дескриптора***); подразумевается соответствующий номер записи в таблице открытых файлов процесса. Размер данной таблицы определяется при настройке операционной системы: этот параметр декларирует предельное количество открытых в одном процессе файлов.

Каждая запись ТОФ содержит целый набор атрибутов, который в данный момент нам не интересен, но в этом наборе имеется один достаточно важный атрибут — это ссылка на номер записи в ***таблице файлов*** операционной системы (***ТФ***). Таблица файлов ОС является системной таблицей, она представлена в системе в единственном экземпляре. В этой таблице происходит регистрация всех открытых в системе файлов.

В таблице файлов ОС помимо прочего содержатся такие атрибуты, как ***указатель чтения/записи*** (ссылающийся на позицию в файле, начиная с которой будет происходить, соответственно, чтение или запись), ***счетчик кратности*** (речь о нем пойдет ниже) и ***ссылка*** на таблицу индексных дескрипторов открытых файлов.

Таблица индексных дескрипторов открытых файлов (***ТИДОФ***) также является системной структурой данных, содержащей перечень индексных дескрипторов всех файлов, открытых в данный момент в системе. Каждая запись этой таблицы содержит актуальную копию открытого в системе индексного дескриптора. Здесь также хранится целый набор параметров, среди которых имеется и счетчик кратности.

Для иллюстрации рассмотрим следующий **пример** (Рис. 131). Пусть в системе запущен *Процесс*$_1$, для которого система при его создании сформировала ТОФ$_1$. Затем этот процесс, посредством обращения к системному вызову *open(),* открывает файл с именем name. Это означает, что в свободном месте этой таблицы заводится файловый дескриптор для работы с данным файлом. В этой записи ТОФ хранится ссылка на соответствующую запись в ТФ. Если файл открывается впервые в системе, то в ТФ заводится новая запись для работы с этим файлом. В данной записи хранится указатель чтения/записи, а также коэффициент кратности, который в начале устанавливается в значение 1 — это означает, что с данной записью ТФ ассоциирована единственная запись из какой-либо ТОФ. И,

конечно, в данной записи ТФ хранится ссылка на запись в ТИДОФ, содержащую актуальную копию индексного дескриптора обрабатываемого файла. Таблицы ТФ и ТИДОФ хранят оперативную информацию, поэтому они располагаются в ОЗУ. Соответственно, файловая система, работая с блоками открытого файла, оперирует данными, хранимыми именно в ТИДОФ.

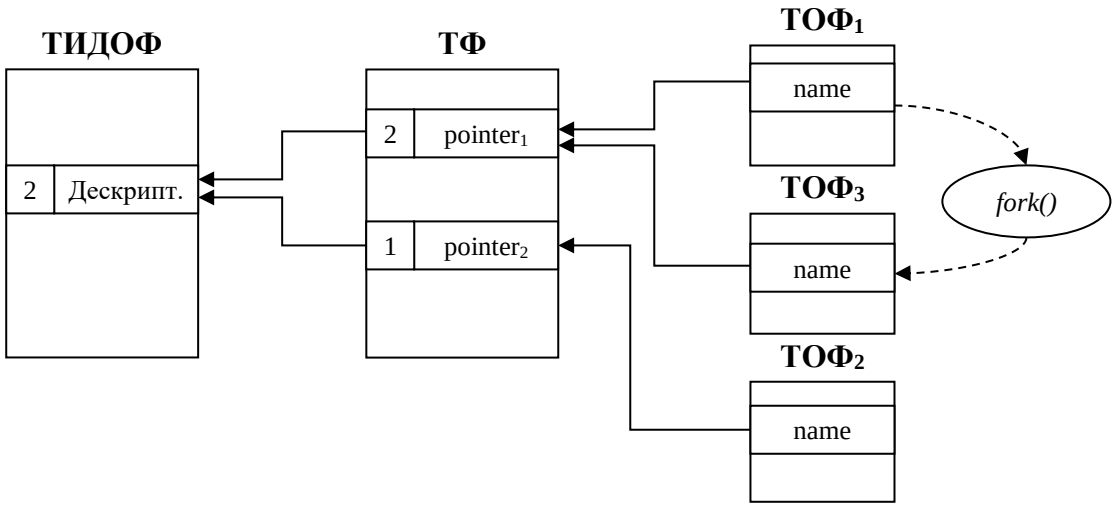


Рис. 131. Организация обмена данными с файлами.

Пусть в системе позже был запущен *Процесс*$_2$, который также открыл файл name. В этом случае в ТФ заводится новая запись, в которой устанавливается свой указатель чтения/записи, но эта запись ТФ будет ссылаться на тот же номер записи в ТИДОФ. Такой механизм позволяет корректно (с системной точки зрения) обрабатывать ситуации одновременной работы с одним и тем же файлом: поскольку в итоге все сводится к единственной актуальной копии индексного дескриптора в ТИДОФ, то работа ведется с соответствующими блоками файла. При этом данные процессы работают с файлом каждый «по-своему», т.к. каждый из них оперирует независимыми указателями чтения/записи, хранимыми в различных записях ТФ.

Теперь предположим, что после открытия файла name, *Процесс*$_1$ обращается к системному вызову *fork()* и порождает своего потомка — *Процесс*$_3$. При обращении к системному вызову *fork()* ТОФ родительского процесса копируется в ТОФ сыновнего процесса. Соответственно, все записи ТОФ$_3$ будут ссылаться на те же записи ТФ, что и записи ТОФ$_1$. Это означает, что при порождении сыновнего процесса в соответствующих записях ТФ происходит увеличение на 1 счетчика кратности. Заметим, что подобный механизм наследования подразумевает, что сыновний процесс будет работать с теми же указателями чтения/записи, что и родительский процесс.

Рассмотренная модель организации обмена данными с файлами имеет свои достоинства и недостатки. Так, ТИДОФ располагается в оперативной памяти. Это означает, что становится эффективнее работа с файловой системой, поскольку уменьшается число обращений к пространству индексных дескрипторов файловой системы, т.е. этот механизм можно считать кэшированием системных обменов. Но эта модель имеет главный недостаток, связанный с некорректным завершением работы операционной системы: если в системе происходит сбой, то содержимое ТИДОФ будет потеряно, а это означает, что будут потери и в файловой системе.

4.2.6 Предопределенные номера индексных дескрипторов

Для корректной и эффективной работы файловой системы номера некоторых индексных дескрипторов зарезервированы, т.е. имеют фиксированное (предопределённое) размещение в области индексных дескрипторов (например, корневой каталог). Некоторые зарезервированные индексные дескрипторы на примере файловой системы Ext4 ОС Linux:
0 - несуществующий ИД
1 - список повреждённых блоков
2 - корневой каталог
3 - квоты пользователей
4 – групповые квоты
…
8 - журнал
…

Незарезервированное под системные нужды номера индексных дескрипторов начинаются со значения 11.

4.2.6.1 Алгоритм поиска ИД файла при его открытии

Рассмотрим упрощенный алгориитм поиска индексного дескриптора файла на примере открытия файла **/dir/filename.** В алгоритме не будем рассматривать действия по проверке прав доступа.

1. Считываем ИД номер 2 (предопределенный номер ИД - корневой каталог «/») из области индексных дескрипторов.
2. Получаем доступ к содержимому корневого файла-каталога (используя массив адресов блоков ИД 2) осуществляем поиск записи в файле корневого каталога, в которой имя совпадает с **dir**, номер индексного дескриптора в этой записи - **ИД_dir**.
3. Считываем содержимое **ИД_dir**, получаем доступ к содержимому файла-каталога **/dir** (используя массив адресов блоков **ИД_dir**) и осуществляем поиск записи, в которой имя совпадает с именем файла-каталога **filename**, номер индексного дескриптор в этой записи - **ИД_ filename**.

5 Управление оперативной памятью

Будем говорить о функциях управления оперативной памятью в контексте решения следующих основных задач. Во-первых, это осуществление контроля использования ресурсов, т.е. учет состояния каждой доступной в системе единицы памяти (свободна она или распределена).

Второй задачей является выбор стратегии распределения памяти. Иными словами, решается задача, какому процессу, в течение какого времени и в каком объеме должен быть выделен соответствующий ресурс. Стратегия распределения памяти является достаточно сложной задачей планирования.

Конкретное выделение ресурса тому или иному потребителю является третьей задачей управления ОЗУ. Эта подзадача следует за предыдущей задачей планирования: после решения задачи, какому процессу сколько выделить памяти и на какое время (в соответствии с наличием ресурса), следует операция непосредственного выделения. Это означает, что для предоставляемого ресурса идет корректировка системных данных (например, изменение статуса занятости), а затем выдача его потребителю.

И, наконец, четвертой задачей является выбор стратегии освобождения памяти. Освобождение памяти можно рассматривать с двух точек зрения. С одной стороны, это окончательное освобождение памяти, происходящее в случае завершения процесса и высвобождения ресурса. В этом контексте задача достаточно детерминирована и не требует каких-либо алгоритмов планирования и принятия решения. С другой стороны, освобождение памяти может рассматриваться как задача принятия решения в случае, когда встает потребность высвободить физическую память из-под какого-то процесса за счет откачивания во внешнюю память, чтобы на освободившееся пространство поместить данные другого процесса. Такая задача уже не тривиальна: необходимо решить, память какого процесса необходимо откачать и какая именно область памяти у выбранного процесса будет освобождаться. В принципе можно откачать весь процесс, но это зачастую неэффективно.

Ниже будут рассмотрены различные стратегии организации оперативной памяти (одиночное непрерывное распределение, распределение разделами, распределение перемещаемыми разделами, страничное распределение, сегментное распределение и сегменто-страничное распределение), а также методы управления ею. При этом при обсуждении каждой стратегии будем обращать внимание на её основные концепции, на те аппаратные средства, которые необходимы для поддержания данной модели, на типовые алгоритмы, а также будем рассматривать основные достоинства и недостатки.

5.1 Одиночное непрерывное распределение

Данная модель распределения оперативной памяти (Рис. 132) является одной из самых простых и основывается на том, что все адресное пространство подразделяется на два компонента. В одной части памяти располагается и функционирует операционная система, а другая часть выделяется для выполнения прикладных процессов.

При таком подходе не возникает особых организационных трудностей. С точки зрения обеспечения корректности функционирования этой модели, необходимо аппаратно обеспечить «водораздел» между пространствами, принадлежащими операционной системе и пользовательским процессом. Для этих целей достаточно иметь один регистр границы: если получаемый исполнительный адрес оказывается меньше значения этого регистра, то это адрес в пространстве операционной системы, иначе – в пространстве процесса. Такая реализация может сочетаться с аппаратной поддержкой двух режимов функционирования: пользовательского режима и режима ОС. Если в режиме пользователя происходит попытка обратиться в область памяти операционной системы, возникает прерывание. Алгоритмы,

используемые при таком распределении, достаточно просты, и мы не будем их здесь обсуждать.

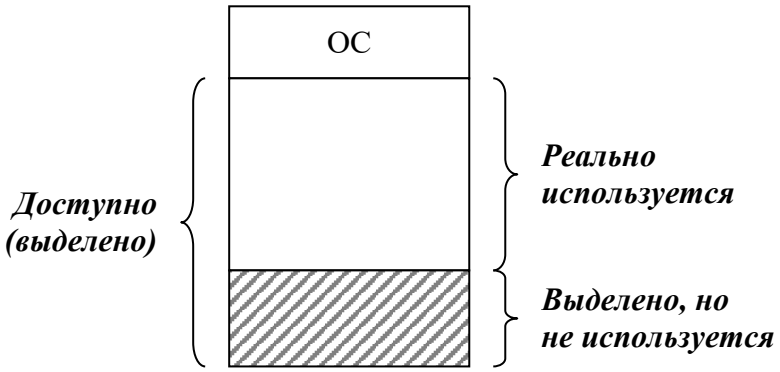


Рис. 132. Одиночное непрерывное распределение.

К достоинствам данной модели относится концептуальная простота во всех отношениях. В частности, минимальные аппаратные требования или отсутствие таковых, как в ОС Microsoft DOS, в которой даже не было регистра границ и пользовательский процесс мог обращаться к области ОС.

Среди недостатков можно отметить, во-первых, неэффективное использование физического ресурса: часть памяти, выделяемой под процесс, никогда реально не используется. Во-вторых, процесс занимает всю память полностью на все время выполнения. Но реально оказывается, что зачастую обращения процесса к памяти происходят в достаточно локализованные участки, а более или менее равномерное обращение ко всему адресному пространству процесса случается очень редко. Получается, что данная модель имеет еще и неявную неэффективность за счет того, что под все адресное пространство процесса отводится сразу все необходимое физическое пространство, хотя реально процесс работает лишь с локальными участками. И, наконец, в-третьих, рассматриваемая модель жестко ограничивает размер прикладного процесса.

5.2 Распределение неперемещаемыми разделами

Данная модель строится по следующим принципам (Рис. 133). Опять же, все адресное пространство оперативной памяти делится на две части. Одна часть отводится под операционную систему, все оставшееся пространство отводится под работу прикладных процессов, причем это пространство заблаговременно делится на N частей (назовем их ***разделами***), каждая из которых в общем случае имеет произвольный фиксированный размер. Эта настройка происходит на уровне операционной системы. Соответственно, очередь прикладных процессов разделяется по этим разделам.

Существуют концептуально два варианта организации этой очереди. Первый вариант (вариант Б) предполагает наличие единственной сквозной очереди, которая по каким-то соображениям распределяется между этими разделами. Второй вариант (вариант А) организован так, что с каждым разделом ассоциируется своя очередь и поступающий процесс сразу попадает в одну из этих очередей.

Существуют несколько способов аппаратной реализации данной модели. С одной стороны, это ***использование двух регистров границ***, один из которых отвечает за начало, а второй — за конец области прикладного процесса. Выход за ту или иную границу ведет к возникновению прерывания по защите памяти.

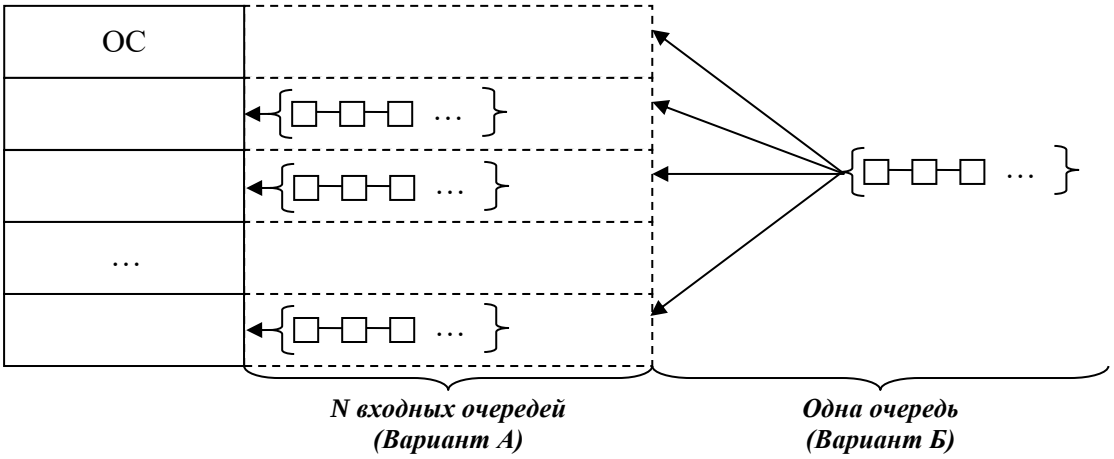


Рис. 133. Распределение неперемещаемыми разделами.

Рассмотрим теперь алгоритмы, применяемые в данной модели распределения памяти. Сначала рассмотрим алгоритм для модели с N очередями. Сортировка входной очереди процессов по отдельным очередям к разделам сводится к тому, что приходящий процесс размещается в разделе минимального размера, достаточного для размещения данного процесса. Заметим, что в общем случае не гарантируется равномерная загрузка всех очередей, что ведет к неэффективности использования памяти. Возможны ситуации, когда к некоторым разделам имеются большие очереди, а к разделам большего размера очередей вообще нет, т.е. возникает проблема недозагрузки некоторых разделов.

Другая модель с единой очередью процессов является более гибкой. Но она имеет свои проблемы. В частности, возникает проблема выбора процесса из очереди для размещения его в только что освободившийся раздел.

Одно из решений указанной проблемы может состоять в том, что из очереди выбирается первый процесс, помещающийся в освободившемся разделе. Такой алгоритм достаточно простой и не требует просмотра всей очереди процессов. Но в этом случае зачастую возможны ситуации несоответствия размеров процесса и раздела, когда процесс намного меньше освободившегося раздела. Это может привести к тому, что маленькие процессы будут «подавлять» более крупные процессы, которые могли бы поместиться в освободившемся разделе.

Другое решение предлагает, напротив, искать в очереди процесс максимального размера, помещающийся в освободившийся раздел. Очевидно, данный алгоритм требует просмотра всей очереди процессов, но зато он достаточно эффективно обходит проблему фрагментации раздела (возникающую, когда «маленький» процесс загружается в крупный раздел, и оставшаяся часть раздела просто не используется). Как следствие, данный алгоритм подразумевает дискриминацию «маленьких» процессов при выборе очередного процесса для постановки на исполнение.

Чтобы избавиться от последней проблемы, можно воспользоваться некоторой модификацией второго решения, основанной на следующем подходе. Для каждого процесса имеется счетчик дискриминации. Под ***дискриминацией*** будем понимать ситуацию, когда в системе освободился раздел, достаточный для загрузки некоторого процесса, но система планирования ОС его пропустила. Соответственно, при каждой такой дискриминации из счетчика дискриминации данного процесса вычитается единица. Тогда при просмотре очереди планировщик сначала проверяет значение этого счетчика: если оно равно равно нулю и процесс помещается в освободившемся разделе, то планировщик обязан загрузить данный процесс в этот раздел.

К достоинствам данной модели распределения оперативной памяти можно отнести простоту аппаратных средств организации мультипрограммирования (например, использование двух регистров границ) и простоту используемых алгоритмов. Сделаем

небольшое замечание. Если речь идет о модели с N очередями, то никаких дополнительных требований к реализации не возникает. Можно так все организовать, что подготавливаемый процесс в зависимости от его размера будет настраиваться на адресацию соответствующего раздела. Если же речь идет о модели с единой очередью процессов, то появляется требование к перемещаемости кода, это же требование добавляется и к аппаратной части. В данном случае это регистр базы, который может совпадать с одним из регистров границ.

К недостаткам можно отнести, во-первых, внутреннюю фрагментацию в разделах, поскольку зачастую процесс, загруженный в раздел, оказывается меньшего размера, чем данный раздел. Во-вторых, это ограничение предельного размера прикладных процессов размером максимального физического раздела ОЗУ. И, в-третьих, опять-таки весь процесс размещается в памяти, что может привести к неэффективному использованию ресурса (поскольку, как упоминалось выше, зачастую процесс работает с локализованной областью памяти).

5.3 Распределение перемещаемыми разделами

Данная модель распределения (Рис. 134) разрешает загрузку произвольного (нефиксированного) числа процессов в оперативную память, и под каждый процесс отводится раздел необходимого размера. Соответственно, система допускает перемещение раздела, а, следовательно, и процесса. Такой подход позволяет избавиться от фрагментации.

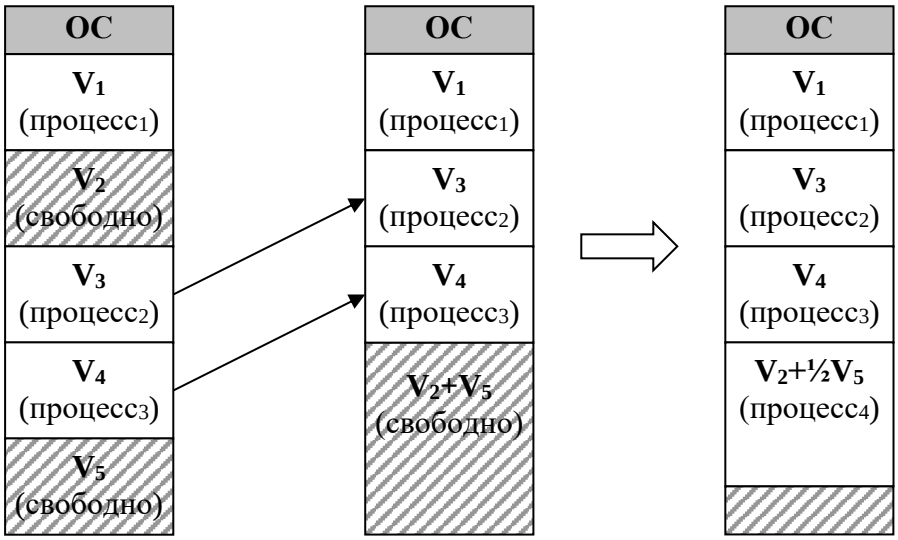


Рис. 134. Распределение перемещаемыми разделами.

По мере функционирования операционной системы после завершений тех или иных процессов пространство оперативной памяти становится все более и более фрагментированным: в памяти присутствует множество небольших участков свободного пространства, суммарный объем которых позволяет поместить достаточно большой процесс, но каждый из этих участков меньше размера этого процесса. Для борьбы с фрагментацией используется специальный процесс компрессии. Данная модель позволяет использовать компрессию за счет того, что исполняемый код процессов может перемещаться по оперативной памяти.

Очевидно, что в общем случае операция компрессии достаточно трудоемкая, поэтому существует ряд подходов для ее организации. С одной стороны, компрессия может быть локальной, когда система для высвобождения необходимого пространства передвигает небольшое количество процессов (например, два процесса). С другой стороны, возможен вариант, когда в некоторый момент система приостанавливает выполнение всех

процессов и начинает их перемещать, например, к начальному адресу оперативной памяти, тогда в конце ОЗУ окажется вся свободная память. Таким образом, стратегии здесь могут быть разными.

Что касается аппаратной поддержки, то здесь она аналогична предыдущей модели: требуются аппаратные средства защиты памяти (регистры границ) и аппаратные средства, позволяющие осуществлять перемещение процессов (в большинстве случаев для этих целей используется регистр базы, который в некоторых случаях может совпадать с одним из регистров границ). Используемые алгоритмы также достаточно очевидны и могут напоминать алгоритмы, рассмотренные при обсуждении предыдущей модели.

Основным достоинством данной модели распределения памяти является ликвидация фрагментации памяти. Отметим, что для систем, ориентированных на работу в мультипрограммном пакетном режиме (когда почти каждый процесс является более или менее большой вычислительной задачей), задача дефрагментации (или компрессии) не имеет существенного значения, поскольку для многочасовых вычислительных задач редкая минутная приостановка для совершения компрессии на эффективность системы не влияет. Соответственно, данная модель хорошо подходит для такого класса систем.

Если же, напротив, система предназначена для обработки большого потока задач пользователей, работающих в интерактивном режиме, то компрессия будет достаточно частой, а продолжительность компрессии, с точки зрения пользователя, будет достаточно большой, что, в конечном счете, будет отрицательно сказываться на эффективности подобной системы.

К недостаткам данной модели необходимо отнести опять-таки ограничение предельного размера прикладного процесса размером физической памяти. И, так или иначе, это накладные расходы, связанные с компрессией. В одних системах они несущественны, в других — напротив, имеют большое значение.

5.4 Страничное распределение

Об этой модели распределения оперативной памяти уже шла речь ранее, но тогда перед нами стояла задача лишь ввести читателя в курс дела, - в этом же разделе будут обсуждаться более подробно современные подходы страничной организации памяти.

Данная модель основывается на том, что все адресное пространство может быть представлено совокупностью блоков фиксированного размера (Рис. 135), которые называются ***страницами***. Есть ***виртуальное адресное пространство*** — это то пространство, с адресами которого оперирует программа, и ***физическое адресное пространство*** — это то пространство, которое есть в наличии в компьютере. Соответственно, при страничном распределении памяти существуют программно-аппаратные средства, позволяющие устанавливать соответствие между виртуальными и физическими страницами. Механизм преобразования виртуального адреса в физический обсуждался ранее, он достаточно прост: берется номер виртуальной страницы и заменяется соответствующим номером физической страницы. Также отмечалось, что для этих целей используется т.н. ***таблица страниц***, которая целиком является аппаратной, что на самом деле является большим упрощением. Если рассмотреть современные машины с современным объемом виртуального адресного пространства, то окажется, что эта таблица будет очень большой по размеру. Соответственно, возникает важный вопрос, как осуществлять указанное отображение виртуальных адресов в физические.

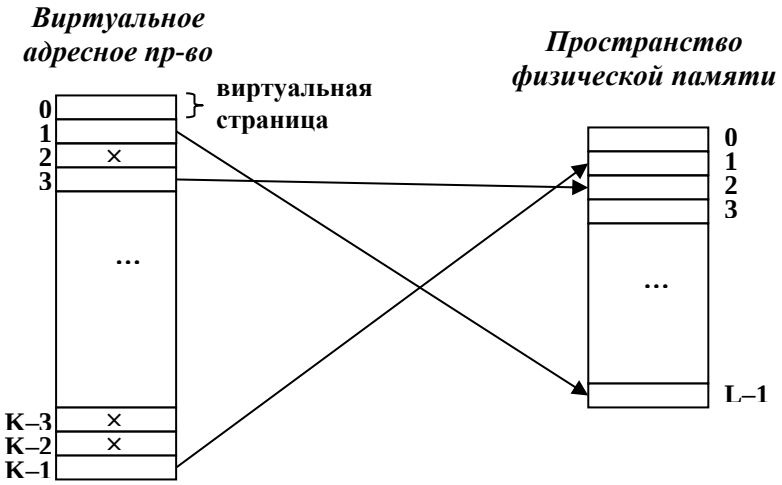


Рис. 135. Страничное распределение.

Ответ на поставленный вопрос, как всегда, неоднозначный и имеет несколько вариантов. Первое решение, приходящее на ум, — это полное размещение таблицы преобразования адресов в аппаратной части компьютера, но это решение применимо лишь в тех системах, где количество страниц незначительное. Примером такой системы может служить машина БЭСМ-6, которая имела 32 виртуальные страницы, и вся таблица с 32 строками располагалась в процессоре. Если же таблица получается большой, то возникают следующие проблемы: во-первых, высокая стоимость аппаратной поддержки, а во-вторых, необходимость полной перезагрузки таблицы при смене контекстов. Но при этом скорость преобразования оказывается довольно высокой.

Альтернативой служит решение, предполагающее хранение данной таблицы в оперативной памяти, тогда каждое преобразование происходит через обращение к ОЗУ, что совсем неэффективно. К аппаратуре предъявляются следующие требования: должен быть регистр, ссылающийся на начало таблицы в ОЗУ, а также должно аппаратно поддерживаться обращение в оперативную память по адресу, хранящемуся в указанном регистре, извлечение данных из таблицы и осуществление преобразования.

Возможно оптимизировать рассмотренный подход за счет использования кэширования L1 или L2. С одной стороны, поскольку к таблице страниц происходит постоянное обращение, странички из данной таблицы «зависают» в КЭШе. Но, если в компьютере используется всего один КЭШ и для потока управления, и для потока данных, то в этом случае через него направляется еще и поток преобразования страниц. Поскольку эти потоки имеют свои особенности, то добавление дополнительного потока со своими индивидуальными характеристиками приведет к снижению эффективности системы.

Стоит также отметить, что в современных системах таблицы страниц каждого процесса могут оказаться достаточно большими, - мультипрограммные ОС поддерживают обработку сотен или даже тысяч процессов, поэтому держать всю таблицу страниц в оперативной памяти также оказывается дорогим занятием. С другой стороны, если в ОЗУ хранить лишь оперативную часть этой таблицы, то возникают проблемы, связанные со сменой процессов: необходимо будет часть таблицы откачивать во внешнюю память, а часть — наоборот, подкачивать, что является достаточно трудоемкой задачей. Соответственно, возникает проблема организации эффективной работы с таблицей страниц, чтобы возникающие накладные расходы не приводили к деградации системы.

Помимо указанных подходов к размещению таблицы страниц, каждый из которых имеет свои преимущества и недостатки, в реальности применяют смешанные, или гибридные, решения.

Что касается используемых алгоритмов и способов организации данных для модели страничного распределения памяти, то традиционно применяются решения, связанные с иерархической организацией этих таблиц.

Типовая структура записи таблицы страниц (Рис. 136) содержит информацию о номере физической страницы, а также совокупность атрибутов, необходимых для описания статуса данной страницы. Среди атрибутов может быть атрибут присутствия/отсутствия страницы, атрибут режима защиты страницы (чтение, запись, выполнение), флаг модификации содержимого страницы, атрибут, характеризующий обращения к данной странице, чтобы иметь возможность определения «старения» страницы, атрибут блокировки кэширования и т.д. Итак, в каждой записи может присутствовать целая совокупность атрибутов, которые аппаратно интерпретируемы: например, при попытке записать данные в страницу, закрытую на запись, произойдет прерывание.

ε	**δ**	**γ**	**β**	**α**	**Номер физической страницы**

Рис. 136. Модельная структура записи таблицы страниц. Здесь: α — присутствие/отсутствие; β — защита (чтение, чтение/запись, выполнение); γ — изменения; δ — обращение (чтение, запись, выполнение); ε — блокировка кэширования.

В качестве одного из первых решений оптимизации работы с памятью стало использование т.н. ***TLB-таблиц*** (Translation Look-aside Buffer — таблица быстрого преобразования адресов, Рис. 137). Данный метод подразумевает наличие аппаратной таблицы относительно небольшого размера (порядка 8 – 128 записей). Данная таблица концептуально содержит три столбца: первый столбец — это номер виртуальной страницы, второй — это номер физической страницы, в которой находится указанная виртуальная страница, а третий столбец содержит упомянутые выше атрибуты.

Виртуальный адрес состоит из номера виртуальной страницы (**VP**) и смещения в ней (**offset**). Страница изымает из этого адреса номер виртуальной страницы и осуществляет оптимизированный поиск (т.е. поиск не последовательный, а параллельный) этого номера по TLB-таблице. Если искомый номер найден, то система автоматически на уровне аппаратуры осуществляет проверку соответствия атрибутов, и если проверка успешна, то происходит подмена номера виртуальной страницы номером физической страницы, и, таким образом, получается физический адрес.

Если же при поиске происходит промах (номер виртуальной странице не найден), то в этом случае система обращается в программную таблицу, выкидывает самую старую запись из TLB, загружает в нее найденную запись из программной таблицы, и затем вычисляется физический адрес. Таким образом, получается, что TLB-таблица является некоторым КЭШем.

Модели отработки промаха могут быть различными. Возможна организация отработки промаха без прерываний, когда система самостоятельно, имея регистр начала программной таблицы страниц, обращается к этой таблице и осуществляет в ней поиск. Возможна модель с прерыванием, когда при промахе возникает прерывание, управление передается операционной системе, которая затем начинает работать с программной таблицей страниц, и т.д. Заметим, что вторая модель менее эффективная, поскольку прерывания ведут к увеличению накладных расходов.

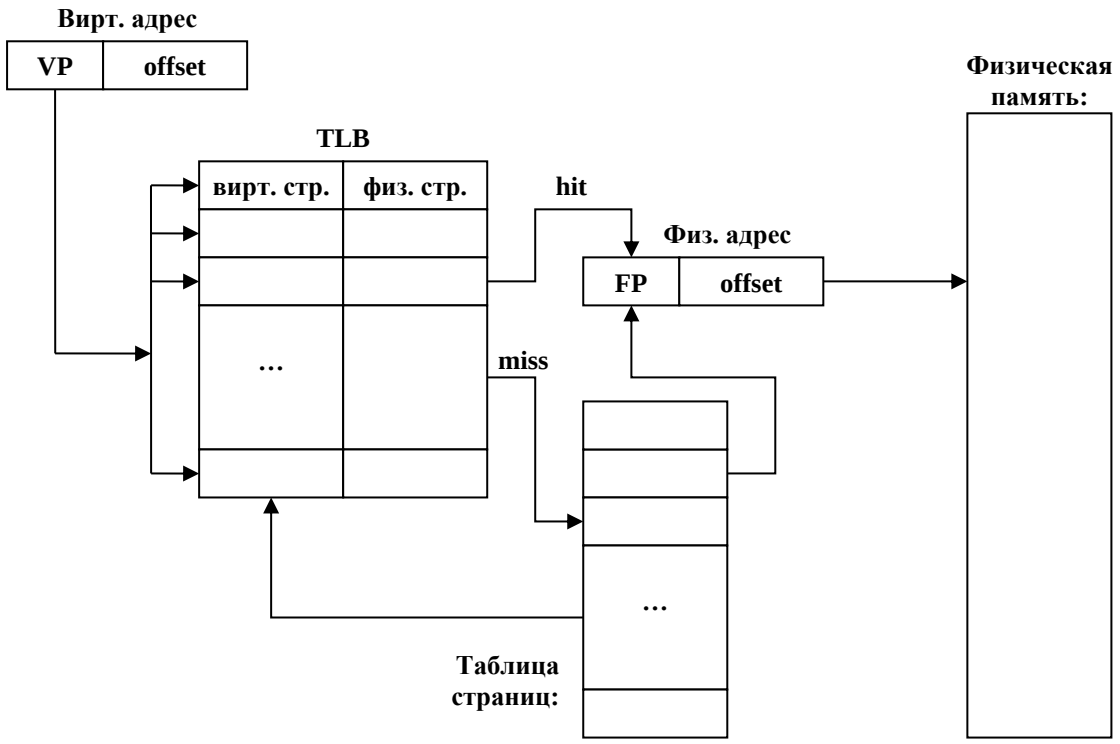


Рис. 137. TLB-таблица (Translation Look-aside Buffer).

Итак, рассмотренная модель использования TLB-таблиц является реальной, по сравнению с той моделью, которая была описана в начале курса. Одной из главных проблем этого подхода является проблема, связанная с большим размером таблицы страниц. Отметим, что большой размер этой таблицы плох по двум причинам: во-первых, при смене контекста система так или иначе обязана поменять эту таблицу, а также содержимое TLB, т.к. там хранится информацию об одном процессе, а во-вторых, это проблема, связанная с организацией мультипроцессирования, — необходимо решать, где размещать все таблицы различных процессов.

Одним из решений, позволяющих снизить размер таблицы страниц, является модель **иерархической организации таблицы страниц** (Рис. 138). В этом случае информация о странице представляется не в виде одного номера страницы, а в виде совокупности номеров, используя которые, можно получить номер соответствующей физической страницы, посредством обращения к соответствующим таблицам, участвующим в иерархии (это может быть 2-х-, 3-х- или даже 4-хуровневая иерархия).

Пусть имеется 32-разрядный виртуальный адрес, который в свете рассмотренной ранее модели может, например, содержать 20-разрядный номер виртуальной страницы и 12-разрядное значение смещения в ней. Если же используется двухуровневая иерархическая организация, то этот же виртуальный адрес можно трактовать, к примеру, как 10-разрядный индекс во «внешней» таблице групп (или кластеров) страниц, 10-разрядное смещение в таблице второго уровня и, наконец, 12-разрядное смещение в физической странице. Соответственно, чтобы получить номер физической страницы необходимо по индексу во «внешней» таблице страниц найти начальный адрес таблицы второго уровня, затем по этому адресу и по смещению по таблице второго уровня находится нужная запись в таблице страниц второго уровня, которая уже и содержит номер соответствующей физической страницы.

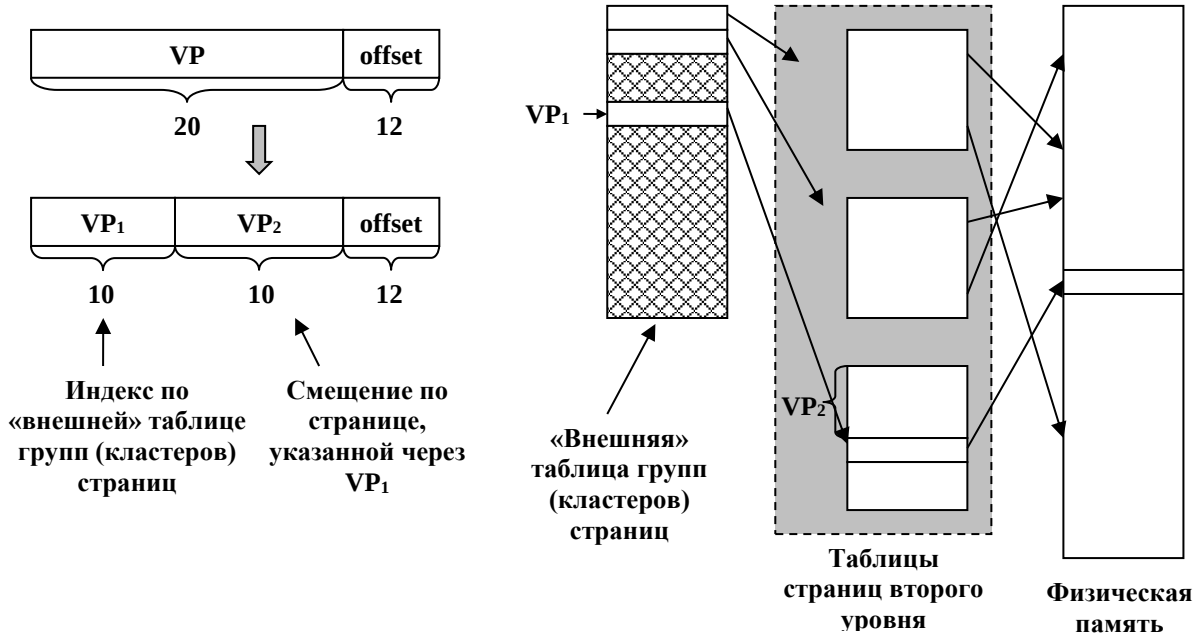


Рис. 138. Иерархическая организация таблицы страниц.

Используя данный подход, может оказаться, что всю таблицу страниц хранить в памяти вовсе необязательно: из-за принципа локализации будет достаточно хранить сравнительно небольшую «внешнюю» таблицу групп страниц и некоторые таблицы второго уровня (они также имеют незначительные размеры); все необходимые таблицы второго уровня можно подкачивать по мере надобности.

Подобные рассуждения можно распространить на большее число уровней иерархии, но, начиная с некоторого момента, эффективность системы начинает сильно падать с ростом числа уровней иерархии (из-за различных накладных расходов), поэтому обычно число уровней ограничено четырьмя.

Существует иное решение, позволяющее также обойти проблему большого размера таблицы страниц, - оно основано на использовании ***хеширования*** (на использовании т.н. ***хеш-таблиц***). Это решение в свою очередь, базируется на использовании ***хеш-функции***, или функции расстановки (Рис. 139). Эти функции используются в следующей задаче: пусть имеется некоторое множество значений, которое необходимо каким-то образом отобразить на множество фиксированного размера. Для осуществления этого отображения используют функцию, которая по входному значению определяет номер позиции (номер кластера, куда должно попасть это значение). Но эта функция имеет свои особенности: при ее использовании возможны коллизии, связанные с тем, что различные значения могут оказаться в одном и том же кластере.

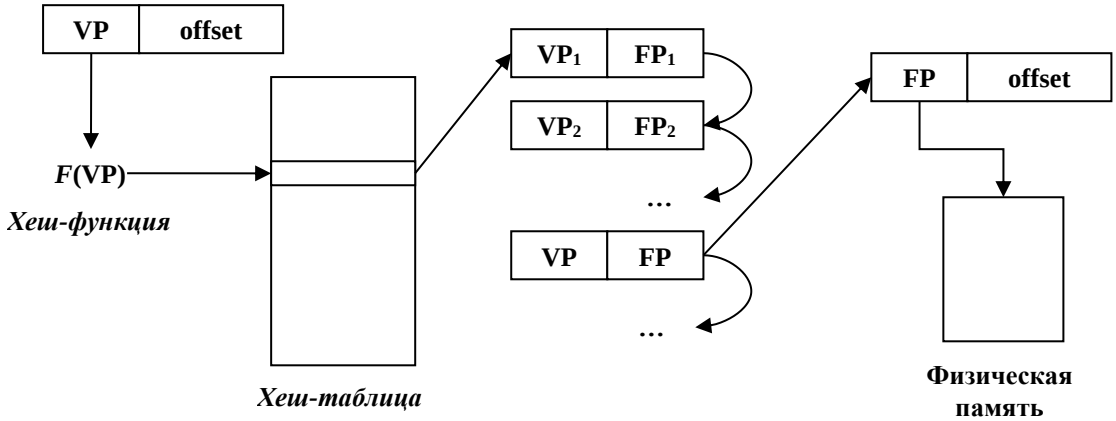


Рис. 139. Использование хеш-таблиц.

Модель преобразования адресов, основанная на хешировании, достаточно проста. Из виртуального адреса аппаратно извлекается номер виртуальной страницы, который подается на вход некоторой хеш-функции, отображающей значение на аппаратную таблицу (т.н. хеш-таблицу) фиксированного размера. Каждая запись в данной таблице хранит начало списка коллизий, где каждый элемент списка является парой: номер виртуальной страницы — соответствующий ему номер физической страницы. Итак, перебирая соответствующий список коллизий, можно найти номер исходной виртуальной страницы и соответствующий номер физической страницы. Подобное решение имеет свои достоинства и недостатки: в частности, возникают проблемы с перемещением списков коллизий.

Еще одним решением, позволяющим снизить размер таблицы страниц, является модель использования т.н. ***инвертированных таблиц страниц*** (Рис. 140). Главной сложностью данного решения является требование к процессору на аппаратном уровне работать с идентификаторами процессов (их PID). Примерами таких процессоров могут служить процессоры серий SPARC и PowerPC.

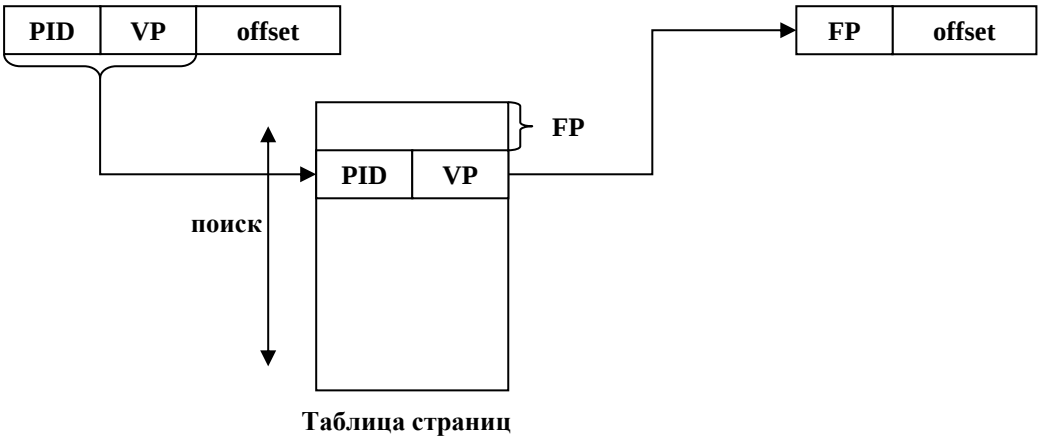


Рис. 140. Инвертированные таблицы страниц.

В этой модели виртуальный адрес трактуется как тройка значений: PID процесса, номер виртуальной страницы и смещение в этой странице. При таком подходе используется единственная таблица страниц для всей системы, и каждая строка данной таблицы соответствует физической странице (с номером, равным номеру этой строки). При этом каждая запись данной таблицы содержит информацию о том, какому процессу принадлежит данная физическая страница, а также какая виртуальная страница этого процесса размещена в данной физической странице. Итак, имея пару PID процесса и номер виртуальной страницы, производится поиск ее в таблице страниц, и по смещению найденного результата определяется номер физической страницы.

К достоинствам данной модели можно отнести наличие единственной таблицы страниц, обновление которой при смене контекстов сравнительно нетрудоемкое: операционная система производит обновление тех строк таблицы, для которых в соответствующие физические страницы происходит загрузка процесса. Отметим, что «тонким местом» данной модели является организация поиска в таблице. Если будет использоваться прямой поиск, то это приведет к существенным накладным расходам. Для оптимизации этого момента возможна надстройка над этим решением более интеллектуальных моделей — например, модели хеширования и/или использования TLB-таблиц.

Революционным достоинством страничной организации памяти стало то, что исполняемый в системе процесс может использовать очень незначительную часть физического ресурса памяти, а все остальные его страницы могут размещаться во внешней памяти (быть откачанными). Очевидно, что и страничная организация памяти имеет свои недостатки: в частности, это проблема фрагментации внутри страницы.

В связи с использованием страничной организации памяти встает еще одна проблема — это проблема выбора той страницы, которая должна быть откачана во внешнюю память при необходимости загрузить какую-то страницу из внешней памяти. Эта задача имеет множество решений, некоторые из которых будут освещены ниже.

Первым рассмотрим алгоритм **NRU** (Not Recently Used — не использовавшийся в последнее время). Этот алгоритм основан на том, что с любой страницей ассоциируются два признака, один из которых отвечает за *обращение* на чтение или запись к странице (R-признак), а второй — за модификацию страницы (M-признак), когда в страницу что-то записывается. Значение этих признаков устанавливается аппаратно. Имеется также возможность посредством обращения к операционной системе обнулять эти признаки.

Итак, алгоритм NRU действует по следующему принципу. Изначально для всех страниц процесса признаки R и M обнуляются. По таймеру или по возникновению некоторых событий в системе происходит программное обнуление всех R-признаков. Когда системе требуется выбрать какую-то страницу для откачки из оперативной памяти, это осуществляется следующим образом. Все страницы, принадлежащие данному процессу, делятся на 4 категории в зависимости от значений признаков R и M.

- **Класс 0**: R = 0, M = 0. Это те страницы, в которых не происходило обращение в последнее время и в которых не сделано ни одно изменение.
- **Класс 1**: R = 0, M = 1. Это те страницы, к которым в последний период не было обращений (поскольку программно обнулен R-признак), но в этой странице в свое время произошло изменение.
- **Класс 2**: R = 1, M = 0. Это те страницы, из которых за последний таймаут читалась информация.
- **Класс 3**: R = 1, M = 1. Это те страницы, к которым за последнее время были обращения, в т.ч. обращения на запись, т.е. это активно используемые страницы.

Соответственно, алгоритм предлагает выбирать страницу для откачивания случайным способом из непустого класса с минимальным номером.

Следующий алгоритм, который мы рассмотрим, — это алгоритм **FIFO**. Если в системе реализован этот алгоритм, то при загрузке очередной страницы в память операционная система фиксирует время этой загрузки. Соответственно, данный алгоритм предполагает откачку той страницы, которая наиболее долго располагается в ОЗУ.

Очевидно, что данная стратегия зачастую оказывается неэффективной, поскольку возможна откачка интенсивно используемой страницы. Поэтому существует целый ряд модификаций алгоритма FIFO, нацеленных на сглаживание обозначенной проблемы.

Модифицированный алгоритм может иметь следующий вид. Выбирается самая «старая» страница, затем система проверяет значение признака доступа к этой странице (R-признак). Если R = 0, то эта страница откачивается. Если же R = 1, то этот признак обнуляется, а также время загрузки данной страницы переопределяется текущим временем (иными словами, данная страница перемещается в конец очереди), после чего алгоритм начинает свою работу с начала.

Данный алгоритм имеет недостатки, связанные с ростом накладных расходов при перемещении страниц по очереди. Поэтому этот алгоритм получил свое развитие, в частности, в виде алгоритма **«Часы»**.

Алгоритм «Часы» подразумевает, что все страницы образуют циклический список (Рис. 141). Имеется некоторый маркер, ссылающийся на некоторую страницу в списке, и этот маркер может перемещаться, например, только по часовой стрелке.

Функционирование алгоритма достаточно просто: если значение R-признака в обозреваемой маркером странице равно нулю, то эта страница выгружается, а на ее место помещается новая страница, после чего маркер сдвигается. Если же R = 1, то этот признак обнуляется, а маркер сдвигается на следующую позицию.

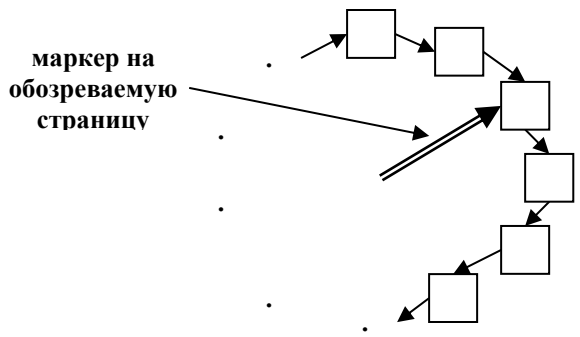


Рис. 141. Замещение страниц. Алгоритм «Часы».

Следующая группа алгоритмов позволяет более адекватно учитывать старение и использование страниц и, соответственно, более адекватно осуществлять выбор страницы для откачки.

Алгоритм **LRU** (Least Recently Used — «наименее недавно» – наиболее давно используемая страница) основан на достаточно сложной аппаратной схеме и действует по следующей схеме.

Пусть имеется N страниц. Для решения задачи в компьютере имеется битовая матрица, размером N × N, которая изначально обнуляется. Когда происходит обращение к i-ой странице, то все биты i-ой строки устанавливаются в 1, а весь i-ый столбец обнуляется. Соответственно, когда понадобится выбрать страницу для откачки, то выбирается та страница, для которой соответствующая строка хранит наименьшее двоичное число.

Рассмотренный алгоритм хорош тем, что достаточно адекватно учитывает интенсивность использования страниц, но этот алгоритм требует сложной аппаратной реализации.

Альтернативой указанному алгоритму может служить алгоритм **NFU** (Not Frequently Used — редко использовавшаяся страница), основанный на использовании программных счетчиков страниц.

Данный алгоритм подразумевает, что с каждой физической страницей с номером i ассоциирован программный счетчик $Count_i$. Изначально для всех i происходит обнуление счетчиков. А затем, по таймеру происходит увеличение значений всех счетчиков на величину интенсивности использования, т.е. на величину R-признака: $Count_i = Count_i + R_i$. Иными словами, если за последний таймаут было обращение к странице, что значение счетчика возрастает, иначе — не изменяется. Соответственно, для откачки выбирается страница с минимальным значением счетчика $Count_i$.

Данная модель также является достаточно адекватной, но она имеет ряд важных недостатков. Первый связан с тем, что счетчик хранит историю: например, если какая-то страница в некоторый период времени интенсивно использовалась, то значение счетчика стало настолько большим, что при прекращении работы с данной страницей значение счетчика достаточно долго не даст откачать эту страницу. А второй недостаток связан с тем, что при очень интенсивном обращении к странице возможно переполнение счетчика.

Чтобы сгладить указанные недостатки, существует модификация данного алгоритма, основанная на том, что каждый раз по таймеру значение счетчика сдвигается на 1 разряд вправо, после чего последний (левый) разряд устанавливается в значение R-признака. Выбирается страница с минимальным значением счетчика.

5.5 Сегментное распределение

Недостатком страничного распределения памяти является то, что при реализации этой модели процессу выделяется единый диапазон виртуальных адресов: от нуля до

некоторого предельного значения. С одной стороны, ничего плохого в этом нет, но это свойство оказывается неудобным по следующей причине. В процессе есть команды, есть статические переменные, которые, по сути, являются разными объединениями данных с различными характеристиками использования. Еще большие отличия в использовании иллюстрируют существующие в процессе стек и область динамической памяти, называемой также ***кучей***. И модель страничной организации памяти подразумевает статическое разделение единого адресного пространства: выделяются область для команд, область для размещения данных, а также область для стека и кучи. При этом зачастую стек и куча размещаются в единой области, причем стек прижат к одной границе области, куча — к другой, и «растут» они навстречу друг другу. Соответственно, возможны ситуации, когда они начинают пересекаться (ситуация переполнения стека). Или даже если стек будет располагаться в отдельной области памяти, он может переполнить выделенное ему пространство. Таковы основные недостатки страничного распределения памяти.

Избавиться от указанных недостатков на концептуальном уровне призвана модель сегментного распределения памяти (Рис. 142). Данная модель представляет каждый процесс в виде совокупности сегментов, каждый из которых может иметь свой размер. Каждый из сегментов может иметь собственную функциональность: существуют сегменты кода, сегменты статических данных, сегмент стека и т.д. Для организации работы с сегментами может использоваться некоторая таблица, в которой хранится информация о каждом сегменте (его номер, размер и пр.). Тогда виртуальный адрес может быть проинтерпретирован как номер сегмента и величина смещения в нем.

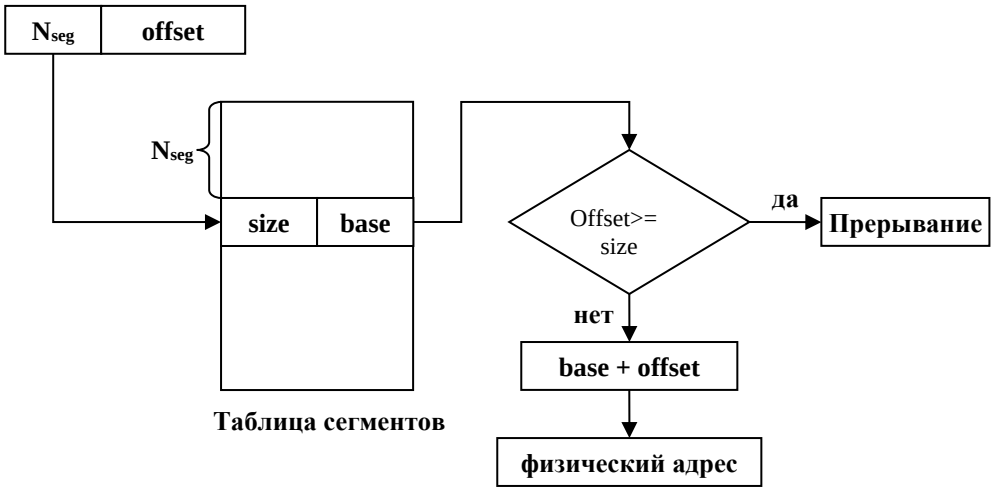


Рис. 142. Сегментное распределение.

Модель сегментного распределения может иметь достаточно эффективно работающую аппаратную реализацию. Существует аппаратная таблица сегментов с фиксированным числом записей. Каждая запись этой таблицы соответствует своему сегменту и хранит информацию о размере сегмента и адрес начала сегмента (т.е. адрес базы), а также тут могут присутствовать различные атрибуты, которые будут оговаривать права и режимы доступа к содержимому сегмента.

Итак, имея виртуальный адрес, система аппаратным способом извлекает из него номер сегмента i, обращается к i-ой строке таблицы, из которой извлекается информация о сегменте. После этого происходит проверка, не превосходит ли величина сегмента размера самого сегмента. Если превосходит, то происходит прерывание; иначе, складывая базу со смещением, вычисляется физический адрес.

К достоинствам данной модели можно отнести простоту организации, которая, по сути, явилась развитием модели распределения разделов. Если модели распределения разделов каждому процессу выделяется только один сегмент (раздел), то при сегментной

модели распределения процессу выделяется совокупность сегментов, каждый из которых будет иметь свои функциональные обязанности.

К недостаткам данной модели необходимо отнести то, что каждый сегмент должен целиком размещаться в памяти (возникает упоминавшаяся выше проблема неявной неэффективности, связанная с принципом локальности). Также возникают проблемы с откачкой/подкачкой: подкачка осуществляется всем процессом или, по крайней мере, целым сегментом, что зачастую оказывается неэффективно. И поскольку каждый сегмент так или иначе должен быть размещен в памяти, то возникает ограничение на предельный размер сегмента.

5.6 Сегментно-страничное распределение

Естественным развитием рассмотренной модели сегментного распределения памяти стала модель сегментно-страничного распределения. Эта модель рассматривает виртуальный адрес как номер сегмента и смещение в нем. Имеется также аппаратная таблица сегментов, посредством которой из виртуального адреса получается т.н. ***линейный адрес***, который, в свою очередь, представляется в виде номера страницы и величины смещения в ней. А затем, используя таблицу страниц, получается непосредственно физический адрес.

Итак, данный механизм подразумевает, что в процессе имеется ряд виртуальных сегментов, которые дробятся на страницы. Поэтому данная модель сочетает в себе, с одной стороны, логическое сегментирование, а с другой стороны, преимущества страничной организации (когда можно работать с отдельными страницами памяти, не требуя при этом полного размещения сегмента в ОЗУ).

Примером реализации может служить реализация, предложенная компанией Intel. Рассмотрим упрощённую модель этой реализации (Рис. 143). Виртуальный адрес в этой модели представляется в виде ***селектора*** (информации о сегменте) и смещения в сегменте.

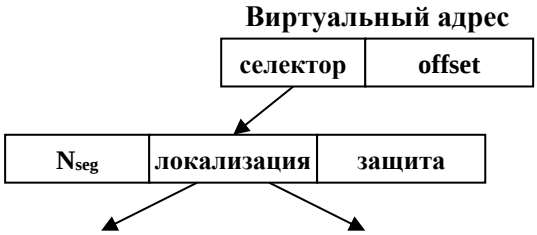


Рис. 143. Сегментно-страничное распределение. Упрощенная модель Intel.

Селектор содержит информацию о локализации сегмента. В модели Intel сегмент может быть одного из двух типов: ***локальный сегмент***, который описывается в таблице локальных дескрипторов **LDT** (Local Descriptor Table) и который может быть доступен лишь данному процессу, или ***глобальный сегмент***, который описывается в таблице глобальных дескрипторов **GDT** (Global Descriptor Table) и который может разделяться между процессами. Заметим, что каждая запись таблиц LDT и GDT хранит полную информацию о сегменте (адрес базы, размер и пр.). Итак, в селекторе хранится тип сегмента, после которого следует номер сегмента (номер записи в соответствующей таблице). Помимо перечисленного, селектор хранит различные атрибуты, касающиеся режимов доступа к сегменту.

Преобразование виртуального адреса в физический имеет достаточно простую организацию (Рис. 144). На основе виртуального адреса, посредством использования

информации из таблиц LDT и GDT, получается 32-разрядный линейный адрес, который интерпретируется в терминах двухуровневой иерархической страничной организации. Последние 12 разрядов отводятся под смещение в физической странице, а первые 20 разрядов интерпретируют как 10-разрядный индекс во «внешней» таблице групп страниц и 10-разрядное смещение в соответствующей таблице второго уровня иерархии.

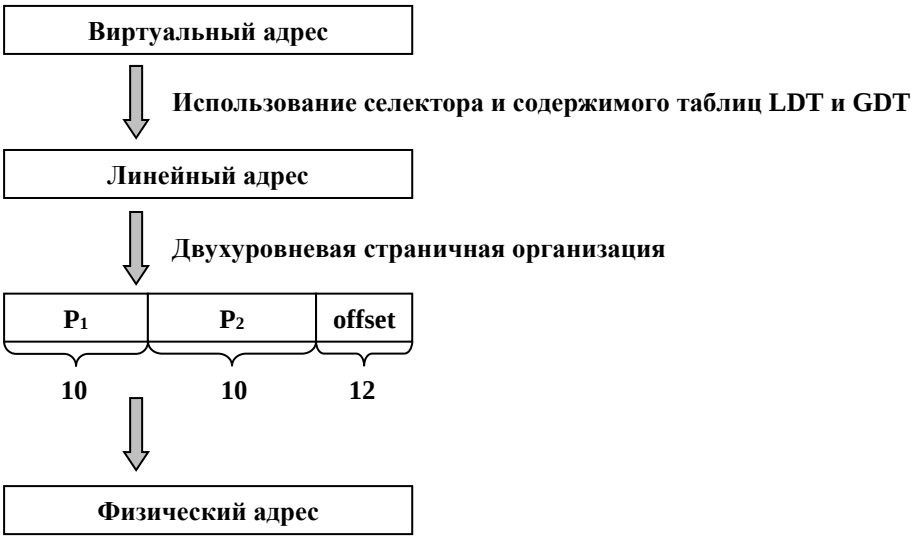


Рис. 144. Схема преобразования адресов.

Среди особенностей данной модели можно отметить, что можно «выключать» страничную функцию, и тогда модель Intel начинает работать по сегментному распределению. А можно не использовать сегментную организацию процесса, и тогда данная реализация будет работать по страничному распределению памяти.

6 Управление внешними устройствами

6.1 Общие концепции

6.1.1 Архитектура организации управления внешними устройствами

Как отмечалось ранее, при организации взаимодействия работы процессора и внешних устройств различают два потока информации: поток управляющей информации (т.е. поток команд какому-либо устройству управления) и поток данных (поток информации, участвующей в обмене обычно между ОЗУ и внешним устройствами). Рассматривая историю вопроса, необходимо отметить, что управление внешними устройствами претерпело достаточно большие изменения.

Первой исторической моделью стало **непосредственное управление** центральным процессором внешними устройствами (Рис. 145.А), когда процессор на уровне микрокоманд полностью обеспечивал все действия по управлению внешними устройствами. Иными словами, поток управления полностью шел через ЦПУ, а наравне с ним через процессор шел и поток данных. Эта модель иллюстрирует синхронное управление: если начался обмен, то, пока он не закончится, процессор не продолжает вычисления (поскольку занят обменом).

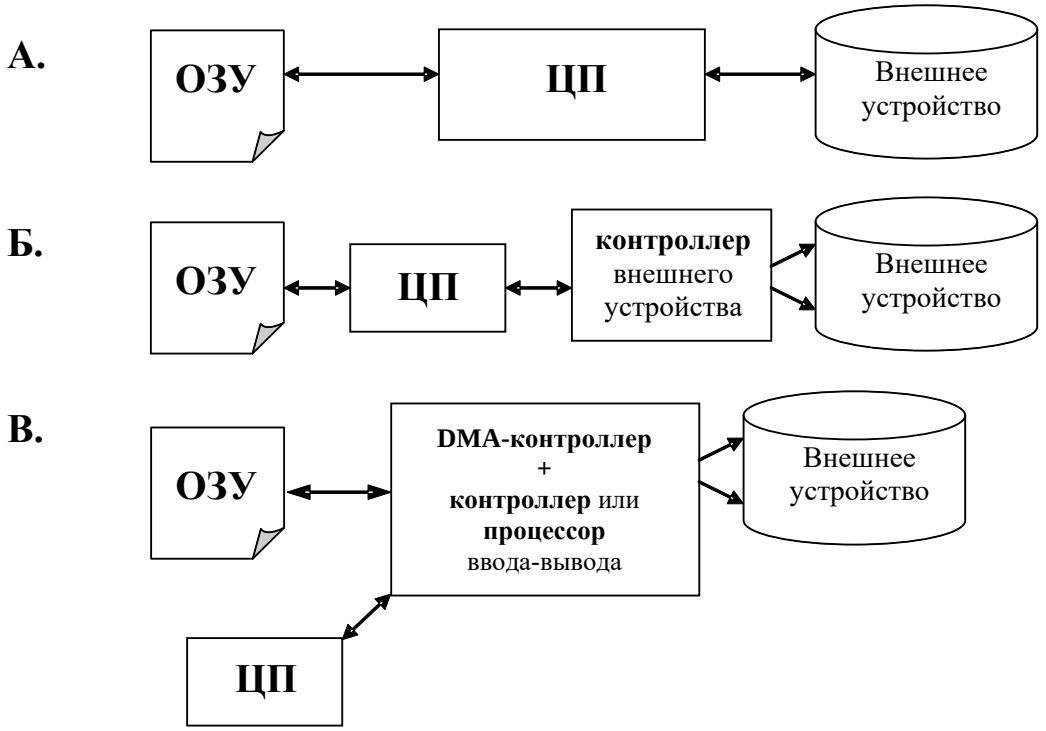


Рис. 145. Модели управления внешними устройствами: непосредственное (А), синхронное/асинхронное (Б), с использованием контроллера прямого доступа или процессора (канала) ввода-вывода (В).

Вторая модель, появившаяся с развитием вычислительной техники, связана с появлением **специализированных контроллеров** устройств, которые концептуально располагались между центральным процессором и соответствующими внешними устройствами (Рис. 145.Б). Контроллеры позволяли процессору работать с более высокоуровневыми операциями при управлении внешними устройствами. Таким образом, процессор частично освобождался от потока управления внешними устройствами за счет того, что вместо большого числа микрокоманд конкретного устройства он оперировал

меньшим количеством более высокоуровневых операций. Но и эта модель оставалась **синхронной**.

Следующим этапом стало развитие предыдущей модели до **асинхронной**, осуществление которой стало возможным благодаря появлению аппарата прерываний. Данная модель позволяла запустить обмен для одного процесса, после этого поставить на счет другую задачу (или же текущий процесс может продолжить выполнять какие-то свои вычисления), а по окончании обмена (успешного или неуспешного) в системе возникнет прерывание, сигнализирующее возможность дальнейшего выполнения первого процесса. Но и эти две модели предполагали, что поток данных идет через процессор.

Кардинальным решением проблемы перемещения обработки потока данных из процессора стало использование появившихся **контроллеров прямого доступа** к памяти (или **DMA-контроллеров**, Direct Memory Access, Рис. 145.B). Процессор генерировал последовательность управляющих команд, указывая координаты в оперативной памяти, куда надо положить или откуда надо взять данные, а DMA-контроллер занимался перемещением данных между ОЗУ и внешним устройством. Таким образом, поток данных шел в обход процессора.

И, наконец, можно отметить последнюю модель, основанную на том, что управление внешними устройствами осуществляется с использованием **специализированного процессора** (или даже **специализированных компьютеров**) или **каналов ввода-вывода**. Данная модель подразумевает снижение нагрузки на центральный процессор с точки зрения обработки потока управления: ЦПУ теперь оперирует функционально-емкими макрокомандами. Решение задачи осуществления непосредственного обмена, а также решение всех связанных с обменом вопросов (в т.ч. оптимизация операций обмена, например, за счет использования аппаратной буферизации в процессоре ввода-вывода) ложится «на плечи» специализированного процессора.

6.1.2 Программное управление внешними устройствами

Рассмотрим архитектуру программного управления внешними устройствами, которую можно представить в виде некоторой иерархии (Рис. 146). В основании лежит **аппаратура**, а далее следуют программные уровни: **программы обработки прерываний**, **драйверы физических устройств**, а на вершине иерархии лежит уровень **драйверов логических устройств**, причем каждый уровень строится на основании нижележащего уровня.

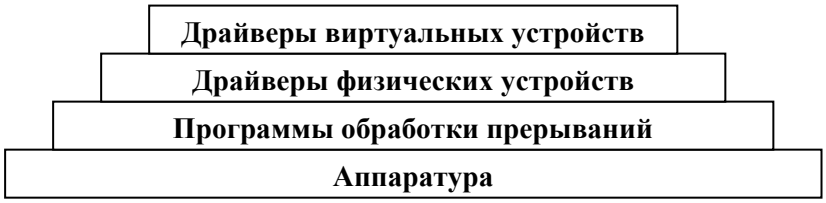


Рис. 146. Иерархия архитектуры программного управления внешними устройствами.

Можно выделить следующие цели программного управления устройствами. Во-первых, это **унификация программных интерфейсов доступа к внешним устройствам**. Иными словами, это стандартизация правил использования различных устройств. Преследуя данную цель, мы абстрагируемся от аппаратных характеристик обмена. Если данная цель достигнута, то, например, пользователю, пожелавшему распечатать текстовый файл, не надо будет заботиться об организации управления конкретным печатающим устройством - ему достаточно воспользоваться некоторым общим программным интерфейсом.

Следующая цель — это **обеспечение конкретной модели синхронизации при выполнении обмена** (синхронный или асинхронный обмен). Отметим здесь, что, несмотря на то, что синхронный вид обмена появился хронологически одним из первых, он остается актуальным и по сей день. Таким образом, ставится цель поддерживать оба вида обмена, а выбор конкретного типа зависит от пользователя.

Еще одной целью является **выявление и локализация ошибок**, а также **устранение их последствий**. Для любой системы справедливо, что чем более она сложна, тем больше статистически в ней возникает сбоев. Соответственно, система должна быть организована таким образом, чтобы она могла выявить момент появления сбоя и стараться обработать эту сбойную ситуацию: либо самостоятельно ее обойти, либо известить пользователя.

Следующая цель — **буферизация обмена** — связана с различной производительностью основных компонентов системы. Известно, что любое внешнее устройство работает медленнее центральной части компьютера, и, соответственно, стоит проблема сглаживания разброса производительностей различных компонентов системы. Причем, если речь идет об устройствах, не являющихся устройствами оперативного доступа (т.е. не являющихся устройствами, к которым идет массовое обращение на обмен от процессов), то для таких устройств проблема сглаживания отодвигается на второй план. Например, если это медленное устройство печати, то для него особо не требуется реализация буфера, а если дело касается магнитного диска, рассчитанного на использование в качестве массового устройства, то тут операции обмена должны обрабатываться по возможности быстро. Решением указанной задачи является организация разного рода кэширования в системе.

Также необходимо отметить такую цель, как **обеспечение стратегии доступа к устройству** (распределенный или монопольный доступ). Во время рассмотрения файловых систем ОС Unix говорилось, что один и тот же файл может быть доступен через множество файловых дескрипторов, которые могут быть распределены между различными процессами, т.е. файл может быть многократно открыт в системе, и система позволяет организовывать распределенный доступ к его информации. Система позволяет организовать управление этим доступом и синхронизацию. Система также позволяет организовать и монопольный доступ к устройству.

И, наконец, последней целью, которую стоит отметить, является **планирование выполнения операций обмена**. Это важная проблема, поскольку от качества планирования может во многом зависеть эффективность функционирования вычислительной системы. Неправильно организованное планирование очереди заказов на обмен может привести к деградации системы, связанной, к примеру, с началом голодания каких-то процессов и, соответственно, зависания их функциональности.

6.1.3 Планирование дисковых обменов

Рассмотрим различные стратегии организации планирования дисковых обменов. При этом преследуется цель проиллюстрировать то многообразие подходов к решению данной проблемы, которые имеют место в мире, с краткими результатами и выводами.

Будем рассматривать некоторое дисковое устройство, обмен с которым осуществляется дорожками (или блоками). Предположим имеется очередь запросов к следующим дорожкам: 4, 40, 11, 35, 7, 14 и пусть изначально головка дискового устройства позиционирована на 15-ой дорожке. Замети, что время на обмен складывается из трех компонентов: выход головки на позицию, вращение диска и непосредственно обмен. Для оценки эффективности алгоритмов будем подсчитывать суммарный путь (выраженный в количестве дорожек), который пройдет головка для осуществления всех запросов на обмен из указанной очереди.

Первая стратегия, которую мы рассмотрим, — стратегия **FIFO** (First In First Out). Эта стратегия основывается лишь на порядке появления запроса в очереди. В нашем случае (Рис. 147) головка сначала начинает двигаться с 15 дорожки на 4, потом на 40 и т.д. После

обработки всей указанной очереди суммарная длина пути составляет 135 дорожек, что в среднем можно охарактеризовать, как 22,5 дорожки на один обмен.

Путь головки	L
15 → 4	11
4 → 40	36
40 → 11	29
11 → 35	24
35 → 7	28
7 → 14	7
Итого: 135 Средний путь: 22,5	

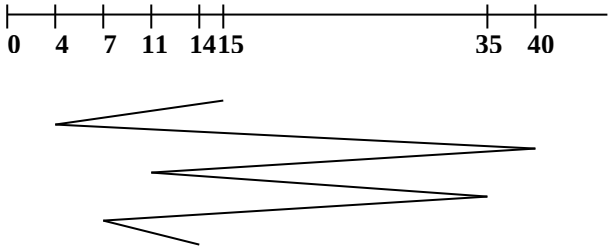


Рис. 147. Планирование дисковых обменов. Модель FIFO.

Альтернативой FIFO является стратегия **LIFO** (Last In First Out). Этот алгоритм в нашем случае имеет примерно те же характеристики, что и FIFO. Но данная стратегия оказывается полезной, когда поступают цепочки связанных обменов: процесс считывает информацию, изменяет ее и обратно записывает. Для таких процессов эффективнее всего будет выполнение именно цепочки обмена, иначе после считывания он не сможет продолжаться, т.к. будет ожидать записи (Рис. 148).

Путь головки	L
15 → 14	1
14 → 7	7
7 → 35	28
35 → 11	24
11 → 40	29
40 → 4	36
Итого: 126 Средний путь: 20,83	

Рис. 148. Планирование дисковых обменов. Модель LIFO.

Следующая стратегия — **SSTF** (Shortest Service Time First) — основана на «жадном» алгоритме. Термин «жадного» алгоритма обычно применяется к итерационным алгоритмам для выделения среди них класса алгоритмов, которые на каждой итерации ищут наилучшее решение. Применительно к нашей задаче данный алгоритм на каждом шаге осуществляет поиск в очереди запросов номера дорожки, требующей минимального перемещения головки диска. В итоге, в нашем примере получаются достаточно хорошие результаты, но данный алгоритм имеет существенный недостаток: ему присуща проблема голодания крайних дорожек (Рис. 149).

Путь головки	L
15 → 14	1
14 → 11	3
11 → 7	4
7 → 4	3
4 → 35	31
35 → 40	5
Итого: 47 Средний путь: 7,83	

Рис. 149. Планирование дисковых обменов. Модель SSTF.

Еще один алгоритм — алгоритм, основанный на приоритетах процессов (**PRI**). Данный алгоритм подразумевает, что каждому процессу присваивается некоторый приоритет, тогда в заказе на обмен присутствует еще и приоритет. И, соответственно, очередь запросов обрабатывается согласно приоритетам. Здесь встает серьезная проблема корректной организации выдачи приоритета, иначе будут возникать случаи голодания низкоприоритетных процессов.

Следующий алгоритм, который мы рассмотрим, — «лифтовый» алгоритм, или алгоритм сканирования (**SCAN**). Данный алгоритм основан на том, что головка диска перемещается сначала в одну сторону (предельно до границы диска), выбирая каждый раз из очереди запрос с номером обозреваемой головкой дорожки, а затем — в другую. Тогда заведомо известно, что для любого набора запросов потребуется перемещений не больше удвоенного числа дорожек на диске. Данный алгоритм может приводить к деградации системы вследствие голодания некоторых процессов в случае, когда идет интенсивный обмен с некоторой локальной областью диска (Рис. 150).

Путь головки	L
15 → 35	20
35 → 40	5
40 → 14	26
14 → 11	3
11 → 7	4
7 → 4	3
Итого: 61 Средний путь: 10,16	

Рис. 150. Планирование дисковых обменов. Модель SCAN.

Некоторой альтернативой является алгоритм циклического сканирования (**C-SCAN**). Этот алгоритм основан на том, что сканирование всегда происходит в одном направлении. В очереди запросов ищется запрос с минимальным (или максимальным) номером, головка передвигается к дорожке с этим номером, а затем за один проход по диску обрабатывается вся очередь запросов. Но проблемы остаются те же самые, что и для алгоритма сканирования (Рис. 151).

Путь головки	L
15 → 4	11
4 → 7	3
7 → 11	4
11 → 14	3
14 → 35	21
35 → 40	5
Итого: 47 Средний путь: 7,83	

Рис. 151. Планирование дисковых обменов. Модель C-SCAN.

Для решения проблемы зависания при интенсивном обмене с локальной областью диска применяются многошаговый алгоритм (**N-step-SCAN**). В этом случае очередь запросов каким-либо образом делится на N подочередей (способ разделения может быть произвольным, в частности, по локализации запросов, по времени поступления и т.д.); затем по какой-либо стратегии выбирается очередь, которая будет обрабатываться (например, по порядку формирования), и начинается ее обработка. Во время обработки очереди блокируется попадание новых запросов в эту очередь (тогда эти запросы могут сформировать новую очередь), другие очереди могут получать заявки. Сама обработка очереди может осуществляться, например, по алгоритму сканирования. Данный алгоритм «уходит» от проблемы голодания.

Итак, мы проиллюстрировали некоторые стратегии организации планирования дисковых обменов. Еще раз отметим, что эти модель очень упрощенные: они основаны на использовании минимальной информации о заказе на обмен. Реальные системы, реальные очереди содержат не одиночные заказы на обмен, а целые цепочки заказов на обмен, которые произвольным образом обрабатывать нельзя. Например, если идет заказ на считывание данных, потом процесс их изменяет, а затем обратно записывает, то эти считывание и запись могут следовать лишь в одном порядке.

6.1.4 RAID-системы. Уровни RAID

Аббревиатура RAID может раскрываться двумя способами. RAID — Redundant Array of Independent (Inexpensive) Disks, или избыточный массив независимых (недорогих) дисков. На сегодняшний день обе расшифровки не совсем корректны. Понятие недорогих дисков родилось в те времена, когда большие быстрые диски стоили достаточно дорого, и перед многими организациями, желающими сэкономить, стояла задача построения такой организации набора более дешевых и менее быстродействующих и емких дисков, чтобы их суммарная эффективность не уступала одному «дорогому» диску. На сегодняшний день цены между различными по характеристикам дисками более сглажены, но бывают и исключения, когда новейший диск чуть ли не на порядок опережает по цене своих предыдущих собратьев. Что касается независимости дисков, то она соблюдается не всегда.

RAID-система — это совокупность физических дисковых устройств, которая представляется в операционной системе как одно устройство, имеющее возможность организации параллельных обменов. Помимо этого образуется избыточная информация, используемая для контроля и восстановления информации, хранимой на этих дисках.

RAID-системы предполагают размещение на разных устройствах, составляющих RAID-массив, порций данных фиксированного размера, называемых ***полосами***, которыми осуществляется обмен в таких системах. Размер полосы зависит от конкретного устройства

(при обсуждении файловых систем была упомянута иерархия различных блоков; RAID добавляет в эту иерархию дополнительный уровень — уровень полос RAID).

На сегодняшний день выделяют семь моделей RAID-систем (от нулевой модели до шестой). Рассмотрим их по порядку.

RAID 0 (Рис. 152). В этой модели полоса соизмерима с дисковыми блоками, соседние полосы находятся на разных устройствах. Организация RAID нулевого уровня чем-то напоминает расслоение оперативной памяти. С каждым диском обмен может происходить параллельно. Каждое устройство независимое, т.е. движение головок в каждом из устройств не синхронизировано. Данная модель не хранит избыточную информацию, поэтому в ней могут храниться данные объемом, равным суммарной емкости дисков, при этом за счет параллельного выполнения обменов доступ к информации становится более быстрым.

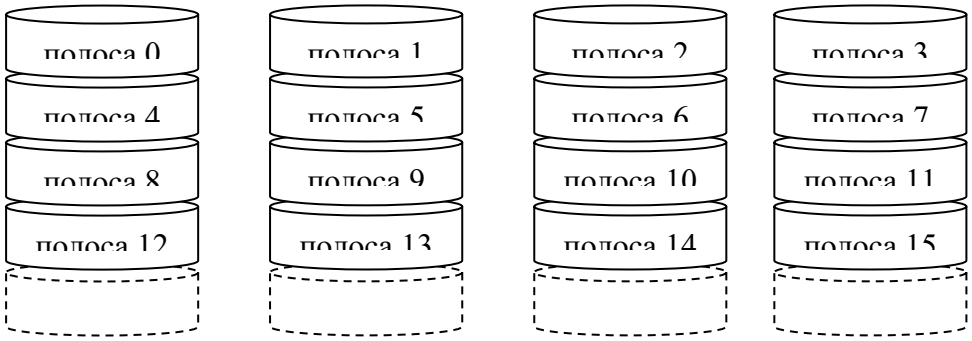


Рис. 152. RAID 0.

RAID 1, или система зеркалирования (Рис. 153). Предполагается наличие двух комплектов дисков. При записи информации она сохраняется на соответствующем диске и на диске-дублере. При чтении информации запрос направляется лишь одному из дисков. К диску-дублеру происходит обращение при утере информации на первом экземпляре диска.

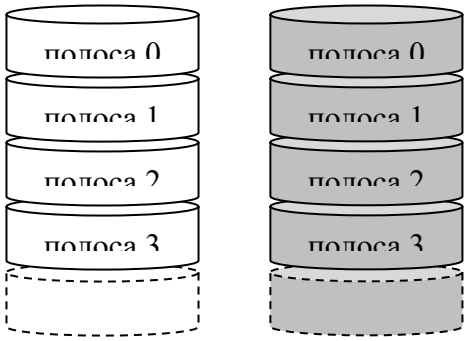


Рис. 153. RAID 1.

Данная модель является достаточно дорогой, причем дороговизна заключается не в непосредственной стоимости дисков (если предположить, что средняя цена диска 100 USD, то на покупку 10 основных дисков и 10 дисков-дублеров организация должна потратить порядка 2000 USD, что не является большой суммой для юридического лица). Цена вопроса встает при обслуживании данной системы: 20 дисков занимают много места, потребляют довольно приличную мощность и выделяют много тепла. К этому можно добавить расходы на обеспечение резервного питания: чтобы эти 20 дисков не отключались при перебоях с электроэнергией, необходимо закупить источники бесперебойного питания с очень емкими аккумуляторами.

Отметим, что модели RAID нулевого и первого уровней могут реализовываться программным способом.

Следующие две модели (**RAID 2**, Рис. 154, и **RAID 3**, Рис. 155) — это модели с т.н. синхронизированными головками, что, в свою очередь, означает, что в массиве используются не независимые устройства, а специальным образом синхронизированные. Эти модели обычно имеют полосы незначительного размера (например, байт или слово). Данные модели содержат избыточную информацию, позволяющую восстановить данные в случае выхода из строя одного из устройств. В частности, RAID 2 использует коды Хэмминга (т.е. коды, исправляющие одну ошибку и выявляющие двойные ошибки). Модель RAID 3 более проста, она основана на четности с чередующимися битами. Для этого один из дисков назначается для хранения избыточной информации — полос, дополняющих до четности соответствующие полосы на других дисках (т.е., по сути, в каждой позиции суммарное число единиц на всех дисках должно быть четным). И в том, и в другом случае при сбое одного из устройства за счет избыточной информации можно восстановить потерянные данные.

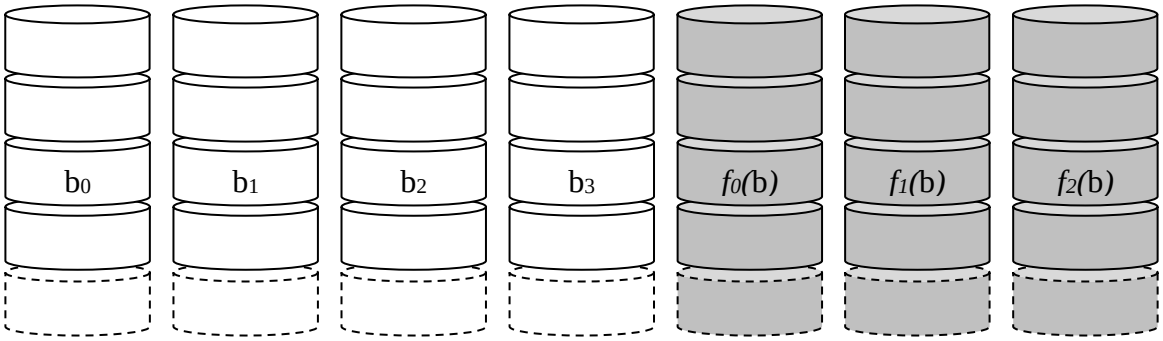
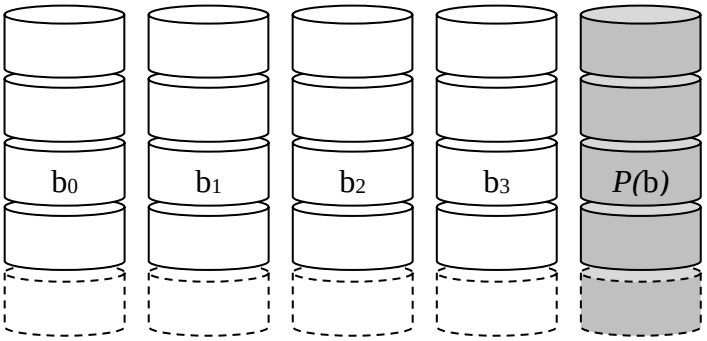


Рис. 154. RAID 2. Избыточность с кодами Хэмминга (Hamming, исправляет одинарные и выявляет двойные ошибки).



В данном случае имеется 4 диска данных и 1 диск четности. Тогда для диска четности:
$X_4(i)=X_0(i) xor X_1(i) xor X_2(i) xor X_3(i)$
Если произойдет потеря данных на первом диске, то для восстановления достаточно воспользоваться формулой:
$X_1(i)=X_0(i) xor X_2(i) xor X_3(i) xor X_4(i)$

Рис. 155. RAID 3. Четность с чередующимися битами.

RAID 4 является упрощением RAID 3 (Рис. 156). Это массив несинхронизированных устройств. Соответственно, появляется проблема поддержания в корректном состоянии диска четности. Для этого каждый раз происходит пересчет по соответствующей формуле.

Модели **RAID 5** (Рис. 157) и **RAID 6** (Рис. 158) спроектированы так, чтобы повысить надежность системы по сравнению с RAID 3 и 4 уровней. Оказывается опасным хранить важную информацию (в данном случае полосы четности) на одном носителе, т.к. при каждой записи происходит обращение всегда к одному и тому же устройству, что может спровоцировать его скорейший выход из строя. Надежнее разнести служебную информацию по разным дискам. Соответственно, RAID 5 распределяет избыточную информацию по дискам циклическим образом, а RAID 6 использует двухуровневую избыточную информацию (которая также разнесена по дискам).

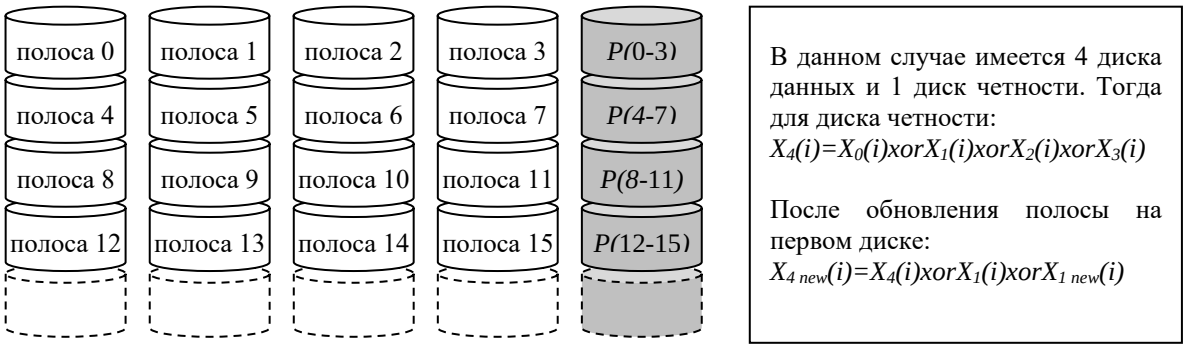


Рис. 156. RAID 4.

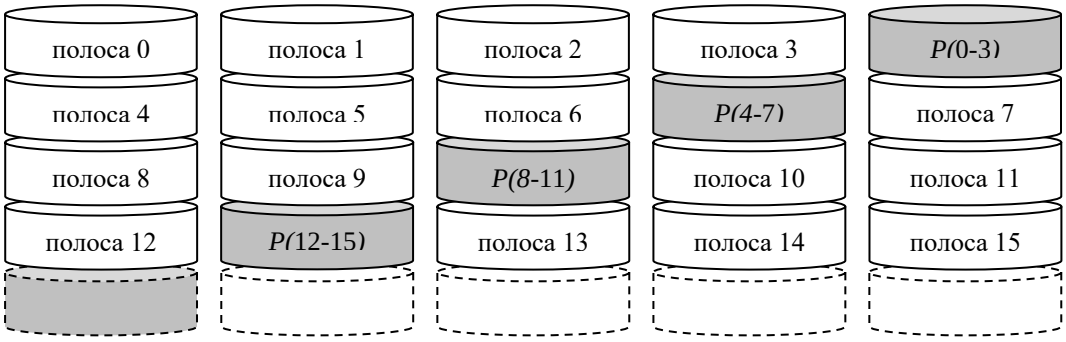


Рис. 157. RAID 5. Распределенная четность (циклическое распределение четности).

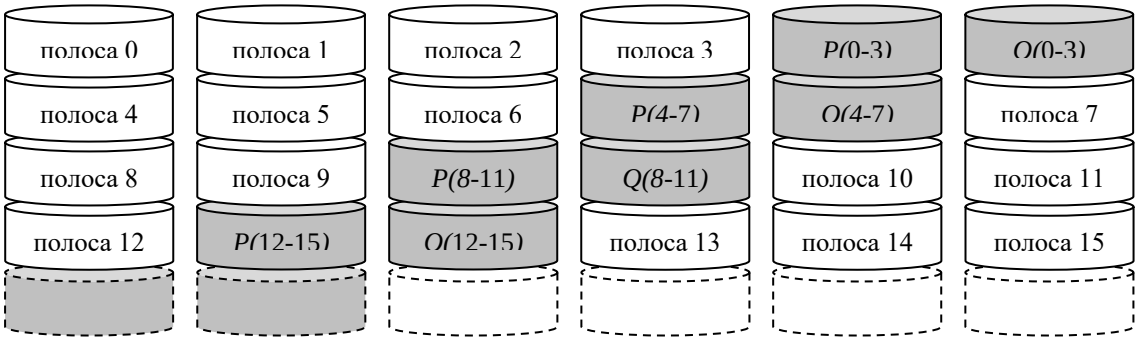


Рис. 158. RAID 6. Двойная избыточность (циклическое распределение четности с использованием двух схем контроля; требуется N+2 диска).

6.2 Работа с внешними устройствами в ОС Unix

6.2.1 Файлы устройств, драйверы

Как уже неоднократно упоминалось, одной из основных особенностей ОС Unix является концепция файлов: практически все, с чем работает система, представляется в виде файлов. Внешние устройства не являются исключением и также представлены в системе в виде специальных файлов устройств, хранимых обычно в каталоге /dev.

С точки зрения интерфейсов организации работы с внешними устройствами система делит абсолютно все устройства на две категории: ***байт-ориентированные*** и ***блок-ориентированные*** устройства. С блок-ориентированными устройствами обмен осуществляется порциями данных фиксированной длины, называемыми блоками. Обычно размер блока кратен степени двойки. Все остальные устройства относятся к байт-ориентированным. Такие устройства позволяют осуществлять обмен порциями данных произвольного размера (от 1 байта до некоторого k). Надо отметить, что к байт-

ориентированным устройствам помимо физических устройств (с которыми можно осуществлять обмен) могут относиться и устройства, с которыми обмен не осуществим. Примером такого устройства может служить таймер: реально обмены с таймером не происходят – он используется для генерации в системе прерываний через определенные промежутки времени, но таймер относится именно к байт-ориентированным устройствам.

Но, говоря о блок- и байт-ориентированных устройствах, следует помнить, что за регистрацию устройств в системе в конечном счете отвечает ***драйвер*** устройства: именно он определяет тип интерфейса устройства. Бывают ситуации, когда одно и то же устройство рассматривается системой и как байт-ориентированное, и как блок-ориентированное. В качестве примера можно привести оперативную память. Заведомо ОЗУ является байт-ориентированным устройством, но при организации обменов или при развертывании в оперативной памяти виртуальной файловой системы ОЗУ может рассматриваться уже как блок-ориентированное устройство. Также отметим, что априори считается, что те устройства, на которых может располагаться файловая система, являются блок-ориентированными.

Рассмотрим системную организацию информации, необходимой для управления внешними устройствами. Как упоминалось выше, в системе имеется специальный каталог устройств, в котором располагаются файлы особого типа — специальные файлы устройств. Эти файлы обеспечивают решение следующих задач:

- именование устройств (если быть более точными, то именование драйверов устройств);
- связывание имени, выбранного для именования устройства, с конкретным драйвером.

Соответственно, структурная организация файлов устройств отличается от организации, например, регулярных файлов. Специальные файлы устройств не имеют блоков файла, хранимых в рабочем пространстве файловой системы. Вся содержательная информация файлов данного типа размещается исключительно в соответствующем индексном дескрипторе. Индексный дескриптор состоит из перечня стандартных атрибутов файла, среди которых, в частности, указывается тип этого файла, а также включает в себя некоторые специальные атрибуты. Эти атрибуты содержат следующие поля: тип файла устройства (блок- или байт-ориентированное), а также еще 2 поля, позволяющие осуществлять работу с конкретным драйвером устройства, — это т.н. ***старший номер*** и ***младший номер***. Старший номер (major number) — это номер драйвера в таблице драйверов, соответствующей типу файла устройства. А младший номер (minor number) — это некоторая дополнительная информация, передаваемая драйверу при обращении. За счет этого реализуется механизм, когда один драйвер может управлять несколькими схожими устройствами.

6.2.2 Системные таблицы драйверов устройств

Для регистрации драйверов в системе используются две системные таблицы: таблица блок-ориентированных устройств — **bdevsw**, и таблица байт-ориентированных устройств — **cdevsw**. Соответственно, старший номер хранит ссылку на драйвер, находящийся в одной из таблиц; тип таблицы определяется типом файла устройств.

Каждая запись этих таблиц содержит структуру специального формата, называемую коммутатором устройства. Коммутатор устройства хранит указатели на всевозможные точки входа (т.е. реализуемые функции) в соответствующий драйвер, либо же в соответствующей записи таблицы вместо указанной структуры хранится специальная ссылка-заглушка на точку ядра.

Стоит отметить следующие типовые имена точек входа в драйвер:

- ***βopen()***, ***βclose()***;
- ***βread()***, ***βwrite()***;
- ***βioctl()***;
- ***βintr()***.

Символ β является аббревиатурой имени устройства: обычно в Unix-системах для именования устройства используют двухсимвольные имена. Например, lp — принтер, mt — магнитная лента и т.п.

В общем случае система специфицирует наиболее полный набор функций, который может предоставить драйвер пользователю. Если какая-либо функция отсутствует, то на ее месте в коммутаторе может стоять заглушка. Заглушки могут быть двух типов: заглушка типа *nulldev()*, которая при обращении сразу возвращает управление, и заглушка типа *nodev()*, которая при обращении возвращает управление с кодом ошибки. Например, для таймера скорее всего будут отсутствовать функции чтения и записи, причем при попытке чтения или записи система должна «ругнуться» (т.е. заглушка типа *nodev()*).

Некоторые из перечисленных точек входа являются специализированными. С помощью функции *βioctl()* можно производить разного рода настройки и управление драйвером. Функция *βintr()* вызывается при поступлении прерывания, ассоциированного с данным устройством.

Традиционно часть функций драйверов может быть реализована синхронным способом, а другая часть — асинхронным способом. Соответственно, синхронная часть драйвера называется **top half**, а асинхронная — **bottom half**.

6.2.3 Ситуации, вызывающие обращение к функциям драйвера

Список ситуаций, при которых происходит обращение к функциям драйверов, четко детерминирован. Во-первых, это старт системы и инициализация устройств и драйверов. При старте система имеет перечень устройств, которые могут быть к ней подключены. Этот перечень — содержимое каталога /dev. После этого она просматривает данный перечень и определяет те устройства, которые есть в наличии, а затем подключает их посредством вызова соответствующей функции коммутатора (функции *βioctl()*).

Во-вторых, это обработка запросов на обмен. Если процессу необходимо произвести считывание или запись данных, то происходит обращение к соответствующей точке входа в драйвер.

В-третьих, это обработка прерывания, связанного с данным устройством. Например, был инициирован обмен, и он закончился (успешно или неуспешно), или же по линии связи пришел какой-то сигнал, который необходимо обработать. В этом случае возникает прерывание, обработка которого происходит в соответствующем драйвере.

И, в-четвертых, это выполнение специальных команд управления устройством. Функции управления могут быть самыми разными, их наполнение зависит от конкретного устройства и от конкретного драйвера.

6.2.4 Включение, удаление драйверов из системы

Изначально Unix-системы (как и большинство систем) предполагали «жесткое» статические встраивание драйверов в код ядра. Это означало, что при добавлении нового драйвера или удалении существующего необходимо было выполнить достаточно трудоемкую операцию перетрансляции (когда ядро создается «с нуля») или, как минимум, перекомпоновку ядра (когда есть готовые объектные модули). Соответственно, эти операции требовали серьезных навыков от системного администратора. Чтобы минимизировать число перекомпоновок ядра, надо было максимизировать число драйверов, встроенных в систему. Но такая модель была неэффективной, поскольку в системе присутствовали драйверы, которые никак не используются.

Альтернативной моделью, существующей и по сей день, является модель динамического связывания драйверов. В этом случае в системе присутствуют программные средства, позволяющие динамически, «на лету» подключить к операционной системе тот или иной драйвер. Данная модель предполагает решение следующих задач. Во-первых, это задача именования устройства. Во-вторых, инициализация драйвера (т.е. формирование

системных областей данных и т.п.) и инициализация устройства (приведение устройства в начальное состояние). В-третьих, добавление данного драйвера в соответствующую таблицу драйверов устройств (либо блок-, либо байт-ориентированных). И наконец, «установка» обработчика прерывания, т.е. предоставление ядру информации, что при возникновении определенного прерывания управление необходимо передать в соответствующую точку входа в данный драйвер.

Для реализации указанной модели в различных системах имеются разные средства: разные системные вызовы и, соответственно, разные команды; при этом обычно присутствуют как команды подключения драйверов, так и симметричные команды удаления драйверов.

6.2.5 Буферизация при блок-ориентированном обмене

Одним из достоинств ОС Unix является организация многоуровневой буферизации при выполнении неэффективных действий. В частности, для организации блок-ориентированных обменов система использует стандартную стратегию кэширования. Действия на разных уровнях в целом очень похожи. Цель кэширования — минимизация обменов с внешними устройствами.

ОС Unix была одной из первых массово распространенных операционных систем, использующих кэширование дисковых обменов. Соответственно, за счет минимизации реальных обращений к физическим устройствам работа системы более эффективная. Но эта организация системы имеет и свои очевидные недостатки. Во-первых, кэширование дисковых обменов приводит к тому, что имеется несоответствие реального содержимого диска и того содержимого, которое должно быть на нем. Соответственно, при сбое системы возможна потеря информации в КЭШах, расположенных в оперативной памяти. В частности, при сбое возможна потеря индексного дескриптора. Конечно, во время работы система сбрасывает актуальную информацию по местам дислокации, но этого недостаточно. Если теряется индексный дескриптор, то теряется список блоков файла. За счет использования избыточной информации можно организовать и восстановление. Но заметим, что при сбое теряется лишь файл, - работоспособность системы остается.

Альтернативными являются системы, работающие без буферизации, когда при каждом обмене происходит реальное обращение к физическому устройству. Эти системы более устойчивы к сбоям в аппаратуре. Примером такой системы может служить Microsoft DOS. Соответственно, при развертывании на ненадежной аппаратуре операционной системы Unix многие ее положительные качества могли теряться.

6.2.6 Борьба со сбоями

Так или иначе, но в ОС Unix есть ряд традиционных средств для минимизации ущерба при отказах. Во-первых, в системе может быть задан параметр, определяющий промежутки времени, через которые осуществляется сброс системных данных по местам дислокации.

Во-вторых, в системе доступна команда sync, позволяющая осуществлять в любой момент этот сброс информации по желанию пользователя.

И, наконец, система использует избыточную информацию, позволяющую восстанавливать данные. Поскольку практически весь ввод-вывод сводится к обменам с файловой системой (т.е., по сути, идет борьба за сохранность файлов и файловой системы), то использование избыточных данных позволяет восстанавливать системную информацию. Обычно безвозвратные потери происходят с частью пользовательской информации; системная информация почти всегда восстанавливаема.